Compiling and Distributing Generic Libraries with Heterogeneous Data and Code Representation

Dissertation
der Fakultät für Informations- und Kognitionswissenschaften
der Eberhard-Karls-Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)

vorgelegt von
Dipl.-Inform. Roland Weiss
aus Ratibor

Tübingen
2003

Tag der mündlichen Qualifikation:	29.01.2003
Dekan:	Prof. Dr. Ulrich Güntzer
1. Berichterstatter:	Prof. Dr. Rüdiger Loos
2. Berichterstatter:	Prof. Dr. Sibylle Schupp
	(Rensselaer Polytechnic Institute, Troy, NY, USA)

To Hannah and Edda

Abstract

Generic programming has evolved at fast pace in recent years. There now exist numerous programming languages that support genericity, and generic libraries were developed that range from small utility libraries to huge general purpose or domain specific libraries. The importance of specification and verification was further emphasized in the context of generic algorithms and data structures, which became major issues in the research community. The largest generic code base is available in C++ which plays a central role for generic programming, because the Standard Template Library (STL) was implemented in C++. The STL ignited the application of its underlying principles to other domains, resulting in influential C++ libraries like the Matrix Template Library (MTL) or the Boost Graph Library (BGL).

This thesis deals with a problem that became apparent for languages like C++, Ada, or Modula-3 that support genericity with heterogeneous data and code representation, especially with regard to large generic libraries. These libraries cannot be compiled and distributed without their source code. Application programs using generic libraries will have to compile the occurring instantiations of generic constructs from the libraries' source code repeatedly during their development cycle.

We first present a general approach to overcome the conceptual problem that necessitates this practice. It mandates a strict separation of compiler front-end and back-end which are connected by a special intermediate representation. Then an overview of the GILF system is given, our incarnation of the solution. Although the GILF system was developed for SUCHTHAT, a generic programming language devised by Sibylle Schupp, GILF is designed to support a wide range of generic programming languages with varying semantics for instantiation, specialization and overloading. We achieve this by separating declarations from definitions, as well as splitting the instantiation process into analysis and application. The information gathered by instantiation analysis is propagated from a compiler's front-end to its back-end as explicit bindings of instantiation parameters to arguments in the mentioned intermediate language. GILF even allows binding multiple algorithms to one function symbol, which provides the basic mechanisms required for online or off-line algorithm selection. Algorithm selection can be exploited with instantiation at load-time or run-time.

Thereafter, we introduce XGILF, the XML-based external GILF representation. An extensive specification is elaborated, covering all core features of GILF. Finally, the GILF prototype implementation is discussed in detail, showing the feasibility of our approach.

Zusammenfassung

Generische Programmierung hat sich in den letzten Jahren mit beträchtlicher Geschwindigkeit entwickelt. Es existiert nun eine Vielzahl von Programmiersprachen die Generizität unterstützen. Weiterhin wurden generische Bibliotheken erstellt, die von kleinen Hilfsbibliotheken bis hin zu großen Universal- oder bereichsspezifischen Bibliotheken reichen. Die Bedeutung von Spezifikation und Verifikation wurde im Kontext generischer Algorithmen und Datenstrukturen weiter betont, deshalb entwickelten sich diese beiden Aspekte zu Kernpunkten in der Forschungsgemeinde. Die größte Basis an generischem Code existiert in C++, das eine zentrale Rolle für generische Programmierung spielt, da die Standard Template Library (STL) in C++ implementiert wurde. Die STL initiierte die Anwendung der ihr zugrunde liegenden Prinzipien auf andere Bereiche, was einflussreiche Bibliotheken wie die Matrix Template Library (MTL) oder die Boost Graph Libraray (BGL) hervorbrachte.

Diese Arbeit beschäftigt sich mit einem Problem, dass bei Sprachen wie C++, Ada95 oder Modula-3 deutlich wurde, die Generizität mit heterogener Daten- und Codedarstellung unterstützen, insbesondere im Hinblick auf große generische Bibliotheken. Diese Bibliotheken können ohne ihren Quellcode nicht übersetzt und verteilt werden. Anwendungsprogramme, die generische Bibliotheken verwenden, müssen in ihrem Entwicklungszyklus die auftretenden Instanzen generischer Konstrukte wiederholt aus dem Quellcode der Bibliotheken übersetzen.

Wir präsentieren zuerst einen allgemeinen Ansatz, der die Überwindung des konzeptuellen Problems ermöglicht, das dieses Vorgehen erforderlich macht. Es schreibt eine strikte Trennung des Übersetzer-Frontends und -Backends vor, die über eine besondere Zwischensprache verbunden sind. Anschließend geben wir einen Überblick über das GILF-System, unsere Realisierung der Lösung. Obwohl das GILF-System ursprünglich für SUCHTHAT entwickelt wurde, einer von Sibylle Schupp konzipierten generischen Programmiersprache, unterstützt sein Design eine weite Spanne von generischen Programmiersprachen mit variierender Semantik für Instanziierung, Spezialisierung und Überladung. Wir erreichen dies durch die Separierung von Deklarationen und Definitionen sowie der Teilung des Instanziierungsprozesses in Analyse und Applikation. Die mittels Instanziierungsanalyse bestimmten Informationen werden in der angesprochenen Zwischensprache als explizite Bindungen von Instanziierungsparameter an ihre Argumente vom Übersetzer-Frontend zum Übersetzer-Backend propagiert. GILF erlaubt es sogar mehrere Algorithmen an einen Funktionsbezeichner zu binden, was den grundlegenden Mechanismus zur Verfügung stellt, der für Offline- oder Online-Algorithmenselektion benötigt wird. Algorithmenselektion kann durch Instanziierung zur Lade- oder Laufzeit ausgenutzt werden.

Danach stellen wir XGILF vor, die XML-basierte externe Repräsentierung von GILF. Eine umfangreiche Spezifikation wird erarbeitet, die alle Hauptmerkmale von GILF behandelt. Abschließend wird die prototypische GILF-Implementierung diskutiert, die die Umsetzbarkeit unseres Ansatzes zeigt.

Contents

Chapter 1

Introduction

1.1 Motivation

Generic programming has attracted increasing attention recently. This applies to the research area as well as to the industrial field. The inclusion of the Standard Template Library (STL) into the C++ International Standard marks a milestone in the development of this relatively young field of research. The generic programming paradigm is especially well suited for structuring and implementing libraries, because it promotes a thorough examination of the problem domain at hand. Ultimately, this approach results in the problem domain's partition into orthogonal components, principally algorithms and data structures. The positive effect of this process is increased code reuse and flexible composability and adaptability of the components of a generic library, without sacrificing efficiency. The outcome of research efforts following the principles of generic programming shows in work like the STL, the Boost Graph Library (BGL), or the Matrix Template Library (MTL).

Despite the apparent success of generic programming and its positive impact on library design, traditional compilation systems support the development of generic libraries in a very limited way only. These compilation systems distribute libraries as archives of compiled object files containing machine code. This general practice fails for generic libraries in languages with nonuniform data representation, which is common to all widely used imperative languages, like C++, Ada95, or Modula-3. For those, the compiler does not know the memory layout of generic, uninstantiated data structures inside libraries, as user provided instantiation arguments can generate an infinite number of data structure layouts. The same holds true for generic algorithms. An operator in a generic algorithm can resolve to a function call or a machine code instruction, resulting in different code representations. In general, everywhere an algorithm manipulates data that depends on the algorithm's instantiation parameters, machine code generation is not possible. These problems lead to the current practice of distributing generic libraries as source code.

1.2 Contribution

The focus of this thesis is to propose a solution to overcome this unfortunate situation. Our work evolved in the context of the SUCHTHAT project. The design of SUCHTHAT, a new programming language introduced by Sibylle Schupp, was directed fundamentally by the motivation to create a generic programming language. Nevertheless, the work presented in this thesis is a general approach to building compiler and linker systems for generic programming languages with nonuniform data and code representation.

A central aspect of our system is the complete separation of a traditional compilation system into two distinct entities, the front-end and the back-end, and making this separation explicit. The front-end program handles lexical, syntactical and semantical analysis of the source code and produces an intermediate representation. Analyzing generic code can be very computation intensive, because it requires a powerful type system that checks the appropriateness of instantiations. Therefore, the intermediate representation should capture all these precomputed information. The back-end manifests itself as a code-generating linker and loader. It operates on units stored in the intermediate representation and creates a machine dependent executable program. The decision to defer code generation to link-, load-, or even runtime enables us to store generic libraries in the form of an intermediate language.

The machine architecture neutral intermediate representation is at the core of our solution to the compilation and distribution problem of generic libraries. The intermediate representation's characteristic of processor architecture independence is shared by SDE, the distribution format of the Oberon-3 system, Java Bytecode, the intermediate representation of the Java Virtual machine, and more recently, CIL, the intermediate language at the heart of Microsoft's .NET platform. All these formats do not support generic components, but rather are centered around the object oriented programming paradigm. Thus, our work concentrates on distilling the demands of generic programming on the constituents of an intermediate representation.

GILF, the **G**eneric **I**ntermediate **L**ibrary **F**ormat, represents our vision of an intermediate representation that supports generic programming adequately. It provides facilities for declaring generic functions and types, defining generic algorithms and data structures, and binding the definitions to the declarations. XGILF, the XML-based external representation of GILF, is the foundation of the prototypical implementation of a GILF back-end in modern C++. The nature of generic programming and GILF pave the way to new high level optimizations like profile driven algorithm selection and replacement at load- or runtime.

The studies of the demands of the generic programming paradigm on a compilation system that resulted in the architecture of the GILF system and the XGILF specification constitute the main contributions of this thesis to generic programming research.

1.3 Overview

The remainder of this thesis will proceed as follows. Chapter 2 first discusses general aspects of generic programming and then walks the reader through the major parts of our generic programming language SUCHTHAT. This includes the specification language, its integration with generic algorithm implementations, and code generation for instances of these generic algorithms. In chapter 3, the GILF compilation system is discussed at length, preceded by a rationale for the intermediate representation. Chapter 4 presents the XGILF specification, followed by a description of the implementation of the GILF prototype in chapter 5. Related work is considered in chapter 6 and chapter 7 concludes with an outlook on future work. The appendix contains more implementation details, a complete translation example, and the GILF core library declarations in XGILF.

Chapter 2

From Concepts to Machine Code

In this chapter we will give an informal introduction to generic programming. A small example guides us through all parts of a generic programming system and the focal interest points are identified and discussed. The discussion starts with the specification of generic concepts, then touches the implementation of generic algorithms and finally ends in looking at the problems of code generation for these algorithms, especially when collected in libraries. While elaborating on the example, the SUCHTHAT project [ScLo98] will be sketched incidentally, the context in which the work presented in this thesis evolved.

After having looked at all parts of a generic programming system we should be ready to formulate the requirements for an intermediate representation that supports compilation of generic programs with nonuniform data and code representation.

2.1 Generic Programming

When looking up the word *generic* in the Merriam Webster Dictionary, we get the following result:

Main Entry	**generic**
Etymology	French générique, from Latin *gener-*, *genus*: birth, kind, class
Date	1676
1 a	relating to or characteristic of a whole group or class
1 b	being or having a nonproprietary name
1 c	having no particularly distinctive quality or application
2	relating to or having the rank of a biological genus

Table 2.1: The entry for *generic* in the Merriam Webster Dictionary.

The first meaning of generic denoted by 1a in table 2.1 portends to the idea of generic programming. A generic algorithm characterizes a possibly unlimited group of concrete algorithms, the generic algorithm's instances. The same generalization is available for data structures.

This generality is achieved by introducing static type parameters in addition to common typed runtime value parameters. Then algorithms can be written using type variables which are not yet bound to concrete data structures. The user of a generic algorithm generates these bindings, either implicitly or explicitly, and thus by performing an instantiation the user creates an instance of the generic construct. Implicit bindings are possible when the static type parameters are deduced from the types of the dynamic value parameters at the calling site. The notion of static type parameters points to the important fact that generic programming is especially suited to be employed in languages with a strong static type system [Gas99][Gas01].

This is of course a rather technical view on generic programming, so let us inspect the objectives of generic programming from the software engineering perspective. Alexander Stepanov, one of the main propagators of generic programming, condenses its goals successfully:

> Generic programming is a discipline that studies systematic organization of useful software components. Its objective is to develop a taxonomy of algorithms, data structures, memory allocation mechanisms, and other software artifacts in a way that allows the highest level of reuse, modularity, and usability. ([MuDeSa01], p. xxi)

He emphasizes the systematic analysis of the problem domain such that one finally conceives a library of software components that can be flexibly combined. These components are primarily generic algorithms and data structures. Code or component reuse is accomplished by two means. The first one is the flexible composability of generic components which results from the separation of orthogonal concerns. The other one is the fact that the compiler performs requested instantiations automatically by substituting the instantiation parameters with the arguments. In traditional libraries, these instantiations have to be written by hand, e.g. LAPACK [AnBaBi$^+$90], SACLIB [CoLo90] or SAC-2 [HoNeSc95]. In these libraries, there is an abundance of concrete algorithms, performing practically the same generic algorithm, but written by hand for all the combinations of supported types, their representations and algorithms operating on them. Types and their representation are encoded in the function name, e.g. SGEBRD from LAPACK is a single precision (S) routine that performs a bidiagonal reduction (BRD) of a real general matrix (GE).

The type abstraction introduced by genericity does not necessarily result in a runtime penalty for instantiated algorithms. This is an important observation, as generic programming tries to yield algorithms that are as efficient as hand-coded ones after instantiation. Schupp mentions the *yardstick of efficiency* in her thesis [Sch96] as an important motif in generic programming[1]. Austern makes this especially clear:

> Generic programming, unlike object oriented programming, does not require you to call functions through extra levels of indirection, it allows you to write a fully general and reusable algorithm that is just as efficient as an algorithm handcrafted for a specific data type. ([Aus98], p. xvii)

The importance of the fact that the abstractness of the components comes without loss of efficiency is further stressed by not only making the requirements on syntax and semantics part of a generic component's interface, but also the requirements on their runtime complexity.

We really have to emphasize that requirements are a central aspect in generic programming. The STL is often called a *set of requirements* and Musser et al. even *define*

[1] She also talks about the other common usage of the word generic in information technology, e.g. generic hardware drivers, which usually counteracts the goal of efficiency. This meaning could be related to entry 1c in table 2.1.

generic programming as requirement oriented programming in [MuScLo98]. Concepts are collections of requirements, and the development of such concepts are the key to reusable generic components. Concepts allow us to rely on correctness of the instantiations and compositions we create form generic components. For Austern, concepts are at the heart of generic programming:

> Defining abstract concepts and writing algorithms and data structures in term of abstract concepts is the essence of generic programming. ([Aus98], p. xvii)

2.2 Exploring Genericity with the Factorial and Gamma Function

We will now examine all relevant aspects of genericity in more detail by looking at the ordinary factorial function $n!$, with $0! = 1$ and satisfying the functional equation $(n+1)! = (n+1) \cdot n!,\ \forall n \in \mathbb{N}$.

2.2.1 Mathematical Background

While generalizing the factorial function we will visit the closely related gamma function. The gamma function for positive reals is defined by the the integral

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt,\ \forall x \in \mathbb{R}^+ \tag{2.1}$$

The gamma function fulfills the functional equation $\Gamma(x+1) = x \cdot \Gamma(x),\ \forall x \in \mathbb{R}^+$ and has the nice property that $\Gamma(n+1) = n!,\ \forall n \in \mathbb{N}$, i.e. the gamma function takes the values of the factorial function for natural numbers, but offset by one. We want to state the following theorem without proof that allows one to characterize the gamma function with the mentioned functional equation. If for functions $F : \mathbb{R}^+ \to \mathbb{R}^+$ these properties hold

$$\begin{array}{ll} \text{(a)} & F(1) = 1 \\ \text{(b)} & F(x+1) = x \cdot F(x),\ \forall x \in \mathbb{R}^+ \\ \text{(c)} & ln \circ F \text{ is convex} \end{array} \tag{2.2}$$

then $F = \Gamma$.

Its is even possible to extend the definition set of the gamma function to negative reals. One way is to use the form contributed to Gauss:

$$\Gamma(x) = \lim_{n\to\infty} \frac{n! \cdot n^x}{x \cdot (x+1) \cdot \ldots \cdot (x+n)},\ \forall x \in \mathbb{R} \backslash \mathbb{Z}_0^- \tag{2.3}$$

Notice the poles at all negative integer values of x. Another popular form for the gamma function is the Weierstrass formula:

$$\frac{1}{\Gamma(x)} = x e^{\gamma x} \prod_{n=1}^{\infty} \left(1 + \frac{x}{n}\right) e^{-\frac{x}{n}} \tag{2.4}$$

where $\gamma = lim_{n\to\infty} 1 + \frac{1}{2} + \ldots + \frac{1}{n} - ln(n)$ is Euler's constant. The reflection theorem is instrumental in the computation of the gamma function on negative reals as it reduces the definition of $\Gamma(-x)$ to $\Gamma(x)$:

$$\Gamma(-x)\Gamma(x) = \frac{-\pi}{x sin(\pi x)} \tag{2.5}$$

To conclude our short overview of the gamma function, a plot of the gamma function is presented in figure 2.1. Proofs for the presented theorems can be found in Analysis textbooks like [Heu00].

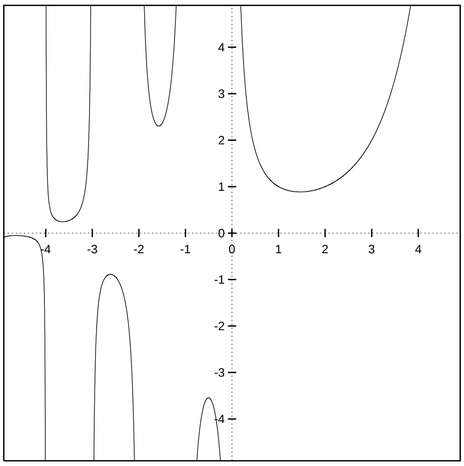

Figure 2.1: Plot of the gamma function.

2.2.2 Genericity

After briefly recapitulating these mathematical facts, we will write a generic algorithm to compute the factorial function in C++.

"../src/examples/factorial.cpp" 6 ≡

```
#include <iostream>
// Function template to compute factorial.
template <typename N>
N factorial(const N& n)
{
  // Initialize the result and counter.
  N m = 1;
  N i = 1;
  // Iteratively compute the factorial.
  while (i <= n)
    m = i++ * m;
  // Return the result.
  return m;
}
// Test two instantiations of the factorial algorithm.
int main()
{
  std::cout << factorial<>(12) << " "     // instantiation type deduced as int
            << factorial<long double>(12) // instantiate with type long double
            << std::endl;
  // Output: 479001600 4.79002e+08
}
```

Function templates are the C++ language feature to write generic algorithms. A function template, or to be more general, a generic algorithm cannot be translated into machine code and executed. Only *instantiations* of generic algorithms result in executable code.

The process of instantiating a generic algorithm is defined as binding the algorithm's type parameters to concrete data structures, i.e. resolving the genericity. In the example above, the first instantiation creates a concrete factorial algorithm for type `int`. The type `int` is deduced from the argument passed to the function. The second instantiation explicitly binds the type parameter to the machine type `long double`. Here, we encounter one of the fundamental questions of generic programming:

How does one determine which instantiations are valid? (P1)

In C++, this question has a very pragmatic answer. If the function template is syntactically correct after replacing all occurrences of the type variables with their bound concrete types, the instantiation is valid. Therefore, we have no guarantees at all that the instantiated generic algorithm will compute any reasonable results, i.e. the instantiation may succeed syntactically, but the operations defined for the bound type lead to erroneous behavior. We have to consider another core issue of generic programming:

How does one state the requirements on the instantiation arguments? (P2)

The requirements imposed on a generic algorithm's type parameters can be either semantical and syntactical, i.e. specification requirements, or algorithmic. To state syntactical requirements is the simplest task and it would be reasonable to expect a C++ language mechanism to specify at least the syntactical requirements that a template argument has to meet. Even in our simple factorial function template, a concrete type that is bound to the template type parameter `N` must support these operations:

default construction	`N()`
assignment	`N& operator=(const N&)`
post increment	`N& operator++(int)`
less than equal comparison	`operator<=(const N&, const N&)`
multiplication	`operator*(const N&, const N&)`
constants	`1`

Table 2.2: List of implicitly required operations in the C++ factorial function template.

The only way a user of the factorial template can obtain this information is by looking at the function's C++ source code. This means, the requirements on template arguments are available implicitly only in C++. This is known as unconstrained genericity [Eve97] or unbounded polymorphism. Modula-3 falls into this category, too.

Ada95 [Bar95], Cecil [Cha98], Theta [LiCuDa$^+$95], and other languages offer possibilities to restrict instantiation parameters to types that conform to a given interface at the syntactical level, as well as to types that have special properties related to the language's type system, like a subtype relationship.

To offer even more powerful instantiation checking capabilities it would be beneficial to be able to state semantical requirements. Thus, integrating methodologies like algebraic specification [AsKrKr99][Kla83] or abstract state machines [Gur00] into generic programming languages is a desirable goal. Even further away are ways to express algorithmic requirements, which enforce the efficient execution of instantiated generic algorithms:

> It should be noted that while the operational semantics of the operations can be specified rigorously by specifying the set of valid expressions and their semantics, the complexity is specified informally; a totally new insight is needed to find a way for specifying complexity requirements in a rigorous but practically useful way. ([MuDeSa01], p. xxiv)

The two discussed problems influence the design of generic programming languages, but another important question comes up when we create code for generic algorithms.

How does one translate generic algorithms to machine code? (P3)

This problem heavily affects the design and implementation of a compiler system for generic programming languages. There are two fundamental approaches.

The first one creates only one machine representation for all instantiations. This way, all concrete types must have a uniform representation. This kind of data representation is usually refered to as boxed or homogeneous [Ler98][Ler97][OdWa97]. The implementation of boxed types consists of a pointer to the heap that holds the data's value and on some metainformation to identify the type. Most functional programming language implementations follow this approach, e.g. Haskell, ML or CAML, but also the proposed extension to Java will support genericity this way [BrCoKe+01].

The other approach creates a distinct machine representation for each instantiation of a generic component. This is necessary if the programming language employs a nonuniform data representation. Not until instantiation time does the compiler know the exact layout of the bound data structures, which it needs in order to be able to create machine code that manipulates this data. C++, Modula-3 and Ada95 act like this. The uniform representation, enabled by adding a pointer indirection, carries a performance hit. This is not the case for nonuniform representations, because the data is accessed directly. Therefore we decided to take this approach, as one goal of genericity is to achieve efficiency on par with hand-coded programs. The implications of this decision will be expatiated upon in section 2.5.

The three problems P1, P2, and P3 are detailed in the following sections, along with approaches and solutions applied in the SUCHTHAT project.

2.3 Algebraic Specification in SUCHTHAT with TECTON

We already mentioned the importance of requirements for generic programming and would therefore recommend that the specification of requirements should be an integral part of a generic programming language. In the SUCHTHAT project, we decided to integrate TECTON [KaMu92][Mus98], an algebraic specification language.

In TECTON, functions and sorts are combined in concepts together with their requirements. Function signatures can be seen as syntactical requirements, whereas the semantics of the operations are introduced with `requires` and `generates` clauses. TECTON allows sort, function and concept replacements in concept instantiations, a powerful ability especially well suited for our purposes.

A library of algebraic concepts [MuScSc+99] was formulated in TECTON and we will now extend this library in order to provide the specification requirements for the factorial example. This extensions are of explanatory nature and are not intended to be proven as the most general ones. They will aid us in motivating some of the constructs that constitute parts of the GILF intermediate representation. All the extensions to the original library were tested with the TECTON checker by Rüdiger Loos.

We start by introducing the concept `Functional-equation-factorial` that contains an unary operator F[2] that fulfills the general functional equation $F(x+1) = g(x) \cdot F(X)$. Therefore, we restrict the domain on which the involved functions operate to be sorts of a semiring with multiplicative identity, which introduces a successor function `succ`. Notice the `Precedence` statement that allows one to extend the set of available operators at run time.

"../src/examples/factorial.sth" 9a ≡

```
Definition: Semiring-with-multiplicative-identity
  refines Semiring, Identity;
  introduces
    succ: domain -> domain;
  requires (for d: domain)
    succ(d) = d + 1.

Precedence: {*} < {F}.

Definition: Functional-equation-factorial
  uses Semiring-with-multiplicative-identity;
  introduces
    F: domain -> domain,
    g: domain -> domain;
  requires (for x: domain)
    F(x + 1) = g(x) * F(x).
```

File defined by parts 9a, 9b, 10, 11, 12a, 12b, 13.

We need to define a semiring with identities as the last prerequisite before we can introduce a general factorial function. This is done by refining the semiring with multiplicative identity. The factorial function is now introduced in an extension to the concept `Semiring-with-identities`. For this purpose, we rename operator F from the factorial functional equation as operator ! and identify function g with the successor function. We still have to add the requirement $0! = 1$ and finally arrive at a very general notion of the factorial function.

"../src/examples/factorial.sth" 9b ≡

```
Definition: Semiring-with-identities
  refines Semiring-with-multiplicative-identity,
          Identity [with + as *, 0 as 1].

Precedence: {*} < postfix{!}.

Extension: Semiring-with-identities
  uses Functional-equation-factorial
    [with Semiring-with-identities as Semiring-with-multiplicative-identity,
          ! as F, succ as g];
  introduces
    ! : domain -> domain;
  requires (for d: domain)
    0! = 1.
```

[2]The reason for declaring F as operator with the `Precedence` sentence is a technical one. When we want to use the postfix operator ! for the factorial function later, it would not be possible to replace F with this operator if F is not an operator itself.

File defined by parts 9a, 9b, 10, 11, 12a, 12b, 13.

Based on the introduced concepts one can implement a factorial algorithm that works for example on naturals. It may be written almost in terms of the given functional requirements. We will examine the interconnections between algorithms and concept specifications in the next section.

But what about the gamma function? The gamma function enables a generalization of the factorial function on naturals to reals, but is a specialization of the introduced general factorial function, offset by one. So in order to come up with a factorial function that can handle reals, we will give a TECTON specification for the gamma function now.

Again, we have to extend and define some concepts first. Function `g` from the concept `Functional-equation-factorial` collapses to the identity function in case of the gamma function. Therefore we add the identity function `id` in an extension of the `Domain` concept. Because we specify the gamma function for positive reals only[3], such a concept is defined. In the concept `Positive-Real` we introduce sort `naturals` as a subsort of sort `positives`, because we want to relate the gamma function at natural number values to the factorial function later on. Finally, the concept `Real` is extended with the logarithm and exponentiation functions, which will be needed to specify the convexity property.

"../src/examples/factorial.sth" 10 ≡

```
Extension: Domain
  introduces
    id: domain -> domain;
  requires (for d: domain)
    id(d) = d.

Definition: Positive-Real
  refines Real, Semiring-with-multiplicative-identity;
  introduces
    positives < reals,
    naturals < positives,
    +: positives x positives -> positives,
    *: positives x positives -> positives,
    1: -> positives;
  requires (for p, q: positives; x, y: reals)
    (p = x and q = y) implies p + q = x + y,
    (p = x and q = y) implies p * q = x * y,
    1 * p = p,
    x: positives = (0 < x),
    (for p: positives)
      p: naturals = (⌊p⌋ = ⌈p⌉).

Extension: Real
  uses Positive-Real;
  introduces
    e: reals -> positives,
    e: -> positives,
    ln: positives -> reals;
  requires (for x, y: reals; p: positives)
```

[3] For the explanatory example it is sufficient to work with the definition of the gamma function for positive reals given in equation 2.1. In the introductory section we showed the extension to negative reals (see equations 2.3 and 2.4), and a complex gamma function exists also. But these more general functions would have required an even larger set of accompanying concepts.

```
    e(x + y) = e(x) * e(y),
    e(1) = e, not(e = 1),
    x < y implies e(x) < e(y),
    ln(e(p)) = p, e(ln(p)) = e.
```

File defined by parts 9a, 9b, 10, 11, 12a, 12b, 13.

Now that the algebraic prerequisites are provided for our purposes, we extend the concept `Real` with the gamma function by renaming the function `F` and `g` from the general functional equation as Γ and `id`, respectively. Notice how we state the requirement that $ln \circ \Gamma$ is convex, which is needed to reduce `F` to be the gamma function uniquely. This requirement could be condensed into a concept of its own. This is not trivial because we have to argue not simply on sorts and functions, but on function compositions, in our case $ln \circ \Gamma$.

The last step is to introduce the factorial function to the concept `Real`. We achieve this by using a `Lemma` sentence which states that operator `!` is just a shift of the gamma function to the left by one. Before, the concept `Natural` is extended with the semiring with identities[4].

"../src/examples/factorial.sth" 11 ≡

```
  Extension: Real
    uses Domain [with positives as domain],
         Functional-equation-factorial [with
           Positive-Real as Semiring-with-multiplicative-identity,
           positives as domain,
           Γ as F, id as g];
    requires (for λ, x, y: positives)
      Γ(1) = 1,
      (0 < λ and λ < 1) implies (for some z: positives)
        ln(Γ(λ * x + (1 - λ) * y)) < (λ * ln(Γ(x)) + (1 - λ) * ln(Γ(y)))
        where z = 1 - λ.

  Extension: Natural
    uses Semiring-with-identities [with naturals as domain, ! as !].

  Lemma: Real requires (for n: naturals) Γ(n + 1) = n!.
```

File defined by parts 9a, 9b, 10, 11, 12a, 12b, 13.

The abstract concepts for defining generic factorial algorithms are developed now and in the next section we will show how to write such algorithms in SUCHTHAT and how they are integrated with the TECTON concepts.

2.4 Integrating Specification and Implementation

We can now give the implementation of the factorial algorithm. We start with the algebraicly general one that works for all semirings with identities.

"../src/examples/factorial.sth" 12a ≡

[4]This is needed as the concept `Natural` is introduced as an elementary concept in [MuScSc$^+$99], i.e. it does not participate in the the concept hierarchy like `Real`, `Integer` etc.

```
Algorithm: r := factorial(n) uses Semiring-with-identities
Input:     n ∈ domain.
Output:    r ∈ domain such that r = n!.
Local:     i ∈ domain.
(1) // Initialize r and i.
    r := 1;
    i := 1.
(2) // Compute factorial.
    while i ≠ n do { r := i * r; i := i + 1 }.
(3) return r □
```

File defined by parts 9a, 9b, 10, 11, 12a, 12b, 13.

This is an iterative factorial algorithm, a straight forward recursive version is also very common. It is crucial to connect the algorithm's implementation to the function signature from a concept. We accomplish this by using the function operator `!` in the `such that` clause. As `n` $\in$ `domain` of the concept semiring with identities, and `domain` is a sort of this concept, the operator `!` is found. Therefore, the algorithm `factorial` will be recognized as one of the operator's implementations.

An important observation on this algorithm is that the termination property is guaranteed only if the input value `n` can be reached from 1 by repeatedly adding 1. Counterexamples are all real numbers that are not naturals. The importance of verifying generic algorithms is stressed in [Sch97].

Writing an algorithm to compute the gamma function is a tougher problem. One common approach to this numerical computation problem is the Stirling formula

$$\left(\frac{x}{e}\right)^x \sqrt{2\pi x} < x! < \left(\frac{x}{e}\right)^x \sqrt{2\pi x} \cdot \left(1 + \frac{1}{12x - 1}\right) \tag{2.6}$$

which gives a good approximation of $x!$ for large positive x, but fails to deliver good results for x close to 1. Very accurate approximations are possible with Lánczos' approximation [Lan64] for the whole definition set, which is an improvement of the Stirling formula that allows thorough control of the relative error. The most complex part in implementing Lánczos' approximation is calculating special coefficients. Spouge [Spo94] proposed a variant of the Lánczos approximation which simplifies this part, the coefficients are given by simple formulas.

Yet another approach for computing the gamma function is mentioned by Bronstein et al [BrSeMu00]. Using the functional equations

$$\Gamma(x) = \frac{\Gamma(x+1)}{x} \quad \text{and} \quad \Gamma(x) = (x-1)\Gamma(x-1) \tag{2.7}$$

it is possible to reduce the calculation of $\Gamma(x)$ to tables on some interval of length 1, e.g. $x \in [1.5, 2.5[$. We resort to an algorithm from Moshier [Mos89] that implements this method, because the elemental arithmetic operations addition, subtraction, multiplication, and division are needed only. It returns good approximations for typical floating point machine types.

"../src/examples/factorial.sth" 12b ≡

```
Algorithm:  r := gamma_moshier(x) uses Integer, Real, Positive-Real
Input:      x ∈ positives.
Output:     r ∈ positives such that r = Γ(x).
Local:      n, k ∈ integers, w ∈ reals, y ∈ positives.
```

```
(1) // Initialize constants.
    if x < 1.5 then n := -⌊(2.5 - x)⌋ else n := ⌊(x - 1.5)⌋;
    w := x - (n + 2).
(2) // Interpolate 1/Γ(l), l ∈ [1.5, 2.5[.
    y := (((((((((((-0.000000199542863674 * w + 0.000001337767384067) * w -
       0.000002591225267689) * w - 0.000017545539395205) * w +
       0.000145596568617526) * w - 0.000360837876648255) * w -
       0.000804329819255744) * w + 0.008023273027855346) * w -
       0.017645244547851414) * w - 0.024552490005641278) * w +
       0.191091101387638410) * w - 0.233093736421782878) * w -
       0.422784335098466784) * w + 0.999999999999999999.
(2) // Main loop:
    // x >= 2.5: Use Γ(x) = (x − 1)Γ(x − 1).
    if n > 0 then
    {
      w := x - 1; k := 2;
      while  k <= n do { w := w * (x - k); k := k + 1 }
    }
    // x < 1.5: Use Γ(x) = Γ(x+1)/x.
    else
    {
      w := 1; k := 0;
      while k > n do { y := y * (x - k); k := k - 1 }
    }.
(3) return w / y □
```

File defined by parts 9a, 9b, 10, 11, 12a, 12b, 13.

An algorithm for the factorial function for positive reals is now just a normalized call to the gamma function.

```
"../src/examples/factorial.sth" 13 ≡
   Algorithm:  r := factorial(x) uses Positive-Real
   Input:      x ∈ positives.
   Output:     r ∈ positives such that r = x!.
   (1) // Normalized call to the gamma function.
       return Γ(x + 1) □
```

File defined by parts 9a, 9b, 10, 11, 12a, 12b, 13.

There are several points we want to elaborate on. They will influence the design of our intermediate representation and point to new areas of further research.

Multiple Implementations and Specializations Before presenting one algorithm to compute the gamma function, we mentioned two others. This is a fundamental property of functions. There is usually more than one generic algorithm to compute the same function, but with different trade-offs. A prominent example are sorting algorithms. They all perform the same task, they sort a sequential range of elements delimited by two iterators according to an order relation. Very general algorithms are applicable to a great variety of input ranges but may not execute as efficient as possible when applied to ranges that would allow more specific algorithms, e.g. quicksort requires random access to the elements in the range, whereas mergesort also works on ranges with sequential access to the elements. Algorithms that operate on a limited set of types, compared to the most general algorithm, are called *specializations*, a very important technique in generic programming. Specializations put more constraints on their type parameters and exploit these properties to produce more efficient realizations in terms of space or run time. A specialization may restrict the type parameters even to one type, an example from the C++ Standard Library is the `vector<bool>` specialization of the `vector` container. It is optimized to occupy only one bit per element in memory. Summarizing, a function can be implemented by multiple generic algorithms, and some of them, the specializations, may be instantiated for a restricted set of type arguments only.

Algorithm Selection The insight that a function may be realized by several algorithms leads to the interesting question which algorithm should be selected from the set of applicable candidates. Furthermore, which criteria should guide the algorithm selection? Obvious assistants are the instantiation types. Specializations are introduced to exploit properties of more specific instantiation types, so it seems natural to select more specialized algorithms or data structures if they are available. But sometimes the size of the input sequence or even the order of the elements in the input sequence to a sorting algorithm will determine which will perform best. For some applications, not the runtime efficiency of an algorithm will be the decisive factor, but the quality of the numerical approximation. This area in generic programming research is still very unexplored and a categorization of the most important aspects combined with the appropriate selection strategies is needed. Of course, the programmer should always have the possibility to force the selection of the one algorithm he thinks is best suited for the given task.

Function Identifiers Generic algorithms are written at an abstract level, i.e. the operations used to express the generic algorithms are generic themselves and are represented by overloaded function identifiers. When instantiating an algorithm, the used operations must be bound to instantiated algorithms. Without the algorithm selection step, this process has to produce a unique algorithm. A first prototypical implementation of SUCHTHAT showed that Baker's overload resolution algorithm [Bak82] extends to generic overload resolution nicely. The function identifiers used in expressing generic algorithms in SUCHTHAT have their origin in TECTON concepts, thus the TECTON availability rules apply. Also, a renaming of the identifiers is possible to adapt function identifiers to be used in other concepts[5]. SUCHTHAT has to provide a convenient way to link an algorithm to the function it implements. This is done with the `such that` clause, which ties the algorithm to a function identifier. In our example, the algorithm `gamma_moshier` is bound to the Γ identifier.

2.5 Generating Code for Generic Algorithms

In the preceding sections we detailed the development of abstract concepts and generic algorithms that implement functions introduced in such concepts. Now we want to examine the transition from generic algorithms to executing code on a real machine.

2.5.1 The Instantiation Process

Generic algorithms cannot be translated into machine code as is based on a nonuniform data representation. The reason is that the size of data structures replacing the generic

[5] In TECTON, the renaming of sorts and concepts is possible, too.

algorithm's type variables is not known until the type variables are actually bound to concrete data structures. The same is true for operations manipulating these data. For instance, if the factorial function is instantiated with machine word integers, addition can be bound to a built-in machine operation. For arbitrary precision integers, addition will be bound to a function call that adds integers in this representation.

Therefore, code generation can proceed after instantiation only, where the missing information is provided. To attain this goal, the following steps are necessary:

1. The generic algorithm must be instantiated, i.e. all type parameters must be bound to instantiation types.
2. The instantiation types must be bound to data structures that realize them on the machine level. If some of these data structures are generic, they have to be instantiated themselves.
3. The function identifiers used in the generic algorithm must be bound to algorithms. If some of these algorithms are generic, they have to be instantiated themselves.
4. The types used in the generic algorithm must be bound to data structures. If some of these data structures are generic, they have to be instantiated themselves.
5. The fully instantiated algorithm is compiled into machine code.

We see that the instantiation of a generic algorithm is a recursive process. The recursion ends when all data structures and algorithms used are finally bound to built-in machine types and operations, respectively. Not until a generic algorithm is fully instantiated code generation is possible. We will exemplify the transition process from generic to concrete algorithm with a simple code fragment that prints the factorial for naturals in the range from 1 to 10.

```
"../src/examples/factorial_test.sth" 15 ≡
   Algorithm: Test_Factorial uses Natural, IOStream.
   Local: i ∈ naturals.
   (1) // Display factorials from 1 to 10.
       for i = 1, ..., 10 do { CharOut << i! } □
```

The call to the factorial function is in the body of the `for` statement. The SUCHTHAT front-end has to check that operator `!` is really available. Then it has to validate its implicit instantiation with naturals. If these conceptual checks succeed, all algorithms that implement the operator `!` have to be collected, as well as data structures that realize the type `naturals`. We assume that the algorithm `factorial` from the previous section and a built-in data structure `uword` are found. The data structure `uword` should represent the native unsigned machine word on any given architecture. Step 1 and 2 of the transition process are finished now.

Step 3 involves binding the function calls in the algorithm body to the realizing algorithms. Table 2.3 shows these bindings for the factorial algorithm. The left column shows the function signature taken directly from TECTON concepts. In the middle column the requested instantiations are listed, for our example the types in the function signature are replaced with the data structure `uword`, the bound type parameter. Finally, the right column shows the selected algorithms.

All functions are bound to machine operations, so the recursion ends. Assignment is also treated as a built-in operation. As the factorial algorithm works exclusively with the type parameter `domain` which is bound to `uword`, step 4 is skipped. So we arrive at a fully instantiated algorithm, which can be compiled into machine code and executed.

function	instance	algorithm/realization
`≠(domain, domain) -> bool`	`≠(uword, uword) -> bool`	machine operation
`+(domain, domain) -> domain`	`+(uword, uword) -> uword`	machine operation
`*(domain, domain) -> domain`	`*(uword, uword) -> uword`	machine operation
`1 -> domain`	`1 -> uword`	conversion to `uword`

Table 2.3: Bindings of function signatures to algorithms in algorithm `factorial`.

2.5.2 Overview of a Traditional Compilation System

For further discussion of code generation for generic programming languages, we will first describe the architecture and functioning of a traditional compilation system. To this end, we identify the entities relevant for the compilation process:

Compilation Unit A compilation unit is an input stream that is passed to the compiler front-end after preprocessing. It is the compiler's view on the source project for each run.

Object File An object file contains machine code for data structures and algorithms that results from processing a compilation unit with the compiler.

Library A library is a collection of object files that contains related algorithms and data structures.

Application An application is a collection composed of object files and libraries that were linked to form an executable file.

Figure 2.2 contains a schematic overview of how an application program is created from source code.

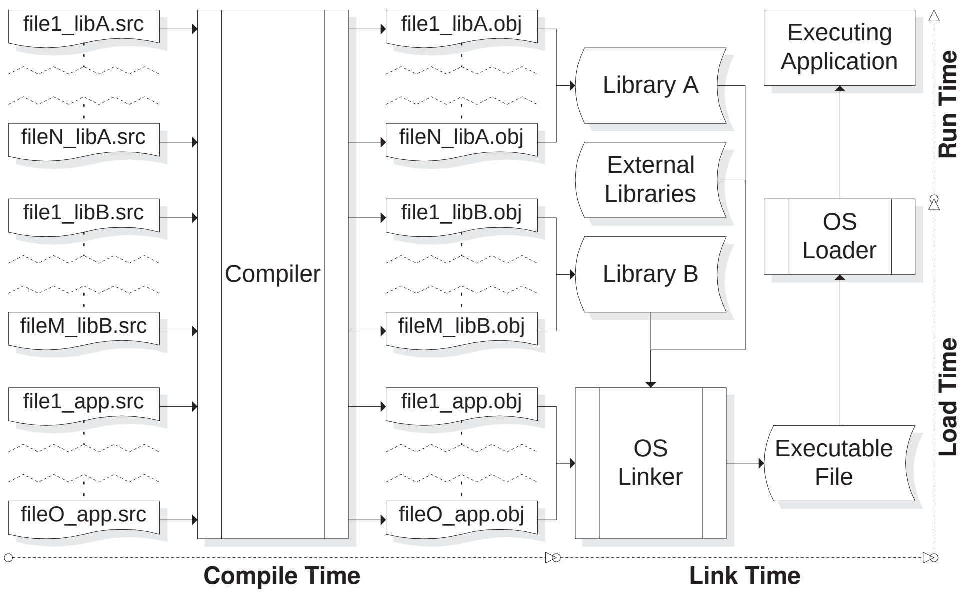

Figure 2.2: Flowchart of a traditional compilation system.

A typical software project consists of several input files written in the source language of choice. These files either belong to libraries or the application main program. All the source files are passed to the compiler. The compiler will generate object files out of these source files that contain machine code ready for execution on the host machine.

The object files generated from source files `*_libA.src` are combined in library A, files `*_libB.src` result in library B. A library is usually an archive of the object files with pointers to their location in the archive as well as an index of the symbols available in the object files. Often, the index is augmented with metainformation about the symbols, like access rights. Libraries are created by the compiler itself or by a simple external tool, like `ar` on most Unix systems.

Now the executable file for the application program is created. This involves linking the program's object files together with application specific libraries and external libraries. The external libraries can be operating system libraries, part of the language environment or some third party libraries. The linker will replace all symbolic references, e.g. in function calls and branches, with the location where the corresponding definition for the construct represented by the symbol is placed inside the application. So the linker's main task is symbol resolution.

The linked application program can now be started by the user. This is done by the loader. The loader loads the application's executable file into main memory, sets the appropriate memory flags and starts the program's execution by jumping to a defined position inside the now occupied memory.

A problem that can not be easily attributed to either the linker or the loader is relocation. Relative addresses in separate parts constituting the application have to be mapped to unique addresses. Linkers typically assume an address space starting at zero and perform relocation based on this assumption. When the loader places the application into memory, this has to be readjusted if the operating system does not support zero-based address spaces for every application.

The most elaborate tool in this system is the compiler and we will give some more details on its operation because the instantiation process must be performed in its property. The compiler performs lexical analysis of every source file, followed by syntactical analysis. Lexical analysis operates on characters and partitions the input into tokens that correspond to terminal symbols of the source language grammar. The parser that performs the syntactical analysis then creates a syntax tree or parse tree out of the tokens according to the source language grammar. Usually the syntax tree is further transformed into an abstract syntax tree that is better suited for further processing. Semantical analysis completes the task of the front end. The input is know checked to conform to the source language rules that go beyond syntactical constraints. This can be the type system of the language, structuring constraints of the language constructs like position of variables, and any other kind of context handling. Having performed all these operations, the compiler has produced an intermediate representation of the input file. This part of the compiler is called front-end, it is highly programming language dependent.

The back-end is responsible for translating the intermediate representation created by the front-end into machine code. This is achieved by continuously applying transformations to the intermediate representation such that the final code will be optimized for runtime or space efficiency without sacrificing correctness. An important technique in the back-end is the lowering of the intermediate representation. The result of the front-end processing is fairly high-level and not well suited for some optimization algorithms. Therefore, the back-end will replace complex constructs in the intermediate representation with more simple ones that are closer to the target machine and allow a more concise formulation of the optimization algorithms. Furthermore, representations closer to the target machine make it easier to exploit optimization opportunities particular to the machine architecture. Figure 2.3 shows the described separation of a compiler into front- and back-end.

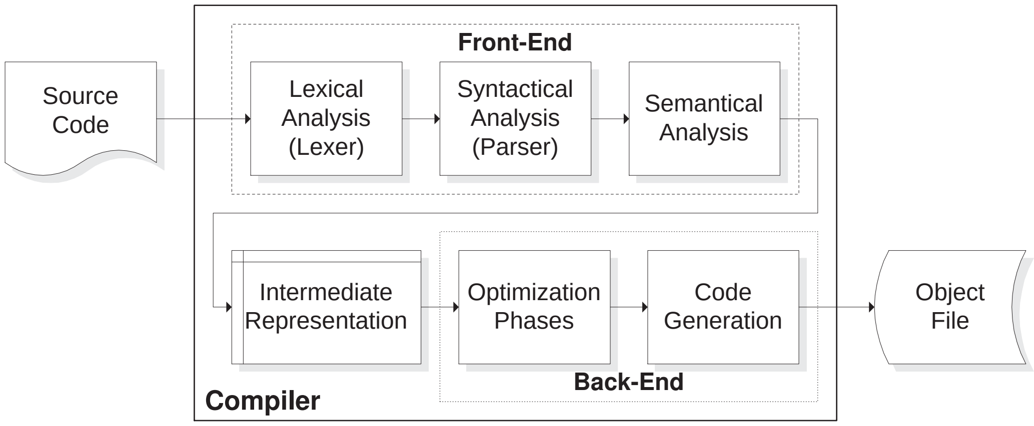

Figure 2.3: Traditional internal compiler structure.

The two parts of a compiler, the front-end and the back-end, are perceived as one component by the developer, the separation was introduced primarily to enable a modular compiler implementation. Therefore, they are normally combined in one program.

To conclude our compiler overview the concepts of time in a traditional compilation system are discussed. They are represented in figure 2.2 by the dashed lines at the lower and right border.

Compile Time This is the time when the compiler executes. The developer is interested in short compilation times because this speeds up the development process, but powerful language features and good optimizers take their toll.

Link Time At link time the linker performs symbol resolution and relocation of the application's object files to produce an executable file.

Load Time This is the period of time from the access of the executable file on secondary storage to the start of execution of the loaded application. Relocation often takes place in this phase, too.

Run Time The whole period of time an application performs operations is called run time. All languages require a more or less sophisticated runtime environment that supports program execution. The runtime system is responsible for tasks like memory management, stack frame management or method dispatch based on runtime type information.

For a more detailed discussion on all aspects of compiler construction refer to textbooks like [AhSeUl86][App98][GrBaJa+00].

2.5.3 Incorporating Instantiation into the Compilation Process

The critical question is how can the instantiation process be incorporated into a traditional compilation system?

The first question that arises is if generic algorithms should be type checked *before* or *after* instantiation takes place. In languages like C++ and Modula-3, which have no means of specifying the requirements on the instantiation types, the only exercisable option is type checking the instantiated constructs. But this has the negative side effect that error messages are not at the level of abstraction at which the developer thinks, because only the symptoms could be diagnosed, not the cause. In a system like SUCHTHAT that is based on a firm foundation of concepts describing the generic components, their semantic check is possible before the instantiation. This is preferable, as error diagnostics should be created as soon in the development pipeline as possible. More important, the error messages are given relative to generic components, not their instances.

The next problem is the stage at which instantiation should be integrated into the compiler. For a monolithic application consisting of one compilation unit the answer is quite easy. First, the front-end checks the input, then all instantiations are performed and the resulting intermediate representation is subject to code generation. In this scheme, instantiation could be seen as the last chain link of the compiler front-end. This is feasible because in a monolithic application generic components are directly instantiated and all instances are visible to the compiler which are further translated into machine code.

But introducing generic libraries complicates matters. Generic libraries are collections of uninstantiated generic components. As we already mentioned, no code generation is possible for these components. Thus, the back-end could not operate. The missing information are the bindings of type parameters to types and their data structure representations, as well as the bindings of algorithms to function signatures. How do current imperative languages like C++ handle this problem? Generic component libraries have to be available in source code such that whenever an instantiation is requested in an application program the template parameters are replaced by the template arguments and the instantiated template will be fed to the compiler for type checking and code generation. This means that linking a generic library in C++ does not only involve symbol resolution and relocation, but actually invoking the whole compiler for all requested instantiations in the main application. This is also true for generic library creation if the library instantiates external generic components itself as well as for applications that are build by linking several object files.

This current practice of linker and compiler interaction has several drawbacks that we want to discuss now.

Repeated Compilations Every instantiation of an external generic component forces a complete compile run. Effectively, the same source code if compiled over and over again which can be a very lengthy operation for generic libraries. The generic library has to be processed at the highest abstraction level many times, at the source code level. This is a clear waste of resources, namely development time and processor usage. The impact of this situation is a well known problem in the C++ community. Adding generic components to a project, like container and algorithms from the C++ Standard Library, result in a drastic increase in compilation and linking time.

Multiple Definitions This is another severe problem that has to be handled by existent compilation systems. All parts of a software project, like libraries and object files, can contain instantiations of generic components. When the parts are linked together, and some of these instantiations were identical, the linker will encounter multiply defined symbols. Consider an instance of the factorial function template for type `int` that is requested in two distinct compilation units of an application. The compiler sees only one compilation unit at a time and therefore machine code will be generated in both resulting object files. The approach taken in most contemporary C++ systems is to simply remove multiply defined symbols in the linking phase that resulted from template instantiations. There are pragmatic approaches to cope with this problem, like the prelinking phase in C++ compilers based on the EDG front-end [Edi00][Lip96]. Requested instantiations are recorded in an external repository and distinct instantiations are assigned to object files. This avoids multiple definitions in one project, but it resurfaces in separate libraries, because their repositories do not know of each other. Thus, the prelinking approach works only for a single project or library, not across these boundaries.

Source Distribution The library creator has to distribute its library in source code. For commercial products this is often a problem because the own source code embodies the company's achievement and should be protected from public exposure. Prepackaged distributions of libraries are also more convenient for the users as they do not have to care about compiling the library and configuring the development environment. Of course, open source projects also have their merits. Programmers can check and fix misbehavior in third party components directly. Furthermore, it leads to a high degree of transparency, which becomes important with regard to security issues.

We have to draw a conclusion from all these observations: A traditional compiler no longer meets the demands of a generic programming language because object files and libraries could not be compiled to machine code if they contain uninstantiated generic components. For languages like C or nongeneric C++, every compilation unit contains sufficient information such that it can be translated into an object file which contains architecture specific machine code. This is no longer true in the case of generic programming languages with a nonuniform data representation. So how should we deal with the fact that a generic component can be translated into machine code only when its complete instantiation is known?

The pivotal insight is that the first time a software project with all its dependencies is available to the compilation system is at link time. At this point the set of all used instances of generic components can be computed and code generation is possible. There exists a tension between the time of code generation endorsed by the traditional design of a compiler, at compile time, and the time when it is first possible in generic settings, at link time.

2.5.4 Easing the Tension

We propose a different approach to a compilation system for generic programming to overcome this tension. The code generating back-end should be no longer part of the compiler. In fact, the compiler should consist of a front-end that performs lexical, syntactical and semantical processing of the source code. This includes overload resolution, operator precedence parsing, type checking and concept checks. The front-end produces an intermediate representation which will be used as distribution format for libraries and possibly application programs. This high level representation of source code contains the full information available in the source code, but simplified and attained information is made explicit for fast processing by later stages in the compilation process. In the next chapter we will introduce the intermediate representation GILF, our generic intermediate library format.

Code generation has to be relocated in another part of either the compilation or the operating system. The best candidates are the linker and the loader. Both have their advantages and drawbacks. We will consider them in the next chapter also.

The big benefit we gain from splitting the compiler up into two programs is that we can now integrate the instantiation process in an elegant and uniform way into the compilation system. For that purpose, the instantiation process is also divided into two steps, instantiation *analysis* and *application*. Instantiation analysis is the more complex one and is performed in the compiler front-end. It has to determine the legality of instantiations, analyze implicit instantiation requests and collect the possible algorithms and data representations for generic components[6]. Finally, the binding information identified this way is stored in the intermediate representation.

The back-end's instantiation step consists of recursively applying the bindings, the instantiation application. This involves replacing type variables in generic constructs and generating code for the now instantiated constructs. Looking back at the factorial example, the processing would be as follows:

1. The front-end reads the source files `factorial.sth` and `test_factorial.sth` and performs lexical, syntactical and semantical analysis. This includes type and availability checking, verifying the legality of instantiating `factorial` with `naturals`, and collecting factorial algorithms and representations for `naturals`.

2. The front-end generates the intermediate representation of the input. It contains a representation of the factorial algorithm as well as bindings of the factorial function call to describe the instantiation in the main test program and the candidate representations.

3. Assuming a code generating loader, the intermediate representation will be translated into an executable for the host machine when starting the factorial test program. This process includes replacing the type variables inside the representation of the factorial algorithm with the chosen data structure for `naturals`.

Figure 2.4 shows the structure of such a compiler that is made up of two separate programs and how instantiation analysis and application is integrated.

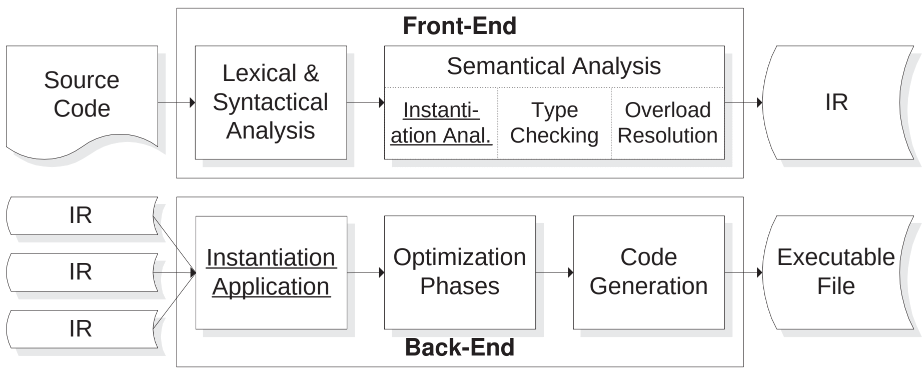

Figure 2.4: Structure of a compilation system for generic programming languages.

[6]This operation is tightly coupled with overload resolution.

2.6 Summary

In this chapter we have taken a complete tour through a generic compilation system, biased towards the SUCHTHAT project. We now want to summarize the three major tiers of such a system and how they contribute to the intermediate representation.

Specification First we described the procedure to specify generic components with the algebraic specification language TECTON. Of course, all the semantic checks have to be performed by the front-end and should not burden the back-end, as the operations carried out on this level can be very expensive in terms of run time. Some decisions needed in the concept checking phase are even delegated to off-line archives that store verified proofs and other knowledge that would lead to unacceptable compilation times; see [Sch97]. But the specifications introduce functions and types. Their signatures will enter the intermediate representation in the form of function and type declarations as they are the common basis to which multiple implementations of algorithms and data structures are bound.

Generic Algorithms and Data Structures The next tier in a generic compilation system is the programming language in which generic algorithms and data structures are expressed. We restricted our evaluation to languages based on the procedural imperative paradigm because it naturally resembles the functioning of current general purpose micro processors. Therefore, such languages allow good control over the efficiency of the compiled program at the programming language level. Languages centered on other paradigms that require a more sophisticated runtime system can be build on top of imperative languages.

Procedural languages are characterized by programming with a state and commands which modify the state. State is captured in typed variables at different scope levels, and commands modify these variables and control the program flow based on its current state. Thus, the intermediate representation contains definitions of algorithms that are expressed in terms of statements. Data structure definitions describe the memory layout of an algorithm's typed variables and constants. An algorithm's set of local variables and constants is collected in storage definitions. Notice that the algorithm and data structure definitions are bound to the generic signatures of functions and types from the specification tier and are therefore generic themselves.

Instantiation and Code Generation The last tier is the execution level on a real platform. To arrive at this level the intermediate representation has to provide two facilities. First, nongeneric, built-in operations and data structures must be supported. These are needed to actually compute results on the host machine. There should also exist a mechanism to convert constants to these built-in data structures, i.e. a feature to set values.

Second, a way to describe instantiations of generic components has to be provided. This will ultimately reduce generic algorithms and data structures to the built-in entities, as described in section 2.5.1. We point out the separation of instantiation analysis and application in figure 2.4. As the back-end should be concerned with instantiation application only, the outcome of the analysis phase should be stored in the intermediate language. Therefore, the bindings of the static instantiation parameters of generic components to their instantiation arguments should be explicitly included in the intermediate representation.

This tiered view of generic programming is shown in figure 2.5, using the example discussed in this chapter. The first tier contains selected concepts and central function signatures. At the top, we have the concept `Functional-equation-factorial` with the function `F`, which is renamed to the factorial operator `!` in `Real` and `Semiring-with-identities`. The next tier consists of algorithms implementing the functions, the example contains no data structures. Finally, these algorithms are instantiated for some data types. Of course, the number of instantiations is possibly unlimited.

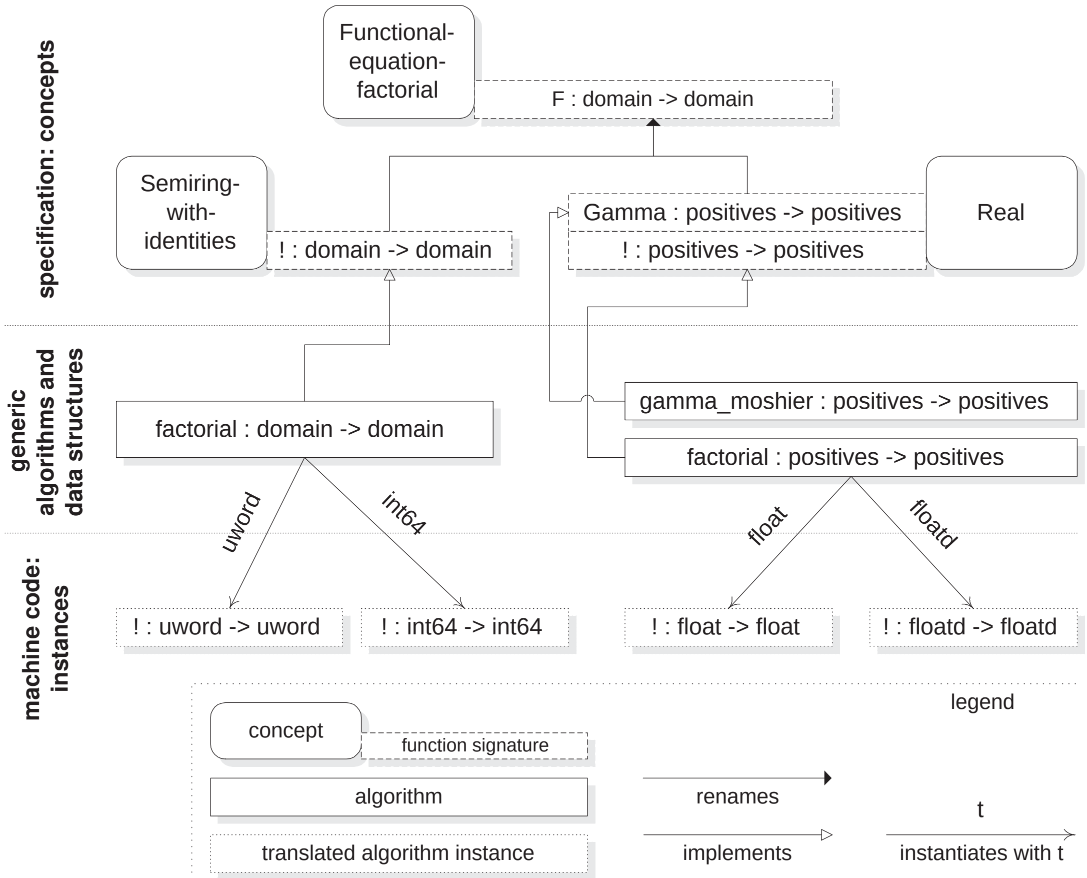

Figure 2.5: Tiered view of the factorial example.

There exist two transitions between the three tiers discussed in this chapter, the first one from algebraic concepts to generic algorithm and data structure definitions, the second one from these definitions to machine code. Both pose interesting challenges for generic programming research. This work concentrates on solving the second one by presenting an intermediate representation for generic source code and the accompanying compilation system infrastructure.

Chapter 3

The GILF Compilation System

In the previous chapter we provided an introduction to generic programming and identified the major problems in contemporary compilation systems, especially when used for generic library construction. Based on the observations made, we proposed an alternative approach for structuring a compilation system that better meets the demands raised by generic programming.

Now we will describe the GILF compilation system in more detail. The major functional units are presented, as well as how they interact. The glue between all these units is GILF, the intermediate language which is targeted by generic programming language front-ends. Therefore, we start our discourse on the GILF compilation system by looking at the rationale for our intermediate representation.

3.1 Rationale for the Intermediate Representation

The design of an intermediate representation has to balance the influence of several parameters. The most important ones are the abstraction level at which language constructs are expressed, and the structure and encoding in which the information extracted by the language front-end are stored.

These possibilities in designing an intermediate language for a compilation system have to be weighed up against the demands of generic programming we identified in the last chapter. Mainly, these include the capabilities to represent uninstantiated generic algorithms and data structures, type and function signatures, as well as bindings between these entities. Summarizing, we state the following design goals for GILF:

Genericity Naturally, this is the mainspring in the design of GILF. The intermediate representation should allow storing the information gathered in the instantiation analysis step of an imperative generic programming language. Furthermore, efficient instantiation application should be possible from GILF.

Portability Another goal of GILF is to provide a portable platform from which code can be generated to an arbitrary machine architecture. This is reasonable, because actual code generation takes place on the target machine, not in the development environment. The generated code should preferably take advantage of the particular features available at the target site.

Machine Processability Although information is stored in GILF at a rather high level, it should make these data easily accessible to the subsequent processing by the compilation system. For example, the language front-end parses the source code and performs operations like associativity and precedence resolving of operator expressions. It would be a waste of resources if the back-end has to compute these results again.

Source Language Neutrality This work is intended to examine the basic requirements of an intermediate representation with respect to generic programming languages. Consequently, GILF was designed to provide these facilities without dedicating itself to one source language. This requires more work when implementing the front-end, because source language constructs have to be mapped to GILF features. On the other hand, the generality of ideas may be explored when targeting GILF from a variety of languages.

Extensibility Finally, GILF in its current form was mainly designed to solve the problems arising in case of nonuniform datatype representation and manipulation of generic components. Therefore, other aspects that are common practice in contemporary intermediate languages were neglected. GILF should be able to provide some of these features through an extension mechanism.

3.1.1 Abstraction Level

The abstraction level at which language constructs are represented is the most central decision to be made in designing an intermediate representation. The language constructs are usually divided into the following three categories:

1. Control Flow
2. Operations and Datatypes
3. Data Flow

Control Flow In general, the higher the abstraction level, the more easily a mapping or transformation from source language constructs to the intermediate representation is performed. For control flow constructs this means that the intermediate language contains constructs that closely resemble those from source languages, like `if` instructions or looping constructs. These conditional control constructs determine the execution of enclosed statement sequences. An additional advantage of this approach is that no information present in the source code is lost, e.g. the presence of loops and nesting of control structures. This enables platform independent optimizations. The drawback of high-level constructs are that condition evaluation and statement sequences are separated. This fact complicates optimizations like common subexpression elimination. Furthermore, the branching structure of the program code is not explicitly available, which precludes optimizations on this lower level. But good instruction scheduling is vital on contemporary architectures, which are very sensitive to pipeline stalls due to data and control hazards (see [HePa96], 3.3ff).

The other extreme is making the branching structure of the source code completely visible. Assembler languages behave this way, conditional and unconditional branches determine the control flow of assembler instructions. Control-flow graphs are an established way of representing control flow at this low level [AhSeUl86]. These directed graphs are very general as arbitrary control flow can be expressed with them. Branches are modeled as edges of the graph, nodes model *basic blocks*. Basic blocks are instruction sequences that contain no branches or labels, i.e. control flow may only enter at the beginning and leave at the end of a basic block. A problem of control-flow graphs is that the original source program's structure is mostly lost, thus the optimizer's work is hindered. There exist algorithms with linear time complexity in the size of the source code that build control flow graphs.

Operations and Datatypes Operations can be expressed at different abstraction levels, also. Analogous to control flow constructs, operations can be represented at the level of the source language. Such a translation means that optimizations at this level will be independent of the target machine. The problem is that the instructions in the intermediate representation will be transformed into several machine instructions, thus effective optimizations at the machine level are not possible.

Another possibility is storing operations as machine instructions. This has the benefit that the operation level of the intermediate representation can stay the same during all processing in the compiler software, because at some point the compiler has to generate machine code. Most algorithms in optimizing compilers can work effectively at this abstraction level. The disadvantage of such a representation is the loss of information inherent in the transformation from source code to machine code. Another problem is relevant when considering a representation for generic programming languages. The source code before instantiation of generic algorithms is not fit for representation at machine code level. Consider the following C++ code snippet, which shows a function `f` that contains two lines of code at some point in its body.

```
template <typename T>
void f() {
  // ...
  T i;          // [1]
  i = i + 1;    // [2]
  // ...
}
```

Line 1 declares a variable `i` of the type parameter `T`, which will be increased by one in line 2. The addition operation could resolve to a function call if type `T` will be bound to a user defined datatype, or to a machine operation if it is bound to a built-in datatype like `int`. Special constructs would be needed in a machine level representation of generic code. Machine code is of course highly target machine specific.

A variant of machine code representations at an even lower abstraction level are register transfer languages [1]. The operations at this level are very simple microcode operations of a hypothetical machine, real machine operations usually perform the work of few micro operations. A register transfer language as intermediate representation is portable, because the micro operations can be implemented by most real architectures. Problems of this low-level representation are the larger required memory space, compared to the other mentioned representations, and the need for a pattern matcher that combines micro operations to machine code.

The representation of datatypes can be either close to high-level programming languages, that means the records, arrays etc. are representable, or machine datatypes are used directly. Again, in a generic setting only instantiated datatypes can be represented with machine datatypes.

[1] The GNU Compiler Collection (GCC) [Sta01] uses a register transfer language as its intermediate representation.

Data Flow Finally, we have to consider data-flow representations. The knowledge at which point variables and temporaries change their values through assignments is vital for compiler optimizations. For example, if a variable is assigned a value that leads to a condition in an `if` statement that will always yield the boolean value `false`, and data-flow analysis shows that this variable is never changed before the conditional evaluation, the dead code can be eliminated effectively. There are two possibilities to design an intermediate language with regard to data-flow, *multi-assignment* and *single-assignment* intermediate languages. Multi-assignment follow the semantics of typical imperative languages closely as they allow a variable to be assigned different values multiple times during program execution. Most algorithms based on multi-assignment languages operate on basic blocks. Traditional data-flow analysis works with bit-vectors, which are computed for basic blocks. A basic block's bit-vector contains information like the availability and liveness of its variables.

More recently, static single-assignment (SSA) form [CyFeRo+91] has been established as data-flow representation in optimizing compilers (see [Mor97] and [Muc97]). The key property of SSA form is that each variable is assigned a value only once. This is achieved by renaming a variable from the source language every time it is assigned a new value. A simple examples illustrates this transformation. The left hand side shows an instruction sequence in multi-assignment form, the right hand side shows the the same code in static single-assignment form:

i	:=	20	i_1	:=	20
j	:=	i + 1	j_2	:=	i_1 + 1
k	:=	i + 10	k_3	:=	i_1 + 10
i	:=	k * 2	i_4	:=	k_3 * 2
x	:=	i + 1	x_1	:=	i_4 + 1

The result is a program that has almost functional form [App98b]. Analysis of the intermediate representation is simplified if it is in SSA form, therefore development of optimization algorithms is facilitated.

In the previous chapter we determined the essential constructs that should be part of an intermediate representation. It became clear that a high-level representation is desirable, as we have to store uninstantiated algorithms. A low-level representation would have to contain special high-level constructs, mostly eliminating the advantages of a low-level representation. Therefore, we favored an intermediate representation that is closer to high-level source languages than to the target machines. This way we have a portable, target independent format that allows us to perform high level transformations. This is important for generic programming, as depending on some of the generic function's input parameters, which can be the instantiation's types or actual values of the call, we may want to switch between different algorithm implementations of the same function. For actual code generation, instantiated algorithms and data structures are lowered to some format that is more suitable for an optimizing code generator.

3.1.2 Structure

After deciding to use a high-level representation for GILF, based on constructs found in imperative languages, the next pending decision is how to structure the information collected in GILF. There are two main constituents that make up the stored information:

- Abstract Syntax Tree

- Symbol Table

The most natural structure for these information in compiler construction is a tree-like representation. Source code has an implicit hierarchical structure that is easily mapped to a tree. For example, procedures contain instructions, which in turn contain expressions, which are made up of variables, constants and function calls. Arbitrary additional data, usually called annotations, can be stored in the tree data structures, which enriches the pure syntax tree. Most compiler construction textbooks devote considerable space to the treatment of syntax trees and related data structures [AhSeUl86][App98][GrBaJa+00]. Global and local symbol tables can simply be treated as special nodes in a tree. A tree representation can contain all the information present in the source code. Figure 3.1 shows a short code sequence and its abstract syntax tree representation.

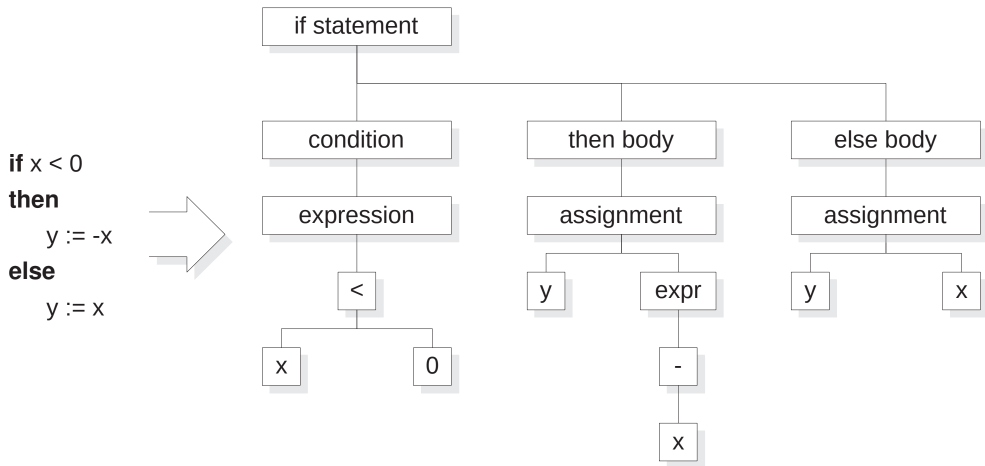

Figure 3.1: A statement sequence and its corresponding abstract syntax tree.

Program source code can be linearized and represented in tabular form without loss of information. This approach is pursued in the Oberon-3 system, which is based on SDE [Fra94], the semantic dictionary encoding. It has similar properties as tree representations.

Another popular representation are abstract stack machines (see [AhSeUl86], 2.8). Abstract stack machines provide primitive instructions for arithmetic, stack manipulation, and control flow. These operations operate almost exclusively on operands from the stack, e.g. branches take the relative offset from the stack. Expressions are converted into postfix notation, which can be translated into stack machine instructions straightforward. Notice that the original source code has to be transformed to be representable as stack machine, and some information is lost in this process, for example expressions have to be flattened in order to fit the evaluation model of stack machines. The most prominent contemporary example for a stack-based intermediate language is the Java Virtual Machine [LiYe99]. Stack machines are easy to interpret, because of the simple instructions and the unfolded expressions. Unfortunately, they do not fit very well the register-centric evaluation model of todays RISC architectures. Thus, for effective optimizations parts of the already performed transformations on the source code have to be undone.

A representation closely related to abstract stack machines overcomes this problem, namely abstract assemblers. An abstract assembler tries to model the most important aspects of a wide range of popular processor architectures. Then, the abstract code should be easily translated into real assembler dialects. Projects related to an universal computer oriented language (UNCOL) date back to the fifties [Con58]. The incentive of these projects was to mitigate the problem in compiler construction of writing $n \cdot m$ compilers, if for n source languages m assembler languages have to be written. With a abstract assembler as intermediate representation, this is reduced to $n + m$ compilers, because for every assembler only one back-end has to be written that translates from the abstract assembler, which is targeted by the front-ends. Most optimizations work on the abstract assembler as it models real processors adequately. Interpretation is more complex than with stack machines. Recently, there is a trend towards enriching assembler dialects with high-level language features and supporting strong static typing [MoCrGl+99] [ECMA01].

GILF has a hierarchical, tree-like structure. This makes it accessible both for computational purposes and for the human reader. Furthermore, we wanted to retain as much information as possible from the original source code in spite of being source language neutral, which is feasible in a tree notation.

3.1.3 Encoding

An intermediate language that is also used as distribution format like GILF has to consider its external as well as its internal representation. The internal representation is subject to the GILF implementation, which is described in chapter 5. There are two general alternatives to encoding the external representation of GILF:

- Binary Encoding
- Textual Encoding

A binary encoding is more compact than its textual counterpart. SDE, later on called *Slim Binaries* [FrKi96], shows that such an encoding can be significantly smaller than executable machine code. This is of importance, because nowadays mobile code is of increasing relevance. Mobile code is transmitted over networks and executed on the receiving target machine, like browser applets. In this cases, a tight encoding saves network resources. The drawback of a binary encoding is that it is not human readable without dedicated tools.

This is the primary advantage of a textual external representation. A large set of available tools can be used to inspect and modify textual formats and the content of a text file can be grasped by a human reader directly. Recently, a lot of work has been invested into XML [XML00], which is a textual tree representation at its core. So we adopted XML as the representation format for the prototype system. All XML capable browsers can be used to display it and the wide range of available software, like parsers, class libraries, and converters, is waiting to be exploited. The linking capabilities of XML make it an ideal choice for a library format, different parts of the library's components can be referenced inside the very file, inside the file system, and even on the world wide web. An intermediate representation formulated in XML has one more powerful feature, we get platform independence at the external representation level as XML is based on the Unicode Standard [Uni00]. For more details on XML, refer to section 4.1.

The GILF system was designed to handle more than one encoding of the external intermediate representation. The current prototype deals only with the XML-based GILF encoding, called XGILF. But the system is prepared for adding other encodings without major rewrites. A more thorough discussion of this feature is given in chapter 5 and appendix B.

3.2 Infrastructure of a GILF System

We have motivated and outlined the content of a GILF entity, which can contain either library or application code. The fact that no machine code is stored inside GILF entities necessitates drastic changes to a system's infrastructure, as discussed in section 2.5. Now we will present the complete infrastructure for a compilation and runtime system build around the GILF intermediate representation. Figure 3.2 displays a general overview of such a system, and we will explain the specific components in the next sections.

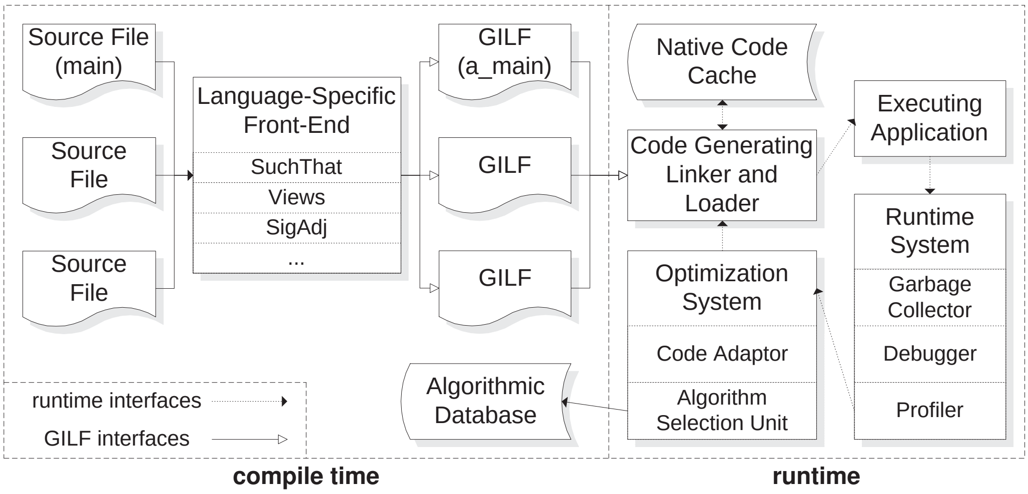

Figure 3.2: A general overview of the complete infrastructure of a GILF-based compilation and runtime system.

3.2.1 Front-Ends

The language specific front-ends perform syntactical and semantical analysis, augmented by instantiation analysis in the case of a generic programming language. The front-end outputs GILF for all processed input. Currently, a SUCHTHAT [Sch96] front-end is under development. GILF was designed to be language independent, but it provides a convenient target as intermediate representation for SUCHTHAT. Most SUCHTHAT constructs map directly to GILF constructs. Views, another generic programming language invented at our group [Gas01], will also consider the GILF system as back-end. Targeting GILF from a variety of generic front-ends will help to refine our intermediate language.

Instead of feeding this intermediate representation directly to the code generating back-end, we use it as the on-disk representation of libraries and executables. Therefore, the work performed at compile time is finished after this step.

Programming languages have different conventions for denoting the function that starts the application program's execution. In C and C++, a function called `main` has a specified signature and is called after initializing the runtime system. In GILF, execution starts by calling the algorithm denoted by the identifier `a_main` that resides in the compilation unit `u_main`. For more details on identifier naming, refer to the XGILF specification given in chapter 4. A GILF entity may contain both library and application code, but only GILF entities that contain the mentioned `main` algorithm may be subject to execution. In figure 3.2, one source file and one generated GILF entity are marked as containing the program entry point `main`.

3.2.2 Code-Generating Linker and Loader

The code-generating linker and loader is one of the most important components of a GILF based system. It executes whenever a user starts an application program on the deployment machine. Figure 3.3 shows a detailed view of the code-generating linker and loader.

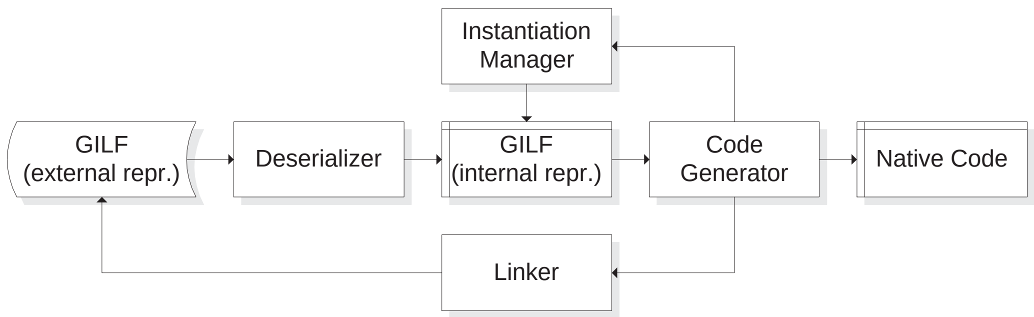

Figure 3.3: Detailed view of the code-generating linker and loader.

The process always starts with an external GILF representation. This can be simply files on the hard disk, but also a system-wide XML database that stores compilation units in XGILF, a http connection or any other input stream. The input stream is processed by the deserializer that turns the external into the canonical internal representation. The internal representation is inherent to the GILF system implementation, all further computations use this form. Currently, a deserializer for XGILF is implemented.

The next step is calling the code generator on the algorithm with the identifier `a_main`, which initiates the code generation of the application. If all symbols are available and no generic components are present, native code is produced. Any unresolved symbols lead to calls to the linker, which will access the needed GILF entities and add the desired units to the internal representation after deserialization. The instantiation manager intervenes if a generic algorithm or data structure is encountered. Then, the instantiation manager checks if instantiation application has already occurred for the requested instance. In this case, it simply skips any processing and the code generator proceeds with the next construct. If the instance is not available, recursive instantiation application is performed. This involves resolving the present bindings and creating an internal representation of the instantiated generic component, which will be further processed by the code generator.

The code-generator of the prototype implementation does not generate native code directly, but rather takes the indirection of first producing C++ code. The C++ code is restricted to basic features, it acts as universal assembler language as proposed by Schupp in [Sch96]. The disadvantage of this approach is that program startup time is increased considerably. On the other hand, we gain platform independence at the target machine, the code generator simply uses the C++ compiler present on the system which emits code for the correct processor architecture. Figure 3.4 shows the prototype's architecture.

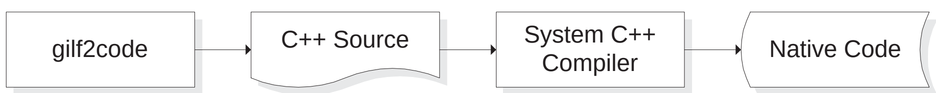

Figure 3.4: Structure of the code-generator's prototype implementation.

We can produce machine code either at link time or at program startup time. In the case of dynamically linked libraries these two choices are virtually the same. We have exercised the startup option for several reasons. Generating code at link time has the advantage that the user does not perceive the time consumed by code generation. On the other hand, we loose the property of platform independence, as native code is produced. Furthermore, optimizations take place on the development machine, not the deployment machine. Also, one is restricted to using statically compiled application programs when generic components are present. But the problem with generic libraries is the main theme of this work, and we do not want to resort to monolithic applications with all library code compiled in at link time.

Thus, the operating system loader's task of reading an executable into memory and starting its execution is augmented by code generation. The code generator becomes a runtime component and turns into an operating system service. For proprietary systems, like Windows NT and various Unix flavors, one does not have the power to extend the system loader. In order to avoid this problem, we create two files for an application program. The first one is the GILF file, holding all semantic information generated by the front-end. The second one is a system conforming executable that starts our code generator.

Let us ponder about the code generator some more. Michael Franz sketched a similar system in his dissertation [Fra94], which was later implemented and publicized [FrKi96] [KiFr99]. His incentive was to provide the foundation for a component based operating system, he was not concerned with the instantiation problem of generic components. He advocates a fast, consequently unsophisticated code generator, because the user will not accept perceivably prolonged startup times. Of course, code quality suffers in this approach upon the first launch of a program. We cope with this problem by introducing a code cache, which holds the latest version of a translated GILF entity. Now we can afford a long compilation run, which applies the whole range of traditional optimizations. Notice that for a main program, we can fill the cache immediately after the compiler front-end emitted its output[2]. This corresponds to code generation at link time.

Another option for code generation that was not mentioned yet is just-in-time (JIT) compilation. Both the Java Runtime Environment (JRE) and the Common Language Runtime (CLR) [MeGo01] employ JIT compilers. The cost of compilation is spread across the whole run time of the application, as at the first call of a function code is generated. Another advantage of JIT compilers is that code is generated only for functions that are actually called.

3.2.3 Native Code Cache

The native code cache was mentioned in the preceding sections. We will discuss it in more detail now.

The prolonged startup time of application programs can become a problem, especially if a fully optimizing compiler is run to create the native code each time. An established solution to this kind of problem, where the same computations are performed multiple times, is caching. We have decided to implement a native code cache at the deployment machine. There are two main aspects that we have to deal with when caching compiled GILF code:

[2] This policy makes only sense if the application is run on the same machine on which it is developed and compiled.

1. Caching code for applications and libraries.

2. Caching local and system-wide code.

The first item points to a behavior specific to generic programs. In a traditional environment, the application code can be compiled on its own, as well as the libraries. When dynamically loading a library, various applications can share the same code of this library, as the code is the same for all applications. Both Unix and Windows operating systems support dynamic libraries in this way, on Unix systems they are called shared libraries using a `.so` extension, on Windows systems they are called dynamic link libraries using a `.dll` extension.

This is no longer the case for generic code. An application can have private data structures and request instantiations of generic components that reside inside a library with these data structures. It makes sense to store these instances with the application only, as no other application will ever make us of these instances. On the other hand, instances that use built-in types or that are part of the GILF core library could be shared among applications. Therefore, one not only has to decide if a compiled instance should be cached, but also if the native code should be stored with the application or the library.

The next issue is where to store the cached code. For system libraries and applications it is reasonable to store often requested and generated instantiations in a system-wide cache. This is no longer true for code that is executed solely by one user or application. The native code cache should be separated into local and system-wide parts.

The instantiation manager is the component in the code generator that deals exclusively with the aspect of genericity that is left to the back-end: instantiation application. The simplest strategy in dealing with instantiation requests is to apply the provided bindings of static instantiation parameters to arguments and generating code for the instantiated generic components. This is what some C++ compilers do, and it leads to long compilation runs and other problems, as observed in section 2.5.3. As just explained, we try to alleviate this observed behavior by caching generated instances for later retrieval in disk caches. Further on, the instantiation manager keeps an internal cache with the instantiations requested for the current compilation unit. The complete cache hierarchy of a GILF system is depicted in figure 3.5.

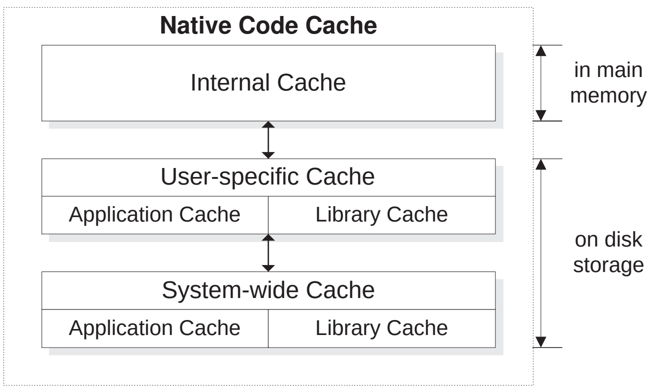

Figure 3.5: The hierarchy of the native code cache in a GILF system.

The internal cache is managed by the code generator and the instantiation manager. After completing the compilation of the current unit, the information in the internal cache is propagated to a cache manager that handles the user-specific (local) and the system-wide cache. The internal cache will be filled from the other caches during the code generation phase, whereas the disk caches are updated with the content from the internal cache after code generation.

Systems that employ a cache have to deal with the problem of data integrity of cached items. This is also a concern for GILF's native code cache. Whenever a GILF file changes on disc, the native code cache may become corrupt, thus loosing its integrity. A simple strategy to overcome this problem is embedding a time stamp into the GILF file that is also present in the cached native compilation. Then, comparing the time stamps of the original GILF file and its cached compiled variant will trigger a recompilation on mismatch. This simple strategy has the disadvantage that a lot of unnecessary recompilation will be performed because of trivial changes in the GILF files. More elaborate techniques will be considered in future GILF system versions.

3.2.4 Runtime System

We have progressed to the stage where the application program is executing. At this time the application communicates with the GILF system only through runtime system interfaces. The runtime system consists of three major parts:

- Garbage Collector
- Debugging System
- Profiler

3.2.4.1 Garbage Collection

Introduction Garbage collection relieves the programmer from manual memory management. This is good as manual memory management is error prone and not necessarily results in efficient execution of the resulting programs. For modern computer programming languages, garbage collection is becoming increasingly important, as language features like object orientation and threads make it difficult to track memory explicitly in complex programs. Not coincidentally, current programming environments like the Java Virtual Machine and the Microsoft Common Language Runtime [Ric00] both feature an integrated garbage collector, which is used as the memory management mechanism in the accompanying languages Java and C#, respectively.

For a better understanding of this issue, we will give a short introduction to memory allocation in programming languages. Memory is available in programming languages in three flavors:

Static Memory Variables residing in static memory occupy the same memory location throughout the whole execution of an application. The compiler has to generate code that is executed by the runtime system to initialize and clean up static memory, depending on the semantics of the programming language.

Stack Memory Parameters of procedure and function calls are pushed onto the stack, as well as the return address and sufficient space for local variables. This allows recursive calls, which is not possible with statically allocated memory.

Dynamic Heap Memory Heap memory may be allocated at any time and in any order during the execution of a program. This allows dynamic allocation of arbitrary data structures that may outlive the function in which they were allocated. But this increased flexibility comes at the cost of more complex management of heap memory.

The common interface for manual heap memory management consists of two functions, one to allocate memory and one to free the formerly allocated memory. These functions may either be part of a runtime library or inherent to the programming language. For example, Pascal has the `new` and `dispose` keywords, C++ has `new` and `delete`, whereas C resorts to the library calls `malloc`, `realloc`, and `free`. The programmer has to pair these calls meticulously in order to avoid memory leaks.

Garbage collection is responsible for automatically reclaiming dynamically allocated heap memory that is no longer used or reachable in the application. Thus, the basic interface for a garbage collected heap consists solely of an allocation function. In Java, objects are allocated on the heap with `new` statements and the garbage collector takes care of recycling memory.

There are two major benefits of garbage collection. First, program robustness is increased as errors due to prematurely released memory[3], as well as memory exhaustion aborts due to memory leaks are prevented. Second, as the programmer does not have to care about memory management details, these concerns do not influence the design and interfaces of software systems. Thus, the abstraction level at which systems are expressed is raised, which is a main goal of software engineering and programming language design.

Garbage collection plays an important role in computer algebra and symbolic computation systems, as applications and algorithms in these systems tend to run for a long time and memory usage is pushed to its limits. The Configurable Memory Management (CMM) system [AtFlIg98] was developed in the context of Posso, a project aimed at advancing tools and software for symbolic computation. SAC-2 [CoLo90] and Saclib [HoNeSc95], two computer algebra libraries, both successfully employ a garbage collector. SUCHTHAT is also intended to work in an garbage collected environment.

All these facts contributed to the decision to integrate a garbage collector into the GILF system.

Basic Techniques There are three fundamental techniques for implementing garbage collection, which we will sketch briefly:

1. Reference Counting
2. Mark-Sweep Garbage Collection
3. Copying Garbage Collection

Reference counting [Col60] is based on the idea that every cell has an associated field that counts the number of active references to the cell. Whenever another cell newly refers to the cell or removes its reference, the invariant has to be maintained that the reference count equals the number of active references to the cell. A cell is reclaimed if the cell's reference count drops to zero. These cells are put back on some kind of freelist.

[3]These kind of errors are usually refered to as the *dangling pointer problem*.

Reference counting is called a direct garbage collection method as no longer reachable cells are returned to the freelist immediately.

One advantage of reference counting lies in the simplicity of the algorithm. Also, the cost of garbage collection is distributed evenly throughout the program's execution, which makes it well suited for real-time and interactive application. Last, the locality of reference of the application is widely preserved, which is an important aspect for todays processor architecture, because cache misses and page faults have a deep impact on performance. On the other hand, reference counting has severe drawbacks. The processing overhead introduced for tracking the reference count at every pointer update is significant. Furthermore, cyclic data structures can not be reclaimed with the traditional reference counting scheme. Finally, reference counting implementations are quite fragile, as simply omitting one adjustment of the count field can lead to errors that are extremely hard to detect.

The other two garbage collection techniques are called indirect as they rely on tracing the heap at certain intervals. Reclaiming unreachable cells is deferred to dedicated garbage collection phases.

The mark-sweep algorithm works as follows. Allocation of cells is accomplished by a lookup into a freelist. Garbage collection starts when the freelist is exhausted, i.e. the request for a cell fails. Processing in the application is no longer possible until the garbage collector finishes. The mark-sweep collector is divided into two phases, a mark and a sweep step. In the marking phase, a global traversal of all live objects is performed. It starts in the root set, which is the set of all cells that are directly available at the point at which the garbage collector was invoked[4]. These live cells are marked as active with a mark-bit, as well as the cells that are recursively accessible from them by following pointers. The recursion ends as only unmarked cells are examined. After the mark phase is finished, all unmarked cells are considered garbage, as they are no longer reachable from the application. Now, the sweep phase starts which linearly scans the heap and puts all unmarked cells back into the freelist, and clears the mark-bit on active cells.

The mark-sweep collection has two major advantages over reference counting. Cycles are handled naturally, no special treatment of these data structures is necessary. In addition, pointer manipulation is completely free of any overhead, thus overall performance is superior to reference counting. On the other hand, program execution is halted during garbage collection, which disqualifies it for real-time and most interactive applications. Notice that during the sweep phase, the whole heap is visited, not only the active cells. Fragmentation of the heap is also a problem, especially when variable sized cells are allowed. This problem is also present in reference counting and manual memory managers that employ freelists.

Finally, we present the copying garbage collector. It addresses the fragmentation problem and also has very fast allocation characteristics. The copying algorithm separates the heap into two semi-spaces, called fromspace and tospace. Fromspace contains the current data and tospace contains abandoned data. The collector starts when allocation of a new cell fails because fromspace's free storage is smaller than the cell's size. Then, the copying collector starts traversing the active cells. Every cell that is visited will be copied from fromspace to tospace. Copying of active cells starts at the bottom of tospace and progresses sequentially, thus defragmenting the heap in every copy phase. Special care has to be taken in order to copy cells that are part of shared and cyclic data

[4]These cells are typically local and global variables. They reside in registers, on the stack, and in static memory.

structures only once. This is achieved by copying *forwarding pointers* over the beginning of already copied cells in fromspace. A recursive version of copying garbage collection must deal with the problem of stack overrun. Cheney developed an elegant iterative version that uses only two pointers [Che70].

The copying algorithm has two main advantages over the mark-sweep algorithm. First, it visits only the active cells, thus its asymptotical complexity is better compared to mark-sweep algorithms, whose sweep phase has to scan the whole heap. Moreover, the heap is defragmented and thus allocation is extremely fast, it is performed by simply increasing the free pointer in current fromspace. The caveats are that active cells have to be copied, which can be expensive for large objects. Also, the address space required for copying garbage collection is doubled, as it requires two equally sized heap semi-spaces.

For all three presented techniques, many improvements were realized and publicized. For example, incremental and generational tracing techniques reduce stop periods of these collectors. For a thorough presentation of garbage collection, refer to [JoLi96].

Garbage Collection for GILF Nishanov and Schupp examined the special needs of garbage collection in the context of generic libraries for the case of C++ [NiSc01]. We considered two aspects of their work. On the one hand, we had to think about the implications of genericity on garbage collection in general. On the other hand, characteristics of garbage collection in C++ had to be evaluated because we use it as target-language in the GILF prototype.

C++ is an imperative general purpose language which expects memory management to be handled explicitly. The layout of objects and runtime components like the stack and heap are not known to user programs. Built-in datatypes are represented by machine words, they are not boxed. A garbage collector for C++ is just another user program with no special support by the compiler. These characteristics lead to problems, as the collection techniques described above are type accurate, that means they assume knowledge about the position of pointers in data structures and the heap layout.

Conservative garbage collection is the method of choice to handle the problematic cases. It does not rely on cooperation of the compiler or any knowledge of the memory subsystem. It has to cope with two fundamental problems:

1. Finding the Root Set

2. Pointer identification

The Boehm-Demers-Weiser collector [BoWe88] is credited for both providing routines for root set finding on all major platforms as well as for deriving efficient heuristics for pointer identification. The collector has a C and a C++ interface. The C interface replaces calls to `malloc`, the C++ interface calls to `new`. The efficiency of the collector can be increased if allocation of objects that contain no pointers are made through a specialized call of the general allocation function. The collector employs a mark-sweep allocation with sweeping spread among allocation calls (so-called lazy or deferred sweeping).

An alternative to using system-dependent heuristics for root finding on the stack, in registers, and static areas is requiring the user to provide pointer finding routines such that pointers in heap-allocated data structures are identified accurately. Objects residing in the heap are now subject to a copying collection algorithm. These collectors are called *mostly copying garbage collectors*, one well known example is the Bartlett collector [Bar88][Bar89], which was developed for usage in the back-end of a Scheme compiler.

So how does genericity complicate matters? The first problem identified by Nishanov and Schupp is end-of-object pointer identification, as these pointers are often used in generic libraries as iterators that mark the end of sequences [MuDeSa01]. If these pointers are not updated after their related sequence was copied, checks for the end of the sequence will fail. The Bartlett collector suffers from this kind of failure. The Boehm-Demers-Weiser collector is not affected as it performs no copying of data structures.

The second problem of generic data structures stems from the fact that its members' actual types are not known until instantiation time. Therefore, up to this time the knowledge is not available which of the structure's members are pointers. This effectively disables the optimized call to the atomic allocation function of the Boehm-Demers-Weiser collector. For large objects like matrices or vectors that hold no pointers this can lead to a significant efficiency degradation.

Both problems are solved in TGC, the collector presented by Nishanov and Schupp. GILF can use both TGC and the Boehm-Demers-Weiser collector. The latter one even with optimized calls for atomic objects, as at instantiation application time the whole type information is present and the correct version of the allocation function call can be synthesized. Once again, the design of the GILF compilation system solves a problem present in current compilers for generic languages.

The GILF Interface Finally, we want to present the interface to dynamic memory allocation in GILF which is part of the GILF core library (see appendix C). It is given in XGILF, the XML-based external representation of GILF. For a thorough discussion of this format see chapter 4.

```
<!-- Dynamically allocate memory from the heap for a single object. -->
<function id="u_func.f_allocate" name="allocate">
 <type-params count="1"> <type-param id="tp_0" name="T"/> </type-params>
 <params count="1">
  <param pass="out_ref!" id="p_0" name="new_obj">
   <static-param-dsg ref="tp_0"/>
  </param>
 </params>
</function>
```

The function declaration introduces the identifier `f_allocate` that resides in the compilation unit `u_func`. For a given type parameter `tp_0` it returns a reference to the newly allocated memory, accessible are parameter `p_0`. If the request for memory could not be fulfilled, even after running the collector, a `null` reference is returned.

3.2.4.2 Debugging System

Another important aspect of every compilation system are the debugging facilities. Debugging rests primarily on two conceptual pillars. First, a programming language should help to avoid writing erroneous code before it is executed at all. Second, if the static semantics of the program have been checked by the front-end and code was generated by the back-end of the compiler, runtime error finding should be supported by powerful debugging mechanisms. We will briefly discuss both aspects, with a special emphasis on the needs of generic programming.

Static Debugging Strong typing has increased the reliability of computer programs. The modification of variables is possible only in a controlled manner as all methods that change state are validated by the type checker. Almost all modern programming languages like Ada95, C++, C#, and Java feature a strong static type system. In chapter 2 we motivated the necessity of enriching the type checks in generic programming languages with semantic concept checks, thus allowing only valid instantiations. SUCHTHAT offers one of the most extensive static type checkers available today.

Notice that in order to be of significant benefit, error messages reporting semantic contract breaches have to be displayed in a accessible form. C++ compilers are notorious for cryptic error messages related to instantiation problems. There even exist software filters that try to make the error messages generated by popular C++ compilers more readable [Zol01]. Compilers based on the EDG front-end [Edi00] show that if some care is put into this aspect of the compiler its usability is increased. However, the general problem that no semantic checks are performed during instantiation in C++ remains. Programmatic approaches try to palliate this shortcoming [SiLu00].

Dynamic/Runtime Debugging Once the source code passes all static checks, debugging is concerned with helping the programmer to find errors in the application logic expressed in algorithms. Methodology to handle this hard problem has been in focus of current research, as debugging tools did not keep pace with the overall evolution of programming languages. Eisenstadt undertook a statistical examination of the pragmatic problems of debugging, how they are treated and the root causes [Eis97]. The two main difficulties in tracking down programming errors are the cause and effect chasm of subtle bugs as well as the inability to apply appropriate debugging tools. The most commonly used approach to finding bugs is data gathering. This can be as simple as inserting *print* statements, but more elaborate techniques are becoming available. Most industrial strength compilers provide a dedicated tool for stepping through the executable and watching variables' values, e.g. gdb, the GNU Debugger. Software visualization [BaDiMa97] is employed to grasp the behavior of the executing program on a more abstract level. Also, tools for monitoring states of the executing program at various time points or even complete execution histories are being developed, enriched with pre- and postcondition checks or query expressions [Len00] [Jah00] [BoJo97]. Another approach is comparing different versions of programs, such that bugs introduced by adding or changing parts of correct software can be located more effectively. This can be done by extensive regression testing [GrHaKi$^+$01], thus erroneous units will be identified by comparison of test sets, or even by automatically comparing changed source code [Gro97]. Finally we want to mention the problem of debugging optimized code. As modern optimizers aggressively reorder code a mapping of source code to optimized machine code is often not possible. Alternate means for debugging such optimized code are needed, for example interpretation of intermediate code [Elm97] in the debugger.

Debugging Support in GILF How does the GILF philosophy match with the needs of debugging tools? GILF entities contain entries that allow source code fragments to be attributed to GILF constructs (see chapter 4 for details). This is the low level support necessary to allow debuggers to display reasonable output. Furthermore, as GILF is intended to be used as back-end by various front-ends, some naming information might be lost in the transformations performed by the front-end. GILF allows this information to be propagated as properties in the intermediate representation.

Extensive and elaborate error messages are to be generated by the front-end. The intermediate representation will only make sanity checks on the binding information created by the front-end, such as checking the number and kind of instantiation parameters.

These checks and the resulting error diagnostics are roughly at the level what C++ compilers generate. This is appropriate for a back-end, which should get correct intermediate code from the front-end.

In general, code generation at load time, as present in GILF, has several advantages for building debugging tools. Instrumentation of code can be applied selectively to the intermediate representation, and only if debugging is actually wanted. For example, new compilation units could be marked as subject for extensive monitoring such that defects in these new components could be identified easily. After a given time period or forced by the user, the instrumentation could be removed at the next program start, resulting in improved runtime performance. Contrary, the compilation system could add debugging and monitoring code to compilation units that resulted in errors. The problem of debugging optimized code could be solved by generating nonoptimized code for units or even functions that will be run through a debugger. Overall, the GILF compilation system provides a good basis for implementing advanced debugging tools.

3.2.4.3 Profiling System

The last component of the runtime system is the profiler. Profiling techniques are increasingly important in assisting modern optimizing compilers, because good runtime performance can be achieved only by generating code with good spatial and temporal locality [HePa96]. An inappropriate mapping of source code to machine code can result in loss of runtime performance. Profilers aid in the process of finding the application's *hot spots* and performance bottlenecks. We will briefly describe profiling techniques, as the profiler is the scaffolding of the optimization system described in section 3.2.5. Profiling techniques can be divided into three main categories:

1. Sampling
2. Instrumentation
3. Hardware Performance Counters

Sampling The profiling data collected by a sampling profiler consists of samples of the processor's program counter. The program counter's value is stored each time the sampling routine is invoked either by a timer interrupt or an interrupt caused by an overflowing hardware counter. Sampling is a statistical approach to gathering profile data. Therefore, it cannot perfectly monitor a program's execution and control flow. Certain execution patterns can result in skewed profiles. On the other hand, sampling can be applied to code without recompilation and it does monitor user and system level code. Furthermore, it causes only a small amount of overhead, typically around 1-3% [AnBeDe$^+$97], and thus can be applied to computational intensive and long running production systems. The only concern here is to control the amount of profiling data generated.

The gathered data can be used to attribute the relative time spent in procedures, source lines or machine instructions. Additional, it is possible to identify instructions that are sources of pipeline stalls. In general, tools are needed to help evaluating the data record produced by the profiler.

Instrumentation Profilers based on instrumentation utilize special binary versions of the programs under observation. The code image contains special instructions that monitor the programs execution, for example the time spent in procedures, the path taken during execution and so on. This is achieved with counters that are manipulated at procedure or basic block entry and exit points, either hardware supported or completely programmatic [BaLa94].

The advantage of instrumented profiling is accuracy. Exact construction of execution paths and dynamic procedure call graphs is possible, as well as other kinds of control-flow graphs. Unfortunately, this comes at the cost of significant runtime overhead, ranging up to 30% for typical applications and benchmarks [BaLa96]. The performance impact disqualifies instrumentation for counting single instructions, as the overhead per instruction would slow down the program immensely. The last problem one has to face with instrumentation is that one has access to the machine code images, as changes have to be performed on them. In most cases, system level profiling is therefore not possible.

Hardware Performance Counters Recently, all major processor implementations[5] introduced hardware registers to measure the execution of code without runtime perturbation. These registers are referred to as hardware performance counters. Hardware performance counters count events related to processor activities, like cycles, cache misses, data reads, data writes, instruction stalls etc. The major concern when applying these counters for profiling is that they are highly machine specific, i.e. inherently nonportable. Development of high-level libraries like PCL [PCL] or PAPI [BrDoGa+00] alleviate this problem by providing abstractions for least common features as well as specialized access to native features with probing possibilities.

In the preceding two sections sampling and instrumentation were presented as the primary means of generating program profiles. Both techniques were extended to work with performance counters in order to minimize the runtime overhead incurred by profiling [AmBaLa97].

Profiling Generic Code Profiling generic code poses no special challenges from the implementation side, the tools available for nongeneric profiling are applicable without modifications, as they operate on executing machine code. The GILF system is well suited for profiling purposes, like for debugging, because instrumentation of the code can be added and removed as needed at load time.

The only conceptual problem we have to handle is that program counter values are given relative to instantiated algorithms. After having detected an algorithm instance as being a performance bottleneck, two ways of further action are possible. First, one can try to aggressively optimize the given algorithm instance. Second, and this is one of our accomplishments, one can select another algorithm that realizes the same generic function. Of course, this is only viable if at least one alternative implementation exists. In order to be able to perform algorithm selection, we have to map the program locations back to the generic function that resulted in the specific algorithm instance indicated by the program counter. Figure 3.6 shows a schematic view of this mapping problem. At the top is the concept `C` that introduces the function `f`. This function is implemented by the two algorithms `alg-f1` and `alg-f2`, respectively. The profiler detects an instance of `alg-f2` as performance bottleneck, depicted by a program counter pointing into the algorithm's body. The PC's value has also been mapped back to the generic algorithm `alg-f2`. The code-generating loader can now replace `alg-f2` at the appropriate invocation site with a call to `alg-f1`.

[5]The Performance Counter Library (PCL) [PCL] supports these architectures: Alpha 21164/21264, MIPS R10000/R12000, UltraSPARC I-III, PowerPC and Power3, as well as Pentium processors.

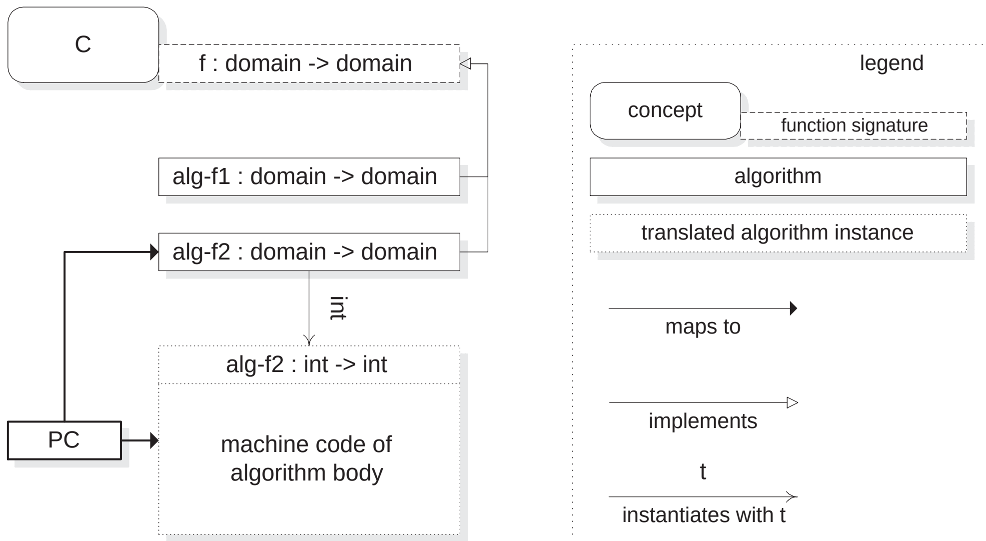

Figure 3.6: Schematic view of the mapping problem between program counter (PC) values to algorithm instances and to the generic algorithm implementations.

The accompanying work by Kreppel [Kre02] deals in more detail with aspects of profiling and selection methods for generic algorithms.

3.2.5 Optimization System

The infrastructure just presented gives us the power to store precompiled generic libraries and create efficient instantiations of its generic components. At startup time all concrete types of the generic algorithms and data structures are known and completely resolved machine code can be produced from the GILF representation, without using boxed representations of built-in types. This comes at the price of prolonged response times whenever GILF code is requested whose architecture specific representation does not already reside in the native code cache. Possible cases are startup time and whenever units are dynamically linked. But owing to the late time of code generation, additional optimization opportunities arise. We will describe the major parts of the optimization system as well as the code enhancements they try to achieve.

3.2.5.1 Algorithm Selection

Let us concentrate our main efforts on transformations specific to generic programming first. In the previous sections and chapter we have stressed several times the fact that a generic function in SUCHTHAT can be implemented by different generic algorithms[6]. Furthermore, a generic function can be specialized for a subset of the possible instantiation arguments, and each specialization can itself be realized by multiple algorithms. Specializations can also be declared for special values of its input parameters, projections in a recursion-theoretical sense. The specialized functions still compute the same generic function, but for a restricted set of parameters. Algorithm selection is the process of selecting one algorithm implementation among possible candidates.

[6]In C++ and Ada95, a generic function can have only one implementation. But out of performance considerations, a generic function can have different specializations.

Having formulated the abilities of a flexible generic language like SUCHTHAT, we now deliberate their impact on the GILF intermediate representation and its compilation system. First of all, GILF has to provide facilities by which the front-end language can propagate its decision which algorithms are valid candidates at a generic function calling site. This means, all appropriate algorithms that implement the specific instance of the generic function in the current context have to be listed in the intermediate representation. In GILF, the following solution is realized. A function call is always represented by a symbol representing the most general generic function, and the value parameters. Thus, the signature of the call is fixed, as well as parameter passing. In library code, this is all that is available. In application code, where all instantiations are resolved, further information has to be provided for the function call:

1. Bindings of the instantiation parameters to their arguments that describe this specific instance.

2. The list of candidate algorithms that result in valid instantiations.

The two items above present the interface to the algorithm selection unit (ASU). Depending on these information, the algorithm best suited to current task has to be selected from the candidate list. Notice that the semantics of the front-end language influence the information presented to the ASU. For example, one language might restrict the list of candidates to the most specialized algorithms, while another could add all possibly applicable algorithms to the list. GILF does not enforce a complex type system at the intermediate level, but rather requires the front-end to make its semantics explicit in the intermediate representation. See section 4.9 for a detailed description.

To our knowledge, SUCHTHAT is the first language that makes algorithm selection totally transparent to the library user. This duty, choosing the fastest algorithm out of a pool of valid algorithms, is left to the programmer in current languages, hard-wiring decisions into the code. Different algorithms are selected by their name, and no way to combine them as a set of alternatives to perform the same task is available. Of course, a SUCHTHAT programmer still has the power to select the algorithm she deems best suited by hand. In this case, the candidate list in GILF is reduced to this one algorithm and algorithm selection is skipped.

After algorithm selection has been executed by the ASU, the next step is to apply the instantiation. Four alternatives to perform this task are feasible, each with its own advantages and drawbacks.

Compile-time Instantiation Application Instantiation application is performed at compile time. This has the advantage that all computations necessary are performed on the development machine, the executable starts and runs at full speed on the deployment machine. Of course, this can severely hinder the development process as an extraordinary increase in compilation times can be observed [AbCo01]. Algorithm selection has to be performed relying on purely static input data. Most important, no machine code-generation is possible for uninstantiated generic code, i.e. library code.

Link-time Instantiation Application At link time, full type information for applying instantiations is available for the first time. Therefore, machine code could be produced for complete applications. On the other hand, this results in statically linked executables. Most current generic imperative languages like C++, Ada95, and Modula-3 use a hybrid of compile and link time instantiation.

Load-time Instantiation Application Load-time instantiation application generates machine code for the selected algorithm at startup of the application. In the GILF prototype, this involves calling `gilf2cpp`, the translator from GILF to C/C++, and running a local C/C++ compiler on the result. This fixes the algorithm that is called at every calling site for the whole runtime of the application. Essential for this approach is an intermediate representation of generic libraries like GILF. Algorithm selection can be based on profiling data gathered in previous runs of the application.

Runtime Instantiation Application Runtime instantiation application takes the central idea of our work even further. It would allow replacing the algorithm that is invoked at a function calling site at runtime. This has two implications for the optimization system. The ASU is becoming part of the runtime system, which can be solved by realizing the ASU with a client-server architecture. The ASU becomes the server for the selection requests of the application, which becomes the client. Also, a dedicated code adaptor is needed to patch the code that is currently executing, such that the newly selected algorithm will be called.

A plethora of new questions to answer are introduced by the latter two schemes for algorithm selection, which we consider primarily. The most obvious problem is which criteria should guide the algorithm selection process?

In order to tackle this problem, let us have a closer look at the well known sorting algorithms. Depending on the type of a container whose elements should be sorted, it is reasonable to choose a different sorting algorithm, like quicksort for arrays and mergesort for linked lists and files. The generic sort algorithm in the STL requires random access iterators and thus cannot be used with lists. This is a very clumsy way to handle this algorithm selection problem. Code that uses the sort function cannot deal with lists, because they have to be sorted with a member function, which has different syntax, defeating the benefit of generic code. Our system makes available a platform where selecting the appropriate algorithm can based on instantiation parameters, like the iterator types.

Not only static instantiation parameters, like the type of the container, are viable criteria for algorithm selection, also the container's attributes like its elements' values and its size can help to choose a better algorithm. Such value parameters can either be compile time constants or dynamic values that change each execution of the program. Since quicksort's complexity degenerates to $O(n^2)$ for certain input sequences, it would be better to apply another sorting algorithm right from the start, if a frequent pattern in the sequence of elements stored inside a container could be determined with a profiling or monitoring tool. Of course, not only containers, but all other types, their values and additional parameters, like the length of a list or the degree of a polynomial, can expose significant information for algorithm selection. Even some well known findings from static complexity analysis or some previously developed heuristics can be useful. David Musser introduced introsort [Mus97], a sorting algorithm based on quicksort that switches to heapsort if a certain recursion depth is reached while partitioning. This effectively protects against running into worst case. He uses a depth limit of $\log n$, where n is the length of the input sequence.

It is also of great importance that the characteristics at the machine level can have an effect on the algorithm selection. Our field of special interest is computer algebra, where algorithms can run for days and switching between one function's different algorithm implementations at the appropriate trade-off points can have a deep impact on its runtime, e.g. using machine addition for numbers that fit into a few machine words and addition for numbers of arbitrary precision (see [Wed96] and [Gra02]) otherwise. Experiments on algorithm selection in this domain are available in [Kre02]. The Algorithmic Database (ADB) is an off-line repository that stores precomputed measurements of algorithm runtimes for different machine architectures. This enables the ASU to fit algorithm selection to the actual hardware available.

The work by Kreppel [Kre02] presents the current ASU. It is designed as a rule-based expert system. Rules are generated automatically from data present in the ADB, ultimately resulting in performance predictions of instances of generic algorithms. Algorithm selection consists of comparing the performance predictions for all algorithms from a candidate list.

To conclude this discourse on algorithm selection criteria, a summary of these criteria that can guide the selection process is given in table 3.1. In the left column, the criterion is listed, the middle column contains the instantiation application time at which this criterion could be exploited, and the right column lists requirements on the system which have to be met such that the algorithm selection could be executed effectively.

Criterion	**Applicability**	**Requirements**
instantiation arguments	compile-, link-, load-, runtime	ADB
compile time constants	compile-, link-, load-, runtime	ADB
runtime constants	load-, runtime	ADB, code adaptor, execution time profiles
parameter values	load-, runtime	ADB, code adaptor, data profiles
hot spots/bottlenecks	compile-, link-, load-, runtime	ADB, code optimizer and adaptor, profiler

Table 3.1: An overview of algorithm selection criteria.

The first row contains instantiation arguments, the second and third row summarize static properties of the parameters, either known at compile- or runtime, the fourth row denotes the parameters' dynamic properties, i.e. their actual values, and the last row deals with hot spots detected by a profiler.

Beside the selection criteria, other problems have to be scrutinized. An infrastructure that allows code replacements has to be devised for runtime instantiation, . Both load- and runtime instantiation require careful choice of the code sections that will undergo optimization. Because of the perceived delay by the user, only selective application of the optimization system will be tolerated. Finally, the heuristics that assess the criteria are a major area of future research.

3.2.5.2 Exploiting Platform-Specific Features

A target independent intermediate representation allows the system to emit code tailored for the processor architecture the program is actually running on. Nowadays this is important even for the same architecture, because different members have heavily varying characteristics. E.g., the family of processors compatible to the 32 bit Intel Architecture IA32 evolved from a pure CISC architecture to a typical RISC architecture internally.

Code for modern implementations of the IA32 architecture have to consider other constraints for efficient code generation than older ones, for example avoiding pipeline stalls and cache misses.

Also, new features are added constantly with every new processor generation, like MMX, Internet Streaming SIMD Extensions (see [Int99] for both), or 3DNow! [Adv99]. Special operations can be sped up significantly when making use of the available extensions. Our approach can handle such varying system configurations in a clean way by compiling only those features into the executable code that are really available at the platform the program is launched on.

In the GILF prototype implementation that translates to C/C++, this boils down to setting the compiler optimization switches that enable processor extensions that are available on the deployment machine.

3.2.5.3 Intermodular Analysis

Another noteworthy point is the ability for intermodular optimizations. At runtime all active compilation units are known and control- and data-flow analysis is not limited to their boundaries. The optimizer knows the context of a called algorithm and can perform individual manipulations like inlining, code movement and register allocation tailored to the situation [Kis99]. A function call that is part of a hot path could be replaced by inline code of the algorithm body, even if the algorithm resides in another compilation unit. Again, late code generation paves the way for new approaches to code optimization.

On the other hand, intermodular analysis can be very expensive for large systems. If it really recoups the increased processing time invested at program startup remains to be shown when the GILF system is applied in large scale development.

3.3 Summary

This chapter gives an extensive overview of the GILF compilation system. It commences by motivating the most important aspects of our intermediate representation, in particular its design goals, the abstraction level, structure and encoding. These aspects are related to contemporary techniques. After these explanations, the infrastructure needed for a complete GILF compilation system is described. The main components of this system are the code-generating linker and loader, the native code cache, the runtime system, and the optimization system. The runtime system's subcomponents were further examined; it is made up of a garbage collector, a debugging system, and a profiling system. The algorithm selection process gained special interest while detailing the optimization system, as it is of primary importance for our vision of generic programming.

Chapter 4

The Annotated XGILF Specification

In the preceding chapter we described the overall structure of the GILF system as well as the motivation and rationale for the GILF intermediate representation. Our work focused on XGILF, an XML based incarnation of GILF.

We will now give a short introduction to the key technologies of XML relevant for our purposes. Thereafter we will present the specification of XGILF in detail, i.e. the elements, their attributes and their content that constitute an XGILF file.

4.1 A Concise Summary of XML

The XML Activity Statement of the World Wide Web Consortium (W3C) describes XML as follows:

> Extensible Markup Language (XML) is a simple, very flexible text format derived from SGML (ISO 8879). Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web. ([XMLA])

XML, the Extensible Markup Language, is a W3C core standard with accompanying standards, tools and technologies that gained immense attention in the industrial and scientific area since its publication as XML 1.0 recommendation in 1998. The key points that make it attractive for all kinds of applications are its simplicity and extensibility. From a high level view, XML is nothing more than a textual tree representation with name-value pairs that are attached to the tree nodes. Therefore, it is appropriate to describe the logical structure and storage of any structured data that should be exchanged between processes, biased towards textual data.

4.1.1 Elements, Attributes, and Text

The building blocks of XML documents are elements, attributes, and text. An element has an associated type which is denoted by the element's name. Properties are attached to elements as attributes, which are name-value pairs. The value of an attribute is given as text string and accessed by its name. Elements can contain other elements and text nodes, the content of an element. The content of an element is delimited by the element's start and end tag. All start tags must be matched by end tags and elements may not overlap. With these facilities at hand data can be stored in a structured manner. Elements can be perceived as tree nodes, attributes and text as leafs of the tree.

4.1.2 Entities, Well-Formedness, and more

XML supports text expansion with so-called entities. An entity reference is replaced by the text provided in the entity declaration. Entity references are allowed almost everywhere in an XML document and play an important role in avoiding redundancies.

A design goal of XML was that the representation is easily accessible to machine processing. This was achieved by formulating strict rules how an XML document may look like. This includes the restriction that elements are not allowed to overlap, the usage of characters that form element tags and the declaration of attributes. XML documents that adhere to these rules are called well-formed. For a complete set of well-formedness constraints refer to the XML specification [XML00].

XML also supports comments, processing instructions (PIs), and CDATA sections. The application of comments is obvious. PIs are used to pass information to the XML processor that works on the XML document and CDATA section allow unescaped text sections.

Another big advantage of XML is the fact that it is build on top of the Unicode standard [Uni00], i.e. XML documents can be written using the whole Unicode character set. The beginning of an XML document usually allows auto-detection of the employed encoding. Thus, XML documents are internationalized by definition.

4.1.3 Valid documents

Well-formedness restricts XML documents to follow simple storage conventions. XML documents whose content follows dedicated rules are called *valid*. The set of valid documents can be defined by means of document type definitions (DTDs), which are integral part of the core XML standard [XML00]. A DTD closely resembles the EBNF specification of grammars. It also allows to specify default values for attributes.

More recently, Schemas [XMLSP01] were added to the list of W3C specifications. An XML Schema can define a set of valid documents more exactly than a DTD. They support a wide variety of structural components [XMLSS01] and a rich set of datatypes [XMLSD01]. In contrast to a DTD, XML Schemas are written in XML itself and do not have their own syntax. These points result in a more verbose specification of an XML document's content, compared to one given with a DTD.

The XGILF specification is given as an annotated document type definition. The annotations usually state additional semantic constraints immanent to our program code representation that cannot be expressed in the DTD. Furthermore, usage examples explain the element type definitions. We have chosen to specify XGILF with a DTD because it results in a more compact written presentation. Furthermore, Schema support has only become available in XML parsers recently[1].

4.1.4 Namespaces

As one intention of XML is world wide data exchange, one has to deal with the problem of name clashes. The relevant names are element type names and attribute names. In order to prevent name clashes, a namespace mechanism was introduced for XML [XMLN99]. The main point is that names from XML namespaces appear as qualified names. A qualified name contains a colon that separates the name into a namespace prefix and the local name. The prefix is mapped to an unique resource identificator (URI)

[1] The Xerces C++ parser from the Apache Project has XML Schema support since December 2001.

with a namespace declaration. The namespace specification gives a special meaning to the colon in names, but an XML document using namespace prefixes is still well-formed XML. An empty element of type `unit` in the `xgilf` namespace is given as example:

```
<xgilf:unit xmlns:xgilf="http://www-ca.informatik.uni-tuebingen.de/gilf/xgilf">
</xgilf:unit>
```

Notice the `xmlns:xgilf` attribute which identifies the namespace prefix for this element and has the associated URI as value.

4.1.5 XML Parsing Technologies: DOM and SAX

The last sections dealt directly with the XML specifications concerning its structure and content. One advantage of XML is its universal applicability, which becomes visible in its tool support. A parser is an important tool to handle XML documents. Simple XML parsers just check the well-formedness of documents, but most modern parsers also allow to check the validity of XML documents according to their DTDs or Schemas. There are two established application programming interfaces to perform XML parsing that follow different approaches, the DOM and the SAX API, respectively.

DOM The Document Object Model (DOM) is an official W3C standard [DOM00]. It provides a tree-like structural view of XML documents and allows to access and manipulate all parts of the document. The XML document is treated like a tree consisting of typed nodes. The core specification features simple navigation in this tree, like advancing and retreating to sibling, children and parent nodes. There also exist more powerful navigation mechanisms like iterators and treewalkers with filters [DOMT00]. They allow very selective traversal of the XML document.

SAX The Simple API for XML is a *de facto* standard that is defined by its Java implementation, but bindings to many other languages like C++, Perl, and ML emerged. It is the joined effort of many programmers on the Internet to develop a common event-driven API for parsing XML documents [SAX]. Event-driven means that when parsing is initiated, the parser reports events to the calling application through callbacks. Events are starting and closing tags, text, attribute values and names, and so on. It is up to the programmer to maintain state between different events if necessary.

DOM versus SAX: The Trade-Offs The advantage of DOM and other tree-like APIs are the availability of the whole document all the time. Of course, this comes at a cost. The XML document is kept in memory by the parser, for this reason DOM parsers are rather resource hungry, especially with respect to main memory. Event-driven parsers like SAX parsers are better suited for simple tasks on the XML document like locating special element nodes. They usually do not keep an internal representation of the XML document, but it is possible to create one for further processing.

Tree-based XML parsers are often more convenient for the programmer to deal with but put more strain on the system resources, whereas event-driven parsers force the programmer to handle state on its own but are more economical with regard to machine resources.

After this short overview of some basic XML technologies we will give the specification of the XGILF core set.

4.2 Prolog

A valid XGILF document provides a GILF declaration in the document's prolog by defining a processing instruction (PI), additional and analogous to the XML declaration (see [XML00], 2.8). The GILF processor operates on the XGILF document, therefore `gilf` is the processing instruction's target. The following information is passed to the GILF processor in the PI's attributes:

Version The `version` attribute identifies the version of the GILF system this file is created with. This work describes the prototype implementation labeled as version 1.0.

Protocol The `protocol` attribute identifies the protocol this file employs. The prototype only supports `xgilf` as value, as only XGILF processing is implemented.

Level The `level` attribute identifies the file's GILF level. The GILF level indicates what kind of features are present in the GILF file. Level 1 indicates only the core features, presented in this work.

Notice that *every* GILF file contains the `gilf` processing instruction. If the protocol is not `xgilf`, the first byte from the alternative encoding starts right after the closing string '?>' of the PI, as depicted in figure 4.1.

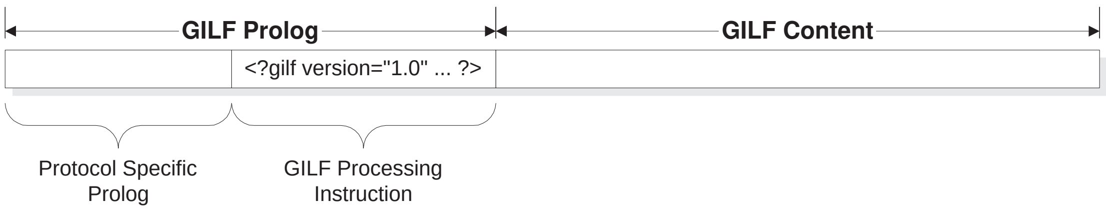

Figure 4.1: The layout of GILF files.

This way, the GILF processor can select the appropriate protocol handler. Furthermore, every XGILF file is a valid XML and GILF file. The `xgilf` protocol specific prolog contains the document type definition that points to the DTD for validation purposes. The complete XGILF prolog with the document type definition and its `gilf` processing instruction looks like this:

```
<?xml version="1.0"?>
<!DOCTYPE xgilf:xgilf SYSTEM "xgilf.dtd">
<?gilf version="1.0" protocol="xgilf" level="1"?>
```

4.3 Namespace

The XGILF elements live in their own namespace with the prefix `xgilf`. We declare entities for referencing the namespace prefix, entity `np` adds a colon at the end, whereas entity `ns` adds one at the front. This way, the prefix can be changed by touching only these entity declarations.

`"../xgf/xgilf.dtd"` 50 ≡

```
<!-- The XGILF 1.0 DTD.
     Location: http://www-ca.informatik.uni-tuebingen.de/gilf/xgilf/xgilf.dtd
     Namespace URI: http://www-ca.informatik.uni-tuebingen.de/gilf/xgilf/ -->
<!ENTITY % np 'xgilf:'>
<!ENTITY % ns ':xgilf'>
```

File defined by parts 50, 51, 52a, 55b, 56, 57a, 57b, 62a, 69, 72a, 74, 76a, 76b, 77a, 77b, 77c, 77d.

We started the file `xgilf.dtd` that contains the complete XGILF DTD. The namespace identifier is also given which should be used in XGILF documents as value for the `xmlns` attribute. Now we list all the XGILF element names.

"../xgf/xgilf.dtd" 51 ≡

```
<!-- Declare all element names in XGILF as entities with namespace prefix. -->
<!ENTITY % xgilf                '%np;xgilf' >
<!ENTITY % unit                 '%np;unit' >
<!ENTITY % import               '%np;import' >
<!ENTITY % declare              '%np;declare' >
<!ENTITY % define               '%np;define' >
<!ENTITY % bind                 '%np;bind' >
<!ENTITY % store                '%np;store' >
<!ENTITY % extend               '%np;extend' >
<!ENTITY % source               '%np;source' >
<!ENTITY % function             '%np;function'>
<!ENTITY % type                 '%np;type'>
<!ENTITY % type-params          '%np;type-params'>
<!ENTITY % type-param           '%np;type-param'>
<!ENTITY % func-params          '%np;func-params'>
<!ENTITY % func-param           '%np;func-param'>
<!ENTITY % const-params         '%np;const-params'>
<!ENTITY % const-param          '%np;const-param'>
<!ENTITY % algorithm            '%np;algorithm' >
<!ENTITY % stat-seq             '%np;stat-seq' >
<!ENTITY % expr                 '%np;expr' >
<!ENTITY % cond                 '%np;cond' >
<!ENTITY % assign               '%np;assign' >
<!ENTITY % if                   '%np;if' >
<!ENTITY % else-if              '%np;else-if' >
<!ENTITY % else                 '%np;else' >
<!ENTITY % while                '%np;while' >
<!ENTITY % repeat               '%np;repeat' >
<!ENTITY % for                  '%np;for' >
<!ENTITY % for-pre              '%np;for-pre' >
<!ENTITY % for-post             '%np;for-post' >
<!ENTITY % label                '%np;label' >
<!ENTITY % branch               '%np;branch' >
<!ENTITY % call                 '%np;call' >
<!ENTITY % return               '%np;return' >
<!ENTITY % data                 '%np;data' >
<!ENTITY % elem                 '%np;elem' >
<!ENTITY % bind-type            '%np;bind-type' >
<!ENTITY % bind-func            '%np;bind-func' >
<!ENTITY % bind-static-params   '%np;bind-static-params' >
<!ENTITY % bind-tp              '%np;bind-tp' >
<!ENTITY % bind-fp              '%np;bind-fp' >
<!ENTITY % bind-cp              '%np;bind-cp' >
```

```
<!ENTITY % bind-params        '%np;bind-params' >
<!ENTITY % bind-param         '%np;bind-param' >
<!ENTITY % var                '%np;var' >
<!ENTITY % const              '%np;const' >
<!ENTITY % val                '%np;val' >
<!ENTITY % unit-dsg           '%np;unit-dsg' >
<!ENTITY % type-dsg           '%np;type-dsg' >
<!ENTITY % func-dsg           '%np;func-dsg' >
<!ENTITY % data-dsg           '%np;data-dsg' >
<!ENTITY % algo-dsg           '%np;algo-dsg' >
<!ENTITY % var-dsg            '%np;var-dsg' >
<!ENTITY % const-dsg          '%np;const-dsg' >
<!ENTITY % binding-dsg        '%np;binding-dsg' >
<!ENTITY % static-param-dsg   '%np;static-param-dsg' >
```

File defined by parts 50, 51, 52a, 55b, 56, 57a, 57b, 62a, 69, 72a, 74, 76a, 76b, 77a, 77b, 77c, 77d.

4.4 General Attributes

A typical phenomena in XGILF documents are elements that share parts of their attribute-list declarations. In order to avoid inconsistencies between these declarations and for documentation purposes, we represent these *general* attributes with parameter-entity declarations. This is common practice, for example the official HTML 4.01 recommendation [HTML99] employs this policy.

```
"../xgf/xgilf.dtd" 52a ≡
   ⟨General Attributes 52b, ... ⟩
```

File defined by parts 50, 51, 52a, 55b, 56, 57a, 57b, 62a, 69, 72a, 74, 76a, 76b, 77a, 77b, 77c, 77d.

Identifiers and References A fundamental ability of an XGILF processor is to locate specific elements within XGILF documents, for example an algorithm wants to refer to its function declaration. This is accomplished by tagging all important elements with an identifier. These identifiers are then referenced by other elements. Elements that have an identifier attribute will become part of the internal representation's symbol table for fast lookup.

```
⟨General Attributes 52b⟩ ≡
   <!ENTITY % Id            'id        ID        #REQUIRED' >
   <!ENTITY % IdReference   'ref       NMTOKEN   #REQUIRED' >
```

Definition defined by parts 52b, 53, 54a, 54b, 54c, 54d, 54e, 55a.
Definition referenced in part 52a.

The reason that `ref` attributes are of type `NMTOKEN` instead of `IDREF` is the constraint that identifiers referenced by `IDREF` attributes have to exist in the same XML document ([XML00], 3.3.1). But our system allows elements to reference parts from other XGILF documents in order to support modularity.

Valid identifier names follow a simple grammar. An identifier name starts with a sequence of letters, the *identifier prefix*, followed by an underscore character '_' and a sequence of characters not containing the dot character '`.`', the *identifier postfix*. The dot character is used as infix operator to create an identifier sequence. The formal grammar is given using the syntax and rules of the XML Recommendation ([XML00], p. 8).

```
xgilf-id  ::= Letter+ '_' ( NameChar - '.' )+
xgilf-ids ::= xgilf-id ( '.' xgilf-id )*
```

The identifier prefix is limited to a defined set of valid prefixes. Every prefix is directly associated with an XGILF element type. Table 4.1 lists all possible prefixes and their associated elements.

identifier prefix	associated element type
u	unit
f	function
a	algorithm
r	required (dependent) function
t	type declaration
d	type definition
p	dynamic value parameter
tp	type instantiation parameter
cp	constant value instantiation parameter
fp	function instantiation parameter
l	label
v	variable
c	constant
e	data structure element
bf	function binding
bt	type binding
bsp	static instantiation parameter bindings
bvp	dynamic value parameter bindings

Table 4.1: Table of valid identifier prefixes in XGILF and their associated elements.

Debugging Information Another important aspect is interoperability with debugging tools. Therefore, most elements can be enhanced with debugging information.

Some elements want to specify their source coarse grained. A source entity can be a file, an unique resource locator (URL), a program repository in a development environment, or some other kind of input stream. In general, the `src` attribute describes the source from which these XGILF elements were generated. The special value `system` denotes XGILF documents belonging to the GILF system.

```
⟨General Attributes 53⟩ ≡
   <!ENTITY % Source         'src         CDATA      #IMPLIED' >
```

Definition defined by parts 52b, 53, 54a, 54b, 54c, 54d, 54e, 55a.
Definition referenced in part 52a.

The `line` and `column` attributes specify the position of a source language construct in the input source that was translated into the current element. If `line-end` and `column-end` are not given, the construct is assumed to continue up to the end of the line from the given position.

```
⟨General Attributes 54a⟩ ≡
```

```
<!ENTITY % SourcePos      'line        CDATA     #IMPLIED
                           column      CDATA     #IMPLIED
                           line-end    CDATA     #IMPLIED
                           column-end  CDATA     #IMPLIED' >
```

Definition defined by parts 52b, 53, 54a, 54b, 54c, 54d, 54e, 55a.
Definition referenced in part 52a.

The `name` attribute can hold an arbitrary string that corresponds to information of a source language construct, e.g. an unmangled function name.

⟨General Attributes 54b⟩ ≡
```
<!ENTITY % SourceName     'name        CDATA     #IMPLIED' >
```

Definition defined by parts 52b, 53, 54a, 54b, 54c, 54d, 54e, 55a.
Definition referenced in part 52a.

Subelement Counter Some elements store the count of specific subelements in the `count` attribute, for example the number of parameters a function has. This is intended for speeding up the computation of XGILF documents and simple correctness checks.

⟨General Attributes 54c⟩ ≡
```
<!ENTITY % Count          'count       CDATA     #IMPLIED' >
```

Definition defined by parts 52b, 53, 54a, 54b, 54c, 54d, 54e, 55a.
Definition referenced in part 52a.

Access Modifiers The `access-mod` attribute controls if the corresponding symbol of the current element is exported, i.e. accessible from other compilation units. The symbol is denoted by its identifier. At the moment, the access modifier can make the symbol either `public` or `private`. Finer control would be possible, e.g. access could be granted or denied to specific compilation units.

⟨General Attributes 54d⟩ ≡
```
<!ENTITY % AccessModifier 'access-mod  (public|private) "public"' >
```

Definition defined by parts 52b, 53, 54a, 54b, 54c, 54d, 54e, 55a.
Definition referenced in part 52a.

Type Modifiers Elements that declare a type can indicate with the `type-mod` attribute that this type should be treated in some way special. Currently, the only usage of the type modifier is to indicate reference instead of value types.

⟨General Attributes 54e⟩ ≡
```
<!ENTITY % TypeModifier   'type-mod    (is-ref)  #IMPLIED' >
```

Definition defined by parts 52b, 53, 54a, 54b, 54c, 54d, 54e, 55a.
Definition referenced in part 52a.

Built-in Algorithms and Data Structures Elements that define an algorithm or data structure can indicate with the `built-in` attribute that this type should be treated specially by the back-end. These elements are recognized by the back-end as built-in algorithms and data structures of the GILF system. Appendix C gives an overview of the GILF core library and its built-in algorithms and data structures.

⟨General Attributes 55a⟩ ≡

```
<!ENTITY % BuiltIn        'built-in    (yes|no)  "no"' >
```

Definition defined by parts 52b, 53, 54a, 54b, 54c, 54d, 54e, 55a.
Definition referenced in part 52a.

4.5 Root Element

The root element of an XGILF file is of type `xgilf`. There is no special meaning associated with this element, its sole purpose is to hold compilation units (see section 4.6). In a traditional programming environment, an XGILF document can correspond to a source language file. But one can imagine more elaborate settings, e.g. all system units might be kept in an XML database. There, the XGILF document could correspond to the whole system library.

We develop the XGILF DTD by presenting the elements and their attributes. The element names were introduced as entities in section 4.3.

"../xgf/xgilf.dtd" 55b ≡

```
<!ELEMENT %xgilf; ((%unit;)+) >
<!ATTLIST %xgilf; %Soure; >
```

File defined by parts 50, 51, 52a, 55b, 56, 57a, 57b, 62a, 69, 72a, 74, 76a, 76b, 77a, 77b, 77c, 77d.

An `xgilf` element must hold at least one compilation unit. The element's `Source` attribute specifies the document's source. This can be a file, a network resource or some other input stream. The attribute value `system` is reserved to brand documents that were not generated from a source language but are part of the XGILF core library.

4.6 Compilation Units

Compilation units are the most coarse-grained structuring element in XGILF. We see compilation units as transformed source code program fragments, analogous to the notion in [Car97]. A program fragment is not self-contained, that means it does not contain definitions for all symbols it references. The external symbols have to be imported from other program fragments, preferably already transformed into a compilation unit. Therefore, it is not possible to compile a program fragment into an executable program.

The primary entities contained in XGILF compilation units are the signatures of functions and types and their definitions, algorithms and data structures. Some declarations and definitions are already linked together by preestablished bindings. The remaining unbound symbols have to be present in the program fragment that holds the program entry point. Moreover, compilation units can contain unit-wide constants and variables. These constituents of a compilation unit are motivated by our research's starting point, the generic language SUCHTHAT [Sch96], which incorporates the imperative statements from Aldes [LoCo92].

The source language has to map its module concept to compilation units, which is straightforward for SUCHTHAT, as it is algorithm centered.

"../xgf/xgilf.dtd" 56 ≡

```
<!ELEMENT %unit; ((%import;)?, (%declare;)?, (%define;)?,
                  (%bind;)?, (%store;)?, (%extend;)?) >
<!ATTLIST %unit;
          %Id; <!-- u -->
```

```
digest       CDATA     #REQUIRED
%Source; %SourcePos; %SourceName; >
```

File defined by parts 50, 51, 52a, 55b, 56, 57a, 57b, 62a, 69, 72a, 74, 76a, 76b, 77a, 77b, 77c, 77d.

Looking at the `unit` element definition, we see that a compilation unit is made up of six primary sections. Here, we give a short overview of their content, which will be detailed in more detail in the following sections.

Import Section (`import` element): The dependencies on symbols from other compilation units have to be made explicit in the import section.

Declaration Section (`declare` element): It holds the declarations of all function and type signatures that belong to the compilation unit.

Definition Section (`define` element): Algorithm and data structure definitions are combined in the definition section.

Binding Section (`bind` element): Bindings of algorithms to function declarations and of data structure definitions to type declarations are stored in the binding section.

Static Storage Section (`store` element): Unit-wide available storage for constants and variables reside in the compilation unit's global storage section.

Extension Section (`extend` element): Information specific to the source language can be stored in the extension section. This section is not specified here but serves as an anchor element for future extensions.

All valid `unit` identifiers start with the prefix 'u'. In the DTD, all `Identifier` attributes are followed by a comment that names the identifier's prefix. For example, the system unit for the boolean type and operations is called `u_bool`:

```
<unit id="u_bool" name="std_boolean" src="system">
```

The `digest` attribute's value is of central importance to GILF's recompilation mechanism. Disregarding the native code cache, GILF files are translated into machine code at program load time. GILF's high level of abstraction avoids any dedication to memory layouts or function calling conventions. The code generating loader sees the whole program and generates a valid executable. Thus, taking advantage of dynamic code generation, every GILF compilation unit can be created from its corresponding program fragment separately.

The native code cache of GILF systems requires more careful considerations with regard to recompilations. Whenever an algorithm or data structure changes in such a way that the binary code no longer resembles the semantics of the changed GILF or XGILF file, the code cache has to be updated. These checks are made with the `digest` attribute. It tries to concentrate the compilation unit's interface into a value, i.e. only changes in the unit that affect its clients should lead to changes of the `digest` value. This implies that all code sections residing in the cache have to be recompiled that are clients of one XGILF unit or document. Alternatively, the `digest` attribute can be given in the `xgilf` processing instruction. An intersting assessment of recompilation effects on development time is presented in [AdTiWe94].

The `unit` element also features the full set of debugging attributes. If the `Source` attribute is not specified in a child node of the compilation unit, it is assumed to be inherited.

4.6.1 Import Section

If a unit uses parts of another unit by referencing types, functions etc., the dependency on such units has to be stated explicitly in a unit's import section.

```
"../xgf/xgilf.dtd" 57a ≡
   <!ELEMENT %import; (%source;+) >

   <!ELEMENT %source; (%unit-dsg;+) >
   <!ATTLIST %source;
            input-src    CDATA     #REQUIRED >
```

File defined by parts 50, 51, 52a, 55b, 56, 57a, 57b, 62a, 69, 72a, 74, 76a, 76b, 77a, 77b, 77c, 77d.

The `input-src` attribute of a `source` element references another GILF input sources that contain the units to import. The unit designators enumerate these units contained in the input source.

4.7 Declaration Section

The declaration section introduces function and type signatures. These are referenced later on by their corresponding definitions using the identifier attribute. With this approach, a signature is given only once, no redundancy is introduced like in C++ declarations and definitions, where the signature has to be repeated.

```
"../xgf/xgilf.dtd" 57b ≡
   <!ELEMENT %declare; (%type; | %function;)*>
   ⟨Type Declarations 58a⟩
   ⟨Function Declarations 59⟩
   ⟨Instantiation Parameter Declarations 58b⟩
   ⟨Value Parameter Declarations 60a⟩
```

File defined by parts 50, 51, 52a, 55b, 56, 57a, 57b, 62a, 69, 72a, 74, 76a, 76b, 77a, 77b, 77c, 77d.

4.7.1 Type Declarations

A type declaration introduces a generic type signature, referenceable by its identifier. The type signature is denoted by its instantiation parameters, these parameters will we explained in the next section. Data structures from other compilation units can reference a type declaration if the `AccessModifier` attribute is set appropriately (see section 4.4 for details), data structures from the same unit have unrestricted access.

```
⟨Type Declarations 58a⟩ ≡
   <!ELEMENT %type; ((%type-params;)?, (%const-params;)?, (%func-params;)?)>
   <!ATTLIST %type;
            %Id; <!-- t -->
            %AccessModifier;
            %SourcePos; %SourceName; >
```

Definition referenced in part 57b.

4.7.2 Instantiation Parameters

Instantiation parameters are fixed at compile time and describe an instantiation, both of types and functions. A type declaration prescribes the number and kind of instantiation arguments that an instantiation has to provide in order to be valid. All semantic checks of the given instantiation arguments must have been performed by the front-end during instantiation analysis. Instantiation application in the back-end simply checks if the instantiation is described completely, i.e. all the instantiation parameters are bound to arguments. Instantiation parameters are divided into the following categories:

Type Parameters: They introduce type variables which are reference by their identifier. Ultimately, type parameters have to be bound to representing data structures which will replace the type variables in the process of instantiation application.

Function Parameters: They allow the type's or function's behavior to be parameterized by other functions. At the moment, this makes sense for functions and their algorithmic implementations only, as no operational properties are available for types. This may change if class types are introduced. An example for such a function parameter is the comparison function that is used in a sorting algorithm. It directly affects the algorithm's behavior and must be present at the instantiation site at compile time.

Constant Value Parameters: This last category of instantiation parameters is used to parameterize algorithms and data structures with a value that is constant at compile time. Examples are the number of elements for a tuple data structure or the size of a static array.

It is important to realize that two instantiations of the same data structure or algorithm are considered equal only if all their instantiation parameters match exactly.

```
⟨Instantiation Parameter Declarations 58b⟩ ≡
   <!ELEMENT %type-params; (%type-param;)+ >
   <!ATTLIST %type-params; %Count; >
   <!ELEMENT %type-param; (%type-dsg;) >
   <!ATTLIST %type-param;
             %Id; <!-- tp -->
             %SourcePos; %SourceName; >

   <!ELEMENT %func-params; (%func-param;)+ >
   <!ATTLIST %func-params; %Count; >
   <!ELEMENT %func-param; (%func-dsg;) >
   <!ATTLIST %func-param;
             %Id; <!-- fp -->
             %SourcePos; %SourceName; >

   <!ELEMENT %const-params; (%const-param;)+ >
   <!ATTLIST %const-params; %Count; >
   <!ELEMENT %const-param; (%type-dsg;) >
   <!ATTLIST %const-param;
             %Id; <!-- cp -->
             %SourcePos; %SourceName; >
```

Definition referenced in part 57b.

All instantiation parameters are structured the same way. A container element (`type-params`, `func-params`, and `const-params`) subsumes parameters from the same category. The number of parameters in the container for this category is captured in the `Count` attribute. Then, every parameter is represented by an element with an unique identifier. The identifier prefixes for the instantiation parameter categories is summarized in table 4.1. References to these identifiers in algorithms and data structures that implement the function or type are replaced during instantiation application.

Function parameters require a function designator which denotes the signature the bound instantiation argument must adhere to. Accordingly, type and constant value parameters require a type designator to fix their signature.

4.7.3 Function Declarations

Functions can have two kinds of parameters. First, they can have instantiation parameters. These are handled exactly the same way as for type declarations. Second, functions have the usual dynamic value parameters. These are passed to the function at every calling site, with arguments matching the types of the function's parameters. The difference between dynamic and constant value parameters is that in the case of constant value parameters the type *and* the value of the parameter influence the function's signature, whereas only a dynamic value parameter's type contributes to the signature.

⟨Function Declarations 59⟩ ≡
```
<!ELEMENT %function; ((%type-params;)?, (%const-params;)?, (%func-params;)?,
                      (%params;)?) >
<!ATTLIST %function;
          %Id; <!-- f -->
          %AccessModifier;
          %SourcePos; %SourceName; >
```
Definition referenced in part 57b.

The `function` element has the same attributes as the `type` element in the previous section. The additional `params` child is worth elaborating on. It is also a container element for the `param` elements that hold the information for every value parameter of the function.

4.7.4 Value Parameters

Value parameters are known in all programming languages with function and procedure abstraction for control flow. In a generic setting, the type of a value parameter may not only be an available type from the current or an imported module, but also a type variable introduced in the function's generic signature. GILF supports passing algorithms as runtime parameters in the form of reference parameters. The information that a parameter is to be treated as reference to a callable entity is made explicit by either the identifier of the static instantiation parameter (see section 4.7.2) or the binding (see section 4.9). Both are pointed to by the corresponding designator. During instantiation application, these designators will be replaced by the fully instantiated algorithm or data structure. Following restrictions apply to the designators: The `static-param-dsg` has to reference either a type or function instantiation parameter, whereas the `binding-dsg` has to reference either a type or function binding.

⟨Value Parameter Declarations 60a⟩ ≡

```
⟨Value Parameter Passing Modifiers 60b⟩
<!ELEMENT %params; (%param;)+ >
<!ATTLIST %params; %Count; >
<!ELEMENT %param; (%static-param-dsg; | %binding-dsg;) >
<!ATTLIST %param;
          %Id; <!-- p -->
          %PassingModifier;
          %SourcePos; %SourceName; >
```

Definition referenced in part 57b.

Value Parameter Passing Conventions The second important aspect of value parameters are the passing conventions enforced by the language. As GILF is intended to support a reasonable amount of diverse imperative generic languages, it supports the most commonly used passing conventions: call-by-value, call-by-result, call-by-value-result, and call-by-reference. The `PassingModifier` attribute in XGILF controls which passing convention is used.

```
⟨Value Parameter Passing Modifiers 60b⟩ ≡
   <!ENTITY % PassingModifier
          'pass       ( in     | out     | inout     | out!     | inout!     |
                        in_val | out_val | inout_val | out_val! | inout_val! |
                        in_ref | out_ref | inout_ref | out_ref! | inout_ref! )
                        #REQUIRED' >
```

Definition referenced in part 60a.

Call-by-value copies the value of an argument to the function into the corresponding parameter on function entry. This is reasonable for objects that fit into a machine register, but sometimes it is also reasonable if you have to create a copy of the object passed anyway. Built-in machine types are passed as value by default when using the `in` value for the `pass` attribute. One can force usage of call-by-value by using the `in_val` value. If a composite data structure is passed by value, it is copied bitwise. This behavior can be overridden by providing a `clone` algorithm for this type and a binding to `clone`'s function declaration from the core library (see appendix C). Then, this algorithm will be called.

Call-by-result is similar to call-by-value but takes place only on algorithm return. It is indicated by the `out` attribute value for machine types and `out_val` forces this passing convention for all data structures. Its main purpose is to transfer computed results to the caller.

Call-by-value-result is the combination of call-by-value and call-by-result. `inout_val` indicates this behavior for all data structures, and `inout` selects it for machine types by default.

One has to keep in mind the performance implications when using these three calling conventions. For data structures that fit into a register, like built-in machine types, this is usually the best choice. But copying large structures like arrays can lead to a performance hit as large amounts of memory are moved.

Call-by-reference is the last supported calling convention in GILF. When an argument is passed by reference, only a memory reference (or pointer) is available to the algorithm body. Thus, the operations on this argument directly operate on the original argument from the calling site. The compiler will usually flag errors if statements in the algorithm body try to manipulate arguments that are passed as read-only references. Data structures that do not fit into a machine register are automatically treated as references, if not forced otherwise by the `_val` suffix to the passing modifier. Analogously, one can force the usage of reference semantics for parameter passing with the `_ref` suffix also for built-in data structures.

Call-by-name is not directly supported in GILF. It is explained here for completeness of this overview on passing conventions. The only significant language that provides call-by-name is Algol 60. Call-by-name is similar to call-by-reference, but the address of the argument is calculated at each access. For example, one can pass an array element by name like `a[i]`. If `i` changes inside the callee, different elements may be accessed.

The last thing to discuss about parameter passing are the `out` and `inout` values that have an exclamation mark '`!`' as suffix. These parameters denote the return type of a function call when it is used as an expression. Only one parameter may be marked this way in a function declaration. More on the simple notion of expressions in GILF will follow in the section about algorithm definitions.

A simple example illustrates the descriptions given in this section, the declaration of a generic addition function $+ : T \times T \to T$. A function declaration is introduced that is accessible by the identifier `u_func.f_+`. It has one generic type parameter that introduces a type variable locally accessible by the identifier `tp_0`. Finally, two input parameters and one output parameter are declared which employ the default passing convention. All three parameters are declared as having the type `tp_0`, which is the function's sole generic type variable. The parameter `p_2` is marked as the function's return type.

```
<function id="u_func.f_+" name="+">
 <type-params count="1"> <type-param id="tp_0" name="T"/> </type-params>
 <params count="3">
  <param pass="in" id="p_0" name="arg1"> <static-param-dsg ref="tp_0"/> </param>
  <param pass="in" id="p_1" name="arg2"> <static-param-dsg ref="tp_0"/> </param>
  <param pass="out!" id="p_2" name="ret"> <static-param-dsg ref="tp_0"/> </param>
 </params>
</function>
```

Now that we have seen how to declare type and function signatures, we will study how to define their implementations.

4.8 Definition Section

The definition element `define` holds children of type `algorithm` and `data`. They describe the implementations that realize declared functions and types.

```
"../xgf/xgilf.dtd" 62a ≡
  <!ELEMENT %define; (%data; | %algorithm;)+ >
  ⟨Algorithms 64a⟩
  ⟨Data Structures 62b⟩
```

File defined by parts 50, 51, 52a, 55b, 56, 57a, 57b, 62a, 69, 72a, 74, 76a, 76b, 77a, 77b, 77c, 77d.

4.8.1 Data Structures

We start with the definition of data structures, they are given in `data` elements. Data structures in GILF define either record or union types, i.e. they are used to define aggregate datatypes. Other built-in data structures like arrays or machine types which subsume integer or floating point types are part of the GILF core library.

The data structures available in GILF simply describe the memory layout of a type and carry no behavior, like classes in object-oriented programming languages or virtual machines like the Java Virtual Machine (JVM, [LiYe99]). Therefore, the content is described by `elem` elements. Each data structure is referenceable through its identifier, which can be recognized by identifier prefix `d`. The data structure's identifier reference attribute establishes its link to a type declaration. Only type declarations may be referenced by `data` elements. The `AccessModifier` attribute states if the data structure is also visible to other compilation units.

A very important aspect of GILF data structures is the `BuiltIn` attribute. If this flag is set to `yes`, the `data` element has to be empty. The back-end will recognize it as special data structure and will synthesize the appropriate code for it. For example, all machine integer types are introduced this way in the GILF core library.

The `kind` attribute determines the data structure's kind, i.e. if it is a record or union type. A record type occupies at least the accumulated memory consumption of its elements, whereas a union occupies at least the memory of its largest element. Padding bytes can lead to a greater memory consumption.

```
⟨Data Structures 62b⟩ ≡
    <!ELEMENT %data; (%elem;)* >
    <!ATTLIST %data;
              %Id; <!-- d -->
              %IdReference; <!-- t -->
              %AccessModifier;
              %BuiltIn;
              kind        (record | union)  #REQUIRED
              %SourcePos; %SourceName; >

    <!ELEMENT %elem; (%static-param-dsg; | %binding-dsg;) >
    <!ATTLIST %elem;
              %Id; <!-- e -->
              %TypeModifier;
              %SourcePos; %SourceName; >
```

Definition referenced in part 62a.

As type declarations are generic, data structures may be generic, too. This means that a data structure's element type can reference a type or function variable from the declaration's instantiation parameter list. This is done with the static instantiation parameter designator. Also, an instantiation can be used as element type, pointed to by the binding designator. The same restrictions apply as for value parameters. Each element from a data structure gets its own identifier such that they may be accessed for reading and writing. The `TypeModifier` attribute can be used to store a reference to data structures instead of embedding all its fields. For algorithms, this is the only option and applied automatically. Elements have the identifier prefix `e`. A STL like pair data structure together with its type declaration would look like this:

```
<!-- Heterogeneous pair type declaration. -->
<type id="u_ext.t_pair" name="pair">
 <type-params count="2">
  <type-param id="tp_0"/> <type-param id="tp_1"/>
 </type-params>
</type>
<!-- Pair data structure definition. -->
<data id="u_ext.d_pair" ref="u_ext.t_pair" kind="record" name="pair">
 <elem id="e_first" name="first"> <static-param-dsg ref="tp_0"/> </elem>
 <elem id="e_second" name="second"> <static-param-dsg ref="tp_1"/> </elem>
</data>
```

4.8.2 Algorithms

Algorithm definitions reside in `algorithm` elements. An algorithm definition consists of three major parts.

The first one, captured in the `stat-seq` element, is quite obvious. It holds the statement sequence that describes the algorithm's behavior. The available statements will be explained in the next paragraphs. The statement child elements inside a statement sequence are executed in the order of their occurrence.

Second, an algorithm has a local storage for the variables and constants it needs. This storage is defined in the `store` element. Currently, GILF algorithms have to collect their variables and constants and list them all in the algorithm-wide storage. Algorithms have a flat scope. This is motivated by SUCHTHAT's algorithm notion (see [Sch96], p. 45f.).

Finally, an algorithm definition has to list its functions that are subject to overload resolution and depend upon the algorithm's instantiation parameters. This list is captured in the `func-params` element. The instantiation of a generic algorithm has to provide bindings for these function symbols, thus describing the outcome of the overload resolution. This approach relieves the back-end from performing overload resolution, which can be expensive for generic programming languages. The function parameters introduced inside the algorithm's `func-params` element are called *required or dependent functions*, and the identifier prefix for them is `r`.

The existence of this element has two reasons. First, one design goal of GILF is to make as much information about the source code explicit as possible, such that further processing by the back-end can proceed quickly. Second, GILF does not enforce a type system in the front-end. The results of overload resolution and instantiation analysis are stored in the bindings section instead of leaving these computations to the back-end. An example should further clarify the purpose of this list of functions. Assuming that inside the algorithm body two values are added, and the addition function depends on one of the algorithm's type variables, then the function symbol representing the addition will be subject to generic overload resolution. If this type variable is bound to a machine integer, the result of the overload resolution will be the processor's built-in integer addition. Otherwise, binding the type variable to an arbitrary precision integer type, the result of the overload resolution will be the algorithm that adds such integers. Thus, the function symbol will either be replaced by a processor instruction or a function call.

```
⟨Algorithms 64a⟩ ≡
  <!ELEMENT %algorithm; ((%func-params;)?, (%store;)?, %stat-seq;)? >
  <!ATTLIST %algorithm;
          %Id; <!-- a -->
          %IdReference; <!-- f -->
```

```
        %AccessModifier;
        %BuiltIn;
        %SourcePos; %SourceName; >
⟨Expressions 64b⟩
⟨Statements 65a, ... ⟩
```

Definition referenced in part 62a.

Algorithms introduce an identifier that begins with the identifier prefix `a`. They have to reference a function declaration with the `IdReference` attribute which describes the algorithm's signature. The `AccessModifier` and `BuiltIn` attributes have the same meaning as they have in data structure definitions.

4.8.2.1 Expressions

Expressions in GILF follow its usual intent to preserve already computed results and be easily accessible to the back-end. Therefore, it is up to the front-end to resolve issues like operator precedence and associativity of fully general expressions. Expressions are usually composed of constants, variables, and function calls and their return values, operators are already resolved to function calls. In the end, expressions yield a typed value. There are three ways to represent a typed value in GILF:

1. as a variable,
2. as a constant,
3. or as a function call.

The first two are obvious, but what type has a function call? GILF supports expressions through function calls in a simple way. One and only one of the `out` or `inout` parameters can be marked as return parameter (see section 4.7.4). This return parameter determines a function's type when used as expression.

```
⟨Expressions 64b⟩ ≡
  <!ELEMENT %expr; (%var-dsg; | %const-dsg; | %call;) >
  <!ATTLIST %expr; %SourcePos; %SourceName; >
```

Definition referenced in part 64a.

The following example shows an expression that corresponds to the addition of a variable `v_i` to the second element of a pair[2] record `v_p`. It assumes a nongeneric binary function `f_int_add`, whose trivial binding is given with the function binding `bf_int_add`.

```
<expr>
 <call ref="u_ext.f_int_add">
  <binding-dsg ref="u_ext.bf_int_add"/>
  <bind-params>
   <bind-param ref="p_0"> <expr><var-dsg ref="v_p.e_second"></expr> </bind-param>
   <bind-param ref="p_1"> <expr><var-dsg ref="v_i"></expr> </bind-param>
  </bind-params>
 </call>
</expr>
```

[2] The pair type was introduced to exemplify data structure definitions in section 4.8.1.

4.8.2.2 Statements

The statements available in GILF should provide enough possibilities to map any imperative vocabulary to GILF without great problems. GILF statements are mainly influenced by the statements available in C++ and Aldes. A sequence of statements is represented by the `stat-seq` element.

```
⟨Statements 65a⟩ ≡
    <!ELEMENT %stat-seq; ( %assign; | %if; | %while; | %repeat; | %for;
                         | %label; | %branch; | %call; | %return;)* >
    <!ATTLIST %stat-seq; %SourcePos; %SourceName; >
```

Definition defined by parts 65a, 65b, 66a, 66b, 66c, 66d, 67, 68a, 68b.
Definition referenced in part 64a.

Assignment A central aspect in imperative languages is changing state, which is performed by assigning values to variables which represent the current state. Therefore, GILF directly supports assignments. The built-in assignment statement bitwise copyies the right-hand side expression to the left-hand side variable. In order to support other semantics, the front-end has to introduce a dedicated assignment function and implement the desired semantics in an algorithm based on the built-in assignment statement.

```
⟨Statements 65b⟩ ≡
    <!ELEMENT %assign; (%var-dsg;, %expr;) >
    <!ATTLIST %assign; %SourcePos; %SourceName; >
```

Definition defined by parts 65a, 65b, 66a, 66b, 66c, 66d, 67, 68a, 68b.
Definition referenced in part 64a.

Structured Control Flow Structured control flow is supported by the `if`, `while`, `repeat`, and `for` elements. These elements are the preferred way in GILF to express control flow, as unstructured control flow complicates the application of optimization algorithms later on in the code generation process (see [Bra95] for a discussion).

The `cond` entities are expressions with an additional validity constraint which requires the expression to be of type `u_bool.t_bool` from the GILF core library. This way we introduce no dependencies in statements other than on the built-in boolean type and its comparison functions.

```
⟨Statements 66a⟩ ≡
    <!ENTITY % cond '%expr;' >
```

Definition defined by parts 65a, 65b, 66a, 66b, 66c, 66d, 67, 68a, 68b.
Definition referenced in part 64a.

The `if` element holds no big surprise. It allows to give an arbitrary number of `else-if` branches and an optional `else` branch. The conditions for branches are expressed with the `cond` entity. It can be any boolean GILF expression.

```
⟨Statements 66b⟩ ≡
    <!ELEMENT %if; (%cond;, %stat-seq;, ((%else-if;)*, (%else;)?)) >
    <!ATTLIST %if; %SourcePos; %SourceName; >

    <!ELEMENT %else-if; (%cond;, %stat-seq;) >
    <!ATTLIST %else-if; %SourcePos; %SourceName; >
```

```
<!ELEMENT %else; (%stat-seq;) >
<!ATTLIST %else; %SourcePos; %SourceName; >
```

Definition defined by parts 65a, 65b, 66a, 66b, 66c, 66d, 67, 68a, 68b.
Definition referenced in part 64a.

The `while` and `repeat` elements represent the typical looping constructs. The condition of the `while` loop is checked prior to entrance of the body, whereas the `repeat` loop first executes its body and then checks the condition.

⟨Statements 66c⟩ ≡

```
<!ELEMENT %while; (%cond;, stat-seq) >
<!ATTLIST %while; %SourcePos; %SourceName; >

<!ELEMENT %repeat; (%cond;, stat-seq) >
<!ATTLIST %repeat; %SourcePos; %SourceName; >
```

Definition defined by parts 65a, 65b, 66a, 66b, 66c, 66d, 67, 68a, 68b.
Definition referenced in part 64a.

The `for` statement in GILF is very general, like its C++ counterpart. First, the statement sequence in the `for-pre` element is executed once. Then the condition is always checked before entering the loop body. After executing the loop body, the statement sequence in the `for-post` element is executed.

⟨Statements 66d⟩ ≡

```
<!ELEMENT %for; (%cond;, for-pre?, for-post?, stat-seq) >
<!ATTLIST %for; %SourcePos; %SourceName; >

<!ELEMENT %for-pre; (stat-seq) >
<!ATTLIST %for-pre; %SourcePos; %SourceName; >

<!ELEMENT %for-post; (stat-seq) >
<!ATTLIST %for-post; %SourcePos; %SourceName; >
```

Definition defined by parts 65a, 65b, 66a, 66b, 66c, 66d, 67, 68a, 68b.
Definition referenced in part 64a.

Unstructured Control Flow Sometimes one cannot avoid direct jumps, especially when creating intermediate code. Branches are split up in two categories, local and nonlocal branches. Local branches jump to locations denoted by a label inside the current algorithm. The implementation of these branches poses no real problem. The same is not the case for non-local branches. The activation of the current algorithm has to be terminated, which is a recursive process if the algorithm is recursive itself (see [GrBaJa$^+$00]).

⟨Statements 67⟩ ≡

```
<!ENTITY % Locality 'local      ( yes | no ) "yes"' >

<!ELEMENT %label; EMPTY >
<!ATTLIST %label;
          %Id; <!-- l -->
          %Locality;
          %SourcePos; %SourceName; >

<!ELEMENT %branch; EMPTY>
```

```
<!ATTLIST %branch;
          %IdReference; <!-- l -->
          %Locality;
          %SourcePos; %SourceName; >
```

Definition defined by parts 65a, 65b, 66a, 66b, 66c, 66d, 67, 68a, 68b.
Definition referenced in part 64a.

The locality of a label or branch is designated by the `local` attribute, which defaults to local. Labels introduce an identifier that can be referenced by `branch` elements. Their identifier prefix is `l`. Notice that non-local labels have to provide a fully qualified identifier, i.e. it has to reside in a identifier chain which contains the algorithm's and the compilation unit's identifiers.

Function Calls The last relevant statement is the function call, which is expressed with the `call` element. A function call in a generic programming language involves the following steps:

Instantiation analysis The front-end processes the generic function call, which consists of an instantiation request. It has to check if the provided instantiation arguments are valid, thus constitute a legal instantiation. Thereafter, a source level representation of the requested instance is generated.

Overload resolution The function symbol representing the function call in the just generated representation is subject to the front-end language's overload resolution, which will finally bind the symbol to one or more valid algorithms.

Instantiation application Code is generated for the algorithm instance(s) created in the previous steps. In GILF, this includes the collection of bindings, which describe the selected instance(s).

Activation The actual function call is performed by passing the value parameters and transferring control to the algorithm.

The first two steps take place at compile time, i.e. statically in the language front-end. This makes sense as each front-end language can have its own semantic idiosyncrasies with regard to instantiation analysis and overload resolution. Therefore, the results of these processes should be stored in GILF such that instantiation application will be straightforward.

After successful instantiation analysis, the front-end can completely describe the requested instance. This can either be done with a function variable introduced by a instantiation parameter, or a complete instantiation description given in a function binding. These possibilities are indicated by the static instantiation parameter designator or the binding designator, respectively. More details on the bindings are given in section 4.9. Furthermore, a feature for indirect function call's is provided. This is made possible with the variable and constant designator child elements. They have to point to algorithm instances, which will be the target of the function call.

Finally, the dynamic value parameters have to be bound to typed expressions in the `bind-params` element. The bindings are either given locally, i.e. nested inside the `call` element, or globally, in the binding section. After following all designators and applying the available bindings, all parameters must be bound to a typed expression. This implies that bindings may be spread across several elements and are linked together with binding designators. The most local bindings replace more global bindings. The passing convention employed for each parameter is denoted in the function signature's `params` element.

```
⟨Statements 68a⟩ ≡
   <!ELEMENT %call; ((%static-param-dsg; | %binding-dsg; | %var-dsg; | %const-dsg;),
                     (%bind-params;)?) >
   <!ATTLIST %call;
             %IdReference; <!-- f -->
             %SourcePos; %SourceName; >
```

Definition defined by parts 65a, 65b, 66a, 66b, 66c, 66d, 67, 68a, 68b.
Definition referenced in part 64a.

Please notice that the function call element in GILF is a generalization of the typical notion of a function call in other intermediate representations. This is strongly motivated by the fact that overload resolution is tightly coupled to generic programming. A function call element is better regarded as an overloaded symbol. Only its bindings will tell if it is actually translated into a function call or into a machine operation.

With the `return` element the current algorithm is left cleanly. This may involve copying of the output parameters if they are passed by result. The optional expression child node will be copied to the function's marked return parameter (see section 4.7.4). Of course, their types must match.

```
⟨Statements 68b⟩ ≡
   <!ELEMENT %return; (%expr;)? >
   <!ATTLIST %return; %SourcePos; %SourceName; >
```

Definition defined by parts 65a, 65b, 66a, 66b, 66c, 66d, 67, 68a, 68b.
Definition referenced in part 64a.

4.9 Binding Section

The binding section in GILF is a novel feature in intermediate representations and is directly motivated by the needs of generic programming. Inside the `bind` container element, semantic properties of the front-end language are stored in order to completely free the back-end form computing this data. Moreover, representing these properties in the intermediate language avoids its proliferation with features that are better handled in the language front-end. Finally, the binding section supports some interesting features of the GILF system like automatic algorithm selection. In general, one should remark that bindings in GILF are performed with the help of identifiers and references to these identifiers, effectively introducing named instantiation and value parameters.

The `bind` element is the general container for bindings in XGILF. They are divided into three different categories:

1. **Function and Type Bindings.** These are used to connect declarations to their definitions, this means data structures are bound to type declarations and algorithms are bound to function declarations, respectively.

2. **Instantiation Parameter Bindings.** These are used to bind instantiation arguments to their corresponding instantiation parameters. Effectively, instantiation parameter bindings describe instantiations of functions and types.

3. **Value Parameter Bindings.** These bindings simply describe the arguments of function calls.

```
"../xgf/xgilf.dtd" 69 ≡
   <!ELEMENT %bind; ((%bind-func;)*, (%bind-type;)*, (%bind-static-params;)*,
                    (%bind-params;)*) >
   ⟨Function and Type Bindings 70a⟩
   ⟨Static Instantiation Parameter Bindings 70b, ... ⟩
   ⟨Dynamic Value Parameter Bindings 71b⟩
```

File defined by parts 50, 51, 52a, 55b, 56, 57a, 57b, 62a, 69, 72a, 74, 76a, 76b, 77a, 77b, 77c, 77d.

4.9.1 Function and Type Bindings

Function and type bindings establish the connection between declarations and their definitions. Function and type bindings are represented by the elements `bind-func` and `bind-type`, respectively. They are structured the same way.

First, if the function or type is generic, an element of type `bind-static-params` describes the instance for which the binding applies. Then, at least one designator points to the implementation of the bound construct. Currently, it makes only sense for functions to provide more than one implementation. In this case, it is up to the algorithm selection unit to choose the best candidate for efficient execution. The selection process can be based on purely static data, most prominently the instantiation arguments, but also on runtime properties, like data gathered in a profile run. Finally, optional binding designators point to other bindings. This feature can be used to separate common pieces of bindings into reusable parts, which will be recombined for the particular cases later.

```
⟨Function and Type Bindings 70a⟩ ≡
   <!ELEMENT %bind-type; ((%bind-static-params;)?, (%binding-dsg;)*, (%data-dsg;)+) >
   <!ATTLIST %bind-type;
             %Id; <!-- bt -->
             %IdReference; <!-- t --> >

   <!ELEMENT %bind-func; ((%bind-static-params;)?, (%binding-dsg;)*, (%algo-dsg;)+) >
   <!ATTLIST %bind-func;
             %Id; <!-- bf -->
             %IdReference; <!-- f --> >
```

Definition referenced in part 69.

Bindings introduce identifiers by which they can be referenced. The prefix for type bindings is `bt`, the one for function bindings is `bf`.

The identifier reference determines the construct's signature by pointing to the appropriate declaration. References in `bind-type` elements have to refer to type declarations, those in `bind-func` elements to function declarations.

The following example shows a function binding that describes the bindings of a generic factorial function `f_fact`, instantiated for the built-in machine type `d_uword`. It binds two algorithms, an iterative one called `a_facti`, and a recursive one called `a_factr`, to the generic declaration. Notice that the bindings of the required functions for both algorithms are omitted.

```
<bind-func id="u_ext.bf_fact" ref="u_ext.f_fact">
 <bind-static-params>
  <bind-tp ref="tp_0"> <binding-dsg ref="u_mtype.bt_uword"/> </bind-tp>
```

```
  </bind-static-params>
  <algo-dsg ref="u_ext.a_facti"/> <!-- iterative algorithm -->
  <algo-dsg ref="u_ext.a_factr"/> <!-- recursive algorithm -->
</bind-func>
```

Specialization, an important technique in generic programming, is supported in GILF by means of function and type bindings. The designators in the binding element describe the valid algorithms for the given instantiation, thus a binding representing a specialization provides only the set of specialized algorithms.

4.9.2 Static Instantiation Parameter Bindings

The binding of instantiation parameters characterizes instances of a generic construct statically. They are specified with `bind-static-params` elements, whose subelements are bindings for type parameters (`bind-tp`), for constant value parameters (`bind-cp`), and for function parameters (`bind-fp`). Typical for bindings, they may be spread over several elements and combined by means of binding designators for sharing purposes. Identifiers introduced by static instantiation parameter bindings are prefixed with `bsp`.

```
⟨Static Instantiation Parameter Bindings 70b⟩ ≡
  <!ELEMENT bind-static-params (bind-tp*, bind-cp*, bind-fp*, binding-dsg*) >
  <!ATTLIST bind-static-params %Id; >
```

Definition defined by parts 70b, 71a.
Definition referenced in part 69.

The purpose of these bindings is to bind instantiation parameters to instances of generic constructs, either data structures or algorithms. These are described with function or type bindings, as discussed above. The bindings' subelements are therefore nested function or type bindings, or designators pointing to such bindings.

```
⟨Static Instantiation Parameter Bindings 71a⟩ ≡
  <!ELEMENT %bind-tp; (%binding-dsg; | %bind-type;) >
  <!ATTLIST %bind-tp;
            %IdReference; <!-- tp --> >

  <!ELEMENT %bind-fp; (%binding-dsg; | %bind-func;) >
  <!ATTLIST %bind-fp;
            %IdReference; <!-- fp --> >

  <!ELEMENT %bind-cp; (%const-dsg;) >
  <!ATTLIST %bind-cp;
            %IdReference; <!-- cp --> >
```

Definition defined by parts 70b, 71a.
Definition referenced in part 69.

The identifier references of instantiation parameter bindings refer to the parameters' identifiers in the context in which they occur. For example, if a generic function is called inside a generic algorithm, the binding may refer to the algorithm's instantiation parameters given in its function signature. This will sometimes require elaborate tracking of the reference structures, thus demanding a good debugger at the GILF level.

4.9.3 Dynamic Value Parameter Bindings

The last binding category is the one for bindings of value parameters of function calls. They are straightforward and specified by bind-param elements, that are subelements of the bind-params container element. Nonlocal value parameter bindings introduce an identifier prefixed with bvp. The bound values are given by expression nodes, which were described in section 4.8.2 on algorithm definitions.

```
⟨Dynamic Value Parameter Bindings 71b⟩ ≡
    <!ELEMENT %bind-params; ((%bind-param;)*, (%binding-dsg;)*) >
    <!ATTLIST %bind-params; %Id; >

    <!ELEMENT %bind-param; (%expr;) >
    <!ATTLIST %bind-param;
              %IdReference; <!-- p -->
              %SourcePos; %SourceName; >
```

Definition referenced in part 69.

The identifier reference in bind-param elements has to refer to parameter identifiers. If the parameter binding is not given locally, these identifiers are determined by the function call's context.

4.10 Static Storage

The store element in XGILF is used to describe static storage. Static storage can appear at two different scope levels in GILF, either as unit wide storage of constants and variables, or as storage local to an algorithm. These scoping rules are quite rigid, and later revisions of GILF may introduce block scoped storage, which is a feature of most contemporary programming languages like C++ and Java. It would facilitate targeting GILF as intermediate representation.

```
"../xgf/xgilf.dtd" 72a ≡
    <!ELEMENT %store; (%var; | %const;)* >
    ⟨Static Attribute 72b⟩
    ⟨Variables 73a⟩
    ⟨Constants 73b⟩
```

File defined by parts 50, 51, 52a, 55b, 56, 57a, 57b, 62a, 69, 72a, 74, 76a, 76b, 77a, 77b, 77c, 77d.

Variables reserve the raw memory needed to represent typed values. In GILF there exist two kinds of variables:

1. Algorithm variables.
2. Unit variables.

Algorithm variables are local to the algorithm they are declared inside, and statements of this algorithm may manipulate the values stored in these variables, only. The type of algorithm variables is fixed for each instance of the algorithm, that means functions operating on these variables may not be instantiated in such a way that they require different "instances" of the same variable.

Sometimes, a group of related algorithm instances need to share state. A prominent example are static data members of a class template in C++. Here, all member functions of each class template instantiation have access to these data members. In such cases, a variable has to be marked as static with the corresponding attribute.

```
⟨Static Attribute 72b⟩ ≡
   <!ENTITY % StaticModifier  'static    (yes|no)  "no"' >
```

Definition referenced in part 72a.

Unit variables allow all algorithm instantiations of a unit to communicate with each other, they even allow cross-unit communication if the access attribute is set to `public`. The type of a unit variable has to be fixed, that means all of the instantiation parameters of the variable's type have to be bound. This is necessary in order to support access from all instantiated algorithms. Otherwise, different instantiations would require different instances of the variable and would ultimately manipulate different memory locations.

Variables introduce an identifier with the prefix `v`. The optional value child node `val` denotes the value to which the variable should be initialized. If it is not present, only the raw memory will be reserved, filled with arbitrary data.

```
⟨Variables 73a⟩ ≡
   <!ELEMENT %var; ((%static-param-dsg; | %binding-dsg;), (%val;)? >
   <!ATTLIST %var;
             %Id; <!-- v -->
             %TypeModifier;
             %AccessModifier;
             %StaticModifier;
             %SourcePos; %SourceName; >
```

Definition referenced in part 72a.

Constants are treated almost the same as variables. The most significant differences are that the value child node has to be present and the manipulation of the initialized memory of the constant may not be changed during algorithm execution. Constants introduce an identifier with the prefix `c`.

Notice that constants are usually treated as nullary functions in high-level languages, e.g. TECTON. But constants play an important role in optimization phases of the backend. Therefore, we decided to represent them explicitly in GILF.

```
⟨Constants 73b⟩ ≡
   <!ELEMENT %const; ((%static-param-dsg; | %binding-dsg;), %val;) >
   <!ATTLIST %const;
             %Id; <!-- c -->
             %TypeModifier;
             %AccessModifier;
             %StaticModifier;
             %SourcePos; %SourceName; >
```

Definition referenced in part 72a.

4.10.1 Representing Value

This far, we have introduced almost all relevant abstractions to map generic, imperative languages to GILF. These abstractions are functions, types, algorithms, data structures, statements, and state. What is still missing is the ability to represent the values of typed variables and constants.

As XGILF is an XML based format, values have to be stored in textual form. Every valid value must have at least one lexical representation. For example, a floating point number can be represented in a human readable format, like `12.7`, or the binary data that represents the best approximation of this number in the native machine format. Depending on the lexical representation, further processing is necessary. It is straightforward to convert a hexadecimal representation to binary data, but finding a good approximation of a floating point number is more demanding.

The W3C recommendation for XML Schemas deals with this problem extensively in Part 2: Datatypes [XMLSD01], which is heavily influenced by the international standard for Language-independent datatypes [ISO11404]. In the Schema recommendation, a large set of built-in datatypes is defined by denoting their range of values and their lexical representation. Furthermore, methods to derive user-defined datatypes from the built-in ones are specified.

The `val` element fulfills the function of lexically representing values in XGILF. The back-end will convert this representation into binary data for the built-in types. Notice that the front-end language has to decide to what type the value should be converted, the value's type is not directly encoded in the `val` element. Only typed entities can have a value child node, thus the back-end always knows to which type the value should be converted.

"../xgf/xgilf.dtd" 74 ≡
```
<!ELEMENT %val; (%val;)* >
<!ATTLIST %val;
          val       CDATA     #IMPLIED
          kind      (bool|dec|hex|real|string|[]|record|union) #REQUIRED
          %Count;
          %IdReference; <!-- d.e -->
          %SourcePos; %SourceName; >
```

File defined by parts 50, 51, 52a, 55b, 56, 57a, 57b, 62a, 69, 72a, 74, 76a, 76b, 77a, 77b, 77c, 77d.

The `val` attribute contains the lexical representation of the value. It can make full use of Unicode characters, as XML is based on the Unicode Standard [Uni00]. The `kind` attribute tells the back-end how the `val` attribute should be interpreted. For example, the integer 255 can either be represented as `255` of decimal kind (`dec`), or as `FF` of hexadecimal kind (`hex`). Table 4.2 lists all available kinds for XGILF value elements and how they should be interpreted by the back-end.

For the representation of nested values of arrays, records, and unions, the `Count` attribute has to be present. It simply states the number of `val` subnodes. Arrays start filling the data structure at index zero and repeat the given value sequence if `Count` is smaller than the array's size. Record and union elements are specified with the identifier reference attribute, unreferenced elements are left uninitialized. Some examples illustrate the `val` element:

```
<val val="true" kind="bool"/>
<val val="-4711" kind="dec"/>
<val val="FEFF" kind="hex"/>
<val val="Hello, World." kind="string"/>
<val val="a &lt; b" kind="string"/>
<val kind="[]" count="2">
  <val val="1" kind="dec"/><val val="2" kind="dec"/>
</val>
<val kind="record">
```

Kind	Interpretation
`bool`	Boolean value, represented as either `true` or `false`.
`dec`	Numerical value, represented in decimal notation. Composed of a sign symbol which is either `+` or `-`, digits from 0 to 9 and a decimal delimiter symbol.
`real`	Decimal fraction, which allows the common scientific notation, consisting of mantissa and exponent. The exponent is separated from the mantissa by the symbol `E` or `e`.
`hex`	Binary data, represented as hexadecimal value of the bit pattern.
`string`	String value, represented as sequence of Unicode characters. Notice that the front-end has to replace the quotation marks used to delimit the attribute's value with the appropriate entity declarations.
`[]`	Array, which recursively contains the values of the array elements.
`record`	Record, which recursively contains the values of the record elements.
`union`	Union, which recursively contains the values of the union elements.

Table 4.2: Available value kinds and their interpretation.

```
    <val val="B&ouml;blingen" kind="string" ref="d_loc.e_town"/>
    <val val="Germany" kind="string" ref="d_loc.e_country"/>
    <val val="71034" kind="dec" ref="d_loc.e_zipcode"/>
</val>
```

One question remains that is directly related to the generality of generic programming. It is common to heavily overload symbols in generic code, a prominent example is the symbol 1 standing for unity. It can represent the integer value 1, which can be converted to machine datatypes trivially, but it can also represent the unity matrix, which requires special code for conversion into the used machine representation of matrices.

The particular problem we have to deal with is how to handle lexical representations of constants whose type is given by a type variable. The interpretation of the literal is possible only after instantiation of the algorithm in which it occurs, because the type variable remains unbound up to this stage. If the type variable resolves to a built-in data structure, the back-end is responsible for producing the binary representation of the literal. But how does one generate the binary representation of values given as literals for user-defined types? Generic programming languages can take two approaches in dealing with literals.

1. The literal representation of a value in the source code implicitly determines the value's type. This reduces the types of literals to types intrinsic to the front-end language, the built-in datatypes. One example in this category is C++. It emulates generic constants by providing possibilities to convert built-in datatypes to user-defined datatypes with overloaded constructors and assignment operators. These functions perform the work of creating a binary representation of the value for the user-defined datatype, starting from values of the built-in datatypes.

2. Constant values are treated as pure nullary functions. Here, constants are represented by identifiers that were introduced with function declarations. The function returns always the same value, the constant which it represents. The generation of the binary representation of the value is performed inside of the function body,

as the whole knowledge of the value is already present. The problem with this approach is that value ranges are handled poorly, every symbol has to be enumerated.

Both ways of dealing with values are supported in GILF in its current form. If types of constants are bound to built-in data structures, the back-end synthesizes the correct binary representation for the given value. Notice that this policy is a little bit more general than in C++, where the literal's type is fixed by its representation, whereas a value node in GILF may represent the value of any built-in type to which it it is convertible. Of course, the behavior of C++ is also possible, one has to bind the constant's type directly to a built-in data structure, not to a type variable from the constant's context.

Nullary functions are handled in GILF by declaring a function that has one output parameter. This parameter specifies the constant's type as its return value, and the algorithm bound to this function generates the value's binary representation inside its parameter's memory. This directly supports the general notion of constant nullary functions.

Left for future work, an integration of the XGILF value representation with XML Schema datatypes is highly desirable. Furthermore, XGILF itself should be expressed as XML Schema. But as mentioned at the beginning of this chapter, we sticked to the compact DTD form in this document for conciseness reasons.

4.11 Designators

Designators are the last elements of the XGILF specification. They are used as kind of *typed references*. In XGILF elements (GILF nodes), the necessity often arises to reference other elements in the XGILF document. In simple cases where no further information has to be present, this can be accomplished by using the identifier reference attribute described in the section on general attributes (see section 4.4). For the following scenarios, this is not feasible:

- One has to reference multiple elements. In XML, an attribute name has to be unique. Here, a designator sequence solves the problem, as elements can be repeated an arbitrary number of times.

- Another important aspect of designators is the ability to have their own subelements, which allows them to describe the reference in more detail, even recursively. For example, algorithm designators contain elements for describing the required instances of their required functions. This can lead to recursive calling chains.

Furthermore, different kinds of designators exist, thus making references in XGILF documents more robust and reliable by exposing more information about the designator's purpose.

The structure of designator elements is always the same. A designator has one identifier reference attribute which points to the referenced XGILF element. The designator's subelements, if any, further describe the reference as follows.

Unit designators have no subelements, they are used to list the dependent units in `import` elements. A unit designator must reference a unit's identifier.

"../xgf/xgilf.dtd" 76a ≡
```
<!ELEMENT %unit-dsg; EMPTY >
<!ATTLIST %unit-dsg; %IdReference; <!-- u --> >
```

File defined by parts 50, 51, 52a, 55b, 56, 57a, 57b, 62a, 69, 72a, 74, 76a, 76b, 77a, 77b, 77c, 77d.

Type and function designators serve the same purpose, they point to a type or function declaration, respectively. If no subelements are provided, all the instantiation parameters introduced by the referenced declaration are left unbound. Otherwise, the instantiation is fully or partially described, either directly with the `bind-static-params` element, or indirectly with binding designators.

"../xgf/xgilf.dtd" 76b ≡
```
<!ELEMENT %type-dsg; ((%bind-static-params;)?, (%binding-dsg;)*) >
<!ATTLIST %type-dsg; %IdReference; <!-- t --> >
<!ELEMENT %func-dsg; ((%bind-static-params;)?, (%binding-dsg;)*) >
<!ATTLIST %func-dsg; %IdReference; <!-- f --> >
```
File defined by parts 50, 51, 52a, 55b, 56, 57a, 57b, 62a, 69, 72a, 74, 76a, 76b, 77a, 77b, 77c, 77d.

Data structure and algorithm designators follow the same structure as type and function designators, but have slightly different semantics. They point to implementations of types or functions, respectively. They can also have binding subelements that describe a full or partial instantiation. Notice that these bindings reference instantiation parameters from the type or function declaration, which is available only by following the identifier reference of the referenced implementation element. This is always possible as data structure and algorithm elements have to reference a declaration.

"../xgf/xgilf.dtd" 77a ≡
```
<!ELEMENT %data-dsg; ((%bind-static-params;)?, (%binding-dsg;)*) >
<!ATTLIST %data-dsg; %IdReference; <!-- d --> >
<!ELEMENT %algo-dsg; ((%bind-static-params;)?, (%binding-dsg;)*) >
<!ATTLIST %algo-dsg; %IdReference; <!-- a --> >
```
File defined by parts 50, 51, 52a, 55b, 56, 57a, 57b, 62a, 69, 72a, 74, 76a, 76b, 77a, 77b, 77c, 77d.

Static instantiation parameter designators reference either instantiation parameters introduced in the declaration of a function or type, or the instantiation parameters introduced as required functions of an algorithm definition.

"../xgf/xgilf.dtd" 77b ≡
```
<!ELEMENT %static-param-dsg; EMPTY >
<!ATTLIST %static-param-dsg; %IdReference; <!-- tp|fp|cp|r --> >
```
File defined by parts 50, 51, 52a, 55b, 56, 57a, 57b, 62a, 69, 72a, 74, 76a, 76b, 77a, 77b, 77c, 77d.

Variable and constant designators are typically used as expressions, variable designators can also designate the target of an assignment. The type of a variable or constant is completely determined by its context, which manifests in its binding or static instantiation parameter designators. Therefore, variable and constant designators accept no subelements.

"../xgf/xgilf.dtd" 77c ≡
```
<!ELEMENT %var-dsg; EMPTY >
<!ATTLIST %var-dsg; %IdReference; <!-- v --> >
<!ELEMENT %const-dsg; EMPTY >
<!ATTLIST %const-dsg; %IdReference; <!-- c --> >
```
File defined by parts 50, 51, 52a, 55b, 56, 57a, 57b, 62a, 69, 72a, 74, 76a, 76b, 77a, 77b, 77c, 77d.

Finally, binding designators are declared. Their sole purpose is to reference binding elements from the binding section, thus help assemble the complete binding information required for instance descriptions or function calls. They must not have subelements.

"../xgf/xgilf.dtd" 77d ≡
```
<!ELEMENT %binding-dsg; EMPTY >
<!ATTLIST %binding-dsg; %IdReference; <!-- bt|bf|bsp|bvp --> >
```
File defined by parts 50, 51, 52a, 55b, 56, 57a, 57b, 62a, 69, 72a, 74, 76a, 76b, 77a, 77b, 77c, 77d.

4.12 Summary

In this chapter we provided a thorough description of XGILF, the XML-based external GILF representation. It started with a concise introduction to basic XML facts and its key technologies. Then, an annotated DTD was presented that detailed all elements of an XGILF document. Decisions that require more explanations were motivated by examining other common approaches.

An XGILF document consists of compilation units that are made up of six major sections, the import, declaration, definition, binding, storage, and extension sections. Figure 4.2 presents an overview of this structure, detailing the most important elements of an XGILF document.

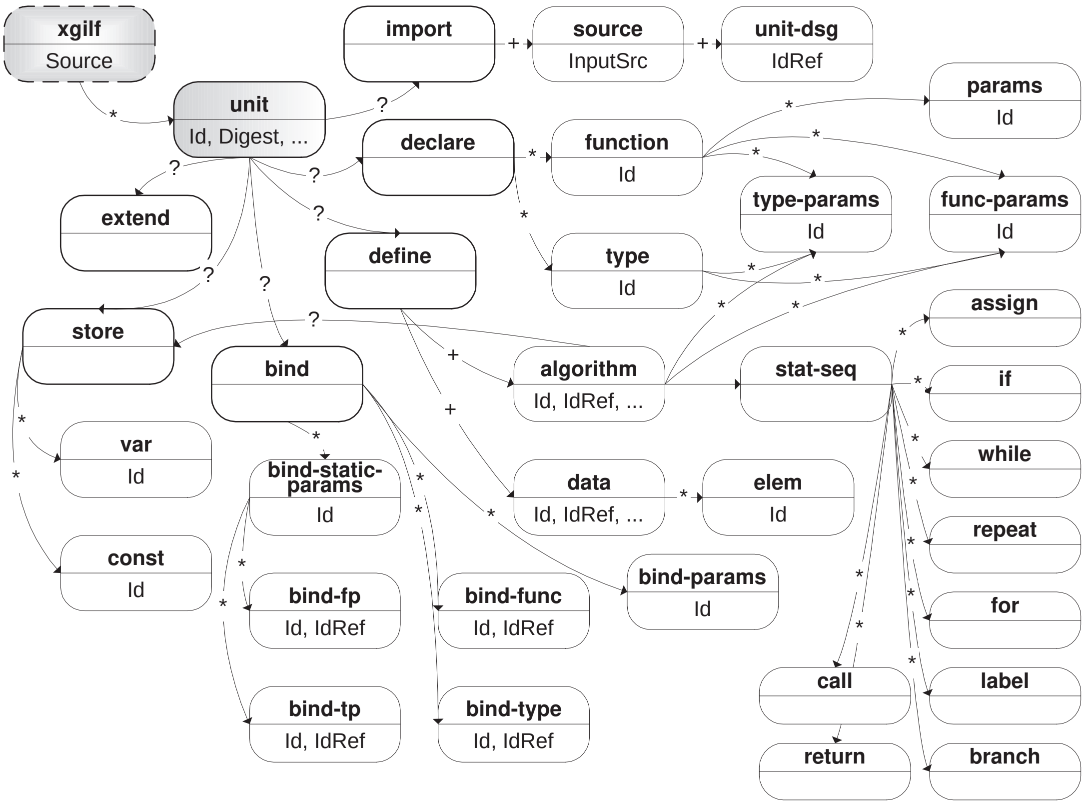

Figure 4.2: Hierarchy of the major XGILF elements.

Finally, designators were explained, which are used in most XGILF elements for referencing other elements in cases where simple identifier reference attributes do not suffice.

Chapter 5

The GILF Prototype

In this chapter we will describe the implementation of the GILF prototype. The implementation language is C++, as we want to exploit the benefits of generic programming for our project. Programming in C++ has undergone major changes in recent years. The emphasize shifted from object oriented programming to writing generic code, and combining these two paradigms.

Genericity allows writing flexible components that are easy to combine without sacrificing efficiency. Techniques like traits, policies, and of course the principles behind the Standard Template Library (see section 6.1.2), are becoming more widely used and accepted.

5.1 Modern C++ Programming

Before discussing the GILF system prototype's implementation, we present a short introduction to important modern C++ programming techniques and idioms. Furthermore, important external libraries are introduced briefly.

5.1.1 Traits

A recurring problem in generic C++ code is that a generic component does not only require bindings for its template parameters, but is also needs further characteristic information of the actual arguments. This information should be available through a uniform interface such that the component making use of it can be written in a generic style. Traits classes are designed for this purpose, they provide a consistent interface to a type's properties through constant type and function member definitions. For example, the C++ Standard Library contains the class `numeric_limits` which is specialized for all built-in types and describes their arithmetic properties. Among others, it has these members:

```
static const bool is_signed = false;
static T epsilon() throw();
```

The member `is_signed` stores true for a type that has a signed representation (which is the case for all predefined floating-point and signed integer types), and function `epsilon()` returns the difference between 1 and the smallest value greater than 1 that is representable for the type. The STL iterator traits are another prominent example, they specify an iterator's category, the value, reference, and pointer type of the elements an iterator refers to, and the difference type of iterator positions ([ISO14882], 24.3.1).

What we achieve with traits classes is exposing properties dependent on the actual type parameter of the traits class. C++ supports this with explicit template specialization. If an explicit specialization of the traits class for a given type exists, the common interface refers to this specialization, otherwise the most general one is taken. Overall, traits introduce a level of indirection for type dependent characteristics. A detailed overview on traits is presented in [Ale00].

5.1.2 Policies

A technique closely related to traits is policy-based programming [Ale01]. However, where traits concentrate on static, type dependent properties, policies try to capture single behavioral aspects of complex entities. An entity could be a smart pointer class, and policies of this entity are the storage policy, which abstracts the structure of the smart pointer, or the ownership policy, which handles how a smart pointer tracks the lifetime of the pointee and eventually deletes it.

The target of policy based programming is to construct a combinatorial set of behaviors for an entity by mixing a small set of core policy classes. Because policy parameters in C++ are resolved at compile time, this flexibility does not come at the expense of increased runtime, like dynamic dispatching in object oriented languages.

Like a traits class, a policy defines a common interface. Any class that respects this interface can be used as policy of the associated entity. Unlike traits, a policy is concerned with behavioral aspects and often carries state.

5.1.3 Template Metaprogramming

Template metaprogramming (TMP) has received some attention in the C++ community recently. The C++ template mechanism can be used to perform operations on types and values of built-in types that take place at compile time. The key idea behind TMP is that nesting a template instantiation inside its own template definition introduces looping by recursion. The base cases of the recursion are given by explicit or partial specializations of the class template, they are often called the if-conditions of TMP. A simple example should clarify the approach which computes the factorial of an integer at compile time:

```
template<int n>   // general case
struct Factorial
{
  enum { value = n * Factorial<n-1>::value };
};

template<>        // recursion base
struct Factorial<0>
{
  enum { value = 1 };
};

static int i = Factorial<7>::value;
```

Return values in TMP are all accessible members of the class template, in the example the enumeration `value`. Relevant examples of TMP are typelists [Ale01], tuples [Jär01], and the call and type traits in Boost.

5.1.4 Boost

The Boost C++ Libraries are a collection of open source libraries that aim at augmenting the C++ Standard Library [Boost]. Boost includes small utility libraries, as well as large, domain specific libraries, like the Boost Graph Library [SiLeLu01]. What sets the Boost effort apart from other open source libraries is that before inclusion into the collection a library has to pass a peer review process where all members scrutinize its design and implementation. This ensures a high quality standard of the Boost library collection. The following libraries play an important role in our implementation:

`shared_ptr`: This class is a nonintrusive smart pointer implementation that features reference counting semantics.

`tuple`: This facility allows storing fixed-size sequences of heterogeneous elements that can be accessed by integer indexes [Jär01].

`any`: The `any` class is designed to hold values of different types but does not attempt conversion between them. The type of the contained value is fixed at object creation time and cannot be altered afterwards.

`tokenizer`: The Boost tokenizer breaks character sequences into tokens and provides an iterator interface to the traversal of the resulting token stream.

5.2 General Structure

The major parts of the GILF system implementation are distributed in two libraries, the utility library `libutility` and the GILF library `libgilf`. The utility library contains code that is of general applicability in contexts other than GILF, whereas the GILF library contains all components directly related to the application logic of our GILF prototype. Furthermore, the command line program `gilf2code` is responsible for reading in a GILF stream and transforming it into fully instantiated code. Figure 5.1 summarizes this structure. The following sections will discuss the GILF library, some additional details on it are available in appendix B. The utility library is presented in appendix A.

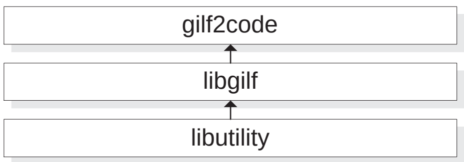

Figure 5.1: The three major building blocks of the GILF prototype implementation.

In chapter 3 the GILF system was introduced. Its core item, the code generating linker and loader, was also the main focus of our implementation. Therefore, the following sections will proceed roughly according to the flow chart presented in figure 3.3.

5.2.1 Coding Conventions

Namespaces All identifiers in our libraries are introduced in namespace `GILF_Core` or one of its sub-namespaces. Identifiers of entities that should not be used in client code of the libraries are introduced in namespaces `detail`. If not mentioned otherwise, all components described in this section and appendix A reside in the top-level namespace GILF_Core.

Identifier Naming Identifiers follow a consistent naming scheme which is influenced by the Boost and STL coding conventions, but also contains some idiosyncrasies. We list the conventions in a tabular form:

Entity	Example	Description
class	Node_Core	Words start uppercase, separated by underscore.
template parameter	NodeType	Words start uppercase, not separated.
member data	m_children	Words all lowercase, prefixed by 'm_', separated by underscore.
member type	sptr_type	Words all lowercase, separated by underscore.
method	get_children	Words all lowercase, separated by underscore.
free function	clone_node	Words all lowercase, separated by underscore.
global data	g_log	Words all lowercase, prefixed by 'g_', separated by underscore.

Table 5.1: Naming conventions for identifiers of C++ constructs in GILF libraries.

Separation of Declarations and Definitions For nongeneric code, we apply the typical separation of declarations into header files using the extension hpp, and definitions into implementation files using the extension cpp. However, even with up to date C++ compilers this is not feasible for generic code. In order to keep class declarations concise, we adopt the following discipline. Both declarations and definitions go into header files, but definitions are written outside of class template declarations. The same is true for member templates. Free function templates are written as unified declaration and definition.

5.3 Internal Representation

The internal representation is located at the center of figure 3.3, most processes in GILF generate it, either by deserialization of another format or by applying transformations on the internal representation.

The utility library provides two classes, Node and Node_Core, for creating a tree-like internal representation. A node's child handling abilities are factored out into Node_Core, which is the default for Node's template parameter NodeCore. Node holds the information present in a node. For this purpose, the class template Indexed_Properties from the utility library is used. Taken as is, these two classes provide static node classes, no overhead due to virtual function lookup is introduced.

5.3.1 Base Class GILF_Node

However, this basic setup is changed for nodes used in libgilf. It introduces a specialized version of the Node_Core called GILF_Node, from which all nodes in GILF's IR inherit. Thus, operations in libgilf are performed on subclasses of GILF_Node. These nodes exhibit changed behavior compared to plain nodes from libutility as follows.

⟨Listing 5.1: `GILF_Node` Declaration⟩ ≡
```
class GILF_Node : public Node_Core<>,
                  public Loki::BaseVisitable<>
```
Code extracted from file `gilf/Node.hpp`, lines 54 to 55.

Class `GILF_Node` derives from `Node_Core` in order to inherit its child handling abilities. In addition, it inherits from `BaseVisitable`, a base class from the Loki library (see section 6.1.2 and A.4) that is needed in order to make a node visitable. This is necessary because transformations on GILF's IR are performed using visitors. `BaseVisitable` has a virtual member function `Accept()`, consequently `GILF_Node`'s behave virtual. Anyhow, `GILF_Node` contains virtual member functions on its own. The class hierarchy of a `Node` present in the GILF prototype is summarized in figure 5.2, ordered by library membership which is given at the bottom.

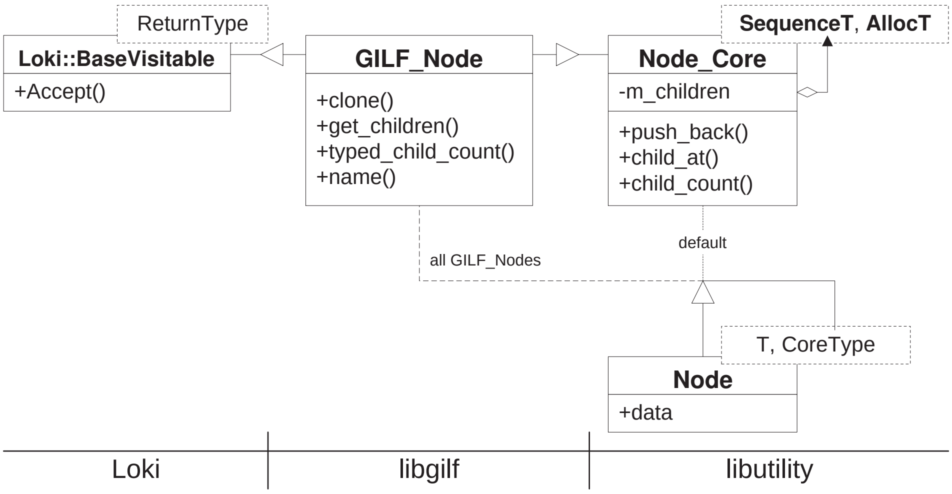

Figure 5.2: Node classes in GILF and their relationships.

It is important to realize that a `GILF_Node` stores only pointers to subclasses of itself in the member `m_children` that is part of `Node_Core`. This way, pointers in this member can always be cast to `GILF_Node` pointers, which enables virtual dispatch. This is reflected in the types exported by `GILF_Node`:

⟨Listing 5.2: `GILF_Node`: Types⟩ ≡
```
public: // Types.
  // Redefine sptr_type to more specialized shared pointer.
  typedef boost::shared_ptr<GILF_Node> sptr_type;
  typedef std::deque<sptr_type> sequence_type;
  typedef sequence_type::size_type size_type;
```
Code extracted from file `gilf/Node.hpp`, lines 60 to 64.

Finally, we take a look at the new member methods introduced by `GILF_Node`.

⟨Listing 5.3: `GILF_Node`: Methods⟩ ≡
```
public: // Methods.
  // Create a new node and clone the current node's properties.
  // Returns a pointer to the new node.
  virtual sptr_type clone() const { return sptr_type(); }
  // Get all child nodes of a given type.
  template <class ChildType>
  void get_children(sequence_type& nodes);
  // Count child nodes of a given type.
```

```
    template <class ChildType>
    size_type typed_child_count();
    // Return a string that gives GILF_Nodes a name.
    virtual std::string name();
    // Enable Loki Visitors.
    DEFINE_VISITABLE()
```
Code extracted from file `gilf/Node.hpp`, lines 68 to 81.

The virtual method `clone()` allows virtual cloning of a GILF node, this means all its properties are retained. This is an elementary C++ idiom, because constructors cannot act virtual (see [ClLoGi99], 21.07). Notice that we pass around shared pointers of cloned nodes which facilitates memory management. We provide an auxiliary free template function `clone_node` that makes writing overrides of the clone method trivial. It has one template parameter that should be a subclass of `GILF_Node`. A shared pointer holding a node of this type is created, and the data member of the function argument `node` is copied. An override of method `clone` is reduced to simply calling this function template.

⟨Listing 5.4: Function Template `clone_node`⟩ ≡
```
template <class NodeType>
GILF_Node::sptr_type clone_node(const NodeType& node)
{
  // Create shared pointer with full type information.
  boost::shared_ptr<NodeType> new_node(new NodeType);
  // Clone the properties.
  new_node->data = node.data;
  // Return cloned node. This implicitly performs an upcast to GILF_Node.
  return new_node;
} // clone_node
```
Code extracted from file `gilf/Node.hpp`, lines 149 to 158.

Method `get_children` retrieves a node's direct children of a given type. The type is the method's template parameter `NodeType`, and the matching child nodes are appended to the sequence container `nodes`, the function's parameter. The implementation of this member template allows some important insights on the interaction of `Node_Core` and `GILF_Node`. The scaffolding of this method is a loop over all child nodes.

⟨Listing 5.5: `GILF_Node`: Member Template `get_children`⟩ ≡
```
template <class ChildType>
void GILF_Node::get_children(sequence_type& nodes)
{
  // Look for subnodes.
  for (Node_Core<>::size_type n = 0; n < child_count(); ++n)
  {
    <see Listing 5.6 on page 84>
  } // for
} // get_children
```
Code extracted from file `gilf/Node.hpp`, lines 93 to 112.

More interesting is the loop's body. We need the exact dynamic type of each child such that we can compare it against the template parameter's type. Unfortunately, this is not possible with the pointers stored in `m_children`, which are of static type `Node_Core<>`. Therefore, we cast each node to a `GILF_Node`, which is legal because all nodes in the GILF IR are subtypes of `GILF_Node`. The cast is performed with a template function provided by Boost's shared pointer. The new pointer is then queried for its exact dynamic type, which is now possible as subclasses of `GILF_Node` act virtually. The general rule in code using `libgilf` is to manipulate nodes only through shared pointers of internal type `GILF_Node`.

⟨Listing 5.6: Member Template get_children: Loop Body⟩ ≡
```
// Cast the Node_Core<> to a GILF_Node.
sptr_type gnode =
  boost::shared_static_cast<GILF_Node, Node_Core<> >(child_at(n));
// Now check the runtime type of the contained pointer.
if (typeid(*gnode.get()) == typeid(ChildType))
{
  g_log(lc_std, lt_lowest) << "Found subnode of type "
                           << typeid(ChildType).name() << ".\n";
  nodes.push_back(gnode);
}
```
Code extracted from file gilf/Node.hpp, lines 100 to 109, referenced in listing 5.5.

Method `typed_child_count` is similar to method `get_children`, but it just counts a node's direct child nodes of a given type, instead of appending them to a sequence container.

The macro `DEFINE_VISITABLE()` has to appear inside a class declaration such that the class is visitable by a subclass of Loki's `Visitor` (see [Ale01], chapter 10).

5.3.2 Defining a Subclass of GILF_Node

After the detailed look at class `GILF_Node` we will now explain how to define a specialized GILF node. We will do this by picking the `algorithm` node from the XGILF specification in chapter 4 and define the corresponding GILF node. This task can be split up in three steps:

1. Provide type definitions for the node's property groups[1].
2. Provide a type definition that selects the node's property groups.
3. Declare the node's class and provide overrides for virtual methods.

Relying on the facilities introduced in section A.1, step 2 is achieved for an algorithm node with this type definition of the `Indexed_Properties` class template:

⟨Listing 5.7: Algorithm Node: Defining its Properties⟩ ≡
```
typedef Indexed_Properties< id_indexed_property, idref_indexed_property,
                            builtin_indexed_property,
                            debug_indexed_property,
                            access_mod_indexed_property >
algorithm_properties;
```
Code extracted from file gilf/nodes/Algorithm.hpp, lines 50 to 54.

The types used as template arguments will be discussed later, they result from accomplishing step 1. Now we show how to declare class `Algorithm`. This is done by inheriting from class `Node` from the utility library, using type definition `algorithm_properties` and class `GILF_Node` as template arguments.

⟨Listing 5.8: Algorithm Node: Class Declaration⟩ ≡
```
class Algorithm : public Node<algorithm_properties, GILF_Node>
```
Code extracted from file gilf/nodes/Algorithm.hpp, line 66.

We still have to override the virtual methods introduced in `GILF_Node`, most importantly method `clone`. As we noted earlier, this work is delegated to the function template `clone_node` (see listing 5.4). Again, Loki's macro `DEFINE_VISITABLE()` has to be present such that the node can be visited.

[1] A property group can consist of a single property.

⟨Listing 5.9: Algorithm Node: Virtual Methods⟩ ≡
```
public: // Virtual methods.
  // Create a new node and clone the current node's properties.
  // Returns a shared pointer to the new node.
  virtual GILF_Node::sptr_type clone() const { return clone_node(*this); }
  // Enable Loki Visitors.
  DEFINE_VISITABLE()
  // Return node's name.
  virtual std::string name() { return std::string("Algorithm"); }
```
Code extracted from file `gilf/nodes/Algorithm.hpp`, lines 71 to 78.

We see that performing steps 2 and 3 is straightforward and displays the node's characteristics in a clean way. What is left are guidelines to define a property group (step 1). For this purpose, we look at debug properties, which are specified for the algorithm's properties as `debug_indexed_property`. Most property groups are collected in the header file `Common_Properties.hpp` and are reused by nodes that share these property groups.

⟨Listing 5.10: Indexed Property: Debug Property Group⟩ ≡
```
enum Debug_Prop_Index
{
  debug_line_iprop = 0, debug_column_iprop,
  debug_name_iprop, debug_source_iprop
};

typedef long debug_line_iprop_type;
typedef long debug_column_iprop_type;
typedef std::string debug_name_iprop_type;
typedef std::string debug_source_iprop_type;

typedef boost::tuples::tuple<
  debug_line_iprop_type, debug_column_iprop_type,
  debug_name_iprop_type, debug_source_iprop_type >
debug_prop_tuple;

typedef GILF_Core::Indexed_Property<Debug_Prop_Index, debug_prop_tuple>
debug_indexed_property;
```
Code extracted from file `gilf/Common_Properties.hpp`, lines 76 to 93.

First, an enumeration has to be defined that that contains index values for all properties collected in this property group. Because Boost tuples are zero-based, the first index is explicitly set to 0. Then, the type of every property is exported as type definition. This is not strictly necessary, but enhances understandability of the code. Using this type definitions, a tuple type definition is introduced that will hold all properties with full type information. Finally, the index enumeration and the tuple type are bound to `Indexed_Property` as template arguments. This final type definition can now be shared by any node that uses the instantiations of class template `Indexed_Properties` as data member.

What we have achieved with this composition of type definitions, class templates and virtual inheritance is an efficient, type safe method for building internal representations based on nodes that can define, share, and recombine property groups. Access to a property consists of a single method call with the property's index as template argument.

5.3.3 The Factory for `GILF_Nodes`

An important aspect of an polymorphic, hierarchical data structure is its ability to be reconstructed from an serialized, external representation. Object factories are a common approach to handling the occurring problem of generating objects from type information present in a form that is not accessible to the C++ type system[2]. We employ a slightly modified version of Alexandrescu's generic factory template (see appendix A.4.1). A global node factory is defined that produces specific GILF nodes based on a type identifier encoded in a string.

⟨Listing 5.11: GILF Node Factory⟩ ≡
```
typedef Factory<GILF_Node::sptr_type, node_id_type, create_node_fptr>
node_factory_type;
// Declaration of global node factory.
extern node_factory_type *g_node_factory;
```
Code extracted from file `gilf/Node_Factory.hpp`, lines 43 to 46.

The initialization of the pointer `g_node_factory` prior to first usage is enforced with a designated initialization object placed in the unnamed namespace. This general approach for initializing static variables in C++ libraries is explained in [Sch89], but the utility library contains a policy based class template `Initializer` for aiding the programmer in this task.

Nodes that participate in the GILF system should be created using the global node factory. One can see how the factory is put to use in section B.1 that elaborates on the deserialization framework.

5.4 General Facilities

5.4.1 Logging

The utility library contains a generic logging class (see section A.2), and the GILF library contains an instantiation of this class template `Log` with four logging categories. A type definition is provided for convenience. The global GILF logging object is called `g_log`.

⟨Listing 5.12: GILF Logger: Type and Declaration⟩ ≡
```
typedef Log<4> logger_type;
extern logger_type g_log;
```
Code extracted from file `gilf/Logger.hpp`, lines 39 to 40.

The four logging categories should be accessed through named constants only. The categories are for logging standard messages, reporting warning and error events, as well as debugging output.

⟨Listing 5.13: GILF Logger: Categories⟩ ≡
```
const logger_type::category_count_type lc_std = 0;
const logger_type::category_count_type lc_warning = 1;
const logger_type::category_count_type lc_error = 2;
const logger_type::category_count_type lc_debug = 3;
```
Code extracted from file `gilf/Logger.hpp`, lines 49 to 52.

Accordingly, five constants are defined that should be used when specifying the priority of the logging output. The gaps between the thresholds constants allow some fine tuning of the priorities.

[2]For more details on the topic, refer to [Ale01], chapter 8.

⟨Listing 5.14: GILF Logger: Thresholds⟩ ≡
```
const logger_type::threshold_type lt_highest = 0;
const logger_type::threshold_type lt_high = 10;
const logger_type::threshold_type lt_medium = 20;
const logger_type::threshold_type lt_low = 30;
const logger_type::threshold_type lt_lowest = 40;
```
Code extracted from file `gilf/Logger.hpp`, lines 64 to 68.

5.4.2 Symbol Table

The GILF library also contains a customizable symbol table. A typical symbol table implementation uses an underlying hash table such that the amortized complexity for getting and putting elements into the symbol table are constant time operations [Sed98]. A space efficient alternative to hash tables for implementing string symbol tables based on tries is presented in [BeSe97]. We have chosen a hash table based implementation, because most implementations of the C++ Standard Library provide a `hash_map` container[3].

Keys into the symbol table are GILF identifiers (see section 4.4), which are standard C++ strings in the internal representation. A type definition `id_iprop_type` is available:

⟨Listing 5.15: Internal Identifier Type⟩ ≡
```
typedef std::string id_iprop_type;
```
Code extracted from file `gilf/Common_Properties.hpp`, line 108.

The value associated with an identifier key in a symbol table consists of two parts and is therefore wrapped inside a C++ `pair` container. The first element is a shared pointer to a GILF node, and the second element can contain arbitrary data describing the node. The type of the second element is specified by the symbol table's template parameter `PropertyType`. This can be used to adapt different symbol tables to store only properties required in the current context.

⟨Listing 5.16: Symbol Table: Class Declaration⟩ ≡
```
template <typename PropertyType>
class Symbol_Table
```
Code extracted from file `gilf/Symbol_Table.hpp`, lines 73 to 74.

The types for key and values are exported through type definitions.

⟨Listing 5.17: Symbol Table: Type Definitions⟩ ≡
```
public: // Types.
  // The key into the symbol table is the symbol's identifier.
  typedef id_iprop::type key_type;
  // Each key is associated with a GILF node and additional properties.
  typedef GILF_Node::sptr_type node_type;
  typedef PropertyType prop_type;
  typedef std::pair<node_type, prop_type> value_type;
```
Code extracted from file `gilf/Symbol_Table.hpp`, lines 79 to 85.

Based on these type definitions, the type of the STL `hash_map` that holds our symbol table can be fixed.

⟨Listing 5.18: Symbol Table: Map Type⟩ ≡
```
typedef HASH_MAP_NAMESPACE::hash_map<const key_type, value_type, Hash_Id> map_type;
```
Code extracted from file `gilf/Symbol_Table.hpp`, line 102.

[3]The Dinkum library contains hash tables, as well as all STL implementations based on the SGI STL, like STLport or Comeau's C++ library.

The function object `Hash_Id` implements the hashing function of the hash table. It does not depend on the class template's template parameter, therefore we have put it into an unnamed namespace. Thus, it is available only to our symbol table class, and does not have to be created for every symbol table instantiation. It is important to remember that hashing functions have to be stateless, i.e. they compute the same result every time when invoked with the same key value. We implemented the string hashing function `djb2`, which is known for its excellent distribution and speed on many different sets of keys and table sizes. It is attributed to Daniel J. Bernstein.

⟨Listing 5.19: Symbol Table: Hashing Function⟩ ≡

```
template <typename PropertyType>
std::size_t
Symbol_Table<PropertyType>::Hash_Id::operator()(const key_type& key) const
{
  // Hash function for strings (djb2 by D.J. Bernstein).
  std::size_t hash = 5381;
  for (std::string::size_type sz = 0; sz < key.size(); sz++)
  {
    hash = ((hash << 5) + hash) + key[sz];
  }
  return hash;
}
```

Code extracted from file `gilf/Symbol_Table.hpp`, lines 139 to 150.

What is left to discuss are the public methods to update and query a symbol table. The `add` and `remove` methods do what their names suggest. They return `true` if the operation finished successful, otherwise they return `false`, for example if an entry with the identifier passed to `add` is already present. Method `is_member` checks if an entry with a given identifier is already present in the symbol table, and `get` also returns the entry's value.

⟨Listing 5.20: Symbol Table: Methods⟩ ≡

```
public: // Methods.
  // Add an entry to the symbol table.
  bool add(const key_type& id, node_type& node);
  // Remove an entry to the symbol table.
  bool remove(const key_type& id);
  // Check if a entry with id exists in symbol table.
  bool is_member(const key_type& id) const;
  // Return a symbol table entry.
  node_type get(const key_type& id) const;
```

Code extracted from file `gilf/Symbol_Table.hpp`, lines 109 to 117.

The GILF library contains two global symbol tables. The first one, `g_symbol_table`, contains all nonlocal symbols that are available in the currently loaded GILF units, and `g_instance_table`, the instantiation table contains symbols of instantiated algorithms and data structures. Their management will be touched in the following sections.

⟨Listing 5.21: Global Symbol Tables⟩ ≡

```
// The global symbol table for symbols inside unchanged units.
// Will be filled during inflation.
typedef Symbol_Table<empty_property> symbol_table_type;
extern symbol_table_type g_symbol_table;
// The global symbol table for instantiated nodes. Will be filled during
// visitation with the Instantiator.
typedef Symbol_Table<empty_property> instance_table_type;
extern instance_table_type g_instance_table;
```

Code extracted from file `gilf/Globals.hpp`, lines 48 to 55.

5.5 Visiting GILF Nodes

Once the external GILF representation has been deserialized (sse section B.1), transformations are applied to the internal representation in order to finally arrive at nongeneric code. In the GILF prototype, we emit C++ code, but using only basic features, no template code is generated. For each distinct transformation, we employ an instantiation of the generic visitor class template ([Ale01], chapter 10). We exemplify the functioning of the visitor pattern by looking at class `Importer`, which allows automatically tracking unit dependencies denoted by GILF's `import` nodes. It is a bit unusual for a transformation visitor, because it creates new units and does not actually transform existing units, but it suffices for our explanatory purposes.

The first step in implementing a visitor is planning ahead. A flow graph of the visited nodes that also contains the visitation order is the essential guideline for a visitor implementation. We call this graph a visitor's *visitation graph*. The following properties of the flow graph will need special treatment:

Cycles. Cycles in the visitation graph can be handled either by explicitly storing information about the visitation process or creating new visitors and calling them recursively.

Edges to non-child nodes. Edges to nodes that are not direct child nodes of the originating node require locating the target node. Often, look-up in a symbol table is the adequate means for this operation.

Multiple ingoing edges. If a node can be reached through different paths, it is important to detect the source node of this visitation. Special state has to be stored in the visitor object to achieve this.

Multiple outgoing edges. If a node has multiple outgoing edges, either the order of their traversal has to be fixed, or conditions have to be stated that allow selective traversal.

The visitation graph for class `Importer` is extremely simple. Only direct child nodes are visited without multiple edges. The only complication is the cycle introduced by recursively tracking unit dependencies of the inflated unit.

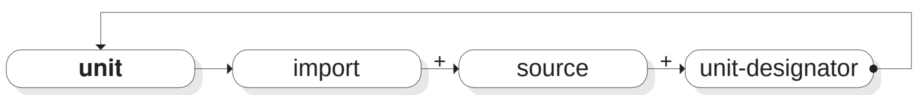

Figure 5.3: Visitation graph of visitor `Importer`.

We use the XGILF element tag names for naming the nodes. Arrows with no strong dot at the beginning are used for paths to direct subnodes, whereas a dot denotes jumps to a referenced node. An unannotated arrow means that exactly one target node should be visited, an arrow with '+' annotation targets one or more nodes, and an arrow with '*' targets zero, one or more nodes. A '?' denotes an optional path. A box at the arrow's beginning denotes a path with alternative target nodes, only one of them should be chosen. The node whose type is set in bold face marks the visitation root node.

A visitor has to inherit from all visitor instantiations with nodes that it wants to visit, in addition to the `BaseVisitor`. The nodes required for `Importer` can be extracted easily from the visitation graph in figure 5.3.

⟨Listing 5.22: Class `Importer`: Declaration⟩ ≡
```
class Importer : public Loki::BaseVisitor,
                 public Loki::Visitor<Unit>,
                 public Loki::Visitor<Import>,
                 public Loki::Visitor<Source>,
                 public Loki::Visitor<Unit_Designator>
```
Code extracted from file `gilf/Importer.hpp`, lines 49 to 53.

Next, a constructor usually sets up internal data specific to the visitor. For `Importer`, this means setting the unit store[4] in which the inflated unit will be stored. Optionally, recursive import dependency tracking can be disabled.

⟨Listing 5.23: Class `Importer`: Constructor⟩ ≡
```
Importer(Unit_Store* store, bool recurse = true) :
  m_store(store), m_recurse(recurse) {}
```
Code extracted from file `gilf/Importer.hpp`, lines 59 to 60.

One of the main benefits of a visitor is its ability to store state inside the object's data members which are accessible to all nodes during visitation. An importer has three members. The pointer to a unit store and the boolean flag that controls recursive dependency tracking are set during object construction. However, the pointer to the current input source `m_input` is reset each time a `source` node is reached, depending on its input source property.

⟨Listing 5.24: Class `Importer`: Data Members⟩ ≡
```
// Pointer to the unit store that will hold the inflated units.
Unit_Store* m_store;
// Input_Source object for the current source.
boost::scoped_ptr<Input_Source> m_input;
// Flag to indicate recursive importing of units.
bool m_recurse;
```
Code extracted from file `gilf/Importer.hpp`, lines 74 to 79.

A visitor's declaration, the available constructors, and its data members are the skeleton of every visitor implementation. Its semantics and behavior are expressed in the visitor's `Visit` methods. For every visited node type an overload of this virtual method is required. The `Visit` methods for unit and import nodes of `Importer` call `visit_children` on their child nodes (see section A.4.3), which results in invoking the corresponding `Visit` methods. In the `import` node's `Visit` method, the member `m_input` is reset. Finally, method `Visit` for unit designators inflates the denoted unit using our deserialization framework.

5.6 Instantiation Application

Once a unit's internal GILF representation has been created together with the units listed in its import dependencies, the instantiator commences operations, the central component of the GILF prototype. Deserialized GILF contains generic constructs (see chapter 4), which will be replaced by nongeneric instances during instantiation application. In this section we will discuss GILF's instantiation engine, which is realized as a visitor in class `Instantiator`, similar to `Importer`, the unit import dependency resolver.

In general, the instantiator is started by calling the virtual method `Accept` of a GILF node on an instantiator object. Instantiation starts at a function or type binding node,

[4] Class `Unit_Store` is an auxiliary class of the GILF library which allows convenient storage and retrieval of GILF unit nodes.

which describes an instantiation. The goal of an instantiation application is to resolve all static instantiation parameters of a generic algorithm or data structure and create a nongeneric, fully instantiated algorithm or data structure. This can require recursive instantiations. After the instantiator returns, the requested instance and all recursively created instances are placed into the global instance table (see section 5.4.2). Notice that the results of instantiation application are still GILF nodes. Code generation can proceed on these nongeneric nodes.

5.6.1 Approach

Before discussing implementation details, we present the general steps of instantiation application. The process is divided into three major phases.

Phase 1: Signature evaluation. Phase 1 is devoted to setting up internal data structures required for successful instantiation application. The algorithm's or data structure's signature is visited and for all instantiation parameters, the corresponding identifier is put as key into an associative container, the so-called *id mapping table*. For algorithms, these identifiers include the dependent functions.

It is important to have access to the set of generated instances, as well as to those instances currently under construction. Only with this information present one can avoid infinite instantiation loops.

Phase 2: Instantiation parameter binding. The goal of the second phase is binding every identifier in the created mapping table to an identifier of a nongeneric instance of an algorithm or data structure. Of course, this can require recursive instantiation applications. During recursive creation of an instantiator, the current instance set has to be retained such that the protection against infinite instantiations in phase 1 remains functional.

The node visitation in phase 2 follows the current node's static instantiation parameter bindings. After phase 2 has completed, all instantiation parameters residing in the mapping table have to be bound to instance identifiers, otherwise the front-end has produced illegal GILF code.

For nongeneric constructs, phase 2 is skipped. This is an important property that ensures termination of instantiation application.

Phase 3: Instance generation. Finally, in phase 3 the requested instance is generated and added to the global instance table. Instances of generic constructs require a mangled identifier name that denotes the generated algorithm or data structure, whereas instances of nongeneric constructs simply keep their original identifier. A good strategy for creating the instance's mangled name is to use the identifiers in the current construct's mapping table. One should also take care to generate the same instance only once.

Instance generation recursively visits the data structure or algorithm node and its subnodes designated in the current binding node. During this visitation, references to instantiation parameters are replaced by the corresponding values found in the mapping table, which was created in the phase 1 and 2, or by recursively created instances. All function, type, and instantiation parameter designators are replaced by their corresponding data structure or algorithm designators.

At this point, one can encounter unresolved function designators in the mapping table. These designators occur because of the infinite instantiation protection in phase 1. Nodes with such incomplete designator identifiers are put into a list and patched in a final loop over the whole list at the end of phase 3.

During instantiation application, the GILF prototype can make inquiries to the algorithm selection unit. This happens if load-time instantiation is configured and more than one algorithm designator is present in a function binding. However, run-time instantiation defers the requests to the selection unit until the execution of the generated program.

5.6.2 Visitation Graphs

The instantiator's implementation is best understood by first looking at its visitation graphs for type and function bindings. The visitation graph differs depending on two orthogonal characteristics of the depicted construct. First, if the construct is generic or nongeneric, and secondly, if the construct is built-in or not, determines the actual visitation graph. Because the differences manifest in additional, but independent paths and nodes, we present the full visitation graph for generic, user defined constructs. We will explain the parts that are not present in the other cases.

Type Binding First, we look at the visitation graph for type bindings, illustrated in figure 5.4.

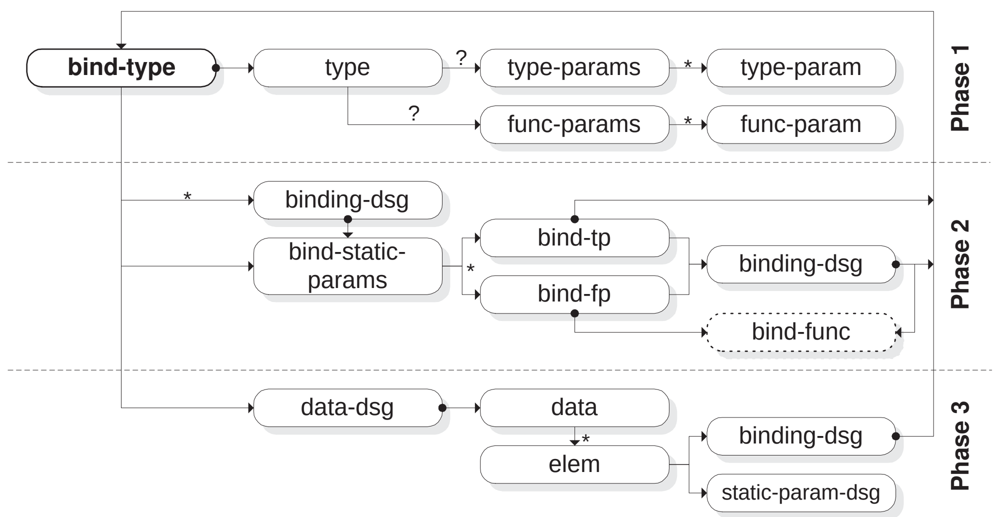

Figure 5.4: Visitation graph for types of visitor `Instantiator`.

In phase 1, we jump to the type node denoted by the identifier reference of the `bind-type` visitation root node. If we arrive at a nongeneric type, traversal stops here.

For generic types, we visit their `type-params` and `func-params` subnodes, and their instantiation parameter subnodes. From these, we extract the identifiers of the static instantiation parameters that have to be bound during instantiation application.

Phase 2 is dedicated to binding the instance identifiers that correspond to all the instantiation parameters found in the type's signature during phase 1[5]. Those bindings are

[5] As mentioned before, if we detected a nongeneric type in phase 1, the second phase is skipped altogether.

stored in `bind-static-params` nodes, which are either direct subnodes of the visitation root node, or reached indirectly through binding designators.

Now we follow the type and function parameter binding nodes one by one. Again, the instance to which they are bound is either given directly through a type or function binding, respectively, or indirectly by means of a binding designator pointing to such bindings. Anyway, a recursive invocation of an instantiator is initiated[6].

Finally, we proceed to phase 3. The type's representation is denoted by a data designator, that points to the corresponding `data` node. Built-in data structures stop here, their layout is implicitly determined. User defined data structures enumerate their layout with element subnodes, which will be visited by the instantiator for instance generation. An element's type is given either by one of the instantiation parameters which were bound to instance identifiers in phase 2, or by a binding designator pointing to a bind type node. This will lead to a recursive instantiator invocation.

Function Binding The visitation graph for function bindings (see figure 5.5) is more ramified than the one for type bindings which we examined above. We will concentrate on the differences and extensions present in the graph for function bindings.

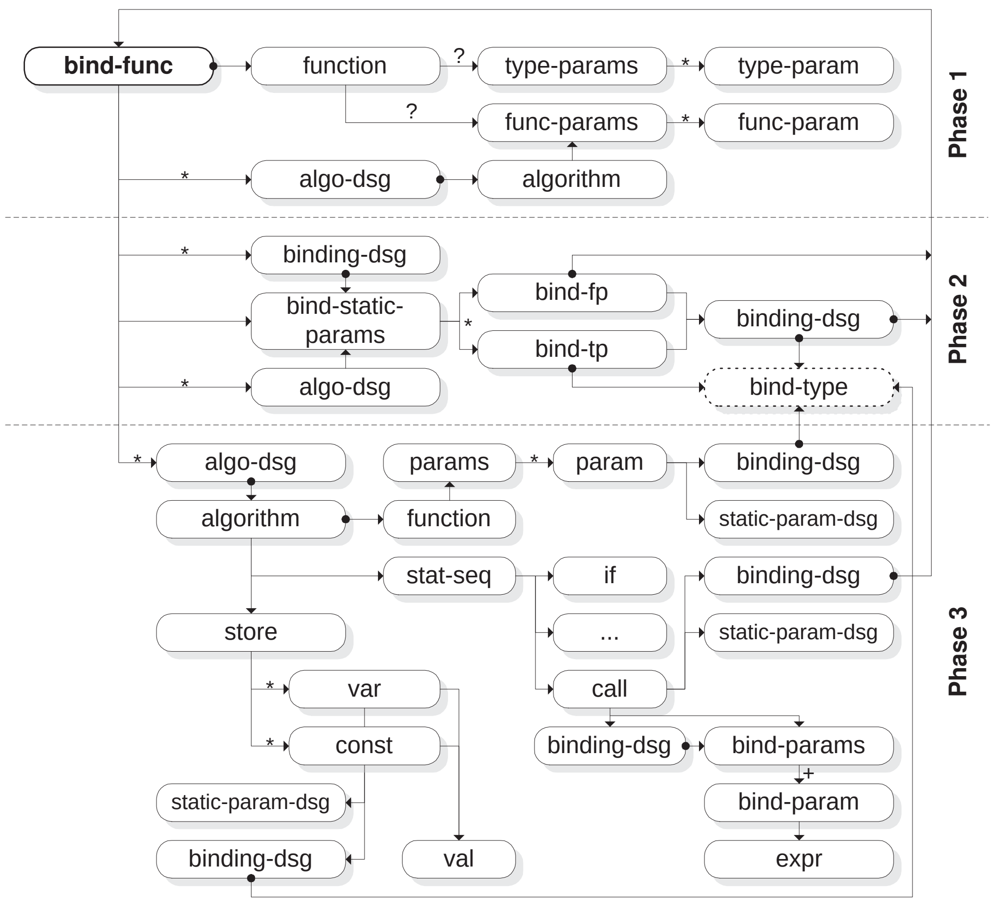

Figure 5.5: Visitation graph for functions of visitor `Instantiator`.

Phase 1 is almost identical to the one for type bindings, however the instantiator

[6]The dotted line around the bind function node indicates this recursive invocation, but the visitation graph for this path is presented separately in figure 5.5.

also has to visit function instantiation parameters present in the designated algorithms and add them to the mapping table. These parameters are the algorithm's overloaded function symbols depending on instantiation parameters that have to be resolved during overload resolution.

Analogously, in order to fill the mapping table in phase 2 one has to track bindings of instantiation parameters to instance identifiers also in the algorithm designators for the corresponding algorithms. These extensions in both phases are only relevant for generic algorithms, because a nongeneric function's instantiator stops visitation at the `function` node, and phase 2 is skipped.

The visitation graphs differ most in phase 3. Via the possibly multiple algorithm designators, the `algorithm` nodes are reached. Here, three major paths are taken. First, the algorithm's associated `function` node is visited in order to bind its dynamic value parameters to this instance's data structures. Interestingly, this step is technically almost identical to processing a data structure's elements. Built-in algorithms can stop here, because their behavior is implicitly defined and the back-end has to generate code for them. Next, the algorithm's body inside the `stat-seq` node is visited. This step is trivial for most nodes, because it simply consists of deeply copying the node's content recursively. The only exception is the `call` node. It has to be bound to the actual algorithm instance that this call is representing, which is either given by an instantiation parameter taken from the mapping table, or by a binding designator that leads to recursive invocation of the instantiator. Furthermore, the call's actual parameters have to be bound to expressions for parameter passing. Finally, an algorithm's local store of variables and constants has to be processed. This is done analogously to the function's parameters, which was described above as the first step of phase 3. However, variables and constants also have a second subnode of type `val` which represents the construct's value. This node is required for constants, and optional for variables.

5.6.3 Implementation

Class `Instantiator` is implemented as a visitor. We have already mentioned the typical problems associated with this strategy is section 5.5. The complex visitation graphs depicted in figures 5.4 and 5.5 indicate that care has to be taken in order to keep the instantiator's implementation tractable. `Instantiator` inherits from Loki's `BaseVisitor` and from `Visitor` instances of all the nodes it has to visit, i.e. all the nodes present in the visitation graphs. We only display the `Visitor` instances of the two root nodes in the declaration for brevity.

⟨Listing 5.25: Class `Instantiator`: Declaration⟩ ≡

```
class Instantiator : public Loki::BaseVisitor,
                     public Loki::Visitor<Bind_Type>,
                     public Loki::Visitor<Bind_Function>,
                     ...
```

Code extracted from file `gilf/Instantiator.hpp`, lines 73 to 107.

Type Definitions As usual, we define the central types for class `Instantiator` in the header file. First, the type of the mapping table `id_mapping_type` is defined, which maps type and function binding identifiers to their corresponding instance identifiers. Similarly, we need a set container type `instance_set_type` that is used for keeping identifiers of generated instances. Last not least, we provide a type for a sequence container holding smart pointers to GILF nodes.

⟨Listing 5.26: Class `Instantiator`: Public Types⟩ ≡
```
// Types that constitute id to id mappings.
typedef id_iprop::type id_key_type;
typedef id_iprop::type id_data_type;
typedef std::map<id_key_type, id_data_type> id_mapping_type;
// Set container type that holds ids of instantiations.
typedef std::set<id_iprop::type> instance_set_type;
// Sequence container type that holds GILF_Node sptrs.
typedef GILF_Node::sequence_type gnode_seq_type;
```
Code extracted from file `gilf/Instantiator.hpp`, lines 113 to 120.

Data Members The instantiator's current phase is stored in member `m_phase`. The values of this member are taken from the enumeration `Phase_Flag`, and each instantiator's constructor initializes `m_phase` to phase 1, which corresponds to the enumeration value `Phase_Signature`.

⟨Listing 5.27: Class `Instantiator`: Phase Enumeration Constants⟩ ≡
```
// Enumeration constants for the three phases of instance generation.
enum Phase_Flag
{
  Phase_Signature = 1, // Phase 1 (Signature Evaluation)
  Phase_Bind,          // Phase 2 (Instantiation Parameter Binding)
  Phase_Generate       // Phase 3 (Instance Generation)
};
Phase_Flag m_phase;
```
Code extracted from file `gilf/Instantiator.hpp`, lines 251 to 258.

The instantiator's complexity requires state of the instantiation process phases to be stored inside the class' data members. The mapping table mentioned during presentation of the visitation graphs is stored in `m_mapping_table`. Preventing infinite instantiation loops is of major importance for every instantiation engine. We avoid recursive reinstantiations of already instantiated constructs with member `m_instance_set` which holds all binding identifiers of the current and recursively called instantiators. Member `m_bind_ref` holds the identifier of the instantiation parameter currently bound to an instance identifier in phase 2. The identifier of the instance generated by this instantiator after successfully completing all phases is stored in member `m_instance_id`. In phase 3, all references to instantiations will be replaced by the actual instance id. However, because of the infinite recursion protection, we can find unresolved function binding identifiers in the mapping table created in phase 2. They are queued in member `m_unresolved_ids` and have to be patched at the end of phase 3. Member `m_father` is used in phase 3 to save the father node during instance generation such that visited child nodes can be appended.

⟨Listing 5.28: Class `Instantiator`: Data Members Related to Phases⟩ ≡
```
// Mapping of instantiation parameter ids to instance ids.
id_mapping_type m_mapping_table;
// The set contains all the recusively requested instantiations of this
// instantiation. Is used to prevent infinite instantiation loops.
instance_set_type m_instance_set;
// Reference id of instance parameter that is currently bound in Phase 2.
idref_iprop::type m_bind_ref;
// The id of the instance that was generated by the Instantiator.
id_iprop::type m_instance_id;
// Sequence of nodes that contains unresolved binding ids. These happen
// because of skipped instantiations to avoid infinite instantiations loops.
gnode_seq_type m_unresolved_ids;
// Holds a pointer to the current father node, thus allowing to append cloned
```

```
// child nodes in visitors. Either an algorithm or data node serves as the
// tree's root of the monomorphic representation of a generic construct.
GILF_Node::sptr_type m_father;
```

Code extracted from file gilf/Instantiator.hpp, lines 262 to 277.

Patching the unresolved identifiers requires bookkeeping of the mappings from function binding identifiers to the identifiers of the instance that resulted from instantiation application. These mappings are stored in data member `m_instance_mappings` which is static, because instantiators need access to all past mappings.

⟨Listing 5.29: Class `Instantiator`: Data Member for Instance Identifier Mappings⟩ ≡

```
static id_mapping_type m_instance_mappings;
```

Code extracted from file gilf/Instantiator.hpp, line 283.

Constructors We define two constructors for class `Instantiator`. The default constructor ensures that the phase is properly initialized, and the copy constructor also saves the state that is required for recursive instantiation application.

⟨Listing 5.30: Class `Instantiator`: Constructors⟩ ≡

```
Instantiator() : m_phase(Phase_Signature) {} // Default ctor.
Instantiator(const Instantiator& inst) :     // Copy ctor, retains state.
  m_phase(Phase_Signature),
  m_instance_set(inst.m_instance_set),
  m_unresolved_ids(inst.m_unresolved_ids) {}
```

Code extracted from file gilf/Instantiator.hpp, lines 125 to 129.

Methods Apart from the virtual `Visit` methods for all visited node types, `Instantiator` provides only one public method which allows retrieving the identifier of the generated monomorphic instance.

⟨Listing 5.31: Class `Instantiator`: Method for Retrieving the Instance Identifier⟩ ≡

```
id_iprop::type get_instance_id() { return m_instance_id; }
```

Code extracted from file gilf/Instantiator.hpp, line 176.

A common action during visitation is to jump to a node given by its reference identifier. We support this with the private method `visit_node_by_ref`. The first parameter is the identifier of the target node at which visitation should continue. This node is looked up in the global symbol table `g_symbol_table`, and a standard exception of type `runtime_error` is thrown of no such node is found. The second parameter is a pointer to an instantiator. If it is set to `0` which is the default value, the current instantiator is reused. Otherwise, visitation continues with the provided instantiator, which is needed for recursive instantiation applications. Method `visit_nodes_by_type` allows visiting direct subnodes restricted by their type, similar to the general facility presented in listing A.31.

⟨Listing 5.32: Class `Instantiator`: Method for Visiting a Node by Identifier Reference⟩ ≡

```
void visit_node_by_ref(const idref_iprop::type& ref,
                       Instantiator* instantiator = 0);
```

Code extracted from file gilf/Instantiator.hpp, lines 198 to 199.

Two methods aid in generating the monomorphic instance of a generic algorithm of data structure, the final outcome of instantiation application. The first one called `append_node` clones the node passed to it and appends it to the current father node stored in data member `m_father`. It also returns a smart pointer to the cloned node.

⟨Listing 5.33: Class `Instantiator`: Method for Appending a Node⟩ ≡

```
GILF_Node::sptr_type append_node(GILF_Node& node);
```

Code extracted from file gilf/Instantiator.hpp, line 220.

The second method in this category, deep_copy, performs a deep node copy starting at node, the sole parameter. The copied hierarchy is appended to the current father node. This method is a shortcut for visiting statement nodes and simply copying them manually inside Visit methods.

⟨Listing 5.34: Class Instantiator: Method for Deeply Copying a Node⟩ ≡

```
void deep_copy(GILF_Node& node);
```

Code extracted from file gilf/Instantiator.hpp, line 236.

During the discussion of visitation graph 5.5 we pointed out the call node as exception in the statement node. It requires special treatment, because it contains parts that have to be transformed. Therefore, method deep_copy checks for call nodes and escapes to regular visitation by calling the call node's Visit method.

⟨Listing 5.35: Class Instantiator: Escaping Method deep_copy to Continue Visitation⟩ ≡

```
if (typeid(node) == typeid(Call))
{
  node.Accept(*this);
}
```

Code extracted from file gilf/Instantiator.cpp, lines 1470 to 1473.

The private method resolve_ids is dedicated to patching the unresolved binding reference identifiers queued in data member m_unresolved_ids. It iterates over all nodes stored in the sequence container and looks up the according reference identifiers in the instance mapping table (see listing 5.29).

⟨Listing 5.36: Class Instantiator: Method for Patching Unresolved References⟩ ≡

```
void resolve_ids();
```

Code extracted from file gilf/Instantiator.hpp, line 246.

Visit Methods A visitor's behavior is defined inside its Visit method overloads. Of course, this is also true for class Instantiator. We will present the most important code sections from these methods that go beyond simply tracking the visitation graphs.

Every instantiation application starts at a Bind_Type or Bind_Function node. We present the Visit method's central parts for the first one, it shows the top-level control flow for entering the three instantiation phases. Method Visit for function bindings exhibits no significant differences to this method.

⟨Listing 5.37: Class Instantiator: Outline of Method Visit(Bind_Type&)⟩ ≡

```
void Instantiator::Visit(Bind_Type& node)
{
  print_prolog(node);
  g_log(lc_std, lt_low) << "Instantiator generates instance for this id:\n";
  g_log(lc_std, lt_low) << node.data.at<id_iprop>() << "\n";

  // (Phase 1) Get the signature.
  <see Listing 5.38 on page 98>
  // (Phase 2) Read instance description.
  <see Listing 5.39 on page 99>
  // (Phase 3) Generate instance.
  <see Listing 5.40 on page 99>
}
```

Code extracted from file gilf/Instantiator.cpp, lines 96 to 148.

Before actually collecting all instantiation parameters for initializing the mapping table, we have to check that we did not already generate this instance. This is achieved by examining the instance set. If the identifier is present, the instantiator's instance identifier is set to this binding's identifier and the method returns prematurely. Otherwise, the identifier is added to the instance set and phase 1 starts by visiting the node indicated by the current node's reference identifier.

⟨Listing 5.38: Class `Instantiator`: Phase 1 for Type Bindings⟩ ≡

```
m_phase = Phase_Signature;
m_mapping_table.clear();
// Check the binding's id for existence in the instance set.
id_iprop::type id = node.data.at<id_iprop>();
if (m_instance_set.count(id) == 0)
{
  // Not present, simply add the id and continue.
  m_instance_set.insert(id);
}
else
{
  // Already present. We have to avoid the endless recursion problem of
  // recursively referenced bindings.
  m_instance_id = id;
  return;
}
// Id reference of Bind_Type points to the type signature.
visit_node_by_ref(node.data.at<idref_iprop>());
```

Code extracted from file `gilf/Instantiator.cpp`, lines 104 to 121, referenced in listing 5.37.

When proceeding to phase 2, the data member `m_phase` is updated accordingly. If the mapping table filled in phase 1 is empty, we skip phase 2 because we deal with a non-generic data structure. For a generic data structure we bind the instantiation parameters in the mapping table by visiting the nested static parameter binding node and those reached via binding designators.

⟨Listing 5.39: Class `Instantiator`: Phase 2 for Type Bindings⟩ ≡

```
m_phase = Phase_Bind;
if (!m_mapping_table.empty())
{
  // Visit the one Bind_Static_Params child node.
  visit_nodes_by_type<Bind_Static_Params>(node, true);

  // Now also follow binding-designators that point to further
  // bind-static-params elements.
  visit_nodes_by_type<Binding_Designator>(node);
}
```

Code extracted from file `gilf/Instantiator.cpp`, lines 125 to 134, referenced in listing 5.37.

Phase 3 generates the instance described by this type binding. For this purpose, it visits the data designator subnode, which will result in a new entry into the global instance table. We also save the mapping from the current binding's identifier to the identifier of the generated instance in data member `m_instance_mappings`.

Notice that function bindings can contain multiple algorithm designators, and if we have chosen load-time instantiation, we have to query algorithm selection unit for the fittest algorithm (see [Kre02]).

⟨Listing 5.40: Class `Instantiator`: Phase 3 for Type Bindings⟩ ≡

```
m_phase = Phase_Generate;
// Visit the data structures pointed to by data designators and generate
// the instances from them.
visit_nodes_by_type<Data_Designator>(node);
// Register the mapping of binding id -> instance id.
g_log(lc_std, lt_medium)
```

```
    << "Generated instance [" << m_instance_id
    << "] for binding [" << id << "].\n";
  m_instance_mappings[id] = m_instance_id;
```
Code extracted from file `gilf/Instantiator.cpp`, lines 138 to 146, referenced in listing 5.37.

Phase 1 executes by walking down the visitation hierarchy until nodes are reached that represent instantiation parameters. These are added to the mapping table. Notice how the value type's default constructor is called to create an empty value along with the parameter's identifier used as key into the table. This code section is extracted from method `Visit(Type_Param& node)`.

⟨Listing 5.41: Class `Instantiator`: Creating Entry in the Mapping Table in Phase 1⟩ ≡
```
  id_mapping_type::const_iterator it = m_mapping_table.find(id);
  if (it == m_mapping_table.end())
  {
    g_log(lc_std, lt_low)
      << "Adding type-param " << id << " to current type map.\n";
    // Add id as key with empty value.
    m_mapping_table[id] = id_data_type();
  }
```
Code extracted from file `gilf/Instantiator.cpp`, lines 368 to 375.

The core functionality of phase 2 is implemented in the visitors for type and function parameter binding nodes. On entrance to these methods we have to validate that the instantiation parameter that will be bound is actually stored in the mapping table. If this is the case, we arrive either directly or through the detour of a binding designator at a type or function binding that describes the instantiation parameter's bound instance. This requires recursively generating this instance first.

The following listing taken from method `Visit` for `Bind_Type_Param` nodes shows how a new instantiator is created with a copy constructor from the current one. Then visitation is started with this instantiator by accepting a type binding, which is stored in the first position of sequence `nodes`. After the instantiator has performed its task, we can update the mapping table with the identifier of the generated instance. This recursive process ends when nongeneric constructs are encountered.

⟨Listing 5.42: Class `Instantiator`: Recursive Instantiation in Phase 2⟩ ≡
```
  // Recursively call the instantiator on a bind-type node.
  Instantiator instantiator(*this);
  nodes[0]->Accept(instantiator);
  // Set the type parameter to the calculated instance id.
  m_mapping_table[m_bind_ref] = instantiator.get_instance_id();
```
Code extracted from file `gilf/Instantiator.cpp`, lines 483 to 487.

Phase 3 processing really starts at data and algorithm nodes, respectively. At these nodes the top-level structure of instance generation is clearly visible. It starts with mangling the identifier of the monomorphic instance. The mangled name is created by appending the instantiation parameter identifiers from the mapping table to the data or algorithm's identifier. For nongeneric constructs, the mangled name equals the original one.

⟨Listing 5.43: Method `Visit(Algorithm& node)`: Name Mangling in Phase 3⟩ ≡
```
  // Name mangling to create new instance id.
  m_instance_id = node.data.at<id_iprop>();
  id_mapping_type::const_iterator it = m_mapping_table.begin();
  while (it != m_mapping_table.end())
  {
    m_instance_id += "." + (*it).first;
    m_instance_id += "(" + (*it).second + ")";
    ++it;
```

```
}
```
Code extracted from file `gilf/Instantiator.cpp`, lines 939 to 947.

Next, we initialize the instantiator's father node by cloning the visited algorithm node. It is crucial to replace the cloned node's identifier with the instance identifier created in the preceding steps, because the cloned node will be the root node of the generated instance.

⟨Listing 5.44: Method `Visit(Algorithm& node)`: Initializing the Monomorphic Instance⟩ ≡
```
// Clone algorithm top node and set id to new mangled id.
m_father = clone_node(node);
// Set the id to the instance id.
boost::shared_ptr<Algorithm> algo_clone =
  boost::shared_dynamic_cast<Algorithm, GILF_Node>(m_father);
if (algo_clone.get() != 0)
{
  algo_clone->data.at<id_iprop>() = m_instance_id;
}
```
Code extracted from file `gilf/Instantiator.cpp`, lines 972 to 980.

Instance generation proceeds according to the visitation graph shown in figure 5.5. First, we visit the algorithm's value parameter nodes. Then, the algorithm's local storage is processed, which consists of variables and constants, and finally the algorithm's body is appended to the monomorphic instance. At this point, we have generated a complete instance which was described by the function binding where instantiation application started. Therefore, we add the father node that holds the instance to the global instance symbol table.

⟨Listing 5.45: Method `Visit(Algorithm& node)`: Instance generation⟩ ≡
```
// First, append the transformed value parameters, to be found in the
// function signature.
visit_node_by_ref(node.data.at<idref_iprop>());
// Then, append the local storage section.
visit_nodes_by_type<Storage>(node);
// Last, append the algorithm body.
visit_nodes_by_type<Statements>(node);
// Add the generated instance to the instance table.
g_instance_table.add(m_instance_id, m_father);
```
Code extracted from file `gilf/Instantiator.cpp`, lines 985 to 993.

One can encounter binding designators while traversing the hierarchy of an algorithm node. These point to function or type bindings, thus requiring recursive instantiation. Listing 5.42 already depicted this procedure. But during instance generation, such nodes have to be replaced by designators to the generated instances. The following listing shows the necessary code. A new designator is created using the global GILF node factory. The type of the created node depends on the type of the designated binding node, which is either a function or a type binding. We recognize this type by looking at the node's identifier prefix. The new node's identifier is then set to the identifier of the recursively created instance. This way we ensure that the algorithm node will not contain any generic constructs.

⟨Listing 5.46: Method `Visit(Binding_Designator& node)`: Designator Replacement⟩ ≡
```
// Use factory to produce node & downcast for id setting.
GILF_Node::sptr_type new_node;
if (id_has_prefix(ref, "bt"))
{
  new_node = g_node_factory->create(c_node_id_data_dsg);
  node_cast<Data_Designator>(new_node)->data.at<idref_iprop>() = id;
}
```

```
else if (id_has_prefix(ref, "bf"))
{
  new_node = g_node_factory->create(c_node_id_algorithm_dsg);
  node_cast<Algorithm_Designator>(new_node)->data.at<idref_iprop>() = id;
}
m_father->push_back(new_node);
```
Code extracted from file `gilf/Instantiator.cpp`, lines 667 to 679.

The same procedure is required for instantiation designators, but the recursive instance generation did already take place in phase 2. The identifiers of the replacement designators are therefore looked up in the mapping table.

A simple translator from nongeneric GILF to C++ is sketched in section B.2, focusing mainly on the problematic constructs. It is also implemented as GILF node visitor.

5.7 Summary

The goal of this chapter was to provide a detailed description of the GILF system's prototype implementation. We first presented a short overview of modern C++ programming, followed by the prototype's module and library structure and the coding conventions employed in the implementation. Then we described the internal GILF representation on which further processing is based after deserialization. We also showed how node allocation is hidden behind a factory interface. Next two general facilities were touched, namely logging and symbol tables. Thereafter, an introduction to visitation of GILF nodes is given, which is the basis for all following transformation steps. The most important one, GILF's instantiation engine, is then discussed in great detail. It transforms full GILF into monomorphic GILF with all generic components removed by instantiation application.

Chapter 6

Related Work

6.1 Genericity

This section will give a short overview of the current state of generic programming research and languages. It will show how GILF aids in constructing compilers for such languages and how different features map to GILF constructs. A general introduction to generic programming is given in chapter 2.

6.1.1 Classification

Code reuse is a major goal in software engineering. Polymorphic languages support the programmer in this task by allowing functions and types to be applicable to parameters of more than one type. The different kinds of polymorphism were first identified by Strachey [Str67]. This classification was later refined by Cardelli and Wegner [CaWe85] and is illustrated in figure 6.1.

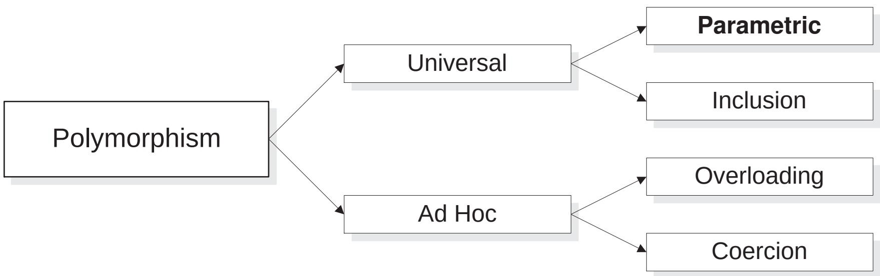

Figure 6.1: Classification of polymorphism according to Cardelli/Wegner.

Polymorphism is subdivided into two main categories, universal and ad hoc polymorphism. Informally, the distinction between them is that universally polymorphic functions operate uniformly on a possibly infinite range of types, whereas ad hoc polymorphism is achieved by a function that works for a fixed set of types, maybe in unrelated ways.

Cardelli and Wegner extended Strachey's original classification with inclusion polymorphism, which models subtyping and inheritance in object oriented languages. Notice that parametric polymorphism usually relies on overloading after instantiation, as function symbols and identifiers like `+`, `*`, `id` and so on will be bound to the correct function call with overload resolution. Identifier renamings like in Theta [LiCuDa+95] or Tecton

[Mus98] are an alternative or complementing approach for solving this particular problem of parametric polymorphism. For a more detailed discussion of the interrelations between the different kinds of polymorphism see [CzEi00], section 6.7. In this thesis, we primarily tried to solve problems associated with parametric polymorphism at the compilation level. However, we do not completely agree with Cardelli and Wegner's assessment of parametric polymorphism, which directly reflects the implementation of ML-style functional programming languages:

> ... in true polymorphic systems code is generated only once for every generic procedure. ([CaWe85], p. 7)

This is achieved by representing data uniformly[1] in some way, usually by using a pointer representation to a structure with more information. Of course, this indirection comes with a run-time overhead and simply hides the different treatment of different instantiation types of generic code. We do not want to accept this penalty and relax their statement by saying that *intermediate* code is generated only once for every generic procedure.

Type parameters can be present either explicitly or implicitly. Most languages apply the rule that declarations of generic constructs have to explicitly state type parameters, whereas type parameters are inferred or deduced when instantiations are used.

In type theory, universal quantification is used to model generic functions and types, bounded quantification for subtypes and type inheritance, and existential quantification for abstract data types. In [CaWe85], the reasonable combinations of these orthogonal concepts are explained and evaluated. However, simple bounded quantification lacks expressive power when applied to generic functions that involve recursively defined types and recursive bounds, like binary methods [BrCaCa$^+$95]. F-bounded polymorphism [CaCoHi$^+$89] is considered a good compromise between expressive power, complexity and tractability, and still allows handling the typical cases where simple bounded quantification fails. Even F-bounded polymorphism fails on some desirable cases, which are discussed in [DaGrLi$^+$95]. Substantial work has been poured into research on type systems that combine inclusion polymorphism, the kind of polymorphism central to object oriented languages, with parametric polymorphism [EiSmTr95] [Bru93] [BrFiSc95] [BrOdWa98] [Lit98].

Constrained Genericity Constraining the legal instantiation of generic constructs is a major issue in designing a generic programming language. A thorough exploration of the design space of this feature is given in [Lit98]. Litvinov establishes the following categories of constrained genericity:

Instantiation-Time Checks Languages in this category allow completely unconstrained generic constructs. Every instantiation is checked by itself, which leads on the one hand to very flexible code, but on the other hand requires the source code of generic components for type checking their instantiations. The role models for this kind of generic programming languages are C++, Ada, and Modula-3. However, the C++ instantiation model permits to enforce instantiation constraints at compile time [SiLu00].

F-bounded Quantification As noted above, F-bounded quantification [CaCoHi$^+$89] allows expressing mutually recursive bounds. Some Java dialects, for example Pizza

[1] The term *boxed representation* is commonly used in this context.

[OdWa97], its successor GJ [BrOdSt+98], and [AgFrMi97], support it as their mechanism for constraining instantiations. Also, the Strongtalk type system for Smalltalk employs F-bounded polymorphism [BrGr93].

Signature Constraints and Structural Subtyping The first programming language that supported constrained parametric polymorphism was CLU [Lis92]. It used `where`-clauses to list signatures that have to be supported by legal instantiation type arguments for generic functions. This signature list introduced a kind of *protocol* type, and all types that conform to this protocol implicitly are treated as subtypes. The same mechanism was later carried over to Theta [LiCuDa+95] and PolyJ [MyBaLi97], a Java extension. It is also present in Cecil [Cha98], but Cecil also offers mutually recursive subtype bounds. This implicit, structural subtyping is usually seen in contrast to explicit subtype relations, as in (F-)bounded quantification. The advantage of explicit subtyping is that accidental false conformance is not possible, which is sometimes an issue with structural subtyping, e.g. in the notorious example with a `draw` method that can belong to both a `GraphicalObject` and a `Cowboy` class. Notice that Strongtalk dropped implicit structural conformance with brands in a later version [Bra96] in favor of explicit subtyping.

SelfType or Matching Another problematic case are methods that return results of its own type. Some languages like Strongtalk [BrGr93] offer a special `SelfType` that can be used in method signatures and will be resolved to the actual instantiated subtype. This allows exact typing of binary and related methods. Matching is another mechanism that exhibits similar expressiveness, it can be found in PolyTOIL [BrFiSc95] and Emerald [RaTeLe+91]. Required functions are related to a type parameter of the function itself. In general, both methods provide convenient syntax for common idioms but are less powerful than F-bounded polymorphism.

Covariant Redefinition Finally, some languages allow the definition of *anchor* types inside functions and classes. The anchor type is the upper bound of the possible instantiation types. Instantiations of the general function or class are performed by explicitly deriving from it and narrowing the anchor type to a more specialized typed. This mechanism was proposed for Java by Thorup [Tho97] [ThTo99], and is also available in Eiffel [Mey92].

The approach taken in the SuchThat project goes beyond these techniques, which all rely on the semantics established by the relations between types. We also rely on a strong type system, but the type hierarchy is determined in the Tecton specification system, which results in extensive semantic guarantees. It is even intended to connect the specification system with an off-line archive of proven properties of the type hierarchies.

In SuchThat, we have a fully static type system, and this is reflected by the available features in GILF. In general, type systems can have both a static and a dynamic part, for example Scheme [KeClRe98] is a dynamically typed functional programming language, all type checks are performed at run time.

Finally, a classification of the typical usage scenarios for generic programming is presented. These can be split in two categories, which complement each other. First, the main target of generic programming is of course writing data structures and algorithms that operate on a wide range of related types in a uniform way. This is exactly how container or collection classes are designed, like in the STL [MuDeSa01] or Java's collection classes and interfaces [BrCoKe+01]. The second important application of parameterization is adapting the behavior of generic constructs in a flexible yet efficient way. The key to this technique is realizing that type parameters can be function types. For example, the way a container is sorted can be changed by passing a function object to the sorting algorithm of the STL, which defines the ordering relation between elements. This approach can be applied also at the module level, which leads to parameterized programming, proposed by Goguen [Gog96]. Ada's package system [Bar95] is a restricted incarnation of this idea. In C++, parameterizing the behavior of components has attracted great attention. Alexandrescu propagates policy based programming [Ale01] that fosters the construction of reusable components, for which almost every behavioral aspect can be adapted by the user. A simple application of this idiom are container adaptors like STL's `stack`. Summarizing, both techniques are well suited for creating libraries that offer a high degree of reusability.

6.1.2 Generic Libraries

Generic programming's primary focus is developing reusable software components. The components are collected and distributed in generic libraries. Significant progress in generic programming manifests in such libraries. This section will provide an overview of the major contributions in this area.

Standard Template Library (STL) The STL [StLe95][MuDeSa01] is perceived as the first library that follows consequently the philosophy of generic programming understood as requirement oriented programming. Structuring a library according to this principle is now often referred to as *in the spirit of the STL* or *STL-like*. The STL has become a part of the C++ Standard Library [ISO14882], but the term is generally used to denote the collection of data structures and algorithm that constitute the original STL. There are many publications available on the STL [Aus98] [MuDeSa01] [Mey01], which give detailed descriptions of its underlying concepts and provide extensive reference material.

Pragmatically, the STL consists of homogeneous container classes that allow accessing and visiting stored elements with iterators. Algorithms operate on and manipulate ranges of elements which are delimited by these iterators. An algorithm's behavior can be adapted by function objects, and allocators allow the customization of memory allocation strategies for containers. Finally, adaptors provide interface adaptions, for example, in some STL implementations a `reverse_iterator` is just an adaptor of a regular `iterator`. So what makes the STL special? There are two points that set the STL apart from previous collection libraries:

1. By separating orthogonal concerns of the problem domain like element traversal and access with iterators, element storage in containers and algorithms operating on element ranges, and parameterizing these concepts appropriately, the STL provides a flexible, efficient and extensive framework for generic components.

2. All concepts in the STL are rigorously annotated with requirements, thus clearly stating which instantiations of data structures and algorithms are valid, and how efficiently they compute.

For todays C++ compilers, STL implementations have to be available in source code for type checking. This problem partially motivated the work on GILF, because distributing generic libraries in a standardized intermediate language is more convenient than in source form.

Matrix Template Library (MTL) One of the first libraries to apply the methodology pioneered in the STL to another problem domain was the Matrix Template Library (MTL), developed at Indiana University. The MTL is an effort to develop a high-performance numerical linear algebra generic library [SiLu99] [SiLu98a]. There are two major goals that the authors want to reach. First, they want to reduce the size of other numerical linear algebra packages which is caused by the combinatorial possibilities to arrange precision types, matrix types, and algorithms operating on them. Second, in order to reach this goal they do not want to sacrifice run-time performance, a very important aspect in scientific computing,

As it turns out, generic programming provides the right mechanisms to reconcile this competing targets. The MTL consists of a small collection of algorithms that operate on vectors and matrices. In MTL, a matrix is a composition of its orthogonal aspects, namely its element type, orientation, shape, and one and two dimensional storage strategy. Every aspect can be extended by the library user if his provided type complies to certain requirements. The possibility to parameterize algorithms with all composed matrix and vector types leads to MTL's comprehensiveness at small source code size. This is carried even further by providing adaptors and special iterators that enable reusing the same algorithms in more situations. For example, in BLAS the dedicated function `daxpy` is responsible for adding a vector to a scaled vector, whereas in MTL this is achieved by calling the generic `add` algorithm with the first argument adapted.

```
// perform: y := ax + y
add(scale(x, a), x, y);       // MTL
daxpy(n, a, x, 1, y, 1);      // BLAS
```

As in the STL, the key to combining algorithms and data structures are iterators. MTL uses an extended iterator concept for dealing with the two dimensional nature of its matrix components. In general, at the first level 2D iterators allow access to rows or columns, which export 1D iterators for accessing single matrix elements. Notice that this fully general approach handles all matrix representations.

The second goal of the MTL, high performance, is guaranteed mainly by two means. Static (parametric) polymorphism in generic libraries removes the overhead associated with inclusion parameterization often found in object oriented libraries[2]. Furthermore, the application of template metaprogramming techniques allow generation of efficient kernel routines in C++ without resorting to external code generation tools. These kernels are used in more general algorithms, which can be configured according to cache hierarchy idiosyncrasies, also using template metaprogramming. This enables portable optimizations even for different architectures [SiLu98b] [SiLu98c].

The VIGRA Computer Vision Library Another generic library that deals with 2D data structures is VIGRA [Köt99] [Köt00], a library designed for image processing. Again, code economy without loss of efficiency is achieved by separating image representation from algorithms manipulating them with the help of iterators. This time, rectangular regions of interest (image windows) are framed by a set of boundary marks. These are all the pixels next to border pixels of the real image. `ImageIterators` present horizontal and vertical views on the region. Algorithms operate on regions delimited by an `upperleft` iterator, pointing to the first pixel in the image, and a `lowerright` iterator, pointing to the boundary mark one pixel right and below the last image pixel. Functors (function objects) introduce the needed flexibility into the small set of algorithms.

[2]Effective small object optimization is crucial in this respect.

One interesting feature of VIGRA is that mutating data access via iterators is more general than in the STL, using the data accessor idiom [KüWe97]. This makes sense for VIGRA, as image representations like banded RGB do not allow application of the STL reference semantics of `operator*()` for data access. Furthermore, iterator adaptors play an important role in visiting pixels in advanced settings. For example, a line iterator adaptor visits pixels along an arbitrarily directed line.

Applying the generic programming paradigm in this problem domain was more successful than object oriented methodology because the inability of compilers to inline and optimize virtual function calls resulted in a big performance impact due to the large data sets typically encountered in image processing.

The Boost Graph Library (BGL) One of the most advanced generic C++ libraries is the Boost Graph Library [SiLeLu01]. Again, graph data structures are parameterized class templates that are connected to algorithms by iterators. Graphs are more complex data structures than standard containers, therefore the interfaces of these components are richer.

Vertices and edges are handled through descriptors. The types of these descriptors can vary for each instantiation and are accessible through the `graph_traits` class. Arbitrary data can be attached to vertices and edges by means of property maps, which provide a general interface to set and query a property's value with a tag, similar to STL's associative containers.

A graph can either be stored as an adjacency list for sparse graphs or an adjacency matrix for dense graphs[3]. Traversal of graphs is handled by iterators, which work either on edges, vertices, or the graph's adjacency structure. For example, the `vertices(g)` function returns a sequence of all edges belonging to graph `g` inside a `std::pair`, which contains two edge iterators, one pointing at the first edge, the other one past the last edge.

Generic graph algorithms operate on these graph data structures, navigating them with the graph's vertex and edge iterators. There are three fundamental algorithm patterns, breadth first search, depth first search, and uniform cost search. All real graph algorithms like Dijkstra's Shortest Path are written in terms of the algorithm patterns. This generality is accomplished with visitors, a extension of the STL's function objects. Instead of just overloading the application operator, visitors provide a set of functions that are called at special event points when traversing the graph, for example `start/finish_edge` or `examine_edge`. Finally, graph adaptors allow to access data structures from other graph libraries, like LEDA and Stanford Graphbase, or to present a different view of a given graph, like `filtered_graph`.

Loki The last library we want to mention is Loki [Ale01], which uses type parameterization and template metaprogramming extensively, but strikes at different goals than the other presented libraries. Loki provides on the one hand low-level facilities like type lists and hierarchy generators. Based on these utilities, highly customizable generic design patterns are built. A generic design pattern tries to capture the essential behavior of the related pattern and makes them customizable by introducing them as special template parameters, so-called policies. Default policies implement the most often used instantiations, but the library user can adapt all parameterized behavioral aspects of the pattern template by providing hand-crafted policies. Loki contains generic patterns for smart pointers, functors, singletons, factories, visitors, and multimethods.

[3] But the actual implementation of the used containers can be selected at instantiation time.

The ability to manipulate types, and especially type lists, at compile time is being recognized as useful in generic programming. Even in functional programming languages, similar observations were made [Hal01]. Furthermore, policy based design leads to extremely flexible generic components. It is even considered to extend STL containers with policy parameters.

6.1.3 Programming Languages

To conclude this section, a short summary of programming languages that support genericity is given.

ML [MiToHa+97] was the first language to feature parametric polymorphism, based on Milner's type calculus. It can be seen as the archetype for most functional programming languages that use type inference to recover some of the positive effects of strong typing, like early error detection and efficient code generation. Structures and signatures allow interface and implementation separation in ML, as well as constraining definitions of polymorphic code. In Haskell [JoHu98], type classes [JoJoMe97] were introduced to structure overloaded functions[4].

The first language to include constrained parametric polymorphism was CLU [Lis92] with where clauses, later also present in Theta [LiCuDa+95]. This approach to constraining genericity was discussed above. Theta is otherwise a pure object oriented language. Many other object oriented programming languages offer type parameterization, for example Eiffel [Mey92], Cecil [Cha98], a dialect of Smalltalk called Strongtalk [Bra96], and extensions to Java. Constraining type parameters by a subtype relation is natural in this setting.

The Java extensions fall into three categories. First, PolyJ [MyBaLi97] introduces structural subtyping to Java. Requiring a subtype relation, for example with F-bounded polymorphism or a related variant, is the most frequent mechanism. It is present in Pizza [OdWa97], NextGen [BrOdSt+98], GJ [BrCoKe+01], and [AgFrMi97]. Finally, an approach based on covariant redefinition, called virtual types, was presented by Thorup [Tho97] [ThTo99].

Even to Oberon, a successor of Pascal, an extension with parametric polymorphism was proposed [RoSz97]. It allows type parameterization for reference types only, thereby simplifying the implementation considerably[5]. In addition, the programmer is aware of the explicit cost due to the pointer indirection.

Last not least, genericity is also available in popular industrial multi-paradigm programming languages. C++ [ISO14882] [Str97] was mentioned throughout this text, and several generic libraries implemented in C++ were described above. Type and value parameterization is available as class and function templates in C++. Type deduction allows omitting explicit type arguments for most instantiations. The most controversial *feature* of C++'s template mechanism is its lack of constrained genericity. Partial solutions were presented [SiLu00], but they only alleviate the problems. Ada [Bar95] is another example of a mainstream language with generics. Contrary to C++, it allows stating requirements on type parameters, which was exploited in an conversion of the STL to Ada95 [ErKo96]. In Ada, all used generic components have to be instantiated explicitly. This has shown unwieldy when generics are extended, but avoid misinterpretations of the programmer's

[4] In [Lit98], type classes are categorized as a kind of F-bounded polymorphism.

[5] In their implementation, the authors used Oberon's reflective features to gain exact type information for operations that were dependent of type variables at run time, for example `new`. The same approach was later proposed for Java implementations [SoAl98] [Vir01].

intent by the compiler. Ada also has a generic package concept, motivated by Goguen's ideas on parameterized programming [Gog96]. However, packages are not valid package parameters themselves. Modula-3 [CaDoGl+89] provides genericity at the module level. In a generic module, some of the imported interfaces are regarded as formal parameters. In order to instantiate such a module, these interfaces have to be bound to actual interfaces. Generics in Modula-3 were kept simple intentionally, because genericity in Ada was considered too complicated.

C++, Ada, and Modula-3 all have in common that type checking is performed at instantiation time. This fact and their nonuniform type system forces them to reuse generic components from source code.

6.1.4 Discussion

In this section we touched general, high-level topics on generic programming. Presently, GILF supports a reasonable amount of features such that a source language with similar design as SUCHTHAT can be easily mapped to GILF. The separation of instantiation analysis and application enables handling most semantic issues in the front-end and pass those decisions to the back-end.

However, two main properties of generic languages and libraries can only be translated into GILF with elaborate manual work. First, dynamic polymorphic behavior as present in most object oriented languages is not supported. This seems a valid omission, as we wanted to concentrate on the problem at hand, instantiating generic data structures and algorithms. Furthermore, template metaprogramming has become an important part in efficient generic C++ libraries. This compile time technique is also not supported in GILF. It relies on partial template specialization. Such a feature would have introduced a significant bottleneck in the back-end, and we wanted to keep high-level semantics decisions in the front-end. Looking at the use cases of template metaprogramming, manipulating types and type lists is most relevant for genericity. No established theory handles this problem satisfactorily. This is a good candidate for a future extension of GILF if more experience is gathered in this area.

Constrained genericity is adequately handled in GILF. At the intermediate language level, the constraints are reduced to a list of required signatures (see section 4.8.2), very similar to where clauses. Binding these signatures to algorithms are the minimal requirements for generating code, and resolving these bindings places no special burden on the back-end, as all semantic considerations of the front-end are passed in the binding section.

6.2 Intermediate Representations

The GILF specification is a central part of this thesis. Other work done in the area of intermediate representations will be examined in this section. We concentrate on systems where the intermediate language was central to the design.

6.2.1 Nongeneric Intermediate Representations

UNCOL The desire for a target independent intermediate representation in compiler construction dates back to historical projects like UNCOL, an attempt to create a *universal computer-oriented language* [Con58]. The aim was to generate an intermediate language that is targeted by compilers of problem-oriented languages. The premise of the UNCOL project was "a computation in terms of a sequence of operations, each of which operates on information stored in machine registers or alters the sequence of operations" ([Con58], p. 5). Various back-ends should translate UNCOL to machine languages of existing hardware. Although discussed, UNCOL was never realized.

P-Code P-Code [NoAmJe+76] is the intermediate language used in most compilers of the programming language PASCAL. Despite its age, P-Code has features that are still used in contemporary systems. P-Code is an assembler language that assumes a virtual stack machine, the P-machine. Instructions are put in four classes, stack manipulation, data control, execution control, and special instructions. The available instructions where influenced by the semantics of PASCAL, for example procedures in P-Code map directly to PASCAL procedures.

Porting a P-Code system to another platform consists of porting either a P-Code interpreter or a P-Code translator to the new machine language, as well as the platform specific standard procedures for input and output, heap management and mathematical functions. P-Code was quite successful, many variants where developed [Nel79] in order to satisfy diverging demands, like space efficiency. The most famous variant is U-Code [PeSi79], which aims at adding more high-level information to P-Code that allow optimizations at the intermediate language level.

DCode A modern descendant of P-Code is DCode [Gou97b], which is the intermediate representation used as target of various language front-ends in the Gardens Point compiler project [Gou97]. Among the front-end languages are Modula-2, Oberon 2, and ANSI C. Code generators for most popular machine architectures exist, like SPARC, MIPS, and IA32. Unlike P-Code, DCode contains no features that relate directly to a specific front-end language, it tries to stay language neutral. Its scope are imperative, procedural programming languages with a more or less conservative feature set.

DCode textually represents the instruction set of an abstract stack machine, the D-Machine. It offers low level data types like `word`, `hugeint`, and `float`, which are manipulated by instructions that operate on values on the stack. The most marking distinction to P-Code is the low level of address expressions, for example the nonexistent load instruction is synthesized from push address and dereference instructions.

For code generation, back-ends that handle DCode parse the ASCII file and create a control flow graph, as well as virtual assembler code. This virtual assembler closely resembles the target architecture, but omits details that do not offer good opportunities for optimization. Finally, register allocation is performed and assembler or object code is written to files.

Overall, the compilers based on DCode show the practical effectiveness of a front-end, back-end decomposition of a compiler.

MLRisc MLRisc is an optimizing back-end that is especially designed to be a good target from divers high-level, typed programming languages [GeLe]. However, it was originally developed as back-end for the Standard ML of New Jersey [SMLNJ] compiler, an implementation of Standard ML'97 [MiToHa+97]. MLRisc wants to help language designers reuse the research investments poured into a modern optimizing back-end.

The intermediate language of MLRisc is an abstract assembler with unlimited pseudo registers, which will be mapped to real registers and memory locations transparently. The main invention of MLRisc is its high degree of adaptability. This approach tries to honor the fact that no intermediate language satisfies all desires of potential front-end languages. Therefore, several aspects of the MLRisc system are specialized according to the semantics of the source language, the target instruction set, the flow graph, and the optimizations applied.

In order to use MLRisc as back-end in a compiler, the source language has to be translated to MLRisc instructions first. This requires specializing the intermediate language to best fit the source language. MLRisc allows the customization of constants, pseudo-ops, regions, annotations, even adding user defined operations. Of course, the subsequent stages operating on this representation have to be aware of this specific information passed to them. The tool MDGen generates most modules for code generation from a machine description. This description specifies register organization, instruction encoding, delay slot mechanism, and so on, of the target machine. Extensions are treated by adding modules that augment the built-in pattern matching system with ML constructs. The tight coupling with ML as implementation language is one major problem of the MLRisc system.

`C--` The motivation behind `C--` [JoRaRe99] was to provide a general assembly language that is appropriate for the task of serving as a generic back-end in programming language research projects. This task is often performed by C, which became a kind of universal assembler because of the wide availability of C compilers on virtually all platforms. Unfortunately, this approach is inadequate if the source language constructs do not map nicely to C constructs. Furthermore, C allows no control over low-level decisions like stack-frame layout or memory aliasing, which hinders garbage collection, exception handling, and optimizations.

`C--` provides abstractions for the main features of processor hardware, i.e. computation, control flow, memory, and registers. Computations work on machine types, providing an expression abstraction instead of lower level constructs. Control flow is modeled with `if` and `goto` statements. Variables will be mapped to registers or memory, depending on the target machine's capabilities. In general, `C--` gives the client much more control over low level decisions than C, but also provides abstractions that are not available in raw machine assemblers.

One major research topic of the `C--` project is efficiently supporting semantically different run-time services for various source languages. This services include garbage collection, profiling, debugging, and exception handling. The difficulty of this task stems from the fact that both front-end and back-end hold knowledge that is needed by these services. Consider the root finding problem in accurate garbage collection. Only the front-end knows which variables are pointers to the heap and which are atoms, whereas the back-end knows the location of these pointers at run time. Similar problems arise in all run-time services.

The philosophy in `C--` is to separate policy from mechanism by providing hooks in the `C--` back-end that allow the implementation of efficient high-level run-time services in the front-end. The front-end and the back-end communicate through a procedural interface whenever the front-end requires information only available to the back-end. The interfaces and mechanisms required for garbage collection and exception handling are described in [JoRaRe99] and [RaJo00], respectively.

As a side note, `C--` uses MLRisc as optimizing code generator.

National Compiler Infrastructure The National Compiler Infrastructure (NCI) project is an effort to provide a high quality compiler infrastructure that is the basis for further compiler research in the industry and at universities. The major components of the NCI are the Zephyr family of tools, and the Stanford University Intermediate Format (SUIF).

SUIF is an extensible C++ framework [KiHö97] whose class hierarchy captures an intermediate language. This internal representation is changed by passes that are applied to each procedure one at a time. An alternative pass model is available that allows a chain of passes to operate on one procedure. The user can easily add his own passes by subclassing the `Pass` or `PipelinablePass` classes. These passes can be combined either by passing the SUIF objects in memory or by reading and writing SUIF objects to and from files with the own persistence technology. The flexible and modular organization was one major invention of the redesign from SUIF 1 to SUIF 2.

The SUIF 2 hierarchy supports objects for annotations, symbol tables, file sets and scoped objects. Scoped objects are mainly statements, instructions and expressions at various levels of abstraction. For example, `do`, `while`, and `for` statements are available for a high-level representation, and several `jump`, `load`, and `store` instructions for low-level representation.

By deriving from the base class, new abstractions can be introduced. Initially, SUIF was intended for representing languages like C and Fortran, which do not differ much at a fundamental level, and are therefore easy to map to a common intermediate language. OSUIF, an effort to extend SUIF's applicability to object oriented languages [DuCoIa+97], illustrates how to extend the class hierarchy for advanced constructs[6], but it also highlights that "program semantics cannot easily be captured with data structures alone" ([DuCoIa+97], p. 2).

A SUIF system first operates on the high-level representation, subsequently lowering the abstraction level until machine code is generated. It is an interesting aspect of the SUIF system that multi-level representations are supported.

The core of the Zephyr compiler infrastructure is a Very Portable Optimizer (VPO) for high-quality code generation [ApDaRa98]. All optimization algorithms are machine independent transformations, but are performed on machine instructions. This somehow contradictory approach relies on code representation as register transfer lists (RTL). The semantics of the RTLs is machine independent, but every RTL has to match a real instruction of the current target machine. Therefore, writing a code expander that turns the compiler's intermediate representation into a valid RTL accepted by VPO and that fulfills the machine invariant constitutes substantial work. A SUIF-to-VPO bridge allows using VPO with SUIF based compiler front-ends.

Zephyr also provides tools that aid compiler writers in using the VPO. The most mature is ASDL, the Abstract Syntax Description Language [WaApKo+97]. It allows the concise description of tree-like data structures, primarily abstract syntax trees. Idiomatic data structures and functions reflecting the syntax description are generated with the tool `asdlGen` for C, C++, Java, and ML. These data structures can be written to and read from secondary storage in a language independent format as *pickles* in order to allow multi language compiler development.

ASDL plays a role comparable to XML Schemas[7], as these also allow the description of specialized tree structures. This is exactly what we did for XGILF. The advantage of XML is that is an established standard and a large amount of tools and APIs are available now, whereas ASDL is more tailored towards the special needs in compiler construction. In [Han99], Hanson mentions that the ASDL toolset was extended to create pickles in

[6] The extension allows dispatch based on one receiver and is part of the SUIF release since version 2.2.

[7] We consider XML DTDs as one XML Schema variant.

XML, which suggests a merging of the related technologies.

MCode Clarity MCode [LeDeGo95] is an intermediate representation developed by Sun at approximately the same time as Java Bytecode, but with different intentions. Clarity is a dialect of C++ with simplified semantics that supports systems and distributed programming especially well. MCode is the intermediate representation generated by the Clarity compiler. It is either interpreted or compiled into machine code at run time, using a simple counting policy for activation of the compiler. For interoperability with legacy C libraries, MCode is wrapped in standard object files that can be processed with standard system tools.

The MCode core is a stack machine with RISC-like instructions. The abstraction level of control flow instructions is at the same level as GILF instructions, constructs for procedure calls and loops exist. The motivation for this design decision was to allow the best possible code generation for the constructs on the actual target machine. Data manipulation is performed by typical stack machine instructions that work on the stack. Data types include native machine types, structs, unions, interfaces and implementations.

The implementation of the Clarity compiler is sketched briefly in [LeDeGo95], it is based on an object oriented design. A new code generator can be added by overloading two base classes, `CGMachine` and `CGValue`, respectively.

The Clarity project seems to have been abandoned by Sun due to the success of the Java programming language.

ANDF/TDF The ANDF project started 1989 as a request for technology by the Open Software Foundation with the intent to establish an architecture neutral distribution format (ANDF) [Mac93]. The technology chosen from the list of candidates was the TenDRA distribution format (TDF) [Cur95], developed at the United Kingdom's Defense Evaluation and Research Agency (DERA). TDF is the intermediate language used in compilers based on TenDRA technology.

TenDRA splits the compilation into two parts, production and installation. The producer translates the source language into portable TDF, which will be translated into target specific code on the deployment machine by the installer. Producers for C, C++, Ada95 [Bun95], and Java [FaFeRo97] are available, either publicly or commercially.

Unlike the representations discussed before, TDF is not an abstraction of processor hardware, but rather a tree-structured intermediate language that contains abstractions for common constructs found in imperative programming languages like C, Pascal, and Ada. In this respect, it is very similar to GILF. The most important TDF constructs are EXPs, which represent statements and expressions, TAGs, which abstract identifiers for variables and labels, and SHAPEs, which abstract data structures. SHAPEs are basic machine types, but recursive definitions are possible with arrays and compounds.

Despite OSF's request for a distribution format, the main emphasis of TDF is on portability. The approach taken by TenDRA is checking source code against target independent application interfaces (APIs). These APIs are specified with TDF's token declarations. The public TenDRA release contains target independent APIs for POSIX, the ANSI C library, Motif 1.2, and other system libraries. The checked source text is then translated into target independent TDF by the appropriate producer, into so-called capsules, a linearized binary encoding of TDF. These capsules are linked against target dependent capsules that contain the token definitions, which replace the unresolved tokens. In a final step, the capsule resulting from the TDF linking process is translated by an installer to target machine code and linked with system libraries, producing an executable for the deployment machine. Programming in a TenDRA compilation environment is all about programming against explicit APIs.

The token construct is the means of parameterization in TDF. Every token has to be replaced by its token definition such that the installer can produce machine code. Tokens can be used almost everywhere in TDF, not only as placeholder in API specifications. Notice that porting a TDF system to a new system involves augmenting all tokens used in the producer by platform specific token definitions that implement these tokens.

An interesting case study is the C++ producer, which is part of the public TenDRA release. It shows how genericity can be handled in a TDF based compilation system. All nongeneric parts of a C++ program are translated directly to TDF by the C++ producer and can be compiled into system object files by installers. But what happens with class and function templates? TDF capsules contain only instantiated templates. At the time the capsule for the main program is generated, the C++ producer holds the list of all instances in a global variable, avoiding reinstantiations. On the other hand, this means that the C++ producer must have access to template code in source form. The C++ producer's documentation describes the idea of dumping the producer's internal representation and using this for linking of C++ template libraries. This comes very close to our concept, but of course is specific to C++ and does defeat the purpose of TDF as distribution format. Furthermore, it is not implemented and work on the C++ producer seems to have stopped since 1998.

SDE/Slim Binaries Slim Binaries [KiFr99] [FrKi96] are an intermediate representation that is based on abstract syntax trees, consequently closely related to GILF and TDF. The technology was introduced by Michael Franz as semantic dictionary encoding (SDE) in his dissertation [Fra94].

At its core, a Slim Binary consists of a parse tree that describes the source program's operations and a symbol table which contains extensive type information about these operations. Slim Binaries are translated into machine code at load time. In order to recoup the time penalty incurred by code generation at load time, the representation exploits source code characteristics to achieve a very compact binary encoding. For example, common expressions that occur multiple times in the same scope are encoded efficiently with a predictive algorithm. From its initial application in the Oberon-3 system, the authors have extended the applicability of SDE/Slim Binaries to portable code because the compact encoding reduces network bandwidth.

What sets Slim Binaries apart from the TDF technology is that it features a dynamic compilation process at load time instead of the more traditional approach taken in TenDRA systems that require a static, off-line linking process. The structure of a the SDE compilation system has directly influenced the GILF system, as can be witnessed in chapter 3. Furthermore, the authors claim that Slim Binaries offer a more dense encoding as well a faster decoding process, compared to TDF capsules and installers. They back this statement with extensive figures. The Oberon-3 distribution for three platforms was reduced from three packages around 2.5MB of data to just one platform independent Slim Binary package of 0.8MB of data.

Java Bytecode The Java programming language [GoJoSt^{+}00] owes much of its success to its innovative run-time environment and execution model. Java source programs are compiled into Java Bytecode, which is executed by an implementation of the Java Virtual Machine (JVM) [LiYe99]. In its simplest form, and also least efficient one, a JVM interprets the bytecode representation. More recently, just-in-time (JIT) compilers have become standard components in contemporary JVM implementations. These JIT compilers selectively translate methods into native machine code, which no longer require interpretation [GrMi00].

The JVM is a stack machine with an instruction set that was designed to facilitate compilation of Java source code into Java Bytecode. The supported data types directly reflect Java's data types: primitive types and reference types. Primitive types are typical integral and floating point machine types, whereas reference types are pointers to class instances and arrays. Instructions operating on JVM types implicitly include the operands' type in the instruction, for example `iload` loads a local variable of type `int` onto the stack. Because the JVM was designed to represent Java, it directly supports the Java object model with single implementation and multiple interface inheritance, and includes method invocation instructions which match Java's dispatch semantics.

Class files are used to transport portable Java Bytecode that can be run on any platform that has a conforming JVM implementation. Additional to the pure instructions, class files contain symbolic information about the class behavior and interface, stored in the constant pool. This type information is needed by the JVM, because it has to perform type checks at load time and even at run time after dynamic class loading, one of its most powerful features. The type checks are necessary to enforce Java's rigid security model that renders some common run-time errors impossible. The security mode is essential for the JVM as it propagates remote execution of mobile, sometimes untrusted code.

The success of Java makes JVMs an ubiquitous software tool on most platforms. This makes Java Bytecode an attractive target for front-ends of languages other than Java, despite the fact that it was not designed for this purpose. At [Tol02], about 160 systems are listed that use the JVM as back-end, including logic, functional, and object oriented programming languages. Nevertheless, the problems of systems taking this road have also been noticed by others [Gou00] [GoCo00] [PeMe01], the most severe ones are the limited methods for parameter passing in the JVM, gaining access to nonlocal variables, and representing function pointers. The lack of tail recursive function calls especially hinders translation of functional languages to the JVM. Moreover, mapping type unsafe features to the JVM is virtually impossible, as the security model is built on top of the strong type system.

CIL Recently, Microsoft also changed the basis of its development platform fundamentally. The common language infrastructure (CLI) provides the techniques and specifications to run managed code, represented in the common intermediate language (CIL) [ECMA01]. It has much in common with the JVM, but also differs in significant design choices. Just like in Java Bytecode, a high-level type system, the common type system (CTS), is central part of the CIL. But in contrast to Java Bytecode, it was designed from the start to support multiple languages equally well. This reflects in the richer set of types in the CTS. It offers primitive types and reference types, but also various pointer types and compound value types. Value types resemble structs in C, and they can be allocated by the virtual execution system (VES) statically, they are not put on the garbage collected heap like all reference types. CIL contains instructions to box and unbox value types for efficient interaction with reference types.

The VES follows the abstract stack machine design, which obviously influences the CIL instruction set. Unlike Java Bytecode, instructions normally do not encode the types on which they operate, the VES has to infer these types from the declarations of stack parameters and local variables. For example, the instruction `add` adds the topmost locations on the stack without overflow check. All arithmetic operations in the CIL are available with and without overflow check that throws an exception, the corresponding instruction to `add` is `add.ovf`. The instructions for method dispatching are also more flexible than in Java Bytecode, a nonvirtual method dispatch is available. Virtual dispatch is built-in for single implementation and multiple interface inheritance. To satisfy the needs of functional programming languages that feature recursion as only means of iteration, the `call` instruction can be marked as a tail call, thus discarding it caller's stack frame. Furthermore, parameter passing by reference for primitive types is supported, whose omission in Java Bytecode is a performance bottleneck in some back-ends targeting it.

The VES differentiates between managed and unmanaged code, the former one is verifiable by the CLI. Managed code is restricted to use features of the instruction set that are completely typesafe. For example, using arithmetic operations on pointers is not allowed in managed code. The verifier guarantees practically the same safety level as the Java Bytecode verifier, performing part of the verification at load time, and relying on run-time checks for some advanced checks, like array bounds checking.

One deliberate design decision of the CIL was not to endorse interpretation. Instead, CIL code is compiled into native code at load time on the deployment machine. This makes Microsoft's compilation strategy very similar to the one chosen both by SDE/Slim Binaries and GILF. Assemblies serve as the component format in CLI. They may contain several class descriptions and their code, top-level methods, and meta-data like versioning information or digital signatures, but also user provided meta information.

Generally, CIL is an interesting intermediate representation for front-ends from multiple source languages. A detailed comparison between CIL and Java Bytecode can be found in [Gou01], using it as back-end of a functional programming language is discussed in [PeMe01].

6.2.2 Generic Intermediate Representations

All the mentioned intermediate representations have one thing in common. They do not take requirements of genericity into account. We will now examine projects that where motivated by alleviating this shortcoming. First, extensions to both Java Bytecode and CIL are examined, followed by an intermediate language that supported parametric polymorphism from the beginning.

Translating Generic Java Dialects to Java Bytecode Java's lack of parametric polymorphism has been perceived as a major drawback from its very first days in 1996, when Java became available. This shows in the numerous literature that treats extending Java with this missing feature. Generally, genericity in Java is realized as parameterized classes and interfaces, and parameterized methods. The first proposals were Pizza [OdWa97], PolyJ [MyBaLi97], and [AgFrMi97]. In this section, these publications are scrutinized in regard to the compilation process.

Odersky and Wadler [OdWa97] established the terminology for work on generics in Java. They identified two major strategies for translating parametric polymorphism in Java, either a homogeneous or a heterogeneous translation. A heterogeneous translation creates specialized code for every instance of a generic construct, whereas a homogeneous translation creates one translation of the generic code that is used by all instantiations, relying on the *generic idiom*. The generic idiom is used to implement collection libraries in current Java without support for parametric polymorphism. It exploits the fact that all classes inherit from the common base class `Object`[8]. Collections store references of type `Object`, enabling them to hold any Java class type. Client code is required to explicitly downcast these objects to the actual stored type in order to access their specific properties. Notice that these downcasts also have to be present in code generated by a generic Java compiler, because otherwise the Java Bytecode verifier would reject the code as type unsafe.

Another important dimension in evaluating generic extensions to Java is the compatibility of generated code to old virtual machines. It is desirable that bytecode created by a generic compiler is executable on unchanged JVMs, this means legal nongeneric Java Bytecode is created that can be processed by an unchanged class loader. Furthermore, an interaction with old collection classes would ease the upgrading process to the new language. The backwards compatibility of generic proposals heavily influenced most work in this area.

The advantages of a homogeneous translation lie in its space economy, because one code is shared by all instantiations. Also, no run-time overhead is introduced except for the type casts required by the bytecode verifier. For realistic software projects, the overhead in run time introduced by type casts is around 3-5%, but synthetic benchmark lead to figures up to 20% [MyBaLi97]. Pizza, and a newer, simplified version called Generic Java (GJ) [BrOdSt+98], both employ the homogeneous translation strategy. Special care was taken in GJ to make parameterized and unparameterized collection classes interoperate at the bytecode level without modifications to the JVM. Recently, GJ was proposed as extension to the Java language [BrCoKe+01] and will presumably become part of Java in version 1.5.

The obvious drawbacks of the homogeneous approach are performance issues. Primitive types can be supported only as boxed values, e.g. `int` as `Integer`, and type casts have to be inserted although static type checking has proven them needless. Therefore, several heterogeneous implementations for parametric polymorphism were proposed [AgFrMi97] [MyBaLi97] [EvKeMe+97] [BoDa98].

The work by Agesen et al. was the first to propose instantiation at load time, which required a revised class loader and extended Java Bytecode. The extensions are prepended to a standard class file. They are made up of an identification number, a counter for type parameters and constraints, followed by the constraints themselves. In the following bytecode, references to the type parameters introduced in the extension section appear, which will be replaced by the loader.

PolyJ requires an extended bytecode, also. First, information about type parameters and where clauses are added to class descriptions, and signatures allow the specification of instantiations. Second, two new instructions are introduced, `invokewhere` and `invokestaticwhere`, which call operations on parameter types, either with virtual or static dispatch.

Security problems occur due to the lost type information through type erasure, first mentioned in [AgFrMi97]. The problem of discarding type information leads to more severe restrictions imposed by the systems based on homogeneous translations, as formalized in [SoAl98]. At the language level, operations that require exact type information are not available for operations involving actual type parameters. These are type casts, object allocation and instance tests. For example, the statement `InIt it = new InIt();` is not valid in GJ, if `InIt` is a type variable. The lack of exact type information also impedes the application of reflective and persistent Java mechanisms in generic code.

[8]Newer implementations us a technique called *type erasure* [BrOdSt+98] that replaces a type parameter with its bound, which is not necessarily the least specific `Object` type.

Recent work about generic facilities in Java concentrated on overcoming these problems while retaining a partial homogeneous translation.

One approach is NextGen [CaSt98], a superset of GJ. It supports full type information at run time by creating wrapper classes for each instantiation, which inherit most of their behavior from a type erased base class. The NextGen implementation can be regarded as a hybrid approach, using a homogeneous translation for the base class, and using a heterogeneous translation for the wrapper classes and interfaces that carry the type information at run time. The wrapper classes are augmented with code *snippets* that implement the type dependent operations that are not possible in a pure homogeneous code base.

The second effort is by Mirko Viroli. He elaborated on the idea by Solorzano and Alagić [SoAl98] of using Java's reflection facilities to make exact type information about instantiations and the involved types available at run time, and to implement type dependent operations in terms of these data. Viroli presents an implementation that computes type descriptors of instantiations at load time and stores references to these descriptors in each parametric object [ViNa00], thus avoiding much of the run time impact associated with a reflective solution to parametric polymorphism. Benchmarks show that it performs better than NextGen regarding load time and space efficiency, but adds a run time penalty. In a following paper, he demonstrates how to extend this technique to virtual parametric methods [Vir01]. An interesting insight found in his work is the condition under which eager recursive generation of parametric types leads to nontermination of the instantiation process [Vir00]. This can be avoided by lazy instantiation, which can also improve performance, as only run time data structures for actually used instantiations are created [Vir02].

Generic CIL Although the specification of CIL and tools to generate CIL have been made available for its first generation only recently, a proposal that handles parametric polymorphism at the intermediate language level already exists [KeSy01], which one more time shows the significance of genericity. The aim of Kennedy and Syme's work is to enrich the common type system present in CIL with constructs that support parametric polymorphism in order to supply a convenient target for various high-level languages with this language feature. More advanced constructs, like type classes and higher-order types in Haskell and Mercury, or the full C++ template mechanism, are not directly provided, but the most expressive Java proposals can be mapped easily to generic CIL. It features exact run-time types, instantiations for value and reference types without boxing, bounded polymorphism, parameterized classes, structs, interfaces and methods, as well as polymorphic inheritance and recursion. The extensions to CIL can be grouped in the following three categories:

1. New polymorphic forms of class, interface, struct, and method declarations.
2. Added instantiated types and type variables to the common type system.
3. New instructions, and generalized forms of the current ones.

Like our approach, generic CIL leverages the advantages of load time instantiation to generate code for different instantiations on demand. An interesting characteristic of their implementation is the hybrid approach to code specialization and code sharing. In general, they apply a heterogeneous code generation approach, but to avoid code bloat, instantiations that yield exactly the same machine code are shared. This gives rise to the same problems encountered by homogeneous Java generics translations, namely to supply exact type information at run time, which is necessary for serialization and reflection mechanisms. For class types this is solved by storing type information in the virtual table, which is specialized for every instantiation. Virtual tables that are used as type identifiers are called *type handles*. Dictionaries of type handles are created during program execution which contain entries for all actually used instantiated types. For parametric methods, the relevant dictionary is passed as extra parameter in order to have access to exact type information.

TAL/TALx86 At Cornell University, a statically typed assembly language (TAL) was developed, first used as target from a call-by-value variant of System F, the polymorphic λ-calculus [MoWaCr$^+$98]. The motivation for this project was to preserve type information in all stages of the compilation process. This information guides several analyses and transformations like closure conversion and unboxing. The typed TAL programs can be checked for various type errors, thus draining notorious error sources, very similar to Java Bytecode and CIL verifiers. This enables TAL to participate in systems that require security guarantees from code running under their control, like plug-ins for applications or operating systems.

TAL has a RISC-like instruction set, augmented with a rich type system that supports parametric polymorphism, tuples and more. This means, it is possible to manipulate generic data types in TAL. On the other hand, TAL instructions in code segments work on fully instantiated types, only. Parametric polymorphism is translated using a boxed type erasure interpretation, common to most compilation systems for functional programming languages.

TAL, intended as a formalism for a low-level, statically typed assembler language, was further improved to a realistic assembly language called TALx86 [MoCrGl$^+$99], based on Intel's IA32 architecture. In TALx86, the focus of the project shifted to providing a preferable alternative to Java Bytecode as target from high-level programming languages. To this end, the already rich type system in TAL was enhanced even more, now supporting higher-order and recursive type-constructors and arbitrary data representation. The relation to the IA32 architecture is very close, it even allows TALx86 programs to be assembled and linked with Microsoft's macro assembler, MASM. Before assembling the code, all instructions will be checked for type safe access of data. Type annotations enrich TALx86 code to enforce its type system on top of the IA32 instructions.

In an additional paper [GlMo99], a linking calculus is presented based on the TALx86 assembler. It handles dynamic linking and loading, and cyclic dependencies of modules, properties commonly needed in modern operating systems.

6.2.3 Discussion

This section has shown the wide variety of intermediate languages used for compiler construction. It clearly underpins the statement that one representation does not fit all needs. The focus of our work was translating generic code, which is written at a very high level of abstraction. In order to retain the source level semantics as much as possible, GILF contains constructs that are very close to imperative programming languages. This is on the one hand motivated by technical problems[9], but mainly by the fact that high-level optimizations like algorithm selection require high-level semantic information.

[9] The inability to generate machine code for generic data structures and algorithms with nonuniform representation was discussed at length in chapter 2.

Therefore, the particular mélange of nongeneric constructs available in GILF is of course influenced by the representations closer to source languages, like ANDF and SDE/Slim Binaries. The built-in algorithms and data types of the GILF core library were in part derived from the instruction sets of CIL, the Java Virtual Machine and prominent contemporary processor architectures. But these are the conservative aspects of GILF only, they do not deal with our main problem, namely creating instantiations of generic constructs.

The advantages of load-time instantiation in GILF are shared by some generic Java proposals and generic CIL, thus the structure of their compilation systems adhere to the same philosophy as GILF's system. However, these solutions all enforce instantiation semantics fixed by a high-level type system, either the Java type system or the Common Type System. GILF tries to be a viable target for source languages with varying instantiation semantics and type systems. This is achieved by explicitly storing binding information in the intermediate representation. It is now possible to split the instantiation process into two phases. Instantiation analysis, which is source language dependent, stores its results in GILF's binding elements, and instantiation application simply collects the bindings and instantiates generic constructs according to these information. Furthermore, explicit binding information naturally provide the basis for algorithm selection, because a function symbol can be bound to multiple algorithms at the intermediate level, according to source language rules. Exact type information are mandatory for serialization and reflection, and therefore are available in advanced Java proposals and generic CIL. They also promote algorithm selection in GILF. Based on the requested algorithm instantiation, the selection unit tries to choose the fittest candidate among the available ones[10].

Code sharing between instantiations, an interesting aspect of the generic CIL system, is currently not supported by GILF. This is strongly motivated by the fact that this technique introduces problems when type information is required. Specialization of all instantiations does not share these drawbacks, but may lead to code bloat.

A positive aspect is the coincidence of the development in building compilation systems around virtual machines and intermediate representation, and our approach to this problem, because they share many characteristics. Thus, further work can concentrate on integrating ideas unique to GILF into these systems and otherwise exploit work alreay invested into their other components.

Recapitulating, GILF offers an versatile intermediate representation for targeting by various generic source languages and directly stimulates experimenting with different instantiation semantics. This is adequate for research in genericity, which is still evolving at a rapid pace.

[10] Of course, algorithm selection can be determined also by factors other than instantiation type arguments.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

The strong acceptance of the Standard Template Library and its inclusion into the International C++ Standard, as well as the emergence of other generic libraries, not only in C++, has proven that genericity is an important and useful language paradigm. The reason for the success of generic programming is the fact that it achieves a stellar goal of software engineering: code reuse, and this without sacrificing efficiency. What is also remarkable is the strong interest of research in the field of genericity in stating and checking the requirements for valid instantiations of generic components.

In this thesis we proposed a solution to the problem that plague current generic programming languages with support for heterogeneous data and code representation. These languages require generic libraries to be distributed as source code. The solution is based on the idea that a compiler for a generic language should be separated into two mostly independent parts, the compiler front-end and the back-end. The front-end is responsible for checking the requirements and constraints of generic components during instantiation analysis, as well as performing overload resolution and other high-level semantic operations. All the decisions of the front-end are propagated to the back-end in an intermediate language by making bindings of the instantiation parameters explicit. The back-end will generated nongeneric instances of algorithms and data structures during instantiation application based on the information present in GILF's binding section. This way, GILF can act as target of generic languages with varying semantics.

7.2 Future Work

The SUCHTHAT project, the context in which our work evolved, was initiated to create a generic programming language and environment for implementing a generic library of computer algebra algorithms called WLOG. Once the SUCHTHAT environment is completed with GILF as its back-end, and extensive tests with the library can be performed, fine tuning the constructs present in GILF will start.

However, the GILF system is intended as a conceptual frame for further research in compiler construction and run-time systems related to generic programming languages. The diversity of these fields made it virtually impossible to deal with all aspects of such software tools in an adequate manner, thus providing incentive for deeper investigations in many areas. We want to list those that seem to offer the most interesting perspectives.

- The interface to algorithm selection in GILF is a novel feature in intermediate representations. Efficiently integrating an algorithm selection unit with run-time instantiation is a major undertaking. The same holds true for strategies that guide algorithm selection. Load-time instantiation can rely on instantiation arguments and the calling context to determine a fittest algorithm, optionally augmented by profiling data gathered in previous runs. Run-time instantiation can even access actual values at algorithm calls, but has to meet more stringent constraints in order to avoid increased run-time due to the overhead introduced by algorithm selection.

- The separation of compiler front-end and back-end by means of an intermediate language has become quite popular in recent years, the most prominent examples are Java Bytecode and Microsoft's Common Intermediate Language (CIL). It would be interesting to exploit the progress made in just-in-time [GrMi00] and dynamic compilation techniques [PoHsEn+99] [Eng96] [Kis99] [PhChEg02] [GrMoPh+00], and integrate our ideas for instantiation management into the virtual machines. For both mentioned intermediate languages exist open source implementations which provide an excellent platform for experimentation.

- GILF in its current form lacks some features that would be desirable but were omitted due to design and time constraints. It supports algorithms and record types, but does not have an integrated object model. This could be an interesting addition to GILF, but devising a flexible object model that supports all major techniques of static and dynamic method dispatch in a language neutral way is far from trivial. Another issue are exceptions, however their integration seems to pose less substantial problems. Work in this area has shown that most concerns do not interfere with genericity.

- Nowadays intermediate languages are considered for mobile computing and distributed applications. In this context security becomes a major concern in intermediate language design. Signature based approaches for authorization and enforcing security are necessary, and the XML Signature Working Group is making progress in this direction. Their work is an attractive candidate for inclusion in XGILF.

There is one final aspect that deserves further work. Although GILF enables one to distribute a generic library without making its defining parts available in source code, a front-end still has to access the declarations of the generic components in the library for checking the validity of instantiations. These declarations can be either integrated into GILF files or be made available externally. The content of these information is of course highly front-end language dependent, and therefore a case by case decision seems necessary. Otherwise, it would be a great achievement if GILF could be extended with a flexible mechanism that eases the provision of declarations, containing concept requirements and algorithmic constraints. This mechanism will then act like precompiled headers for generic concepts.

Appendix A

The Utility Library

In chapter 5 we discussed the GILF system prototype and its implementation. Here, we will present the auxiliary library `libutility` that contains additional facilities used by the prototype implementation. Code in `libutility` is not directly related to GILF system's application logic and is therefore factored out into a self-contained library.

A.1 Representing Nodes with Properties

GILF is a tree data structure at its core. XGILF, its XML based external representation, makes this visually apparent when displayed in an XML editor or browser. All nodes can be expanded and collapsed, allowing for a comfortable examination of the intermediate code. Thus, we need data structures for holding an internal representation of GILF trees during the system's runtime.

We have split the relevant data structures in two parts. The utility library holds general classes that enable us to store tree-like data, whereas the GILF library contains extensions of these classes specialized for GILF's specific application behavior. The approach is a variation of the scheme presented in [SiWe99].

A.1.1 Nodes

Storing nodes and child nodes is achieved with two class templates. Class `Node` holds the information present in every node, which is of type `DataType`, its first template parameter. `DataType` has to meet no requirements. This gives us the flexibility to store arbitrary data in a node. Notice that `data`, the member used to represent information in the node, is public and therefore can be read and modified freely. This is motivated by the fact that `Node`'s sole purpose is accessing this member, and this should be as convenient as possible. Any kind of encapsulation is delegated to type `DataType`.

⟨Listing A.1: Class Template `Node`⟩ ≡

```
template <typename DataType, class CoreType = Node_Core<> >
class Node : public CoreType
{

public: // Types.
  typedef DataType data_type;
  typedef CoreType core_type;

public: // Members.
  data_type data; // The data/information stored inside the node.
```

```
}; // class Node
```

Code extracted from file utility/Node.hpp, lines 97 to 108.

Node has a second template parameter called NodeCore, from which it inherits publicly. NodeCore is a concept that specifies all aspects that deal with handling child nodes. We examine the requirements on models of NodeCore by looking at the default implementation Node_Core. It has two template parameters, both are template template parameters. The first one must be bound to a model of a STL conformant sequence container. It defaults to deque from the C++ Standard Library. Child nodes will be stored inside a container of this type, which manifests in the private member m_children. For a discussion of the second template parameter, see [SiWe99].

⟨Listing A.2: Class Template Node_Core⟩ ≡

```
template <template <typename, class> class SequenceType = std::deque,
          template <typename> class AllocatorType = std::allocator>
class Node_Core
{

public: // Types.
  <see Listing A.3 on page 125>

public: // Methods.
  <see Listing A.4 on page 125>

private: // Members.
  // The sequence of child nodes.
  sequence_type m_children;

}; // class Node_Core
```

Code extracted from file utility/Node.hpp, lines 46 to 86.

The two public sections of Node_Core represent the requirements on models of the concept NodeCore. We begin with the public type definitions that have to be present. The first one specifies the node type stored inside the private member mentioned above. In our implementation, we use shared pointers from the Boost library to hold nodes. This has two reason. Pointers allow nodes to act polymorphically, and shared pointers take care of the nodes' memory management. The second type definition, sequence_type, exports the type of the used sequence container. Finally, size_type exports the sequence's type used to declare its size.

⟨Listing A.3: Class Template Node_Core⟩ ≡

```
// The shared pointer type.
typedef boost::shared_ptr<Node_Core> node_type;
// The child nodes sequence type.
typedef SequenceType<node_type, AllocatorType<node_type> > sequence_type;
// The corresponding size_type.
typedef typename sequence_type::size_type size_type;
```

Code extracted from file utility/Node.hpp, lines 53 to 58, referenced in listing A.2.

The second public section enumerates the required methods. Method child_count returns the number of child nodes, child_at returns the requested child node by position, and push_back appends a node at the end of the child node sequence. In general, these methods simply act as forwarding calls to methods that are part of the sequence container's interface. It should be noted that this is a minimal set of methods, but they were sufficient to implement our prototype without great inconveniences.

⟨Listing A.4: Class Template Node_Core⟩ ≡

```
// Return the number of child nodes.
```

```
size_type child_count() const
{
  return m_children.size();
}

// Return child node at position n.
node_type child_at(size_type n) const
{
  return m_children.at(n);
}

// Push node on the back of the child sequence.
void push_back(const node_type& node)
{
  m_children.push_back(node);
}
```

Code extracted from file `utility/Node.hpp`, lines 63 to 79, referenced in listing A.2.

A.1.2 Properties

Now that we have a generic representation of nodes and child nodes, the last issue to resolve is finding a general, yet safe way of storing information in `Node`'s public member `data` (see listing A.1). The requirements on such a class are:

- Access to single properties is type safe, i.e. line numbers should be stored as integers, file names as strings, and so on.
- Related properties can be grouped together, e.g. information relevant for debugging.
- Groups of properties can themselves be combined to completely describe the data stored inside a node.

Our solution to this problems takes advantage of Boost's `tuple` library, Loki's `Tuple` class and typelist facility, and template metaprogramming. It consists of two class templates. Class `Indexed_Property` is a small wrapper around tuple. Tuples in C++ are heterogeneous, fixed-size containers that store elements that can be accessed by index [Jär01]. The wrapper has two template parameters, `IndexType` and `TupleType`. The latter one is the tuple that stores the property's elements. `IndexType` is an enumeration that introduces a named index for every property.

⟨Listing A.5: Class Template `Indexed_Property`⟩ ≡

```
template <typename IndexType, class TupleType>
class Indexed_Property
```

Code extracted from file `utility/Indexed_Properties.hpp`, lines 51 to 52.

The member `m_property_tuple` holds the properties stored inside object instances of class `Indexed_Property`.

⟨Listing A.6: `Indexed_Property`: Members⟩ ≡

```
private: // Members.
  TupleType m_property_tuple; // Tuple that stores the properties.
```

Code extracted from file `utility/Indexed_Properties.hpp`, lines 112 to 113.

The `at()` methods return references to indexed properties. They either return a `const` or a non-`const` reference to the element, depending on the type of access. Notice the `template` keyword in the function call to get(), which is needed if a member template is called that belongs to a template parameter. The traits class access_traits is part of tuple's public interface.

⟨Listing A.7: Indexed_Property: Access Methods⟩ ≡

```
// Return const reference to element at position i.
template <IndexType i>
typename access_traits<typename element<i, TupleType>::type>::const_type
at() const
{
  return m_property_tuple.template get<i>();
}

// Return non-const reference to element at position i for modifying access.
template <IndexType i>
typename access_traits<typename element<i, TupleType>::type>::non_const_type
at()
{
  return m_property_tuple.template get<i>();
}
```

Code extracted from file utility/Indexed_Properties.hpp, lines 72 to 86.

At this point, we have fulfilled the first two requirements for the properties representation, mostly relying on functionality already provided by Boost's tuple class. The real work is making indexed property instances recombineable, but keeping access to elements of different property groups transparent to the user. This is achieved with class Indexed_Properties, the idea is as follows. Indexed_Properties takes $n \geq 1$ template parameters, each representing one group of related properties, thus these parameters are all instances of Indexed_Property. Accessing a property looks exactly the same as for a simple indexed property, method at() is called with an enumeration constant. However, this time the process of getting the reference to the property element involves two steps. First, the enumeration constant's type selects the Indexed_Property that contains the desired property, and thereafter the constant's value selects the tuple's element that holds the typed property (see figure A.1).

The class template declaration reveals one drawback of indexed properties, the number of index property template parameters is fixed[1]. Currently, ten property groups are supported. All template parameters, except the first property type, default to an empty property type from sub-namespace detail. This will be exploited to store only provided property types.

⟨Listing A.8: Class Template Indexed_Properties⟩ ≡

```
template <class PropertyType0,
          class PropertyType1 = detail::Empty_Property1,
          class PropertyType2 = detail::Empty_Property2,
          class PropertyType3 = detail::Empty_Property3,
          class PropertyType4 = detail::Empty_Property4,
          class PropertyType5 = detail::Empty_Property5,
          class PropertyType6 = detail::Empty_Property6,
          class PropertyType7 = detail::Empty_Property7,
          class PropertyType8 = detail::Empty_Property8,
          class PropertyType9 = detail::Empty_Property9>
class Indexed_Properties
```

Code extracted from file utility/Indexed_Properties.hpp, lines 164 to 174.

[1]This restriction may be removed if the Boost Metaprogramming Library becomes part of the official Boost Library

The index type and the tuple type of every property type parameter is exported with a type definition. We only display the first one.

⟨Listing A.9: `Indexed_Properties`: Types⟩ ≡

```
typedef typename PropertyType0::index_type index_type0;
typedef typename PropertyType0::tuple_type tuple_type0;
```

Code extracted from file `utility/Indexed_Properties.hpp`, lines 180 to 181.

Now we start with the template metaprogramming. First, all tuple types are put into typelist `all_tuple_types`. From this list, we remove all types that are empty types, which is the type used in default properties. This way, we compute the typelist `used_tuple_types` that contains only used tuple types.

⟨Listing A.10: `Indexed_Properties`: Typelists (Tuple Types)⟩ ≡

```
// First, we put all the tuple types into a type list.
typedef TYPELIST_10(tuple_type0, tuple_type1, tuple_type2, tuple_type3,
                    tuple_type4, tuple_type5, tuple_type6, tuple_type7,
                    tuple_type8, tuple_type9)
  all_tuple_types;

// Now we create a new type list without all empty tuple types. This
// way we know how many property types are actually used. The length of the
// type list containing no empty tuple types denotes this number.
typedef typename Loki::TL::EraseAll<
  all_tuple_types, boost::tuples::tuple<Loki::EmptyType>
>::Result used_tuple_types;
```

Code extracted from file `utility/Indexed_Properties.hpp`, lines 213 to 224.

The length of the computed typelist is used to generate another typelist which contains only used property types. This is achieved with an extension to Loki, the typelist metafunction `FirstN`, see section A.4.

⟨Listing A.11: `Indexed_Properties`: Typelists (Property Types)⟩ ≡

```
// Second, we put all the property types into a type list.
typedef TYPELIST_10(PropertyType0, PropertyType1, PropertyType2, PropertyType3,
                    PropertyType4, PropertyType5, PropertyType6, PropertyType7,
                    PropertyType8, PropertyType9)
  all_property_types;

// With the information gathered in the first step, we create a type list
// containing only the used property types.
typedef typename Loki::TL::FirstN<
  all_property_types, Loki::TL::Length<used_tuple_types>::value
>::Result used_property_types;
```

Code extracted from file `utility/Indexed_Properties.hpp`, lines 228 to 238.

The tuple that holds the indexed property instances takes this typelist as type parameter for its constructor. We have to use Loki's `Tuple` class, because `tuple` from Boost currently does not support constructing a tuple from a typelist. Template metaprogramming helped to trim down the tuple's memory requirements only to the indexed property types actually present.

⟨Listing A.12: `Indexed_Properties`: Member⟩ ≡

```
private: // Members.
  // Member that holds the properties, created by template generator.
  // Remark: Tuple uses the GenScatterHierarchy class template.
  Loki::Tuple<used_property_types> m_properties;
```

Code extracted from file `utility/Indexed_Properties.hpp`, lines 450 to 453.

What is left to show is how transparent access to elements of the just declared member `m_properties` is implemented in method `at()`. As already mentioned, two aspects of template parameter `i` are exploited. Its type is responsible for selecting the correct specialization of the member template, in the shown example the one that is specialized for `index_type0`. Therefore all Loki typelist operations can operate at position 0. This happens two times, first to access the tuple type that contains the return type of member template `at()`, and also to access the correct index property element in member `m_properties`. The template parameter value is then used to access the correct return type or field, respectively.

⟨Listing A.13: `Indexed_Properties`: Access⟩ ≡

```
template <index_type0 i>
typename access_traits<typename element<
  i, typename Loki::TL::TypeAt<all_tuple_types, 0>::Result >::type
>::const_type at() const
{
  return (Loki::Field<0>(m_properties)).template at<i>();
}
```

Code extracted from file `utility/Indexed_Properties.hpp`, lines 265 to 271.

For examples on usage of the described utility classes, see section 5.3. Figure A.1 displays the two-stage element selection process that takes place when method `at()` is called. In the code snippet at the figure's top, the indexed properties' instance is embedded in a node as described at the beginning of this section.

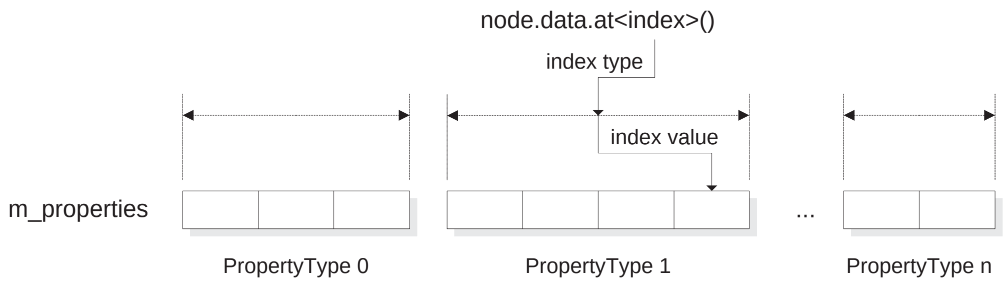

Figure A.1: Two-stage selection process for elements of indexed properties.

Summarizing, class template `Indexed_Properties` provides a facility to store typed objects inside a two-level tuple hierarchy. On most compilers, the tuple type introduces no space overhead compared to a handwritten class with each member specified explicitly. It is useful when the number of properties stored in a node is fixed and does not change frequently. A similiar technique was developed independently from our work by Emily Winch [Win01]. Instead of using names of enumarations, she uses names of empty classes to access elements inside a tuple. However, she offers no facility for combining groups of named objects.

The utility library also contains the class template `Named_Properties`. It also stores typed elements, however it allows dynamic addition and deletion of elements belonging to a type. The idea behind this class is to store one map from the standard library for every type. A member template selects the appropriate map by its template instantiation argument and returns the element depending on a key. `Named_Properties` is not discussed further, because it is not used in the GILF prototype currently.

A.2 A Generic Logging Facility

Logging a running system's activities is one of the major aids in monitoring and diagnosing the system. Our utility library contains a generic logging facility that can be customized to meet different system's demands. The following design criteria guided the implementation:

- A log can be used like an output stream, and all types that provide an output operator can be written to such a log stream.
- A log has a fixed number of categories, each category can be redirected to a different output stream.
- Each category has an associated threshold value that can be changed during the system's run time.
- Each logging message will prepend a string generated in a policy function, for example to print date and time.

Using the `log` class template is quite easy. One has to instantiate a log for the system, and then one can write to the log's categories like to an output stream with the typical output operator notation.

`Log` requires three template parameters. `CategoryCount`, the first one, is a value parameter that specifies the number of categories `Log`'s instantiation should have. Accessing a category outside the valid range `[0, ..., CategoryCount-1]` results in a standard exception of type `runtime_error`. `PrologPolicy` is a policy template parameter. A prolog policy has to provide a function `print_prolog()` that returns a `std::string`. Finally, `ThresholdType` specifies the type of the categories' thresholds. A category's threshold determines if a logging message will be written at all.

⟨Listing A.14: Class Template `Log`⟩ ≡

```
template <std::size_t CategoryCount = 1,
          class PrologPolicy = detail::Log_Default_Prolog_Policy,
          typename ThresholdType = threshold_type>
class Log : public PrologPolicy
```

Code extracted from file `utility/Log.hpp`, lines 91 to 94.

The sole `Log` constructor has two parameters, `default_stream` is the default output stream for all categories, and `default_threshold` is the default threshold for all logging categories. Both value sets can be changed by the appropriate `set` methods later on.

⟨Listing A.15: `Log`: Constructor⟩ ≡

```
Log(std::ostream& default_stream = std::cerr,
    const threshold_type& default_threshold = 1);
```

Code extracted from file `utility/Log.hpp`, lines 113 to 114.

Because each category can have its own associated output stream and threshold, they have to be stored in containers. As the size is fixed, we use arrays. Furthermore, the member `m_current_category` determines the category whose stream will be used for output. Also, this category's threshold will be compared against member `m_current_threshold`.

⟨Listing A.16: `Log`: Members⟩ ≡

```
private: // Members.
  // Holder of the streams for every category.
  std::ostream* m_ostreams[CategoryCount];
  // Holder of the thresholds for every category.
  threshold_type m_thresholds[CategoryCount];
```

```
    // The current category.
    category_count_type m_current_category;
    // The current threshold.
    threshold_type m_current_threshold;
```
Code extracted from file `utility/Log.hpp`, lines 188 to 196.

The workhorse of `Log` is its member template `write`. It checks if the category is valid and the threshold does not suppress output. Output is suppressed if $current\_threshold \geq threshold_{current\_category}$. After these checks, it feeds the argument `x` into the output stream associated with the current logging category.

⟨Listing A.17: `Log`: Method `write`⟩ ≡
```
template <typename T>
Log& write(const T& x);
```
Code extracted from file `utility/Log.hpp`, lines 123 to 124.

The interface most visible to the user however are the application operator and of course the output operator. The output operator simply forwards to member template `write` discussed above. The application operator deserves more elaboration. The parameter `category` selects the logging category for the following message. Parameter `threshold` determines the importance of the message and `skip_prolog` can be used to suppress output of the string generated by the prolog policy.

⟨Listing A.18: `Log`: Application Operator⟩ ≡
```
Log& operator()(category_count_type category = 0,
                threshold_type threshold = 0,
                bool skip_prolog = false);
```
Code extracted from file `utility/Log.hpp`, lines 131 to 133.

There is one technical detail worth mentioning. Streams allow formatting of their output through manipulators. If logging should keep this formatting, the manipulators have to be applied to the output streams. This has to be done to three stream classes, which can be seen in the code listing.

⟨Listing A.19: `Log`: Stream Manipulators⟩ ≡
```
Log& operator<<(std::ostream& (*f)(std::ostream&));
Log& operator<<(std::ios& (*f)(std::ios&));
Log& operator<<(std::ios_base& (*f)(std::ios_base&));
```
Code extracted from file `utility/Log.hpp`, lines 164 to 166.

A simple example will demonstrate the typical usage of class `Log`. A log `l` is instantiated that has three logging categories, uses `cerr` as default output stream, and has a default threshold of two. Category three is then bound to a file stream. Finally, all logging output is wrapped inside a try-catch block.

⟨Listing A.20: `Log`: Testcode⟩ ≡
```
namespace gilf = GILF_Core; // namespace alias
try
{
  // Create instance.
  gilf::Log<3> l(std::cerr, 2);

  // Bind category 2 to a file.
  std::ofstream logfile("test/logfile");
  l.set_stream(2, logfile);

  // Examples.
  l(0,0) << "Hello, world.\n";
  l(1,1) << "Need " << std::setw(10) << 12.7 << std::string(" help!") << std::endl;
  l(0,2) << "more tests " << 4711 << std::endl;
```

```
    l(2,0) << "Category 2 should go to a file " << 123 << std::endl;
    l(2,2) << "seems ok... " << 3.1415926 << std::endl;
  }
  catch (std::exception& e)
  {
    std::cerr << "## std::exception caught ## - " << e.what() << "\n";
  }
```

Code extracted from file test/Log_test.cpp, lines 41 to 61.

The output generated by the example shows the default prolog policy, which prints a message counter, followed by date and time. On the command line, this output is produced:

```
[0] 20:19:09, 07/08/02: Hello, world.
[1] 20:19:09, 07/08/02: Need       12.7 help!
```

A.3 XML Utilities

The code required to handle XML in an XGILF input stream is factored out into the XGILF inflator classes (see appendix B.1). This code relies on XML utility classes and functions located in the utility library.

Our project uses the XML library Xerces, developed by the Apache project. Like most XML libraries, it uses its own string class. We like to use standard C++ components, therefore a small function `to_string` converts a `DOMString` to `string` from the C++ Standard Library. Xerces contains a function that transcodes an XML string into an ASCII character string. With this character string, a `std:string` can be constructed. To avoid memory leaks, the transcode buffer has to be deleted explicitly.

⟨Listing A.21: Convert DOMString to C++ String⟩ ≡

```
std::string to_string(const DOMString& s)
{
  // Transcode DOMString into an ASCII string.
  char *transcode_buffer = s.transcode();
  // Copy this ASCII string into a std::string.
  std::string tmp_string(transcode_buffer); // Copy the buffer.
  // Delete transcode buffer, required by transcode().
  delete [] transcode_buffer;
  return tmp_string;
}
```

Code extracted from file utility/dom/Utility.cpp, lines 117 to 126.

Another important operation on XML nodes is to check whether they are of a given type. In XML terminology, a node has type T if the node is an element node and its tag name is equal to T.

⟨Listing A.22: Check Type of XML DOM Node⟩ ≡

```
bool check_element_type(const DOM_Node& node, const std::string& type)
{
  // Check if node is of element type.
  if (node.getNodeType() != DOM_Node::ELEMENT_NODE)
  {
    return false;
  }
  // Check the element's type.
  const DOM_Element& elem = static_cast<DOM_Element&>(const_cast<DOM_Node&>(node));
  if (!(elem.getTagName().equals(type.c_str())))
```

```
    {
      return false;
    }
    // Passed all tests, return true as result.
    return true;
  }
```
Code extracted from file utility/dom/Utility.cpp, lines 271 to 286.

DOM level 2 [DOM00] introduced NodeIterators and TreeWalkers, which allow selective traversal of a DOM document. If a node will be part of the traversal is specified by NodeFilters, which can be attached to NodeIterators and TreeWalkers. The utility library contains several node filters. We will demonstrate the general procedure for writing a custom node filter with the example of a node filter that accepts nodes whose element types are given by a vector of tag names.

The constructor expects this vector of tag names as input argument and initializes the member m_types with it. The DOM interface requires a NodeFilter to override the virtual function acceptNode, which has an input parameter of type DOM_Node.

⟨Listing A.23: NodeFilter Declaration⟩ ≡

```
class Element_Nodes_By_Types : public DOM_NodeFilter
{

public: // Ctor.
  Element_Nodes_By_Types(const std::vector<std::string>& types)
    : m_types(types) {}

public: // Methods.
  // Required override of acceptNode.
  virtual short acceptNode(const DOM_Node& n) const;

private: // Members.
  std::vector<std::string> m_types; // Types of valid element nodes.

}; // Element_Nodes_By_Types
```
Code extracted from file utility/dom/Utility.hpp, lines 91 to 105.

The implementation of acceptNode is straightforward because of the STL capabilities. First a check is made that we deal with an DOM element node. With algorithm find from the C++ Standard Library the vector of types is compared against the element node's tag name. If a match is found, the node is accepted, otherwise rejected.

⟨Listing A.24: acceptNode Implementation⟩ ≡

```
short Element_Nodes_By_Types::acceptNode(const DOM_Node& n) const
{
  if (n.getNodeType() == DOM_Node::ELEMENT_NODE)
  {
    DOM_Element elem = static_cast<DOM_Element&>(const_cast<DOM_Node&>(n));
    if (find(m_types.begin(), m_types.end(), to_string(elem.getTagName()))
        != m_types.end())
    {
      return DOM_NodeFilter::FILTER_ACCEPT;
    }
  }
  return DOM_NodeFilter::FILTER_REJECT;
}
```
Code extracted from file utility/dom/Utility.cpp, lines 161 to 173.

The XML utilities also include functions for writing a DOM document to an output stream. This is necessitated by the fact that the DOM recommendations does not contain facilities for this task. At level 3, DOM will contain the long awaited Load and Save Specification. However, at the time of this writing (Summer 2002), only a working draft of this specification exists [W3Tr]. We provide an output operator for DOM nodes and DOM strings. With these operators, XML nodes and their attributes can be written in a common C++ style.

⟨Listing A.25: DOM Output Operators⟩ ≡

```
std::ostream& operator<<(std::ostream& target, const DOMString& toWrite);
std::ostream& operator<<(std::ostream& target, const DOM_Node& toWrite);
```

Code extracted from file `utility/dom/Stream_Helper.hpp`, lines 85 to 86.

A.4 Loki Extensions and Modifications

Loki, the generic component library developed by Andrei Alexandrescu [Ale01], is designed to be customizable to the concrete application's needs. This approach has proven very successful, and Loki is a major building block of the GILF system implementation. Nevertheless, in some places it lacks functionality or is not as flexible as desired. In these cases, we provided extensions for, or modifications to Loki.

A.4.1 Factory and Smart Pointers

The factory design pattern [GaHeJo+95] is a useful aid in situations where class objects belonging to a class hierarchy have to be created based on external type identifiers. For example, GILF nodes in external representation contain a type identifier such that in the deserialization process specific nodes can be identified relying solely on this information. In Loki, the `Factory` class template provides a generic component that implements the factory design pattern ([Ale01], chapter 8).

According to Loki's design concept, most aspects of the factory's behavior can be customized with template parameters that represent policies. However, at one point `Factory` suffers from over-specification. The return type of the member method `CreateObject` is `AbstractProduct*`, a pointer type. This effectively precludes object factories from returning smart pointers as abstract product, because these are usually returned as value types, not as pointers. The `Factory` that is part of `libutility` removes this restriction, which becomes apparent when looking at its method `create`:

⟨Listing A.26: Factory Method `create`⟩ ≡

```
/// Ask for product creation by identifier.
AbstractProduct create(const Identifier& id);
```

Code extracted from file `utility/Factory.hpp`, lines 72 to 73.

In GILF, the internal representation uses smart pointers for memory management of its tree-like node structure, see chapter 5. Therefore, this modification of Loki's `Factory` class template was essential for our implementation.

A.4.2 Truncating a Typelist

The algorithm for truncating a typelist to its first n elements exemplifies the functional style of template metaprogramming now common in advanced C++ libraries. All typelist related algorithms reside in namespace `TL`, which itself is part of namespace `Loki`.

⟨Listing A.27: Template Metafunction `FirstN`⟩ ≡

```
namespace TL
{
  // FirstN class template declarartion.
  <see Listing A.28 on page 135>

  // Partial template specializations that end recursion.
  <see Listing A.29 on page 135>

  // General case: take type at head position and continue for n-1 elements.
  <see Listing A.30 on page 135>
} // namespace TL
```

Code extracted from file `utility/Loki_Extension.hpp`, lines 83 to 113.

First, the declaration of the general class template `FirstN` is given. It has two template parameters, which represent the input parameters of the template metafunction `FirstN`. `TList` is the input typelist, and `N` is the number of elements to which `TList` should be truncated. If this number is larger than the typelist's size, a compilation error will occur.

⟨Listing A.28: `FirstN`: Declaration⟩ ≡

```
template <class TList, unsigned int N> struct FirstN;
```

Code extracted from file `utility/Loki_Extension.hpp`, line 87, referenced in listing A.27.

Then, the two template specializations that end the recursion are given. The first one handles empty typelists by simply creating a type definition `Result` that is set to `Null-Type`. The second one handles an argument value of 1 for the input parameter `N`. In this case, the type definition `Result` is set to the head of the input typelist.

⟨Listing A.29: `FirstN`: Terminate Recursion⟩ ≡

```
// Empty typelists.
template <unsigned int N> struct FirstN<NullType, N>
{
  typedef NullType Result;
};
// End recursion for N = 1 by returning the head.
template <class Head, class Tail>
struct FirstN<Typelist<Head, Tail>, 1>
{
  typedef TYPELIST_1(Head) Result;
};
```

Code extracted from file `utility/Loki_Extension.hpp`, lines 92 to 102, referenced in listing A.27.

Finally, the most general template specialization handles the recursive invocation of the `FirstN` class template pattern matching. It sets the typelist `Result` to the head of `TList`, appended with the tail of `TList`, truncated to $N-1$ elements.

⟨Listing A.30: `FirstN`: Recursion⟩ ≡

```
template <class Head, class Tail, unsigned int N>
struct FirstN<Typelist<Head, Tail>, N>
{
  typedef Typelist<Head, typename FirstN<Tail, N-1>::Result > Result;
};
```

Code extracted from file `utility/Loki_Extension.hpp`, lines 107 to 111, referenced in listing A.27.

A.4.3 Visiting Subnodes

Loki's `Visitor` class template represents the visitor design pattern [GaHeJo$^+$95]. In the GILF prototype, transformations after deserialization of the external representation are performed with visitors that are derived from the Loki `Visitor` component. A common task is to restrict the visitation of subnodes to those of one type. This is exactly the purpose of the function template `visit_children`.

The input parameter `node` is the node whose children will be visited, and the template parameter `ChildType` specifies the type to which the visitation should be restricted. Finally, `visitor` is a reference to the visitor whose `Visit` methods will be called.

⟨Listing A.31: Function Template `visit_children`⟩ ≡

```
template <class ChildType>
void visit_children(GILF_Core::GILF_Node& node, BaseVisitor& visitor)
{
  // Get the sequence type for GILF nodes.
  typedef GILF_Core::GILF_Node::sequence_type seq_type;

  // Look for subnodes of type ChildType.
  seq_type nodes;
  node.template get_children<ChildType>(nodes);
  for (seq_type::iterator it = nodes.begin(); it != nodes.end(); ++it)
  {
    (*it)->Accept(visitor);
  }
}
```

Code extracted from file `utility/Loki_Extension.hpp`, lines 133 to 146.

The algorithm proceeds as follows. First, a sequence container of `GILF_Nodes` is filled with all subnodes of `node` that are of type `ChildType`. This relies on the function template `get_children`, which is part of the `Node` class template (see section A.1). Then, a loop iterates over all these nodes and dispatches based on their type with the virtual method `Accept`.

Appendix B

Auxiliary `libgilf` Components

B.1 Transforming External into Internal Representation

A major building block of the GILF system is the deserialization framework that is responsible for transformation of an external into the internal GILF representation. The following design constraints describe the cornerstones of the framework.

Multiple Input Sources Different input streams can act as input source, for example files, http connections, and so on.

Multiple External Representations The facilities provided by the GILF library should support multiple external representations. A GILF input source can designate its serialization format and the GILF system should be able to handle different formats transparently.

The goal of the deserialization framework is to provide the GILF library user with a simple application interface whose outcome is a valid GILF node hierarchy.

B.1.1 The Application Interface

The interface presented to the library user consists primarily of two classes.

`Input_Source` This class is intended as the high-level interface for accessing GILF input sources. Different constructors control the selection of the input stream. Based on the information found in the prolog (see section 4.2) which describe the used serialization protocol, a corresponding accessor object is created.

`Accessor` The accessor encapsulates the access to the raw data using the serialization protocol of the external format.

Class `Input_Source` In order to initialize reading from a file stream, we provide a constructor that takes a file name as input parameter.

⟨Listing B.1: Input Source: Constructor for Files⟩ ≡

```
Input_Source(const std::string& file_name);
```

Code extracted from file `gilf/Input_Source.hpp`, line 66.

Providing constructors for various input streams is the key to fulfilling our first requirement on the deserialization framework. We will show the basic functioning of such a constructor using the example from above. The constructor for files performs the following steps:

1. It tries to open the specified input file, honoring a list of search paths while locating the file.
2. The file is scanned for a GILF processing instruction.
3. The serialization protocol and its version are extracted from the processing instruction.
4. Based on the protocol information, the appropriate accessor object is created.
5. If an error condition occurred in the above steps, a C++ standard exception of type `runtime_error` is thrown. This can happen if the file is not found, none or an incomplete processing instruction is present, or the protocol is not supported.

Step 4 is the key to fulfilling our second design constraint. By creating an accessor that is able to handle the detected protocol, every input source can be encoded with a different protocol. This code listing shows how an accessor is created for XGILF inflation:

⟨Listing B.2: Input Source: Creating an XGILF Accessor⟩ ≡

```
if (m_protocol == "xgilf")
{
  g_log(lc_std, lt_medium)
    << "Creating xgilf::Accessor with file: " << this_name << ".\n";
  m_accessor.reset(new xgilf::Accessor(this_name));
}
```

Code extracted from file `gilf/Input_Source.cpp`, lines 171 to 176.

The created accessor will be assigned to the data member `m_accessor`, and the accessor is residing in namespace `xgilf` in which all components are defined that are responsible for deserialization of XGILF nodes. This will be discussed in conjunction with class `Accessor`. Member `m_accessor` is a shared pointer to an accessor.

⟨Listing B.3: Input Source: Accessor Members Type⟩ ≡

```
typedef boost::shared_ptr<Accessor> accessor_type;
```

Code extracted from file `gilf/Input_Source.hpp`, line 71.

The most important method of `Input_Source` returns a constant reference to the source's accessor object. With this object the deserialization process can be started. Method `add_search_path` adds the string parameter `path` to the list of search paths that will be scanned while locating the input source file.

⟨Listing B.4: Input Source: Public Methods⟩ ≡

```
// Access to the input source's instantiated accessor object.
const accessor_type& accessor() const;

// Add path to the search paths.
static void add_search_path(const std::string& path);
```

Code extracted from file `gilf/Input_Source.hpp`, lines 76 to 80.

Class `Accessor` The purpose of class `Accessor` is to hide the actual protocol handling behind a uniform interface. As we have seen in the last section, an `Input_Source` object holds an accessor, which is created depending on the information in the prolog of the GILF input stream. This reveals the nature of class `Accessor`, which is intended as pure base class. Each protocol will have its own subclass of `Accessor`, and an `Input_Source` will keep a pointer to one of these subclasses.

The goal of deserializing an external format is receiving a GILF node that represents the external format's content. Therefore, an accessor has to provide a method that returns a GILF node. The remaining issue is at what granularity extracting parts of the external GILF input stream should be allowed. Currently, an accessor supports specifying units. Adding other methods that allow extracting functions, algorithms, and so on pose no conceptual challenge.

⟨Listing B.5: Class Accessor⟩ ≡

```
class Accessor
{

public: // Ctors & dtor.
  // Virtual destructor.
  virtual ~Accessor() {};

public: // Methods.
  GILF_Node::sptr_type get_unit(const std::string& unit_id) const;

private: // Virtual methods.
  virtual GILF_Node::sptr_type do_get_unit(const std::string& unit_id) const = 0;

}; // class Accessor
```

Code extracted from file `gilf/Accessor.hpp`, lines 39 to 52.

The `Accessor` base class declaration follows the nonvirtual interface idiom [Sut01], which mandates separating the interface of a class from its implementation details, like virtual methods for hooks. Therefore, method `get_unit` is the nonvirtual, public interface for accessors, and subclasses override the virtual, but private method `do_get_unit`. The implementation of the public method is simply a forward call to the virtual hook.

⟨Listing B.6: Class Accessor: Forwarding to Virtual Method⟩ ≡

```
GILF_Node::sptr_type Accessor::get_unit(const std::string& unit_id) const
{
  return do_get_unit(unit_id);
}
```

Code extracted from file `gilf/Accessor.cpp`, lines 31 to 34.

Now that the general application interface and the interaction between `Input_Source` and accessors have been discussed, we will show how to implement a specific accessor.

B.1.2 Implementing an Accessor

We will look at the accessor for XGILF input sources. All components involved in deserializing an XGILF stream are located in namespace `xgilf`. The central role in this process plays the subclass of base class `Accessor`.

⟨Listing B.7: XGILF Accessor: Declaration⟩ ≡

```
class Accessor : public GILF_Core::Accessor
```

Code extracted from file `gilf/xgilf/Accessor.hpp`, line 55.

It is also called `Accessor`, but because it resides in namespace `xgilf`, no name clash results. A constructor is present that mirrors the input source constructor for files, it takes the same argument, a file name. It is responsible for initializing the XML system that will be used to process the file. In turn, the virtual destructor has to terminate the XML system. `Accessor` contains the static counter `m_initialized` that is used to prevent multiple initializations and terminations of the XML system.

⟨Listing B.8: XGILF Accessor: Constructor and Destructor⟩ ≡

```
Accessor(const std::string& file_name);
virtual ~Accessor();
```

Code extracted from file `gilf/xgilf/Accessor.hpp`, lines 62 to 63.

We employ the Xerces C++ library from the Apache XML project [ASF], in particular its DOM [DOM00] implementation. The constructor also sets up a DOM parser that will be used to operate on the XML input. This also comprises checking the XML input for well-formedness, and optionally validating it against the XGILF DTD (see chapter 4).

⟨Listing B.9: XGILF Accessor: DOM Parser Member⟩ ≡

```
// DOM parser for working on the document.
boost::scoped_ptr<DOMParser> m_DOMparser;
```

Code extracted from file `gilf/xgilf/Accessor.hpp`, lines 78 to 79.

All these prerequisites show that a subclass of the base class `Accessor` hides all the details of handling a specific GILF serialization format inside the library. However, the major task in writing an accessor still remains to be performed, which is implementing the virtual method `do_get_unit`. It expects a unit identifier and returns a GILF node containing the deserialized unit. The task can be split up in two steps.

1. The starting point of the unit specified by the identifier has to be found in the located XGILF document.

2. Starting at the found unit element, we have to expand its properties. Further on, its child elements and their properties have to be expanded recursively.

The first step is almost trivial with a DOM parser, with method `getElementsByTagName` one can retrieve a list of all units in the XGILF document, and this list can then be traversed, comparing its `id` attribute against the given unit identifier.

⟨Listing B.10: XGILF Accessor: Get Node List of Unit Elements⟩ ≡

```
DOM_NodeList nl = (m_DOMparser->getDocument()).getElementsByTagName("unit");
unsigned int nll = nl.getLength();
g_log(lc_std, lt_low) << nll << " unit(s) found." << "\n";

// Look for unit by id.
DOM_Node unit_node;
for (size_t i = 0; i < nll; i++)
{
  DOM_Node mynode = nl.item(i);
  if (((DOM_Element &) mynode).getAttribute("id").equals(unit_id.c_str()))
  {
    unit_node = mynode;
    break;
  }
}
```

Code extracted from file `gilf/xgilf/Accessor.cpp`, lines 180 to 194.

Step two is what deserializing XGILF elements is all about. Expanding properties is performed by reading the XML element's attributes, which are strings, and convert them to the type of the corresponding properties in GILF's internal representation (see section 5.3). For this purpose, special function templates, so called *property converters*, are defined.

Expanding child elements involves traversing these child elements and creating GILF nodes based on the elements' types. Then, their attributes have to be converted to typed properties, also. This recursive process is controlled with *inflators*.

Property Converters XGILF property converters are function templates that all have the same declaration, except for the function name. They have one template parameter NodeSptrType, whose valid bindings are to shared pointers of GILF nodes that contain the property which is the target of the conversion[1]. The first value parameter is such a shared pointer to a GILF node holding the target property of the conversion, and the second value parameter is the XML element which holds the attribute that will be subject to conversion. This listing shows the declaration of a property converter that handles value attributes (see section 4.10.1):

⟨Listing B.11: XGILF Value Property Converter: Declaration⟩ ≡

```
template <class NodeSptrType>
void convert_value(NodeSptrType& node, const DOM_Element& element);
```

Code extracted from file gilf/xgilf/Property_Converter.hpp, lines 102 to 103.

We elaborated on two details from the implementation of this property converter. The next listings shows how the value property is accessed in a GILF node using method at of class Indexed_Properties, which is used as data member GILF nodes. We get a reference to the value property in order to manipulate its content directly in the function body. The listing also displays how the kind attribute of the XGILF element is read into a standard string.

⟨Listing B.12: XGILF Value Property Converter: Accessing a Property⟩ ≡

```
// Get reference to value property.
value_iprop::type& value = node->data.template at<value_iprop>();
// Get the kind attribute.
std::string kind_attr = to_string(element.getAttribute(c_attr_value_kind));
```

Code extracted from file gilf/xgilf/Property_Converter.cpp, lines 341 to 344.

The specification of XGILF element val states that values of different types can be stored in the corresponding attribute. Therefore, we store the value property in a struct holding a Boost any container and a kind enumeration. The enumeration allows us to cast the any object back to its contained type, which can be seen in the logging output line at the end of the listing. The listing shows how a value struct is filled in the case of a floating point value.

⟨Listing B.13: XGILF Value Property Converter: Converting a Floating Point Value⟩ ≡

```
value.kind = kind_real;
// Convert the attribute string to a double.
char ** end_ptr = 0;
value.value = std::strtod(value_attr.c_str(), end_ptr);
g_log(lc_debug, lt_low) << "value[real] detected: "
                        << boost::any_cast<double>(value.value) << ".\n";
```

Code extracted from file gilf/xgilf/Property_Converter.cpp, lines 464 to 469.

Inflators Property converters perform the task of transforming a node's serialized properties into typed properties in GILF's internal representation, in our case study these properties are made persistent as XML attributes. In order to deserialize the whole content of a node, this has to be done recursively for the child nodes. This is exactly the objective of inflators. Looking back at the XGILF accessor, we can see how an empty unit node is inflated using the corresponding inflator:

⟨Listing B.14: XGILF Accessor: Inflate Unit Element⟩ ≡

```
// First, we create an empty unit GILF node.
GILF_Node::sptr_type unit = gilf::g_node_factory->create(c_node_id_unit);
// Second, we create a unit inflator.
boost::shared_ptr<Inflator> unit_inflator =
```

[1] Violation of this requirement is reported at compile time.

```
  g_inflator_factory->create(c_node_id_unit);
// Finally, we inflate the empty unit using the inflator.
unit_inflator->inflate(unit, unit_node);
```
Code extracted from file `gilf/xgilf/Accessor.cpp`, lines 211 to 217.

First, an empty unit node is created with the global node `g_node_factory`, using a string constant as type identifier. In the same way, a unit inflator is created with the global inflator factory[2].

Now, we can call the public method `inflate` on the unit inflator, which will deserialize the content of the detected XGILF unit element `unit_node` (see listing B.10) into the empty node `unit`. The base class `Inflator` also employs the nonvirtual interface idiom, thus the public method `inflate` forwards to the virtual, private implementation method `do_inflate`.

⟨Listing B.15: Class `Inflator`: Declaration⟩ ≡
```
class Inflator
{
public:
  void inflate(GILF_Node::sptr_type& node, const DOM_Node& dom_node,
               bool recursive = true) const;
private:
  virtual void do_inflate(GILF_Node::sptr_type& node, const DOM_Node& dom_node,
                          bool recursive) const = 0;
}; // class Inflator
```
Code extracted from file `gilf/xgilf/Inflator.hpp`, lines 54 to 62.

A third argument to method `inflate` can restrict inflation to direct child nodes. Every GILF node has a twin inflator. The knowledge present in an inflator is what properties have to be set and what kind of child nodes have to be inflated. The final question that remains to be answered is how does one define a specific inflator? We take a pragmatic approach and examine the XGILF inflator of the algorithm element.

⟨Listing B.16: Class `Algorithm_Inflator`: Type Definition⟩ ≡
```
typedef Inflator_Core<detail::Algorithm_Inflator_Traits> Algorithm_Inflator;
```
Code extracted from file `gilf/xgilf/Algorithm_Inflator.hpp`, line 124.

The algorithm inflator is just a type definition to an instantiation of class `Inflator_Core`. This class is a traits and policy based skeleton for inflators. In general, a new inflator is created by defining the corresponding `Inflator_Traits` class and providing a type definition like in the code snippet presented above.

An instantiation of `Inflator_Core` is a valid inflator, because it inherits from the base class `Inflator`. The adaptability is reached by also inheriting from a traits class.

⟨Listing B.17: Class `Inflator_Core`: Declaration⟩ ≡
```
template <class InflatorTraits>
class Inflator_Core : public Inflator, public InflatorTraits
```
Code extracted from file `gilf/xgilf/Inflator_Core.hpp`, lines 78 to 79.

Inflator traits consist of three type definitions and two methods. The type definition `node_type` specifies the inflation's target node type, `node_sptr_type` the type of a shared pointer to such a node. Furthermore, `node_id_type` specifies the type of identifiers used to query the inflator factory. As GILF and XGILF identifiers are both strings, they are of the same type.

⟨Listing B.18: Algorithm Inflator Traits: Type Definitions⟩ ≡
```
typedef GILF_Core::Algorithm node_type;
```

[2]The inflator factory is an almost identical twin of the node factory (see section 5.3.3), but it returns inflators instead GILF nodes.

```
typedef boost::shared_ptr<node_type> node_sptr_type;
typedef GILF_Core::node_id_type node_id_type;
```

Code extracted from file gilf/xgilf/Algorithm_Inflator.hpp, lines 72 to 74.

More interesting are the two protected policy methods that every inflator inherits from its inflator traits. Implementing these methods is the main liability for adding an XGILF inflator.

⟨Listing B.19: Algorithm Inflator Traits: Method Declarations⟩ ≡

```
// Set values of GILF node node_sptr by reading the XML element.
void set_node_values(node_sptr_type& node_sptr,
                     const DOM_Element& element) const;
// Inflate all children of node node_sptr by reading the XML element.
void inflate_children(node_sptr_type& node_sptr,
                      const DOM_Element& element) const;
```

Code extracted from file gilf/xgilf/Algorithm_Inflator.hpp, lines 80 to 85.

Method set_node_values sets the properties of the GILF node pointed to by node_sptr. To this end, it calls property converters which operate on the XGILF element. Implementing set_node_values boils down to a sequence of calls to the property converters, for example this call sets the algorithm's identifier and reference property:

⟨Listing B.20: Algorithm Inflator Traits: Setting the Identifier Property⟩ ≡

```
convert_id(node_sptr, element);
node_sptr->data.at<idref_iprop>() =
  to_string(element.getAttribute(c_attr_idref));
```

Code extracted from file gilf/xgilf/Algorithm_Inflator.cpp, lines 73 to 75.

The identifier property converter adds identifiers of global nodes to the global symbol table (see section 5.4.2). These identifiers are all prefixed with the unit's identifier, therefore the detection of appropriate nodes is straightforward.

The second policy method inflate_children is responsible for inflating child nodes of the XGILF element and appending them to node_sptr. The core of this work is traversing the element's child node list and performing inflation recursively. The inflator core provides auxiliary function templates that allow selective inflation based on node type identifiers. Thus, an inflate_children implementation is simplified to a sequence of consistency checks and calls to these auxiliary functions. We will have a look at one of those function templates:

⟨Listing B.21: Inflator Core: Auxiliary Method for Child Node Inflation⟩ ≡

```
template <typename NodeSptrType, typename NodeIdType>
void inflate_typed_children(NodeSptrType& node_sptr,
                            const DOM_Element& element,
                            NodeIdType node_id,
                            bool single = false);
```

Code extracted from file gilf/xgilf/Inflator_Core.hpp, lines 106 to 110.

This function template inflates all child nodes of element whose node type identifier is equal to node_id. If the boolean parameter single is true, then only the first child node is inflated. All inflated child nodes are appended to the GILF node denoted by the shared pointer node_sptr. The template parameters are typically deduced from the argument's types at the function's calling site. There exists another overload of the function template that expects a list of node type identifiers, which allows inflation of nodes of various types.

However, the main achievement of Inflator_Core is its generic implementation of the virtual method do_inflate (see listing B.15). The function body relies on type information from the inflator traits and its actual behavior is adapted by the policy methods set_node_values and inflate_children. Implementing a generic version of do_inflate in terms of these type information and policy methods proceeds as follows:

1. The node gilf_node is cast to the actual subclass it holds using the type information present in the traits class. The same is done with the DOM node, which is cast to an DOM element.

2. With full type information present, the properties of the node are set using the policy method set_node_values.

3. The last step is to inflate the child nodes using the policy method inflate_chidren. This happens only if the boolean flag recursive is true.

B.1.3 Roundup

Figure B.1 is an overview on the central classes involved in the serialization framework, and their relationships. It is restricted to the example used in the preceding deliberation of the framework, which deals with inflating XGILF elements.

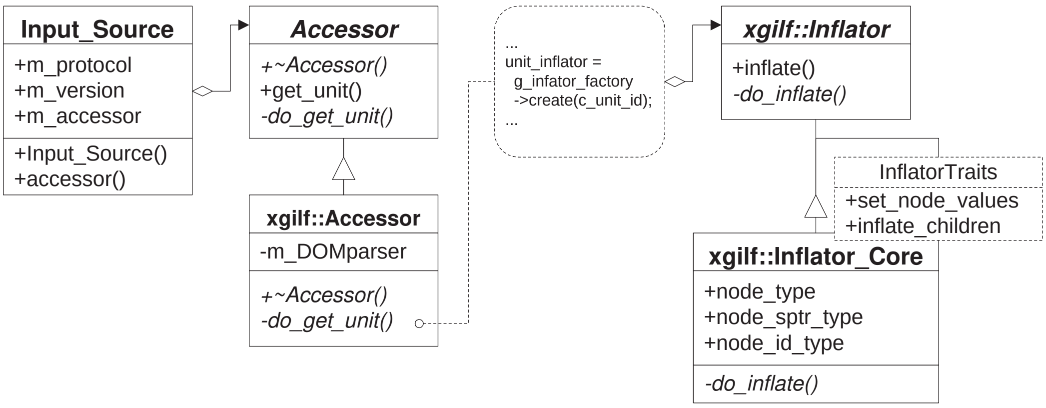

Figure B.1: Class relationships inside the serialization framework.

Finally, we list desirable actions in the framework and what is needed to carry them out, and where to integrate the changes.

Adding a different input stream. One has to add a constructor to class Input_Source that specifies the new input stream. Likewise, a constructor that matches this one has to be added to Accessor classes that support this kind of input stream.

Adding a new inflator. The corresponding traits class has to be defined. This is divided into defining three type definitions and implementing two policy methods.

Supporting a different serialization protocol. At the top level, an accessor subclass has to be implemented. In general, this includes implementing a new inflator hierarchy. This requires considerable work, but most aspects can be automated, as was shown for XGILF inflators with class Inflator_Core and the related auxiliary function templates.

B.2 Code Generation

We explained how type and function bindings transformed with an `Instantiator` result in monomorphic algorithm and data structure nodes in section 5.6. At this point, for the first time we can actually generate real machine code from our intermediate representation. Code generation is defined by two major tasks.

1. Translating GILF's statement nodes into the corresponding code sequences in the target language or machine architecture.

2. Providing target code for all built-in algorithms and data structures.

The basis for code generation is the global instance table that was filled with monomorphic instances of algorithms and data structures using our instantiation engine. A code generator that participates in the GILF system has to translate these nodes into the desired target language. Most of the work required for implementing such a code generator poses no conceptual challenges, because we have already eliminated the generic parts with the transformations applied during instantiation application. Only if we want to support run-time instantiation, special dispatch code is necessary that initiates just-in-time compilation.

The GILF prototype contains a translator from GILF's internal representation to C++. However, we generate completely nongeneric C++ code, the instantiation engine of C++ will not be used. This ensures that no idiosyncrasies of C++'s instantiation semantics will interfere with the front-end semantics. This was one of the goals of the GILF project. We will now briefly describe the prototype's code generation component.

B.2.1 Visitation Graphs

A data structure's visitation graph is quite simple. Its root node is a `data` node, and built-in data structures can even stop here. User defined data structures also have to walk their element subnodes. Such data structures require the creation of either a record or union compound, whose field types are given by algorithm or data structure designators. Algorithm designators result in synthesis of typed function pointers, whereas data structure designators create ordinary entries in the corresponding record or union compound.

Figure B.2: Visitation graph of visitor `Emitter` for data structures.

Like for data structures, built-in algorithms simply process the algorithm node and synthesize the correct code, optinally including a library that implements the required GILF core library features. Visitation of algorithm nodes by an emitter with user defined content proceeds in three major steps.

First, the all value parameters are visited and the algorithm's signature is emitted. Next, we assemble the algorithm's local symbol table by traversing its `store` node and its subnodes. This way we will generate code for all variables and constants used inside the algorithm body. Finally, the statements describing the algorithm's behavior are visited and corresponding code is emitted. The most elaborate node to handle in this hierarchy is the `call` node which represents a function call. A function call can be either direct, in this case the call target is given by an algorithm designator, or indirect, in this case the target is denoted by a variable or constant designator. If the target language does not support named parameters, we have to visit the algorithm's function node and traverse its parameters. We will then impose some order on the parameters. Finally, we can generate code for the function call by visiting the parameter bindings, which will specify this call's arguments. They will be emitted in the predefined order.

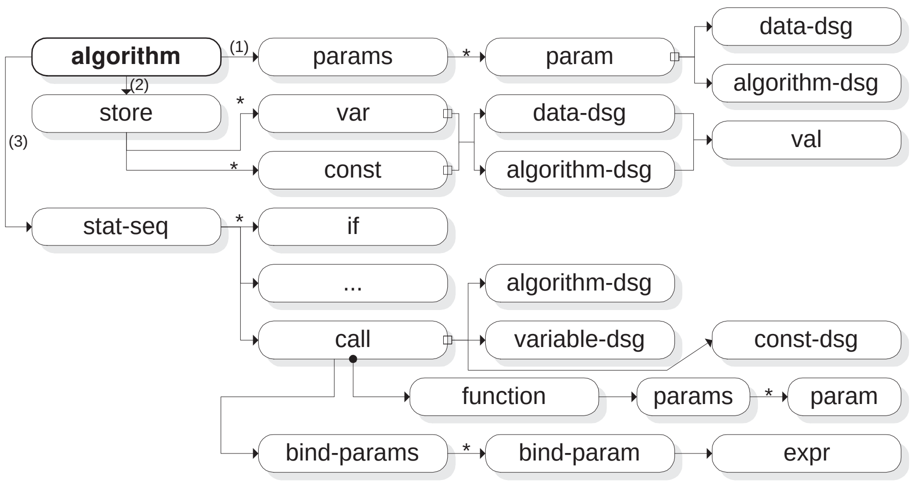

Figure B.3: Visitation graph of visitor `Emitter` for algorithms.

B.2.2 Implementation

Declaration The C++ code emitter is implemented as Loki visitor in class `Emitter`. As usual, it has to inherit from all the node classes that it will visit during data structure or algorithm hierarchy traversal.

⟨Listing B.22: Class `Emitter`⟩ ≡

```
class Emitter : public Loki::BaseVisitor,
                public Loki::Visitor<Data>,
                public Loki::Visitor<Element>,
                ...
```

Code extracted from file `gilf/Emitter.hpp`, lines 54 to 86.

Types The emitter has two public types. Type `line_type` is used to store one line of emitted code, we create the code in standard strings. Multiple lines are stored in the line sequence type, which is a `std::deque` container holding elements of `line_type`.

⟨Listing B.23: Class `Emitter`: Public Types⟩ ≡

```
// The type of output for one single line.
typedef std::string line_type;
// The container sequence type that holds lines.
typedef std::deque<line_type> line_seq_type;
```

Code extracted from file `gilf/Emitter.hpp`, lines 101 to 104.

Methods The result of an emitter can be queried with method `get_code`. It returns the line sequence container `m_code` that holds the complete code which was generated during emitter visitation.

⟨Listing B.24: Class Emitter: Public Method get_code⟩ ≡

```
// Access the code of this emitter.
const line_seq_type& get_code() const { return m_code; }
```

Code extracted from file gilf/Emitter.hpp, lines 109 to 110.

Most identifier names used in GILF are obviously not C++ standard conformant. This is accomplished by method trim_symbol, that expects an arbitrary identifier and turns it into a *trimmed*, i.e. a standard conformant, C++ identifier. We see the power of STL algorithms in the shape of std::for_each, which lets this work collapse into one line of code. However, we still have to provide the function object Trimmer, which checks each character iterated over by for_each and performs transformations where applicable.

⟨Listing B.25: Class Emitter: Private Method trim_symbol⟩ ≡

```
id_iprop::type Emitter::trim_symbol(const id_iprop::type& symbol)
{
  return (std::for_each(symbol.begin(), symbol.end(), Trimmer())).symbol();
}
```

Code extracted from file gilf/Emitter.cpp, lines 1060 to 1063.

The subnodes of statement sequences have to be visited in their original order, and method visit_sequential performs exactly this job.

⟨Listing B.26: Class Emitter: Private Method visit_sequential⟩ ≡

```
// Visit all subnodes of node sequentially.
void visit_sequential(GILF_Node& node);
```

Code extracted from file gilf/Emitter.hpp, lines 285 to 286.

Visit Methods The Visit method overrides of a visitor contain the bulk of its implementation, which is also true for the C++ emitter. Visitation begins either at a Data or an Algorithm GILF node. We start by looking at the emitter methods responsible for data structure code generation.

The method is split up in two sections. The first one handles built-in data structures and the second one generates code for user defined data structures. Built-in data structures from the core library (see appendix C) can either be mapped directly to C++ types or require types from small support libraries. Data structures in GILF are denoted by their identifier, and therefore we provide C++ type definitions for the built-in data structures such that we can simply use the trimmed identifier name as C++ type name. For example, the architecture specific unsigned machine word u_mtype.d_uword will result in the following C++ type definition:

```
typedef unsigned int u_mtype_DOTd_uword;
```

GILF supports two kinds of user defined data structures, record and union types, respectively. These are both available in C++, the former one as struct, and the latter one as union. The data structure's kind can be retrieved from the node's kind property. The data structure's trimmed identifier is appended after the compound's kind name, and the whole code line is pushed at the end of the code container m_code. Then we visit the nested element nodes in order to create the data structure's field entries. Eventually, the closing parenthesis is added to the code line container, which completes the C++ compound.

⟨Listing B.27: Class Emitter: Generating User Defined Data Structures⟩ ≡

```
// Check the kind of the data structure.
if (node.data.at<data_kind_iprop>() == data_kind_record)
{
  code = "struct ";
}
```

```
else if (node.data.at<data_kind_iprop>() == data_kind_union)
{
  code = "union ";
}
// Add the trimmed id as struct/union name.
code += trim_symbol(id);
code += " {";
m_code.push_back(code);
// Now visit the element subnodes and create field entries.
visit_nodes_by_type<Element>(node);
// Now close the struct/union.
m_code.push_back("};");
```
Code extracted from file `gilf/Emitter.cpp`, lines 143 to 159.

Element node visitation proceeds straightforward. The field's type is extracted from the designator subnode (see figure B.2), and the field name is given by the element's identifier. We also have to check for pointer types and annotate the field's type name accordingly. The following code lines show the code generated for an instantiation of a heterogeneous pair type.

```
struct
u_main_DOTd_pair_het_DOTtp_0_OPu_mtype_DOTd_uword_CP_DOTtp_1_OPu_mtype_DOTd_byte_CP {
  u_mtype_DOTd_uword e1;
  u_mtype_DOTd_byte e2;
};
```

Method `Visit(Algorithm&)` controls code emission for algorithms. The structure is also divided in two parts like for data structures as it handles either built-in or user defined algorithms. We provide inline definitions for built-in algorithms, analogously to the type definitions provided for built-in data structures. The inline definitions are slightly more complex, because we have to generate the return type and the correct number of typed parameters. The `inline` specifier ensures that we do not introduce a function call indirection for simple operations that often map directly to machine code instructions.

The top-level code for generating user defined algorithms closely resembles the corresponding visitation graph path (see figure B.3). It starts a new code line by adding the trimmed algorithm identifier. This signature line will be completed during traversal of the parameter nodes. The algorithm's body is made up of its local symbol table and its statement sequence.

⟨Listing B.28: Class `Emitter`: Generating User Defined Algorithms⟩ ≡
```
// Add the trimmed id as algorithm name.
m_code.push_back(trim_symbol(id));
// First, the function signature is created.
visit_nodes_by_type<Params>(node);
// Now emit the algorithm's body.
m_code.push_back("{");
// Write local output parameter if present.
handle_local_output_var();
// Add the variables and constants.
visit_nodes_by_type<Storage>(node, true);
// Now generate the statements.
visit_nodes_by_type<Statements>(node, true);
m_code.push_back("}");
```
Code extracted from file `gilf/Emitter.cpp`, lines 305 to 317.

One of the central issues in translating GILF to C++ is generating function signatures and function calls. The problems arises from the disparity in specifying function parameters in C++ and GILF. The former one uses positional parameters, and the latter one named parameters. Furthermore, the return parameter in C++ is only given by type and is not present in the parameter list in parentheses. These facts force us to impose an order on the algorithm's parameters specified in GILF. In the prototype, we simply use their occurrence in the internal representation.

Another interesting point in code emission for algorithms are the employed passing conventions. We use either reference or value semantics, based on the size of parameter's type. This decision process is factored out into the private method `resolve_passing_mod`, which returns a qualified passing modifier, if a unqualified was present in the GILF code. The only additional parameter to this method is the parameter's type, represented by its `type_id`.

⟨Listing B.29: Class `Emitter`: Deciding Parameter Passing Conventions⟩ ≡

```
// Get the parameter's passing convention.
passing_mod_iprop::type passing =
  resolve_passing_mod(node.data.at<passing_mod_iprop>(), type_id);
```

Code extracted from file `gilf/Emitter.cpp`, lines 406 to 408.

The local variables and constants are created very similar to fields in user defined data structures, which was discussed above. The only novel feature here is that they can be initialized to values. Most of the work for generating values is delegated to Boost's `lexical_cast` library. We have to extract the value from the Boost `any` type based on its kind property, the following code snippet shows the conversion of a 32-bit interger value.

⟨Listing B.30: Class `Emitter`: Writing Values with Boost `lexical_cast`⟩ ≡

```
m_code.back() +=
  boost::lexical_cast<std::string>(boost::any_cast<boost::uint32_t>(value.value));
```

Code extracted from file `gilf/Emitter.cpp`, lines 646 to 647.

Most statement constructs present in GILF map to C++ statements without problems. We have to be careful that the sequence ordering is retained, but otherwise code generation is mostly a mechanical process. Sequential traversal of statement sequences is enforced in method `visit_sequential`, for example in the following code:

⟨Listing B.31: Class `Emitter`: Visiting Statement Sequences in Order⟩ ≡

```
void Emitter::Visit(Statements& node)
{
  visit_sequential(node);
}
```

Code extracted from file `gilf/Emitter.cpp`, lines 664 to 667.

As noted before, code generation for function calls is more challenging because of C++'s positional parameter identification. We proceed as follows. The call's target is given by the call's designator subnode's identifier reference property. Next, an empty ordering table is pushed onto a stack. The stack is necessary, because GILF support nested function calls. This table is then filled in method `order_parameters` based on the ordering of the called function's parameters in GILF's internal representation. With this information at hand, binding value parameters can begin. It starts by visiting all parameter bindings and caches the argument expressions, and then creates the function call with the precomputed order of parameters.

⟨Listing B.32: Class `Emitter`: Translating the `call` Node⟩ ≡

```
// Push an empty parameter order table on its stack.
m_params_order.push_back(order_table_type());
// Fill the parameter order table.
order_parameters(node.data.at<idref_iprop>());
// Push an argument container with the size of the order table's size.
m_arguments.push_back(line_seq_type(m_params_order.back().size()));
```

```
// Now, generate the function call arguments.
m_code.back() += "(";
visit_nodes_by_type<Bind_Params>(node);
m_code.back() += ")";
// Remove the argument container from its stack.
m_arguments.pop_back();
// Remove the parameter order table from its stack.
m_params_order.push_back(order_table_type());
```

Code extracted from file `gilf/Emitter.cpp`, lines 917 to 930.

We finally want to emphasize the point that the C++ code generator present in the GILF prototype is intended for explanatory purposes only. The generated code was not optimized for execution speed or space efficiency. However, the resulting programs run at reasonable speed.

Appendix C

The XGILF Core Library

This appendix contains the specification of the GILF core library, written in XGILF. The core library and the GILF specification (see chapter 4) represent the interface a compiler front-end has to target when generating GILF. In addition to the features available in GILF, the core library provides the interface to built-in data structures and algorithms, as well as some predefined general purpose function and type signatures.

C.1 Boolean

Unit `u_bool` contains a boolean type and related operations. The type's signature is accessible by identifier `t_bool`, the data structure by identifier `d_bool`, and the type binding by identifier `bt_bool`. In the next sections, we will in general omit the definition and binding nodes for brevity, because most built-in data structures and algorithms nodes can be derived mechanically from the declaration.

At the beginning of the file, the XML processing instructions are visible, which denote that we deal with an XGILF file, using version `1.0`. The digest is currently computed with the tool `md5sum` and embedded in the `xgilf` processing instruction.

⟨Listing C.1: System Unit for Boolean Type⟩ ≡

```
<?gilf version="1.0" protocol="xgilf" level="1"?>
<?xgilf version="1.0" digest="e817a0e2b01f15bcff3dd6d4571ed67f"?>
...
<xgilf>

 <!--========================================================================-->
 <unit id="u_bool" name="std_boolean">

  <!-- unit dependencies -->
  <import>
   <source input-src="function.xgf"> <unit-dsg ref="u_func"/> </source>
  </import>

  <!-- declaration part, shareable among system files -->
  <declare>
   <!-- type bool -->
   <type id="u_bool.t_bool" name="bool"/>

   <!-- boolean operations -->
   <function id="u_bool.f_and" name="and">
    <params count="3">
```

```
      <param pass="in" id="p_0" name="arg1"> <binding-dsg ref="bt_bool"/> </param>
      <param pass="in" id="p_1" name="arg2"> <binding-dsg ref="bt_bool"/> </param>
      <param pass="out!" id="p_2" name="result"> <binding-dsg ref="bt_bool"/> </param>
     </params>
    </function>

    <function id="u_bool.f_or" name="or">
     <params count="3">
      <param pass="in" id="p_0" name="arg1"> <binding-dsg ref="bt_bool"/> </param>
      <param pass="in" id="p_1" name="arg2"> <binding-dsg ref="bt_bool"/> </param>
      <param pass="out!" id="p_2" name="result"> <binding-dsg ref="bt_bool"/> </param>
     </params>
    </function>

    <function id="u_bool.f_not" name="not">
     <params count="2">
      <param pass="in" id="p_0" name="self"> <binding-dsg ref="bt_bool"/> </param>
      <param pass="out!" id="p_1" name="result"> <binding-dsg ref="bt_bool"/> </param>
     </params>
    </function>
   </declare>

   <!-- definition part, for code generation -->
   <define>
    <data id="u_bool.d_bool" ref="u_bool.t_bool" built-in="yes"/>

    <algorithm id="u_bool.a_and" ref="u_bool.f_and" built-in="yes"/>
    <algorithm id="u_bool.a_or" ref="u_bool.f_or" built-in="yes"/>
    <algorithm id="u_bool.a_not" ref="u_bool.f_not" built-in="yes"/>
    <algorithm id="u_bool.a_==" ref="u_func.f_==" built-in="yes"/>
    <algorithm id="u_bool.a_!=" ref="u_func.f_!=" built-in="yes"/>
   </define>

   <!-- bindings -->
   <bind>
    <!-- bind type definitions to type declarations -->
    <bind-type id="u_bool.bt_bool" ref="u_bool.t_bool">
     <data-dsg ref="u_bool.d_bool"/>
    </bind-type>

    <!-- bind algorithms to functions -->
    <bind-func id="u_bool.bf_and" ref="u_bool.f_and">
     <algo-dsg ref="u_bool.a_and"/>
    </bind-func>
    ...

    <bind-func id="u_bool.bf_==" ref="u_func.f_==">
     <bind-static-params id="bsp_local">
      <bind-tp ref="tp_0"><binding-dsg ref="u_bool.bt_bool"/></bind-tp>
     </bind-static-params>
     <algo-dsg ref="u_bool.a_=="/>
    </bind-func>
    ...
   </bind>

  </unit> <!-- std_boolean [u_bool] -->

 </xgilf>
```

Code extracted from file `../xgf/core_units/boolean.xgf`, lines 3 to 108.

C.2 Machine Types

The unit `u_mtype` contains types and related functions that provide access to machine types and operations on them natural to the deployment architecture. Therefore, no guarantees about sizes in bits can be made, but we make sure that the operations require no special code for canonization. Type `t_byte` contains the smallest addressable memory unit, and type `t_word` is the native integral computation unit for the host architecture. Both types have an unsigned counterpart. Also, a type for single precision floating point computations is available by identifier `t_float`, and one for double precision by identifier `t_floatd`.

For all these types exist algorithms that perform comparison and arithmetic operations. They reference function signatures from unit `u_func`, see section C.6.

At the beginning of the function bindings one can see how the built-in comparison for equality algorithm for bytes is bound to the corresponding general purpose function signature. First, the instantiation parameter is bound to `bt_byte`, and then the built-in algorithm `a_==_byte` is designated as binding target.

⟨Listing C.2: System Unit for Native Machine Types⟩ ≡

```
<unit id="u_mtype" name="std_machtypes">

 <!-- unit dependencies -->
 <import>
  <source input-src="function.xgf"> <unit-dsg ref="u_func"/> </source>
 </import>

 <!-- declaration part, shareable among system files -->
 <declare>
  <!-- machine types, architecture dependent -->
  <type name="byte" id="u_mtype.t_byte"/>
  <type name="ubyte" id="u_mtype.t_ubyte"/>
  <type name="word" id="u_mtype.t_word"/>
  <type name="uword" id="u_mtype.t_uword"/>

  <type name="float" id="u_mtype.t_float"/>
  <type name="floatd" id="u_mtype.t_floatd"/>
 </declare>

 <!-- definition part, for code generation -->
 <define>
  <!-- the built-in machine types -->
  <data id="u_mtype.d_byte" ref="u_mtype.t_byte" built-in="yes"/>
  <data id="u_mtype.d_word" ref="u_mtype.t_word" built-in="yes"/>
  <data id="u_mtype.d_ubyte" ref="u_mtype.t_ubyte" built-in="yes"/>
  <data id="u_mtype.d_uword" ref="u_mtype.t_uword" built-in="yes"/>

  <data id="u_mtype.d_float" ref="u_mtype.t_float" built-in="yes"/>
  <data id="u_mtype.d_floatd" ref="u_mtype.t_floatd" built-in="yes"/>

  <!-- comparison algorithms for the built-in machine types -->
  <algorithm id="u_mtype.a_==_byte" ref="u_func.f_==" built-in="yes"/>
  <algorithm id="u_mtype.a_==_ubyte" ref="u_func.f_==" built-in="yes"/>
  <algorithm id="u_mtype.a_==_word" ref="u_func.f_==" built-in="yes"/>
  <algorithm id="u_mtype.a_==_uword" ref="u_func.f_==" built-in="yes"/>
  <algorithm id="u_mtype.a_==_ufloat" ref="u_func.f_==" built-in="yes"/>
  <algorithm id="u_mtype.a_==_ufloatd" ref="u_func.f_==" built-in="yes"/>
  ...
```

```
  </define>

  <!-- bindings -->
  <bind>
   <!-- bind type definitions to type declarations -->
   <bind-type id="u_mtype.bt_byte" ref="u_mtype.t_byte">
    <data-dsg ref="u_mtype.d_byte"/>
   </bind-type>
   <bind-type id="u_mtype.bt_ubyte" ref="u_mtype.t_ubyte">
    <data-dsg ref="u_mtype.d_ubyte"/>
   </bind-type>
   <bind-type id="u_mtype.bt_word" ref="u_mtype.t_word">
    <data-dsg ref="u_mtype.d_word"/>
   </bind-type>
   <bind-type id="u_mtype.bt_uword" ref="u_mtype.t_uword">
    <data-dsg ref="u_mtype.d_uword"/>
   </bind-type>

   <bind-type id="u_mtype.bt_float" ref="u_mtype.t_float">
    <data-dsg ref="u_mtype.d_float"/>
   </bind-type>
   <bind-type id="u_mtype.bt_floatd" ref="u_mtype.t_floatd">
    <data-dsg ref="u_mtype.d_floatd"/>
   </bind-type>

   <!-- bind algorithms to functions -->
   <!-- comparison functions -->
   <bind-func id="u_mtype.bf_==_byte" ref="u_func.f_==">
    <bind-static-params>
     <bind-tp ref="tp_0"> <binding-dsg ref="u_mtype.bt_byte"/> </bind-tp>
    </bind-static-params>
    <algo-dsg ref="u_mtype.a_==_byte"/>
   </bind-func>
   ...
  </bind>

</unit> <!-- std_machtypes [u_mtype] -->
```

Code extracted from file `../xgf/core_units/machtype.xgf`, lines 29 to 457.

C.3 Integers

The unit `u_int` provides integral types of fixed size. The size n is given in bits, however only values for n are valid where $n = 8 \cdot 2^m$ with $m \geq 0$ holds. On 32-bit architectures, 64-bit operations have to be realized in software.

In front of the binding section the constants for all required sizes (8, 16, 32, and 64) are defined in the storage section. The binding section contains the built-in bindings, for example `bf_==_int8` compares two signed 8-bit integers and returns a boolean result value. We see that both the constant value parameter and the type parameter are bound in the static parameter binding subnode.

⟨Listing C.3: System Unit for Fixed Size Integer Types⟩ ≡

```
<unit id="u_int" name="std_integer">

  <!-- unit dependencies -->
  <import>
   <source input-src="function.xgf"> <unit-dsg ref="u_func"/> </source>
```

```
</import>

<!-- declaration part, shareable among system units -->
<declare>
 <!-- signed integers, parameterized by size in bytes -->
 <type id="u_int.t_int#n" name="int#n">
  <const-params count="1">
   <const-param id="cp_0" name="#bits">
    <type-dsg ref="u_mtype.t_uword"/>
   </const-param>
  </const-params>
 </type>

 <!-- unsigned integers, parameterized by size in bytes -->
 <type id="u_int.t_uint#n" name="uint#n">
  <const-params count="1">
   <const-param id="cp_0" name="#bits">
    <type-dsg ref="u_mtype.t_uword"/>
   </const-param>
  </const-params>
 </type>
</declare>

<!-- definition part, for code generation -->
<define>
 <data id="u_int.d_int8" ref="u_int.t_int#n" built-in="yes"/>
 <data id="u_int.d_int16" ref="u_int.t_int#n" built-in="yes"/>
 <data id="u_int.d_int32" ref="u_int.t_int#n" built-in="yes"/>
 <data id="u_int.d_int64" ref="u_int.t_int#n" built-in="yes"/>
 <data id="u_int.d_uint8" ref="u_int.t_uint#n" built-in="yes"/>
 <data id="u_int.d_uint16" ref="u_int.t_uint#n" built-in="yes"/>
 <data id="u_int.d_uint32" ref="u_int.t_uint#n" built-in="yes"/>
 <data id="u_int.d_uint64" ref="u_int.t_uint#n" built-in="yes"/>

 <algorithm id="u_int.a_==_int8" ref="u_func.f_==" built-in="yes"/>
 ...
 <algorithm id="u_int.a_!=_int8" ref="u_func.f_!=" built-in="yes"/>
 ...
 <algorithm id="u_int.a_lt_int8" ref="u_func.f_lt" built-in="yes"/>
 ...
 <algorithm id="u_int.a_gt_int8" ref="u_func.f_gt" built-in="yes"/>
 ...
 <algorithm id="u_int.a_lte_int8" ref="u_func.f_lte" built-in="yes"/>
 ...
 <algorithm id="u_int.a_gte_int8" ref="u_func.f_gte" built-in="yes"/>
 ...
</define>

<store>
 <!-- numerical constants for bit counts -->
 <const id="u_int.c_8">
  <binding-dsg ref="u_mtype.bt_uword"/><val val="8" kind="dec"/>
 </const>
 <const id="u_int.c_16">
  <binding-dsg ref="u_mtype.bt_uword"/><val val="16" kind="dec"/>
 </const>
 <const id="u_int.c_32">
  <binding-dsg ref="u_mtype.bt_uword"/><val val="32" kind="dec"/>
 </const>
 <const id="u_int.c_64">
```

```
<binding-dsg ref="u_mtype.bt_uword"/><val val="64" kind="dec"/>
</const>
</store>

<!-- bindings -->
<bind>
 <!-- bind type definitions to type declarations -->
 <bind-type id="u_int.bt_int8" ref="u_int.t_int8">
  <bind-static-params>
   <bind-cp ref="cp_0"> <const-dsg ref="u_int.c_8"/> </bind-cp>
  </bind-static-params>
  <data-dsg ref="u_int.d_int8"/>
 </bind-type>
 ...
 <!-- bind algorithms to functions -->
 <bind-func id="u_int.bf_==_int8" ref="u_func.f_==">
  <bind-static-params>
   <bind-tp ref="tp_0"> <binding-dsg ref="u_int.bt_int8"/> </bind-tp>
   <bind-cp ref="cp_0"> <const-dsg ref="u_int.c_8"/> </bind-cp>
  </bind-static-params>
  <algo-dsg ref="u_int.a_==_int8"/>
 </bind-func>
 ...
</unit> <!-- std_integer [u_int] -->
```

Code extracted from file ../xgf/core_units/integer.xgf, lines 16 to 598.

C.4 Arrays

All declarations and definitions related to arrays are available through unit u_array. In GILF, we differentiate between fixed-size arrays whose size is static at compile time, and those whose size is set at runtime, or to be more precise, at allocation time with a variable argument. The former one's type is referenced by identifier t_[], and the latter one's by t_[c]. Both have one type instantiation parameter that declares the array elements' type, and t_[c] has an additional constant instantiation parameter that indicates the array's size. For dynamically allocated arrays, a special allocate function f_allocate[] is declared, which has one value parameter which expects the array's size. Furthermore, built-in algorithms for querying an array's size (a_size[]), as well as setting (a_set[n]) and retrieving (a_get[n]) elements by index are provided.

⟨Listing C.4: System Unit for Array Types⟩ ≡

```
<unit name="std_array" id="u_array">

 <!-- unit dependencies -->
 <import>
  <source input-src="machtype.xgf"> <unit-dsg ref="u_mtype"/> </source>
 </import>

 <!-- declaration part, shareable among system units -->
 <declare>
  <!-- static array, size fixed at run time -->
  <type name="runtime array" id="u_array.t_[]">
   <type-params count="1"> <type-param id="tp_0" name="T"/> </type-params>
  </type>

  <!-- static array, size fixed at compile time -->
  <type name="compile time array" id="u_array.t_[c]">
```

```
 <type-params count="1"> <type-param id="tp_0" name="T"/> </type-params>
 <const-params count="1">
  <const-param name="size" id="cp_0">
   <type-dsg ref="u_mtype.t_uword"/>
  </const-param>
 </const-params>
</type>

<!--
* allocate[] function
* Allocates memory for a fixed size array on the heap and returns
* reference to the new array.
-->
<function name="allocate[]" id="u_array.f_allocate[]">
 <type-params count="1"> <type-param name="T[]" id="tp_0"/> </type-params>
 <params count="2">
  <param pass="in" id="p_0" name="size"><binding-dsg ref="u_mtype.bt_uword"/></param>
  <param pass="out_ref!" id="p_1" name="new[]"><static-param-dsg ref="tp_0"/></param>
 </params>
</function>

<!--
* size[] function
* Returns number of items storeable in the given array.
-->
<function name="size[]" id="u_array.f_size[]">
 <type-params count="1"> <type-param name="T[]" id="tp_0"/> </type-params>
 <params count="2">
  <param pass="in" id="p_0" name="T[]"><static-param-dsg ref="tp_0"/></param>
  <param pass="out!" id="p_1" name="size"><binding-dsg ref="u_mtype.bt_uword"/></param>
 </params>
</function>

<!--
* get[n] function
* Returns the item at position n in the given array.
-->
<function name="get[n]" id="u_array.f_get[n]">
 <type-params count="2">
  <type-param name="T[]" id="tp_0"/>
  <type-param name="T" id="tp_1"/>
 </type-params>
 <params count="3">
  <param pass="in_ref" id="p_0" name="[]"><static-param-dsg ref="tp_0"/></param>
  <param pass="in" id="p_1" name="n"><binding-dsg ref="u_mtype.bt_uword"/></param>
  <param pass="out!" id="p_2" name="item"><static-param-dsg ref="tp_1"/></param>
 </params>
</function>

<!--
* set[n] function
* Sets the item at position n in the given array.
-->
<function name="set[n]" id="u_array.f_set[n]">
 <type-params count="2">
  <type-param name="T[]" id="tp_0"/>
  <type-param name="T" id="tp_1"/>
 </type-params>
 <params count="3">
  <param pass="in_ref" id="p_0" name="[]"><static-param-dsg ref="tp_0"/></param>
```

```
      <param pass="in" id="p_1" name="n"><binding-dsg ref="u_mtype.bt_uword"/></param>
      <param pass="in" id="p_2" name="item"><static-param-dsg ref="tp_1"/></param>
     </params>
    </function>
   </declare>

   <!-- definition part, for code generation -->
   <define>
    <data id="u_array.d_[]" ref="u_array.t_[]" built-in="yes"/>
    <data id="u_array.d_[c]" ref="u_array.t_[c]" built-in="yes"/>

    <algorithm id="u_array.a_allocate[]" ref="u_array.f_allocate[]" built-in="yes"/>
    <algorithm id="u_array.a_size[]" ref="u_array.f_size[]" built-in="yes"/>
    <algorithm id="u_array.a_get[n]" ref="u_array.f_get[n]" built-in="yes"/>
    <algorithm id="u_array.a_set[n]" ref="u_array.f_set[n]" built-in="yes"/>
   </define>
   ...
  </unit> <!-- std_array [u_array] -->
```

Code extracted from file `../xgf/core_units/array.xgf`, lines 17 to 143.

C.5 Unicode Characters

GILF supports Unicode characters with two types in unit `u_unicode`. Type `t_UCchar` represents one Unicode character, and an array of Unicode characters is represented with type `t_UCchar[]`. An equality check for Unicode characters is also provided, which compares the character codes of two `t_UCchar`s, but does ignore any compositions. More elaborate comparisons of Unicode character strings can be built on top of this basic character check.

⟨Listing C.5: System Unit for Unicode Characters⟩ ≡

```
  <unit name="std_unicode" id="u_unicode" digest="86F2983E">

   <!-- unit dependencies -->
   <import>
    <source input-src="array.xgf"> <unit-dsg ref="u_array"/> </source>
    <source input-src="function.xgf"> <unit-dsg ref="u_func"/> </source>
   </import>

   <!-- declaration part, shareable among system files -->
   <declare>
    <!-- type t_UCchar: single, 16 bit wide unicode character -->
    <type name="UC character" id="u_unicode.t_UCchar"/>
    <!-- type t_UCchar[]: array of unicode characters -->
    <type name="UC character[]" id="u_unicode.t_UCchar[]"/>
   </declare>

   <!-- definition part, for code generation -->
   <define>
    <data id="u_unicode.d_UCchar" ref="u_unicode.t_UCchar" built-in="yes"/>
    <data id="u_unicode.d_UCchar[]" ref="u_unicode.t_UCchar[]" built-in="yes"/>

    <!-- comparison algorithm for equality of two unicode characters, compares
         just the unicode character code, not some dynamically composed characters -->
    <algorithm id="u_unicode.a_==_UCchar" ref="u_func.f_==" built-in="yes"/>
   </define>

   <!-- bindings -->
   ...
```

```
</unit> <!-- std_unicode [u_unicode] -->
```
Code extracted from file `../xgf/core_units/unicode.xgf`, lines 17 to 69.

C.6 Functions

A collection of general purpose function signatures is available in unit `u_func`, for example for comparison and arithmetic operations. Additionally, it contains function signatures that trigger special behavior in the back-end when they are bound to user defined algorithms.

Function `main` Instantiation application starts by looking for a unit's function binding for `f_main`. Then, all dependent instantiations are generated recursively and program execution starts by calling the designated algorithm.

Function `allocate` The `allocate` function presents the interface to GILF's dynamic memory allocation system. It will try to allocate heap memory for a single object of the instantiation type parameter and returns a reference to the newly allocated object. If the allocation request could not be fulfilled even after a garbage collector invocation, a null reference is returned.

Function `clone` This function is relevant in two contexts. First, if a function argument is passed by value, and a binding for the argument's type to the clone function exists, the designated `clone` algorithm will be called to clone the original argument. The same holds true for assignments. Notice that only bindings for user defined data structures are valid instantiation parameters, because built-in data structures are handled implicitly.

⟨Listing C.6: System Unit for Function Declarations⟩ ≡
```
<unit id="u_func" name="std_function">

 <!-- unit dependencies -->
 <import>
  <source input-src="boolean.xgf"> <unit-dsg ref="u_bool"/> </source>
  <source input-src="unicode.xgf"> <unit-dsg ref="u_unicode"/> </source>
  <source input-src="machtype.xgf"> <unit-dsg ref="u_mtype"/> </source>
  <source input-src="array.xgf"> <unit-dsg ref="u_array"/> </source>
 </import>

 <!-- declaration part, shareable among system files -->
 <declare>
  <!--
  * main function
  * Every user program has to provide an algorithm for this function in its
  * main unit. The function will be called at program startup.
  -->
  <function id="u_func.f_main" name="main">
   <params count="2">
    <param pass="in" id="p_0" name="args">
     <binding-dsg ref="u_func.bt_mainargs"/>
    </param>
    <param pass="out!" id="p_1" name="return state">
     <binding-dsg ref="u_mtype.bt_word"/>
    </param>
```

```
 </params>
</function>

<!--
* allocate function
* Allocates memory from the heap for a single dynamic object.
-->
<function id="u_func.f_allocate" name="allocate">
 <type-params count="1"> <type-param id="tp_0" name="T"/> </type-params>
 <params count="1">
  <param pass="out_ref!" id="p_0" name="new_obj"><static-param-dsg ref="tp_0"/></param>
 </params>
</function>

<!--
* clone function
* This function will be called by the back-end when a reference object is
* passed by value. Thus, any reference type that wants to be called in such a
* manner has to provide an implementation of and a binding to this function.
-->
<function name="clone" id="u_func.f_clone">
 <type-params count="1"> <type-param name="T" id="tp_0"/> </type-params>
 <params count="2">
  <param pass="in" id="p_0"  name="arg1"> <static-param-dsg ref="tp_0"/> </param>
  <param pass="out_ref!" id="p_1" name="result"><static-param-dsg ref="tp_0"/></param>
 </params>
</function>

<!-- comparison functions -->
<function name="==" id="u_func.f_==">
 <type-params count="1"> <type-param name="N" id="tp_0"/> </type-params>
 <params count="3">
  <param pass="in" id="p_0" name="arg1"> <static-param-dsg ref="tp_0"/> </param>
  <param pass="in" id="p_1" name="arg2"> <static-param-dsg ref="tp_0"/> </param>
  <param pass="out!" id="p_2" name="result"><binding-dsg ref="u_bool.bt_bool"/></param>
 </params>
</function>

<function name="!=" id="u_func.f_!=">
 <type-params count="1"> <type-param name="N" id="tp_0"/> </type-params>
 <params count="3">
  <param pass="in" id="p_0" name="arg1"> <static-param-dsg ref="tp_0"/> </param>
  <param pass="in" id="p_1" name="arg2"> <static-param-dsg ref="tp_0"/> </param>
  <param pass="out!" id="p_2" name="result"><binding-dsg ref="u_bool.bt_bool"/></param>
 </params>
</function>

<function name="lt" id="u_func.f_lt">
 <type-params count="1"> <type-param name="T" id="tp_0"/> </type-params>
 <params count="3">
  <param pass="in" id="p_0" name="arg1"> <static-param-dsg ref="tp_0"/> </param>
  <param pass="in" id="p_1" name="arg2"> <static-param-dsg ref="tp_0"/> </param>
  <param pass="out!" id="p_2" name="result"><binding-dsg ref="u_bool.bt_bool"/></param>
 </params>
</function>

<function name="gt" id="u_func.f_gt">
 <type-params count="1"> <type-param name="T" id="tp_0"/> </type-params>
 <params count="3">
  <param pass="in" id="p_0" name="arg1"> <static-param-dsg ref="tp_0"/> </param>
```

```
  <param pass="in" id="p_1" name="arg2"> <static-param-dsg ref="tp_0"/> </param>
  <param pass="out!" id="p_2" name="result"><binding-dsg ref="u_bool.bt_bool"/></param>
 </params>
</function>

<function name="lte" id="u_func.f_lte">
 <type-params count="1"> <type-param name="T" id="tp_0"/> </type-params>
 <params count="3">
  <param pass="in" id="p_0" name="arg1"> <static-param-dsg ref="tp_0"/> </param>
  <param pass="in" id="p_1" name="arg2"> <static-param-dsg ref="tp_0"/> </param>
  <param pass="out!" id="p_2" name="result"><binding-dsg ref="u_bool.bt_bool"/></param>
 </params>
</function>

<function name="gte" id="u_func.f_gte">
 <type-params count="1"> <type-param name="T" id="tp_0"/> </type-params>
 <params count="3">
  <param pass="in" id="p_0" name="arg1"> <static-param-dsg ref="tp_0"/> </param>
  <param pass="in" id="p_1" name="arg2"> <static-param-dsg ref="tp_0"/> </param>
  <param pass="out!" id="p_2" name="result"><binding-dsg ref="u_bool.bt_bool"/></param>
 </params>
</function>

<!-- arithmetic functions -->
<function name="+" id="u_func.f_+">
 <type-params count="1"> <type-param name="T" id="tp_0"/> </type-params>
 <params count="3">
  <param pass="in" id="p_0" name="arg1"> <static-param-dsg ref="tp_0"/> </param>
  <param pass="in" id="p_1" name="arg2"> <static-param-dsg ref="tp_0"/> </param>
  <param pass="out!" id="p_2" name="result"><static-param-dsg ref="tp_0"/></param>
 </params>
</function>

<function name="++" id="u_func.f_++">
 <type-params count="1"> <type-param name="T" id="tp_0"/> </type-params>
 <params count="1">
  <param pass="inout!" id="p_0" name="self"><static-param-dsg ref="tp_0"/></param>
 </params>
</function>

<function name="-" id="u_func.f_-">
 <type-params count="1"> <type-param name="T" id="tp_0"/> </type-params>
 <params count="3">
  <param pass="in" id="p_0" name="arg1"> <static-param-dsg ref="tp_0"/> </param>
  <param pass="in" id="p_1" name="arg2"> <static-param-dsg ref="tp_0"/> </param>
  <param pass="out!" id="p_2" name="result"><static-param-dsg ref="tp_0"/></param>
 </params>
</function>

<function name="--" id="u_func.f_--">
 <type-params count="1"> <type-param name="T" id="tp_0"/> </type-params>
 <params count="1">
  <param pass="inout!" id="p_0" name="self"><static-param-dsg ref="tp_0"/></param>
 </params>
</function>

<function name="*" id="u_func.f_*">
 <type-params count="1"> <type-param name="T" id="tp_0"/> </type-params>
 <params count="3">
  <param pass="in" id="p_0" name="arg1"> <static-param-dsg ref="tp_0"/> </param>
```

```
      <param pass="in" id="p_1" name="arg2"> <static-param-dsg ref="tp_0"/> </param>
      <param pass="out!" id="p_2" name="result"><static-param-dsg ref="tp_0"/></param>
    </params>
  </function>

  <function name="/" id="u_func.f_/">
    <type-params count="1"> <type-param name="T" id="tp_0"/> </type-params>
    <params count="3">
      <param pass="in" id="p_0" name="arg1"> <static-param-dsg ref="tp_0"/> </param>
      <param pass="in" id="p_1" name="arg2"> <static-param-dsg ref="tp_0"/> </param>
      <param pass="out!" id="p_2" name="result"><static-param-dsg ref="tp_0"/></param>
    </params>
  </function>
</declare>

<bind>
  <!-- type binding for array of unicode strings -->
  <bind-type id="u_func.bt_mainargs" ref="u_array.t_[]">
    <bind-static-params>
      <bind-tp ref="tp_0"> <binding-dsg ref="u_unicode.bt_UCchar[]"/> </bind-tp>
    </bind-static-params>
    <data-dsg ref="u_array.d_[]"/>
  </bind-type>
</bind>

</unit> <!-- std_function [u_func] -->
```

Code extracted from file ../xgf/core_units/function.xgf, lines 17 to 187.

Appendix D

Examples

D.1 Mapping SUCHTHAT to GILF

The goal of this project was to provide a back-end for generic programming languages, with special interest in diectly supporting SUCHTHAT. The imperative statements in SUCHTHAT are adopted from Aldes [LoCo92], and mapping them to the high-level statements available in gilf (see section 4.8.2) is almost trivial. In [Wei97] such a mapping to C++ statements is presented, which can be reused without great modifications.

The task of mapping the declarative TECTON part present in SUCHTHAT to GILF is more challenging[1]. At the lowest level, TECTON provides function declarations, composed of function identifiers and their arity, which consists of sort identifiers. These constructs can be mapped to GILF function and type declarations. TECTON concepts group functions and sorts. This can be simulated in GILF by introducing a unit node for every concept that contains corresponding function and type declarations.

At this point, we can translate SUCHTHAT to GILF and let the back-end perform its work. However, the intermediate code does not contain any semantic information expressed in TECTON that restrict legal concept instances. This kind of front-end language specific information can be stored in GILF's `extend` node[2].

D.2 Factorial

D.2.1 XGILF Representation

This section presents the example discussed in chapter 2. The factorial function is translated to XGILF manually, providing an iterative and a recursive algorithm for its computation. The declarations and defintions are located in unit `u_math`. Unit `u_main` contains a simple main application that calls the factorial function with a constant value, which requires complete binding information. Notice how the called function becomes a required function of algorithm `a_main`.

"../xgf/examples/factorial.xgf" 163 ≡

```
<?xml version="1.0" standalone="yes"?>
<?gilf version="1.0" protocol="xgilf"?>
<?xgilf digest="b78065c29fbbd41ea10ae9ebd7db3607" version="1.0"?>
<!--===========================================================================
```

[1] TECTON declarations are discussed in section 2.3.

[2] The conclusions mention this aspect of GILF in more detail.

```
* Example file for XGILF: factorial.xgf
=============================================================================-->
<xgilf>

 <!--===========================================================================-->
 <!-- user defined unit: u_math -->
 <unit name="math" id="u_math">
  <!-- unit dependencies -->
  <import>
   <source input-src="function.xgf"> <unit-dsg ref="u_func"/> </source>
  </import>

  <!-- declaration part, shared among system components -->
  <declare>
   <!-- function factorial: Calculates the factorial for a given number. -->
   <function id="u_math.f_fact" name="factorial">
    <type-params count="1"> <type-param id="tp_0" name="N"/> </type-params>
    <params count="2">
     <param pass="in" id="p_0" name="n"> <static-param-dsg ref="tp_0"/> </param>
     <param pass="out!" id="p_1" name="n!"> <static-param-dsg ref="tp_0"/> </param>
    </params>
   </function>
  </declare>

  <!-- definition part, for code generation -->
  <define>
   ⟨Iterative Factorial Algorithm 165⟩
   ⟨Recursive Factorial Algorithm 166⟩
  </define>
 </unit> <!-- math [u_math] -->

 <!--===========================================================================-->
 <!-- user defined main unit [u_main] -->
 <unit name="main" id="u_main">
  <!-- unit dependencies -->
  <import>
   <source input-src="function.xgf"> <unit-dsg ref="u_func"/> </source>
   <source input-src="machtype.xgf"> <unit-dsg ref="u_mtype"/> </source>
   <source input-src="factorial.xgf"> <unit-dsg ref="u_math"/> </source>
  </import>

  <define>
   <!-- main algorithm -->
   <algorithm ref="u_func.f_main" id="u_main.a_main" name="main">
    <!-- list of required generic functions -->
    <func-params count="1">
     <func-param id="r_fact"> <func-dsg ref="u_math.f_fact"/> </func-param>
    </func-params>

    <!-- local symbol table -->
    <store>
     <const id="c_0">
      <binding-dsg ref="u_mtype.bt_uword"/><val kind="dec" val="21"/>
     </const>
     <const id="c_1">
      <binding-dsg ref="u_mtype.bt_word"/><val kind="dec" val="0"/>
     </const>
    </store>

    <!-- algorithm body -->
```

```
    <stat-seq>
     <call ref="u_math.f_fact">
      <binding-dsg ref="u_main.bf_fact"/>
      <bind-params>
       <bind-param ref="p_0"> <expr><const-dsg ref="c_0"/></expr> </bind-param>
      </bind-params>
     </call>
     <return> <expr><const-dsg ref="c_1"/></expr> </return>
    </stat-seq>
   </algorithm>
  </define>

  <!-- binding table for the main unit -->
  <bind>
   <!-- bind main function -->
   <bind-func id="u_main.bf_main" ref="u_func.f_main">
    <algo-dsg ref="u_main.a_main"/>
   </bind-func>
   <!-- bind algorithm(s) to factorial function -->
   ⟨Bindings for Function Factorial 168⟩
  </bind>
 </unit>

</xgilf>
```

The iterative algorithm introduces three dependent function symbols, two for arithmetic operations and one for comparison. The implemenation of the algorithm is straightforward, it uses the GILF for statement node.

⟨Iterative Factorial Algorithm 165⟩ ≡

```
<!-- ## Iterative implementation of factorial. ## -->
<algorithm id="u_math.a_facti" ref="u_math.f_fact" name="factorial_iter" >

 <!-- list of required generic functions -->
 <func-params count="3">
  <func-param id="r_*"> <func-dsg ref="u_func.f_*"/> </func-param>
  <func-param id="r_lte"> <func-dsg ref="u_func.f_lte"/> </func-param>
  <func-param id="r_++"> <func-dsg ref="u_func.f_++"/> </func-param>
 </func-params>

 <!-- symbol table local to algorithm -->
 <store>
  <!-- counter i -->
  <var name="i" id="v_0"> <static-param-dsg ref="tp_0"/> </var>
  <!-- constant 1 -->
  <const id="c_0"> <static-param-dsg ref="tp_0"/><val val="1" kind="dec"/> </const>
 </store>

 <!-- now the algorithm definition -->
 <stat-seq>
  <label id="l_0">step 1: init n!</label>
  <assign>
   <var-dsg ref="p_1"/> <expr> <const-dsg ref="c_0"/> </expr>
  </assign>

  <label id="l_1">step 2: compute factorial</label>
  <for>
   <expr> <!-- i<=n -->
    <call ref="u_func.f_lte">
     <static-param-dsg ref="r_lte"/>
```

```
      <bind-params>
       <bind-param ref="p_0"> <expr><var-dsg ref="v_0"/></expr> </bind-param>
       <bind-param ref="p_1"> <expr><var-dsg ref="p_0"/></expr> </bind-param>
      </bind-params>
     </call>
    </expr>
    <for-pre> <!-- i=1 -->
     <stat-seq>
      <assign>
       <var-dsg ref="v_0"/>
       <expr> <const-dsg ref="c_0"/> </expr>
      </assign>
     </stat-seq>
    </for-pre>
    <for-post> <!-- i++ -->
     <stat-seq>
      <call ref="u_func.f_++">
       <static-param-dsg ref="r_++"/>
       <bind-params>
        <bind-param ref="p_0"> <expr><var-dsg ref="v_0"/></expr> </bind-param>
       </bind-params>
      </call>
     </stat-seq>
    </for-post>
    <stat-seq>
     <assign> <!-- n! = n! * i -->
      <var-dsg ref="p_1"/>
      <expr>
       <call ref="u_func.f_*">
        <static-param-dsg ref="r_*"/>
        <bind-params>
         <bind-param ref="p_0"> <expr><var-dsg ref="p_1"/></expr> </bind-param>
         <bind-param ref="p_1"> <expr><var-dsg ref="v_0"/></expr> </bind-param>
        </bind-params>
       </call>
      </expr>
     </assign>
    </stat-seq>
   </for>

   <label id="l_2">step 3</label>
   <return> <expr><var-dsg ref="p_1"/></expr> </return>
  </stat-seq>
 </algorithm>
```

Definition referenced in part 163.

The recursive algorithm for computing a number's factorial differs technically in one major aspect from the iterative one presented before. It introduces one additional dependent function symbol `r_factr`. This symbol represents the recursive invocation of the algorithm itself. Its use is exemplified in the algorithm's body.

⟨Recursive Factorial Algorithm 166⟩ ≡

```
 <!-- ## Recursive implementation of factorial. ## -->
 <algorithm name="factorial_rec" id="u_math.a_factr" ref="u_math.f_fact">

  <!-- list of required generic functions -->
  <func-params count="4">
   <func-param id="r_*"> <func-dsg ref="u_func.f_*"/> </func-param>
   <func-param id="r_lte"> <func-dsg ref="u_func.f_lte"/> </func-param>
```

```
 <func-param id="r_-"> <func-dsg ref="u_func.f_-"/> </func-param>
 <func-param id="r_factr"> <func-dsg ref="u_math.f_fact"/> </func-param>
</func-params>

<!-- symbol table local to algorithm -->
<store>
 <!-- constant 1 -->
 <const id="c_0"> <static-param-dsg ref="tp_0"/> <val val="1" kind="dec"/> </const>
</store>

<stat-seq>
 <if>
  <!-- if (n <= 1) return 1 -->
  <expr>
   <call ref="u_func.f_lte">
    <static-param-dsg ref="r_lte"/>
    <bind-params>
     <bind-param ref="p_0"> <expr><var-dsg ref="p_0"/></expr> </bind-param>
     <bind-param ref="p_1"> <expr><const-dsg ref="c_0"/></expr> </bind-param>
    </bind-params>
   </call>
  </expr>
  <stat-seq>
   <return> <expr><const-dsg ref="c_0"/></expr> </return>
  </stat-seq>
  <!-- return n*fact(n-1) -->
  <else>
   <stat-seq>
    <return>
     <expr><call ref="u_func.f_*"> <!-- n*fact(n-1) -->
      <static-param-dsg ref="r_*"/>
      <bind-params>
       <bind-param ref="p_0"> <expr><var-dsg ref="p_0"/></expr> </bind-param>
       <bind-param ref="p_1">
        <expr><call ref="u_math.f_fact"> <!-- fact(n-1) -->
         <static-param-dsg ref="r_factr"/>
         <bind-params>
          <bind-param ref="p_0">
          <expr><call ref="u_func.f_-"> <!-- n-1 -->
           <static-param-dsg ref="r_-"/>
           <bind-params>
            <bind-param ref="p_0"> <expr><var-dsg ref="p_0"/></expr> </bind-param>
            <bind-param ref="p_1"> <expr><const-dsg ref="c_0"/></expr> </bind-param>
           </bind-params>
          </call></expr>
          </bind-param>
         </bind-params>
        </call></expr>
       </bind-param>
      </bind-params>
     </call></expr>
    </return>
   </stat-seq>
  </else>
 </if>
</stat-seq>

</algorithm>
```

Definition referenced in part 163.

Finally, we have to provide the bindings of the factorial function `f_fact` to its implementing algorithms. We present the binding that will result in an instantiation with the built-in unsigned integer type, in GILF available through data structure `u_mtype.bt_uword`.

```
⟨Bindings for Function Factorial 168⟩ ≡
  <bind-func id="u_main.bf_fact" ref="u_math.f_fact">
   <bind-static-params>
    <bind-tp ref="tp_0"> <binding-dsg ref="u_mtype.bt_uword"/> </bind-tp>
   </bind-static-params>
   <!-- bind recursive algorithm to factorial function -->
   <algo-dsg ref="u_math.a_factr">
    <bind-static-params>
     <bind-fp ref="r_lte">
      <binding-dsg ref="u_mtype.bf_lte_uword"/>
     </bind-fp>
     <bind-fp ref="r_*">
      <binding-dsg ref="u_mtype.bf_*_uword"/>
     </bind-fp>
     <bind-fp ref="r_-">
      <binding-dsg ref="u_mtype.bf_-_uword"/>
     </bind-fp>
     <bind-fp ref="r_factr">
      <binding-dsg ref="u_main.bf_fact"/>
     </bind-fp>
    </bind-static-params>
   </algo-dsg>
   <!-- bind iterative algorithm to factorial function -->
   <algo-dsg ref="u_math.a_facti">
    <bind-static-params>
     <bind-fp ref="r_lte">
      <binding-dsg ref="u_mtype.bf_lte_uword"/>
     </bind-fp>
     <bind-fp ref="r_*">
      <binding-dsg ref="u_mtype.bf_*_uword"/>
     </bind-fp>
     <bind-fp ref="r_++">
      <binding-dsg ref="u_mtype.bf_++_uword"/>
     </bind-fp>
    </bind-static-params>
   </algo-dsg>
  </bind-func>
```

Definition referenced in part 163.

D.2.2 Generated C++ Representation

The following C++ code is produced by the GILF prototype. We assume that the iterative algorithm is selected and the instantiator (see section 5.6) is called on binding `u_main.bf_fact`. It would be desirable to replace the mangled names by more accessible identifiers. Notice how the generic constructs in the GILF representation presented above where removed by the instantiator.

```
typedef bool u_bool_DOTd_bool;
typedef unsigned int u_mtype_DOTd_uword;
inline u_bool_DOTd_bool u_mtype_DOTa_lte_uword_DOTtp_0_OPu_mtype_DOTd_uword_CP
(u_mtype_DOTd_uword a, u_mtype_DOTd_uword b)
{ return a <= b; }
inline u_mtype_DOTd_uword u_mtype_DOTa__PLUS_PLUS_uword_DOTtp_0_OPu_mtype_DOTd_uword_CP
(u_mtype_DOTd_uword& a)
{ return ++a; }
```

```
inline u_mtype_DOTd_uword u_mtype_DOTa__MULT_uword_DOTtp_0_OPu_mtype_DOTd_uword_CP
(u_mtype_DOTd_uword a, u_mtype_DOTd_uword b)
{ return a * b; }
u_mtype_DOTd_uword u_math_DOTa_facti_DOTr__MULT_OPu_mtype_DOTa__MULT_uword_DOTtp_0_OPu_\
mtype_DOTd_uword_CP_CP_DOTr__PLUS_PLUS_OPu_mtype_DOTa__PLUS_PLUS_uword_DOTtp_0_OPu_mtype_\
DOTd_uword_CP_CP_DOTr_lte_OPu_mtype_DOTa_lte_uword_DOTtp_0_OPu_mtype_DOTd_uword_CP_CP_\
DOTtp_0_OPu_mtype_DOTd_uword_CP(u_mtype_DOTd_uword p_0)
{
u_mtype_DOTd_uword p_1;
u_mtype_DOTd_uword v_0;
const u_mtype_DOTd_uword c_0 = 1;
l_0:
p_1 = c_0;
l_1:
// for precode
v_0 = c_0;
for (;u_mtype_DOTa_lte_uword_DOTtp_0_OPu_mtype_DOTd_uword_CP(v_0, p_0);) {
p_1 = u_mtype_DOTa__MULT_uword_DOTtp_0_OPu_mtype_DOTd_uword_CP(p_1, v_0);
// for postcode
u_mtype_DOTa__PLUS_PLUS_uword_DOTtp_0_OPu_mtype_DOTd_uword_CP(v_0);
}
l_2:
return p_1;
}
```

D.3 Regression Tests

The file regress.xgf contains regression tests that check the basic functionality of our GILF implementation.

"../xgf/examples/regress.xgf" 169 ≡

```
<?xml version="1.0" standalone="yes"?>
<?gilf version="1.0" protocol="xgilf"?>
<?xgilf digest="94594ccdb12b395c433dfabf27f4b4fe" version="1.0"?>

<xgilf>
 <unit id="u_main">

  <import>
   <source input-src="machtype.xgf">
    <unit-dsg ref="u_mtype"/>
   </source>
  </import>

  <store>
   <const id="u_main.c_test">
    <binding-dsg ref="u_math.bt_uint32"/>
    <val count="0" kind="dec" val="4711"/>
   </const>
   <const id="u_main.c_test2">
    <binding-dsg ref="u_math.bt_uint32"/>
    <val count="0" kind="hex" val="FFFF"/>
   </const>
   <const id="u_main.c_test3">
    <binding-dsg ref="u_mword.bt_bool"/>
    <val count="0" kind="bool" val="true"/>
   </const>
   <const id="u_main.c_test4">
    <binding-dsg ref="u_math.bt_float"/>
```

```
   <val count="0" kind="real" val="3.1415926"/>
  </const>
 </store>

 <declare>
  <!-- Homogeneous Pair (declaration). -->
  <type id="u_main.t_pair" name="pair">
   <type-params count="1">
    <type-param id="tp_0" name="T"/>
   </type-params>
  </type>

  <!-- Heterogeneous Pair (declaration). -->
  <type id="u_main.t_pair_het" name="pair">
   <type-params count="2">
    <type-param id="tp_0" name="T0"/>
    <type-param id="tp_1" name="T1"/>
   </type-params>
  </type>

  <!-- Pair of words. -->
  <type id="u_main.t_pair_word"/>

  <!-- Allocate test funtion. -->
  <function id="u_main.f_allocate_tst"/>

 </declare>

 <define>
  <!-- Homogeneous Pair (definition). -->
  <data id="u_main.d_pair" ref="u_main.t_pair" kind="record">
   <elem id="e1"><static-param-dsg ref="tp_0"/></elem>
   <elem id="e2"><static-param-dsg ref="tp_0"/></elem>
  </data>

  <!-- Heterogeneous Pair (definition). -->
  <data id="u_main.d_pair_het" ref="u_main.t_pair_het" kind="record">
   <elem id="e1"><static-param-dsg ref="tp_0"/></elem>
   <elem id="e2"><static-param-dsg ref="tp_1"/></elem>
  </data>

  <!-- Pair of words. -->
  <data id="u_main.d_pair_word" ref="u_main.t_pair_word" kind="record">
   <elem id="e1"><binding-dsg ref="u_mtype.bt_word"/></elem>
   <elem id="e2"><binding-dsg ref="u_mtype.bt_word"/></elem>
  </data>

  <!-- Test allocate special funtion generation. -->
  <algorithm ref="u_main.f_allocate_tst" id="u_main.a_allocate_tst">
   <!-- local symbol table -->
   <store>
    <var id="v_0" type-mod="is-ref"><binding-dsg ref="u_mtype.bt_uword"/></var>
   </store>
   <stat-seq>
    <assign>
     <var-dsg ref="v_0"/>
     <expr><call ref="u_func.f_allocate">
      <binding-dsg ref="u_mtype.bt_uword"/>
     </call></expr>
    </assign>
```

```
  </stat-seq>
 </algorithm>

</define>

<bind>
 <!-- Homogeneous Pair (instantiation). -->
 <bind-type id="u_main.bt_pair_0" ref="u_main.t_pair">
  <!-- (A) describe instance -->
  <bind-static-params id="local">
   <bind-tp ref="tp_0"><binding-dsg ref="u_mtype.bt_uword"/></bind-tp>
  </bind-static-params>
  <!-- (B) bind type to data structure -->
  <data-dsg ref="u_main.d_pair"/>
 </bind-type>

 <!-- Heterogeneous Pair (instantiations). -->
 <bind-type id="u_main.bt_pair_het0" ref="u_main.t_pair_het">
  <!-- (A) describe instance -->
  <bind-static-params id="local">
   <bind-tp ref="tp_0"><binding-dsg ref="u_mtype.bt_uword"/></bind-tp>
   <bind-tp ref="tp_1"><binding-dsg ref="u_mtype.bt_byte"/></bind-tp>
  </bind-static-params>
  <!-- (B) bind type to data structure -->
  <data-dsg ref="u_main.d_pair_het"/>
 </bind-type>

 <bind-type id="u_main.bt_pair_het1" ref="u_main.t_pair_het">
  <!-- (A) describe instance -->
  <bind-static-params id="local">
   <bind-tp ref="tp_0"><binding-dsg ref="u_mtype.bt_word"/></bind-tp>
  </bind-static-params>
  <binding-dsg ref="u_main.bsp_test"/>
  <!-- (B) bind type to data structure -->
  <data-dsg ref="u_main.d_pair_het"/>
 </bind-type>

 <bind-type id="u_main.bt_pair_het2" ref="u_main.t_pair_het">
  <!-- (A) describe instance -->
  <bind-static-params id="local">
   <bind-tp ref="tp_0"><binding-dsg ref="u_mtype.bt_uword"/></bind-tp>
   <bind-tp ref="tp_1"><binding-dsg ref="u_main.bt_pair_het1"/></bind-tp>
  </bind-static-params>
  <!-- (B) bind type to data structure -->
  <data-dsg ref="u_main.d_pair_het"/>
 </bind-type>

 <!-- Pair of words. -->
 <bind-type id="u_main.bt_pair_word" ref="u_main.t_pair_word">
  <data-dsg ref="u_main.d_pair_word"/>
 </bind-type>

 <!-- Array of words. -->
 <bind-type id="u_main.bt_array_word" ref="u_array.t_[]">
  <!-- describe instance -->
  <bind-static-params id="local">
   <bind-tp ref="tp_0"><binding-dsg ref="u_mtype.bt_word"/></bind-tp>
  </bind-static-params>
  <data-dsg ref="u_array.d_[]"/>
 </bind-type>
```

```
    <!-- Array of array of words (flattened with bind-dsg). -->
    <bind-type id="u_main.bt_aarray_word" ref="u_array.t_[]">
     <!-- describe instance -->
     <bind-static-params id="local">
      <bind-tp ref="tp_0"><binding-dsg ref="u_main.bt_array_word"/></bind-tp>
     </bind-static-params>
     <data-dsg ref="u_array.d_[]"/>
    </bind-type>

    <!-- Array of array of words (nested bind-type). -->
    <bind-type id="u_main.bt_aarray2_word" ref="u_array.t_[]">
     <!-- describe instance -->
     <bind-static-params id="local">
      <bind-tp ref="tp_0">

       <bind-type id="u_main.bt_local_1" ref="u_array.t_[]">
       <!-- describe instance -->
        <bind-static-params id="local">
         <bind-tp ref="tp_0"><binding-dsg ref="u_mtype.bt_word"/></bind-tp>
        </bind-static-params>
        <data-dsg ref="u_array.d_[]"/>
       </bind-type>

      </bind-tp>
     </bind-static-params>
     <data-dsg ref="u_array.d_[]"/>
    </bind-type>

    <bind-static-params id="u_main.bsp_test">
     <bind-tp ref="tp_1"><binding-dsg ref="u_mtype.bt_uword"/></bind-tp>
    </bind-static-params>

    <!-- Bind allocate test funtion. -->
    <bind-func id="u_main.bf_allocate_tst" ref="u_main.f_allocate_tst">
     <algo-dsg ref="u_main.a_allocate_tst"/>
    </bind-func>
   </bind>

  </unit>
 </xgilf>
```

Colophon

This document was created with the LaTeX typesetting system using a modified report style. The modifications mainly affect chapter and section headings which are typeset in sans serif font.

We use postscript fonts throughout this document. The serif font is Palatino, Avant Garde is used as sans serif font, and the font for monospaced characters is Letter Gothic. The font size for standard text is eleven points.

All illustrations were created with Visio, except for figure 2.1 which was generated with GnuPlot. Limitations in the EPS export filter of Visio forced us to use "standard postscript fonts" in illustrations, which are Times New Roman, Helvetica, and Courier, respectively. For PDF generation, the exported EPS illustrations are converted to PDF with `epstopdf` which leaves the bounding boxes intact, in contrast to Adobe Acrobat Distiller.

The XGILF specification and SUCHTHAT examples are documented in literate programming style, this means the displayed code is embedded in the document source. It is the single source for both documentation and code, like the XGILF DTD loaded by XML parsers for validation. The literate programming tool employed is a modified version of `nuweb` [BrRaMe02] which supports syntax highlighting for languages available in the `listings` LaTeX package.

Another approach to the one source philosophy for documentation and code is taken in the ProgDOC system [Sim02]. Code listings of the GILF prototype implementation, of related code, and of the XGILF core library are extracted from the original source files. The extracted lines have to be marked up with special comments, and the ProgDOC system is used as extractor. The extracted listings are wrapped into the `listings` LaTeX package with a handcrafted C++ program. This generated LaTeX code is very similar to the one in the mentioned nuweb extensions.

Both approaches to code documentation have their merits, depending on the size and kind of documented code. A large code base with many technical details and repetetive sections lends itself to a ProgDOC documentation style, whereas concise and dense program or specification code is well suited for traditional literate programming.

The original document can be processed with either LaTeX or `pdf`LaTeX in order to create Postscript or PDF output. The PDF output is completely hyper-linked with help of the `hyperref` LaTeX package.

Bibliography

[AADEBUG97] *Third International Workshop on Automated Debugging.* Linköping, Sweden, May 26-27, 1997. Electronic proceedings available at www.ep.liu.se/ea/cis/1997/009.

[AADEBUG00] *Fourth International Workshop on Automated Debugging.* Munich, Germany, August 28-30th, 2000. Electronic proceedings available at xxx.lanl.gov/abs/cs.SE/0010035.

[AbCo01] DAVID ABRAHAMS, AND CARLOS PINTO COELHO. *Effects of Metaprogramming Style on Compilation Time.* Available at www.boost.org.

[AdTiWe94] ROLF ADAMS, WALTER F. TICHY, AND ANNETTE WEINERT. *The Cost of Selective Recompilation and Environment Processing.* ACM, Transactions on Software Engineering and Methodology (TOSEM), Volume 3(1), 1994.

[Adv99] ADVANCED MICRO DEVICES, INC. *3DNow! Technology Manual*, Order Number 21928, Advanced Micro Devices, Inc., 1999.

[AgFrMi97] OLE AGESEN, STEPHEN N. FREUND, AND JOHN C. MITCHELL. *Adding Type Parameterization to the Java^{TM} Language.* ACM, SIGPLAN Notices 32(10), Proceedings of [OOPSLA97], 1997.

[AhSeUl86] ALFRED V. AHO, RAVI SETHI, AND JEFFREY D. ULLMAN. *Compilers - Principles, Techniques and Tools.* Addison-Wesley Publishing Company, 1986.

[Ale00] ANDREI ALEXANDRESCU. *Traits: The else-if-then of Types.* SIGS Publications, C++ Report, Vol. 12(4), April 2000.

[Ale01] ANDREI ALEXANDRESCU. *Modern C++ Design. Generic Programming and Design Patterns Applied.* Addison-Wesley Publishing Company, 2001.

[AmBaLa97] GLENN AMMONS, THOMAS BALL, AND JAMES R. LARUS. *Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling.* Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI'97), Las Vegas, Nevada, USA, SIGPLAN Notices 32(5), May 1997.

[AnBaBi$^+$90] E. ANDERSON, Z. BAI, C. BISCHOF, J. DEMMEL, J. DONGARRA, J. DU CROZ, A. GREENBAUM, S. HAMMARLING, A. MCKENNEY, AND D. SORENSEN. *LAPACK: A portable linear algebra library for high-performance computers.* Computer Science Deptartment Technical Report CS-90-105, University of Tennessee, Knoxville, Tennessee, May 1990.

[AnBeDe+97] JENNIFER M. ANDERSON, LANCE M. BERC, JEFFREY DEAN, SANJAY GHEMAWAT, MONIKA R. HENZINGER, SHUN-TAK A. LEUNG, RICHARD L. SITES, MARK T. VANDEVOORDE, CARL A. WALDSPURGER, AND WILLIAM E. WEIHL. *Continuous Profiling: Where Have All the Cycles Gone?* ACM Transactions on Computer Systems (TOCS), Vol. 15(4), November 1997.

[ApDaRa98] ANDREW W. APPEL, JACK DAVIDSON, AND NORMAN RAMSEY. *The Zephyr Compiler Infrastructure.* University of Virginia, USA, 1998. Available at www.cs.virginia.edu/zephyr.

[App98] ANDREW W. APPEL. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.

[App98b] ANDREW W. APPEL. *SSA is Functional Programming.* ACM, SIGPLAN Notices 33(4), pp. 17-20, 1998.

[ApMa94] ANDREW W. APPEL, AND DAVID B. MACQUEEN. *Separate Compilation for Standard ML.* Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94), Orlando, Florida, USA, ACM, SIGPLAN Notices 29(6), June 1994.

[ASF] APACHE SOFTWARE FOUNDATION. *The Apache XML Project.* Available at xml.apache.org.

[AsKrKr99] EGIDIO ASTESIANO, HANS-JÖRG KREOWSKI, AND BERND KRIEG-BRCKNER (EDS.). *Algebraic Foundations of System Specification.* IFIP State-of-the-Art Reports, Springer, 1999.

[AtFlIg98] GIUSEPPE ATTARDI, TITO FLAGELLA, AND PIETRO IGLIO. *A Customisable Memory Management Framework for C++.* John Wiley & Sons, Ltd, Software – Practice and Experience, Vol. 28, No. 11, pp. 1143-1183, 1998.

[Aus98] MATTHEW H. AUSTERN. *Generic Programming and the STL.* Addison-Wesley Publishing Company, 1998.

[AyJoPe+98] ANDREW AYERS, STUART DE JONG, JOHN PEYTON, AND RICHARD SCHOOLER. *Scalable Cross-Module Optimization.* Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98), Montreal, Canada, ACM, SIGPLAN Notices 33(5), May 1998.

[Bak82] THEODORE P. BAKER. *A One-Pass Algorithm for Overload resolution in Ada.* ACM Transactions on Programming Languages and Systems, Vol. 4, No. 4, pp. 601-614, October 1982.

[BaLa94] THOMAS BALL, AND JAMES R. LARUS. *Optimally Profiling and Tracing Programs.* ACM Transactions on Programming Languages and Systems, Vol. 16, No. 3, July 1994.

[BaLa96] THOMAS BALL, AND JAMES R. LARUS. *Efficient Path Profiling.* Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture, Paris, France, December 2-4, 1996.

[Bar88] JOEL F. BARTLETT. *Compacting Garbage Collection with Ambigous Roots.* DEC Research Report 88/2, February 1988.

[Bar89] JOEL F. BARTLETT. *Mostly-Copying Garbage Collection Picks Up Generations and C++.* DEC Technical Note TN-12, October 1989.

[Bar95] JOHN G. P. BARNES. *Programming in Ada95.* Addison-Wesley Publishing Company, 1995.

[BaDiMa97] RONALD BAECKER, CHRIS DIGIANO, AND AARON MARCUS. *Software Visualization for Debugging.* ACM, Communications of the ACM, Vol. 40, No. 4, pp. 44-54, April 1997.

[BeSe97] JON LOUIS BENTLEY, AND ROBERT SEDGEWICK. *Fast Algorithms for Sorting and Searching Strings.* Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 360-369, New Orleans, Louisiana, USA, January 1997.

[Boost] BOOST COMMUNITY. *Boost C++ Libraries.* Available at www.boost.org.

[BöGuPo00] LÁSLÓ BÖSZÖRMÉNYI, JÜRG GUTKNECHT AND GUSTAV POMBERGER (EDITORS). *The School of Niklaus Wirth: The Art of Simplicity.* Morgan Kaufmann Publishers, San Fransisco, California, USA, 2000.

[BoDa98] BORIS BOKOWSKI AND MARKUS DAHM. *Poor Man's Genericity for Java.* Lecture Notes in Computer Science, Vol. 1543, Springer, 1998.

[BoJo97] SIMON P. BOOTH, AND SIMON B. JONES. *Walk Backwards to Happiness – Debugging by Time Travel.* Proceedings of [AADEBUG97], 1997.

[BoWe88] HANS-JUERGEN BOEHM, AND MARK WEISER. *Garbage Collection in an Uncooperative Environment.* John Wiley & Sons, Ltd, Software – Practice and Experience, Vol. 18, No. 9, pp. 807-820, 1988.

[BrRaMe02] PRESTON BRIGGS, JOHN D. RAMSDELL, AND MARC W. MENGEL. *Nuweb Version 1.0b1. A Simple Literate Programming Tool.* Available at nuweb.sourceforge.net.

[Bra95] MARC MICHAEL BRANDIS. *Optimizing Compilers for Structured Programming Languages.* Ph.D. thesis, Swiss Federal Institute of Technology Zürich, 1995.

[Bra96] GILAD BRACHA. *The Strongtalk Type System for Smalltalk.* Workshop on Extending the Smalltalk Language at [OOPSLA96], 1996. Available at java.sun.com/people/gbracha.

[BrCoKe+01] GILAD BRACHA, NORMAN COHEN, CHRISTIAN KEMPER, STEVE MARX, MARTIN ODERSKY, SVEN-ERIC PANITZ, DAVID STOUTAMIRE, KRESTEN THORUP, AND PHILIP WADLER. *Adding Generics to the Java Programming Language: Participant Draft Specification.* Sun Microsystems, Inc., 2001. Available at developer.java.sun.com.

[BrGr93] GILAD BRACHA, AND DAVID GRISWOLD. *Strongtalk: Typechecking Smalltalk in a Production Environment.* Proceedings of [OOPSLA93], ACM, SIGPLAN Notices 28(10), 1993.

[BrDoGa+00] SHIRLEY BROWNE, JACK DONGARRA, N. GARNER, KEVIN S. LONDON, AND P. MUCCI. *A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters.* Supercomputing 2000, Dallas, Texas, USA, November 4-10, 2000. Electronic proceedings available at www.supercomp.org/sc2000/Proceedings/start.htm.

[BrSeMu00] ILJA N. BRONSTEIN, KONSTANTIN A. SEMENDJAJEW, AND GERHARD MUSIOL. *Taschenbuch der Mathematik.* 5. Auflage, Verlag Harri Deutsch, Franfurt/Main, September 2000.

[BrOdSt+98] GILAD BRACHA, MARTIN ODERSKY, DAVID STOUTAMIRE, AND PHILIP WADLER. *Making the future safe for the past: Adding Genericity to the Java^{TM} Programming Language.* Proceedings of [OOPSLA98], ACM, SIGPLAN Notices 33(10), 1998.

[Bru93] KIM B. BRUCE. *Safe Type Checking in a Statically-Typed Object-Oriented Programming Language.* ACM, Conference Record of [POPL93], 1993.

[BrCaCa+95] KIM B. BRUCE, LUCA CARDELLI, GIUSEPPE CASTAGNA, JONATHAN EIFRIG, SCOTT F. SMITH, VALERY TRIFONOV, GARY T. LEAVENS, AND BENJAMIN C. PIERCE. *On Binary Methods.* Theory and Practice of Object Systems (TAPOS), Volume 1(3), 1995.

[BrFiSc95] KIM B. BRUCE, ADRIAN FIECH, AND ANGELA SCHUETT. *PolyTOIL: A typesafe polymorphic object-oriented language.* Proceedings of ECOOP'95. Lecture Notes in Computer Science, Vol. 952, Springer, 1998.

[BrOdWa98] KIM B. BRUCE, MARTIN ODERSKY, AND PHILIP WADLER. *A Statically Safe Alternative to Virtual Types.* Proceedings of ECOOP'98. Lecture Notes in Computer Science, Vol. 1445, Springer, 1998.

[Bun95] JØRGEN BUNDGAARD. *An ANDF Based Ada 95 Compiler System.* Lecture Notes in Computer Science, Vol. 1031, Springer, 1995.

[CaCoHi+89] PETER S. CANNING, WILLIAM COOK, WALTER L. HILL, WALTER G. OLTHOFF, AND JOHN C. MITCHELL. *F-Bounded Polymorphism for Object-Oriented Programming.* FPCA '89, Conference on Functional Programming Languages and Computer Architecture, London, England, September 1989.

[Car97] LUCA CARDELLI. *Program fragments, Linking, and Modularization.* ACM, Conference Record of [POPL97], pp. 266-277, 1997.

[CaDoGl+89] LUCA CARDELLI, JAMES DONAHUE, LUCILLE GLASSMAN, MICK JORDAN, BILL KALSOW, AND GREG NELSON. *Modula-3 Report (revised).* Research Report SRC-52, Systems Research Center, Digital Equipment Corporation, November 1989.

[CaWe85] LUCA CARDELLI, AND PETER WEGNER. *On Understanding Types, Data Abstraction, and Polymorphism.* ACM Computing Surveys, Volume 17(4), pp. 471-522, December 1985.

[CaSt98] ROBERT CARTWRIGHT, AND GUY L. STEELE JR. *Compatible Genericity with Runtime Types for the Java^{TM} Programming Language.* Proceedings of [OOPSLA98], ACM, SIGPLAN Notices 33(10), 1998.

[CyFeRo+91] RON CYTRON, JEANNE FERRANTE, BARRY K. ROSEN, MARK N. WEGMAN, AND F. KENNETH ZADECK. *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph.* ACM Transactions on Programming Languages and Systems, Vol. 13, No. 4, pp. 451-490, 1991.

[Cha98] CRAIG CHAMBERS, AND THE CECIL GROUP. *The Cecil Language. Specification and Rationale.* Department of Computer Science and Enginieering, University of Washington, Seatle, USA, December 1998.

[Che70] C. J. CHENEY. *A Nonrecursive List Compacting Algorithm.* ACM, Communications of the ACM, Vol. 13, No. 11, pp. 677-678, November 1970.

[ClLoGi99] MARSHALL CLINE, GREG LOMOW, AND MIKE GIROU. *C++ FAQs. Second Edition.* Addison-Wesley Publishing Company, 1999.

[Col60] GEORGE E. COLLINS. *A method for overlapping and erasure of lists.* ACM, Communications of the ACM, Vol. 3, No. 12, pp. 655-657, 1960.

[CoLo90] GEORGE E. COLLINS, AND RÜDIGER G. K. LOOS. *Specifications and Index of SAC-2 Algorithms.* Technical Report WSI 90-4, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, 1990.

[Con58] MELVIN E. CONWAY. *Proposal for an UNCOL.* ACM, Communications of the ACM, Vol. 1, No. 10, pp. 5-8, October 1958.

[Cur95] I. F. CURRIE. *TDF Specification, Issue 4.0.* Technical report, Defense Research Agency, Worcestershire, United Kingdom, 1995. Available at www.tendra.org.

[CzEi00] KRZYSZTOF CZARNECKI, AND ULRICH W. EISENECKER. *Generative Programming. Methods, Tools, and Applications.* Addison-Wesley Publishing Company, 2000.

[DaGrLi+95] MARK DAY, ROBERT GRUBER, BARBARA LISKOV, AND ANDREW C. MYERS. *Subtypes vs. Where Clauses: Constraining Parametric Polymorphism.* Proceedings of [OOPSLA95], ACM, SIGPLAN Notices 30(10), 1995.

[DOM00] ARNAUD LE HORS, PHILIPPE LE HGARET, LAUREN WOOD, GAVIN NICOL, JONATHAN ROBIE, MIKE CHAMPION, AND STEVE BYRNE. *Document Object Model (DOM) Level 2 Core Specification. Version 1.0.* W3C Recommendation, November 2000. Available at [W3Tr].

[DOMT00] ARNAUD LE HORS, PHILIPPE LE HÉGARET, LAUREN WOOD, GAVIN NICOL, JONATHAN ROBIE, MIKE CHAMPION, AND STEVE BYRNE. *Document Object Model (DOM) Level 2 Traversal and Range Specification. Version 1.0.* W3C Recommendation, November 2000. Available at [W3Tr].

[DuCoIa+97] ANDREW DUNCAN, BOGDAN COCOSEL, COSTIN IANCU, HOLGER KIENLE, RADU RUGINA, URS HÖLZLE, AND MARTIN RINARD. *SUIF: SUIF 2.0 With Objects.* Second SUIF Compiler Workshop, Stanford University, USA 1997. Available at suif.stanford.edu/suifconf/suifconf2.

[ECMA01] ECMA. *Common Language Infrastructure (CLI). Partitions I to IV.* Standard ECMA-335, December 2001. Available at www.ecma.ch.

[ECOOP98] *12. European Conference on Object-Oriented Programming (ECOOP'98).* Brussels, Belgium, 1998.

[Edi00] EDISON DESIGN GROUP, INC. *C++ Front End. Internal Documentation.* Edison Design Group, Inc., New Jersey, USA, December 2000. Available at www.edg.com/cpp.html.

[EiSmTr95] JONATHAN EIFRIG, SCOTT F. SMITH, AND VALERY TRIFONOV. *Sound Polymorphic Type Inference for Objects.* Proceedings of [OOPSLA95], ACM, SIGPLAN Notices 30(10), 1995.

[Eis97] MARC EISENSTADT. *"My Hairiest Bug" War Stories.* ACM, Communications of the ACM, Vol. 40, No. 4, pp. 30-37, April 1997.

[Elm97] KIM ELMS. *Debugging Optimised Code Using Function Interpretation.* Proceedings of [AADEBUG97], 1997.

[Els99] MARTIN ELSMAN. *Static Interpretation of Modules.* Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99), Paris, France, ACM, SIGPLAN Notices 34(9), September 1999.

[Eng96] DAWSON R. ENGLER. *VCODE: A Retargetable, Extensible, Very Fast Dynamic Code Generation System*, Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI'96), Philadephia, Pennsylvania, USA, ACM SIGPLAN Notices 31(5), May 1996.

[ErKo96] ULFAR ERLINGSSON, AND ALEXANDER V. KONSTANTINOU. *Implementing the C++ Standard Template Library in Ada 95.* Techincal Report TR96-3, Rensselaer Polytechnic Institute, Troy, January 1996.

[Eve97] MARK EVERED. *Unconstraining Genericity.* Proceedings of Conference on Technology of Object-Oriented Languages and Systems (TOOLS Asia'97), Beijing, China, 1997.

[EvKeMe+97] MARK EVERED, JAMES L. KEEDY, GISELA MENGER, AND AXEL SCHMOLITZKY. *Genja – A New Proposal for Parameterised Types in Java.* Proceedings of Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific'97), Melbourne, Australia, 1997.

[FaFeRo97] CHRISTIAN FABRE, FRANCOIS DE FERRIERE, AND FRED ROY. *Java-ANDF Feasibility Study. Final Report.* Open Software Foundation Reseach Institute, 1997.

[Fra94] MICHAEL FRANZ. *Code-Generation On-the-Fly: A Key to Portable Software.* Doctoral Dissertation, Verlag der Fachvereine, Zürich, 1994.

[FrKi96] MICHAEL FRANZ, AND THOMAS KISTLER. *Slim Binaries.* Department of Information and Computer Science, Technical Report TR 96-24, University of California, Irvine, 1996.

[GaHeJo+95] ERICH GAMMA, RICHARD HELM, RALPH JOHNSON, AND JOHN VLISSIDES. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, 1995.

[Gas99] HOLGER GAST. *Considerations on Genericity for Programming Language Design.* Technical Report WSI 99-5, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, 1999.

[Gas01] HOLGER GAST. *Generic Programming with Views: Type- and Class-Inference with Polymorphic Subsumption by Resolution Theorem Proving.* Technical Report WSI 2001-17, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, 2001.

[GeLe] LAL GEORGE, AND ALLEN LEUNG. *MLRISC – A framework for retargetable and optimizing compiler back ends.* Available at www.cs.nyu.edu/leunga/www/MLRISC/Doc/html/index.html.

[GlMo99] NEAL GLEW, AND GREG MORRISETT. *Type-Safe Linking and Modular Assembly Language.* ACM, Conference Record of [POPL94], pp. 250-261, 1994.

[Gog96] JOSEPH GOGUEN. *Parameterized Programming and Software Architecture.* Fourth International Conference on Software Reuse, IEEE Computer Society, April 1996.

[GoJoSt$^+$00] JAMES GOSLING, BILL JOY, GUY STEELE, AND GILAD BRACHA. *The Java^{TM} Programming Language. Second Edition.* Addison-Wesley Publishing Company, 2000. Available at java.sun.com/docs/books/jls/index.html.

[Gou97] K. JOHN GOUGH. *Multi-language, Multi-target Compiler Development: Evolution of the Gardens Point Compiler Project.* Proceedings of the Joint Modula Languages Conference, Linz, Austria, March 1997. In Lecture Notes in Computer Science, Vol. 1204, Springer, 1997.

[Gou97b] JOHN GOUGH. *The DCode Intermediate Representation: Reference Manual and Report.* Revision 3.2, November 1997. Available at sky.fit.qut.edu.au/~gough.

[GoCo00] K. JOHN GOUGH, AND DIANE CORNEY. *Evaluating the Java Virtual Machine as a target for Languages Other than Java.* Joint Modula Languages Conference, Zürich, Switzerland, September 2000.

[Gou00] K. JOHN GOUGH. *Parameter Passing for the Java Virtual Machine.* 23rd Australian Computer Science Conference, ACSC-2000, Canberra, February 2000.

[Gou01] K. JOHN GOUGH. *Stacking them up: a Comparison of Virtual Machines.* Proceedings of Australian Computer Systems and Architecture Conference, ACSAC-2001, Gold Coast, Australia, February 2001.

[Gra02] TORBJÖRN GRANLUND. *GNU MP – The GNU Multiple Precision Arithmetic Library*, Edition 4.0.1, Free Software Foundation, 2002. Available at swox.com/gmp/.

[GrMoPh$^+$00] BRIAN GRANT, MARKUS MOCK, MATTHAI PHILIPOSE, CRAIG CHAMBERS, AND SUSAN J. EGGERS. *DyC: An Expressive Annotation-Directed Dynamic Compiler for C.* Theoretical Computer Science, Volume 248(1-2), pp. 147-199, October 2000.

[GrHaKi$^+$01] TODD L. GRAVES, MARY JEAN HARROLD, JUNG-MIN KIM, ADAM PORTER, AND GREGG ROTHERMEL. *An Empirical Study of Regression Test Selection Techniques.* ACM Transactions on Software Engineering and Methodology (TOSEM), Vol. 10, Issue 2, April 2001.

[Gri99] ROBERT GRIESEMER. *Generation of Virtual Machine Code at Startup.* Virtual Machine Workshop at [OOPSLA99], 1999.

[GrBaJa$^+$00] DICK GRUNE, HENRI E. BAL, CERIEL J. H. JACOBS, AND KOEN G. LANGENDOEN. *Modern Compiler Design.* John Wiley & Sons, Ltd, 2000.

[GrMi00] ROBERT GRIESEMER, AND SRDJAN MITROVIC. *A Compiler for the Java^{TM} HotSpot Virtual Machine.* pp. 133-152 in [BöGuPo00], 2000.

[Gro97] THOMAS GROSS. *Bisection Debugging.* Proceedings of [AADEBUG97], 1997.

[Gur00] YURI GUREVICH. *Sequential Abstract State Machines Capture Sequential Algorithms.* ACM Transactions on Computational Logic, vol. 1, no. 1, July 2000, 77-111.

[Hal01] THOMAS HALLGREN. *Fun with Functional Dependencies.* Proceedings of the Joint CS/CE Winter Meeting, Varberg, Sweden, January 2001.

[Han99] DAVID R. HANSON. *Early Experience with ASDL in lcc.* John Wiley & Sons, Ltd, Software – Practice and Experience, Vol. 29, No. 5, pp. 417-435, 1999.

[Heu00] HARRO HEUSER. *Lehrbuch der Analysis. Teil 2.* 11. Aufl., B.G. Teubner Verlag, 2000.

[HePa96] JOHN L. HENNESSY, AND DAVID A. PATTERSON. *Computer Architecture. A Quantitative Approach. Second Edition.* Morgan Kauffmann Publishers, Inc., 1996.

[HoNeSc95] HOON HONG, ANDREAS NEUBACHER, AND WOLFGANG SCHREINER. *The Design of the SACLIB/PACLIB Kernels.* Journal of Symbolic Computation 19(1-3): 111-132, Academic Press, London, 1995.

[HTML99] DAVE RAGGETT, ARNAUD LE HORS, AND IAN JACOBS (EDITORS). *HTML 4.01 Specification.* W3C Recommendation, December 1999. Available at www.w3.org/TR.

[Int99] INTEL CORPORATION. *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference.* Order Number 243191, Intel Corporation, 1999.

[ISO14882] JTC1/SC22 – PROGRAMMING LANGUAGES, THEIR ENVIRONMENT AND SYSTEM SOFTWARE INTERFACES. *Programming Languages – C++.* International Organization for Standardization, ISO/IEC 14882, 1998.

[ISO11404] JTC1/SC22/WG11 – BINDING TECHNIQUES. *Language-independent datatypes.* International Organization for Standardization, ISO/IEC 11404:1996, 1996.

[Jah00] ERWAN JAHIER. *Collecting Graphical Abstract Views of Mercury Program Executions.* Proceedings of [AADEBUG00], 2000.

[JäHaGe99] BERND JÄHNE, HORST HAUSSECKER, AND PETER GEISSLER. *Handbook on Computer Vision and Applications.* Volume 3, Academic Press, 1999.

[Jär01] JAAKKO JÄRVI. *Tuple Types and Multiple Return Values.* C/C++ Users Journal, August 2001.

[JaLoMu98] MEHDI JAZAYERI, RÜDIGER G. K. LOOS, AND DAVID R. MUSSER (EDITORS). *Generic Programming.* International Seminar on Generic Programming, Dagstuhl Castle, Germany, April/May 1998, Lecture Notes in Computer Science, Vol. 1766, Springer, 1998.

[JoLi96] RICHARD JONES, AND RAFAEL LINS. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, Ltd, 1996.

[JoHu98] SIMON PEYTON JONES, AND JOHN HUGHES (EDS.). *Haskell 98: A Non-strict, Purely Functional Language.* Language Report, 1998. Available at www.haskell.org.

[JoJoMe97] SIMON PEYTON JONES, MARK JONES, AND ERIK MEIJER. *Type classes: Exploring the Design Space.* Proc. of ACM SIGPLAN Haskell Workshop, June 1997.

[JoRaRe99] SIMON PEYTON JONES, NORMAN RAMSEY, AND FERMIN REIG. *C--: a portable assembly language that supports garbage collection.* Lecture Notes in Computer Science, Vol. 1702, Springer, 1999.

[KaMu92] DEEPAK KAPUR, AND DAVID R. MUSSER. *Tecton: a framework for specifying and verifying generic system components.* Computer Science Technical Report 92-20, Rensselaer Polytechnic Institute, July, 1992.

[KeClRe98] RICHARD KELSEY, WILLIAM CLINGER, AND JONATHAN REES (EDS.). *Revised^5 Report on the Algorithmic Language Scheme.* ACM, SIGPLAN Notices 33(9), 1998.

[KeSy01] ANDREW KENNEDY, AND DON SYME. *Design and Implementation of Generics for the .NET Common Language Runtime.* Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, ACM, SIGPLAN Notices 36(5), May 2001.

[KiHö97] HOLGER KIENLE, AND URS HÖLZLE. *Introduction to the SUIF 2.0 Compiler System.* Technical Report TRCS97-22, University of California, December 1997.

[KiFr99] THOMAS KISTLER, AND MICHAEL FRANZ. *A Tree-Based Alternative to Java Byte-Codes.* International Journal of Parallel Programming, Vol. 27, No. 1, pp. 21-33, 1999.

[Kis99] THOMAS KISTLER. *Continuous Program Optimization.* Ph.D. thesis, University of California, Irvine, 1999.

[Kla83] HERBERT KLAEREN. *Algebraische Spezifikation - Eine Einführung.* Springer, 1983.

[Köt99] ULLRICH KÖTHE. *Reusable Software in Computer Vision.* Chapter 6 in [JäHaGe99], 1999.

[Köt00] ULLRICH KÖTHE. *STL-Style Generic Programming with Images.* SIGS Publications, C++ Report, Vol. 12(1), January 2000.

[Kre02] ADRIAN U. KREPPEL. *Algorithm Selection Based On Empirical Data.* Ph.D. thesis, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, 2002.

[KüWe97] DIETMAR KÜHL, AND KARSTEN WEIHE. *Data Access Templates.* SIGS Publications, C++ Report, Vol. 9(7), July/August 1997.

[Lan64] CORNELIUS LANCZOS. *A Precission Approximation of the Gamma Function.* SIAM Journal of Numerical Analysis, Vol. 1, pp. 86-96, 1964.

[Len00] RAIMONDAS LENCEVICIUS. *On-the-fly Query-Based Debugging with Examples.* Proceedings of [AADEBUG00], 2000.

[Ler94] XAVIER LEROY. *Manifest Types, Modules, and Separate Compilation.* ACM, Conference Record of [POPL94], 1994.

[Ler97] XAVIER LEROY. *The effectiveness of type-based unboxing.* Workshop "Types in Compilation", Amsterdam, June 1997.

[Ler98] XAVIER LEROY. *An overview of Types in Compilation.* Proceedings of Workshop "Types in Compilation", Lecture Notes in Computer Science, Vol. 1743, Springer, 1998.

[LeDeGo95] BRIAN T. LEWIS, L. PETER DEUTSCH, AND THEODORE C. GOLDSTEIN. *Clarity MCode: A Retargetable Intermediate Representation for Compilation.* Technical Report SMLI TR-95-43, Sun Microsystems Laboratories, Inc., 1995.

[Lip96] STANLEY B. LIPPMAN. *Indide the C++ Object Model.* Addison-Wesley Publishing Company, 1996.

[LiLa98] STANLEY B. LIPPMAN, AND JOSÉE LAJOIE. *C++ Primer. Third Edition.* Addison-Wesley Publishing Company, 1998.

[Lis92] BARBARA LISKOV. *A History of CLU.* Technical Report TR-561, Massachusetts Institute of Technology, Cambridge, 1992.

[LiCuDa+95] BARBARA LISKOV, DOROTHY CURTIS, MARK DAY, SANJAY GHEMAWAT, ROBERT GRUBER, PAUL JOHNSON, AND ANDREW C. MYERS. *Theta Reference Manual.* Programming Methodology Group Memo 88, MIT Laboratory for Computer Science, Cambridge, Massachusetts, USA, February 1995.

[Lit98] VASSILIY LITVINOV. *Constraint-Based Polymorphism in Cecil: Towards a Practical and Static Type System.* In [OOPSLA98], 1998.

[LoCo92] RÜDIGER G. K. LOOS, AND GEORGE E. COLLINS. *Revised Report on the Algorithm Description Language ALDES*, Technical Report WSI 92-14, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, 1992.

[LiYe99] TIM LINDHOLM, AND FRANK YELLIN. *The Java^T M Virtual Machine Specification, Second Edition.* Addison-Wesley Publishing Company, 1999.

[Mac93] STAVROS MACRAKIS. *From UNCOL to ANDF: Progress in Standard Intermediate Languages.* Open Software Foundation, Inc., 1993.

[MiToHa+97] ROBIN MILNER, MADS TOFTE, ROBERT HARPER, AND DAVID MACQUEEN. *The Definition of Standard ML - Revised.* MIT Press, May 1997.

[MeGo01] ERIK MEIJER, AND JOHN GOUGH. *Technical Overview of the Common Language Runtime.* Available at research.microsoft.com/~emeijer/, 2001.

[Mey92] BETRAND MEYER. *Eiffel: The Language.* Prentice Hall, New York, 1992.

[Mey01] SCOTT MEYERS. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library.* Addison-Wesley Publishing Company, 2001.

[Mor97] ROBERT MORGAN. *Building an Optimizing Compiler.* Digital Press, Butterworth-Heinemann, 1997.

[MoCrGl+99] GREG MORRISETT, KARL CRARY, NEAL GLEW, DAN GROSSMAN, RICHARD SAMUELS, FREDERICK SMITH, DAVID WALKER, STEPHANIE WEIRICH, AND STEVE ZDANCEWIC. *TALx86: A Realistic Typed Assembly Language.* ACM SIGPLAN, Second Workshop on Compiler Support for System Software, Atlanta, May 1999.

[MoWaCr+98] GREG MORRISETT, DAVID WALKER, KARL CRARY, AND NEAL GLEW. *From System F to Typed Assembly Language.* ACM, Conference Record of [POPL98], 1998.

[Mos89] STEPHEN L. MOSHIER. *Methods and Programs for Mathematical Functions.* Prentice Hall, New York, 1989.

[Mös00] HANSPETER MÖSSENBÖCK. *Compiler Construction: The Art of Niklaus Wirth.* pp. 55-68 in [BöGuPo00], 2000.

[Muc97] STEVEN S. MUCHNICK. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, San Fransisco, California, USA, 1997.

[MuDeSa01] DAVID R. MUSSER, GILLMER J. DERGE, AND ATUL SAINI. *STL Tutorial and Reference Guide, Second Edition.* Addison-Wesley Publishing Company, 2001.

[Mus97] DAVID R. MUSSER. *Introspective Sorting and Selection Algorithms*, John Wiley & Sons, Ltd, Software – Practice and Experience, Vol. 27, No. 8, pp. 983-993, 1997.

[Mus98] DAVID R. MUSSER. *The Tecton Concept Description Language.* Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, 1998.

[MuScLo98] DAVID R. MUSSER, SIBYLLE SCHUPP, AND RÜDIGER LOOS. *Requirement Oriented Programming.* pp. 12-24 in [JaLoMu98], 1998.

[MuScSc$^+$99] DAVID R. MUSSER, SIBYLLE SCHUPP, CHRISTOPH SCHWARZWELLER, AND RÜDIGER LOOS. *The Tecton Concept Library.* Technical Report WSI 99-2, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, 1999.

[MyBaLi97] ANDREW C. MYERS, JOSEPH A. BANK, AND BARBARA LISKOV. *Parametized Types for Java.* ACM, Conference Record of [POPL97], 1997.

[Nel79] PHILIP A. NELSON. *A Comparison of PASCAL Intermediate Languages.* Proceedings of the ACM SIGPLAN SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices 14(8), pp. 208-213, August 1979.

[NiSc01] GOR V. NISHANOV, AND SIBYLLE SCHUPP. *A mostly-copying collector component for class templates.* John Wiley & Sons, Ltd, Software – Practice and Experience, Vol. 31, No. 5, pp. 445-470, May 2001.

[NoAmJe$^+$76] KESAV NORI, URS AMMANN, K. JENSEN, H. NÄGELI, AND CHRISTIAN JACOBI. *The PASCAL (P) Compiler: Implementation Notes.* Technical Report 10, revised edition, ETH Zürich, Switzerland, 1976.

[OdWa97] MARTIN ODERSKY, AND PHILIP WADLER. *Pizza into Java: Translating theory into practice.* ACM, Conference Record of [POPL97], pp. 146-159, 1997.

[OOPSLA93] *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97).* Washington, DC, USA, October 1993.

[OOPSLA95] *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95).* Austin, Texas, USA, October 1995.

[OOPSLA96] *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96).* San Jose, California, USA, October 1996.

[OOPSLA97] *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97).* Atlanta, Georgia, USA, October 1997.

[OOPSLA98] *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'93).* Vancouver, British Columbia, Canada, October 1998.

[OOPSLA99] *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99).* Denver, Colorado, USA, October 1999.

[OOPSLA00] *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2000).* Minneapolis, Minnesota, USA, October 2000.

[PCL] *PCL - The Performance Counter Library.* Version 2.1, February 2002. Available at www.fz-juelich.de/zam/PCL.

[PeSi79] DANIEL L. PERKINS, AND RICHARD L. SITES. *Machine-Independent Pascal Code Optimization.* Proceedings of the ACM SIGPLAN SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices 14(8), pp. 201-207, August 1979.

[PeMe01] NIGEL PERRY, AND ERIK MEIJER. *Implementing Functional Languages on Object-Oriented Virtual Machines.* Technical Report, Microsoft Research, 2001. Available at research.microsoft.com/~emeijer.

[PhChEg02] MATTHAI PHILIPOSE, CRAIG CHAMBERS, AND SUSAN J. EGGERS. *Towards automatic construction of staged compilers.* ACM, Conference Record of [POPL02], pp. 113-125, 2002.

[PoHsEn+99] MASSIMILIANO POLETTO, WILSON C. HSIEH, DAWSON R. ENGLER, AND M. FRANS KAASHOEK. *'C and tcc: A Language and Compiler for Dynamic Code Generation.* ACM, Transactions on Programming Languages and Systems (TOPLAS) 21(2), pp. 324-369, 1999.

[POPL93] *The 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'93).* Charleston, South Carolina, USA, January 1993.

[POPL94] *The 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94).* Portland, Oregon, January 1994.

[POPL97] *The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97).* Paris, France, January 1997.

[POPL98] *The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98).* San Diego, CA, USA, January 1998.

[POPL99] *The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99).* San Antonio, TX, USA, January 1999.

[POPL02] *The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02).* Portland, Washington, USA, January 2002.

[RaTeLe+91] RAJENDRA K. RAJ, EWAN D. TEMPERO, HENRY M. LEVY, ANDREW P. BLACK, NORMAN C. HUTCHINSON, AND ERIC JUL. *Emerald: A General-Purpose Programming Language.* John Wiley & Sons, Ltd, Software – Practice and Experience, Vol. 21, No. 1, pp. 91-118, 1991.

[RaJo00] NORMAN RAMSEY, AND SIMON L. PEYTON JONES. *A single intermediate language that supports multiple implementations of exceptions.* Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, ACM, SIGPLAN Notices, 35(5), May 2000.

[Ric00] JEFFREY RICHTER. *Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework.* MSDN Magazine, Novermber 2000.

[RoSz97] PAUL ROE, AND CLEMENS SZYPERSKI. *Lightweight Parametric Polymorphism for Oberon.* Proceedings of Joint Modular Languages Conference (JMLC'97), Linz, Austria, 1997.

[SaOd90] VATSA SANTHANAM, AND DARYL ODNERT. *Register Allocation Across Procedure and Module Boundaries.* Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, White Plains, New York, USA, ACM, SIGPLAN Notices 25(6), June 1990.

[SAX] *The Official SAX Homepage.* Available at www.saxproject.org.

[Sch96] SIBYLLE SCHUPP. *Generic programming — SuchThat one can build an algebraic library.* Ph.D. thesis, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, 1996.

[Sch89] JERRY SCHWARZ. *Initializing Static Variables in C++ Libraries.* SIGS Publications, C++ Report, Vol. 1(2), February 1989.

[Sch97] CHRISTOPH SCHWARZWELLER. *Mizar Verification of Generic Algebraic Algorithms.* Ph.D. thesis, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, 1997.

[ScLo98] SIBYLLE SCHUPP, AND RÜDIGER LOOS. *SuchThat — Generic Programming Works.* pp. 133-145 in [JaLoMu98], 1998.

[Sed98] ROBERT SEDGEWICK. *Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching.* Addison-Wesley Publishing Company, 3rd Edition, 1998.

[SiLeLu01] JEREMY G. SIEK, LIE-QUAN LEE, AND ANDREW LUMSDAINE. *The Boost Graph Library. User Guide and Reference Manual.* Addison-Wesley Publishing Company, 2001.

[SiLu98a] JEREMY G. SIEK, AND ANDREW LUMSDAINE. *The Matrix Template Library: A Unifying Framework for Numerical Linear Algebra.* Workshop on Parallel Object-Oriented Scientific Computing at [ECOOP98], 1998.

[SiLu98b] JEREMY G. SIEK, AND ANDREW LUMSDAINE. *A Rational Approach to Portable High Performance: The Basic Linear Algebra Instruction Set (BLAIS) and the Fixed Algorithm Size Template (FAST) Library.* Workshop on Parallel Object-Oriented Scientific Computing at [ECOOP98], 1998.

[SiLu98c] JEREMY G. SIEK, AND ANDREW LUMSDAINE. *The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra.* Second International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE 98), Santa Fé, NM, USA, December 1998.

[SiLu99] JEREMY G. SIEK, AND ANDREW LUMSDAINE. *The Matrix Template Library: Generic Components for High-Performance Scientific Computing.* IEEE Computer Society, Computing in Science & Engineering, 1999.

[SiLu00] JEREMY G. SIEK, AND ANDREW LUMSDAINE. *Concept Checking: Binding Parametric Polymorphism in C++.* Workshop on C++ Template Programming, Erfurt, Germany, October 2000.

[SiWe99] VOLKER SIMONIS, AND ROLAND WEISS. *Heterogeneous, Nested STL Containers in C++.* Andrei Ershov Third International Conference: Perspectives of System Informatics, Novosibirsk, Russia, July 1999. Lecture Notes in Computer Science, Vol. 1755, Springer, 1999.

[Sim02] VOLKER SIMONIS. *ProgDoc – A Program Documentation System.* Available at www.progdoc.org.

[ShAp93] ZHONG SHAO, AND ANDREW W. APPEL. *Smartest Recompilation.* ACM, Conference Record of [POPL93], 1993.

[SMLNJ] LUCENT TECHNOLOGIES, BELL LABORATORIES. *Standard ML of New Jersey.* Available at cm.bell-labs.com/cm/cs/what/smlnj/index.html.

[SoAl98] JOSE H. SOLORZANO, AND SUAD ALAGIC. *Parametric Polymorphism for Java: A Reflective Solution.* Proceedings of [OOPSLA98], ACM, SIGPLAN Notices 33(10), 1998.

[Spo94] JOHN L. SPOUGE. *Computation of the Gamma, Digamma, and Trigamma Functions.* SIAM Journal of Numerical Analysis, Vol. 31, No. 3, pp. 931-944, 1994.

[SrWa92] AMITABH SRIVASTAVA, AND DAVID W. WALL. *A Practical System for Intermodule Code Optimization at Link-Time.* Western Research Laboratory (WRL), Digital Equipment Corporation, Research Report 92/6, December 1992.

[Sta01] RICHARD M. STALLMAN. *Using and Porting the GNU Compiler Collection. For GCC Version 3.0.* Free Software Foundation, Inc., Boston, MA, USA, 2001.

[StLe95] ALEXANDER STEPANOV, AND MENG LEE. *The Standard Template Library.* Technical Report HPL-95-11, Hewlett-Packard Laboratories, November 1995.

[Str67] CHRISTOPHER STRACHEY. *Fundamental Concepts in Programming Languages.* Lecture notes for the International Summer School in Computer Programming, Copenhagen, Denmark, 1967. Reprinted in Higher-Order and Symbolic Computation, Volume 13, Number 1-2, April 2000.

[Str97] BJARNE STROUSTRUP. *The C++ Programming Language, 3rd Edition.* Addison-Wesley Publishing Company, 1997.

[Sut01] HERB SUTTER. *Virtuality.* C/C++ Users Journal, 19(9), July 2001.

[Tho97] KRESTEN KRAB THORUP. *Genericity in Java with Virtual Types.* Lecture Notes in Computer Science, Vol. 1241, Springer, 1997.

[ThTo99] KRESTEN KRAB THORUP, AND MADS TORGERSEN. *Unifiying Genericity – Combining the Benefits of Virtual Types and Parameterized Classes.* Lecture Notes in Computer Science, Vol. 1628, Springer, 1999.

[Tol02] ROBERT TOLKSDORF. *Programming Languages for the Java Virtual Machine.* Available at grunge.cs.tu-berlin.de/~tolk/vmlanguages.html.

[Uni00] THE UNICODE CONSORTIUM. *The Unicode Standard, Version 3.0.* Addison-Wesley Publishing Company, 2000.

[ViNa00] MIRKO VIROLI, AND ANTONIO NATALI. *Parametric Polymorphism in Java: an Approach to Translation Based on Reflective Features.* ACM, SIGPLAN Notices 35(10), Proceedings of [OOPSLA00], 2000.

[Vir00] MIRKO VIROLI. *On the recursive generation of parametric types.* Technical Report DEIS-LIA-00-002, Università degli Studi di Bologna, 2000.

[Vir01] MIRKO VIROLI. *Parametric Polymorphism in Java: an Efficient Implementation for Parametric Methods.* Proceedings of the 2001 ACM Symposium on Applied Computing (SAC), Las Vegas, NV, USA, March 2001.

[Vir02] MIRKO VIROLI. *A Lazy Type-Passing Approach for the Translation of Generics in Java.* Unpublished draft, 2002. Available a www.ingce.unibo.it/~mviroli/LM/index.htm.

[W3Tr] WORLD WIDE WEB CONSORTIUM. *W3C Technical Reports and Publications.* Available at www.w3.org/TR.

[Wal86] DAVID W. WALL. *Global Register Allocation at Link Time.* Western Research Laboratory (WRL), Digital Equipment Corporation, Research Report 86/3, 1986.

[Wal89] DAVID W. WALL. *Link-Time Code Modification.* Western Research Laboratory (WRL), Digital Equipment Corporation, Research Report 89/17, September 1989.

[Wal90] DAVID W. WALL. *Predicting Program Behavior Using Real or Estimated Profiles.* Western Research Laboratory (WRL), Digital Equipment Corporation, Technical Note TN-18, December 1990.

[WaPo87] DAVID W. WALL, AND MICHAEL L. POWELL. *The Mahler Experience: Using an Intermediate Language as the Machine Description.* Western Research Laboratory (WRL), Digital Equipment Corporation, Research Report 87/1, 1987.

[WaApKo+97] DANIEL C. WANG, ANDREW W. APPEL, JEFF L. KORN, AND CHRISTOPHER S. SERRA. *The Zephyr Abstract Syntax Description Language.* Conference on Domain-Specific Languages (DSL), Santa Barbara, California, USA, October 1997.

[Wed96] SEBASTIAN WEDENIWSKI. *Piologie – Eine exakte arithmetische Bibliothek in C++ ,* Technical Report WSI 96-35, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, 1996.

[WeSi01] ROLAND WEISS, AND VOLKER SIMONIS. *Exploring Template Template Parameters.* Andrei Ershov Fourth International Conference: Perspectives of System Informatics, Novosibirsk, Russia, July 2001. Lecture Notes in Computer Science, Vol. 2244, Springer, 2001.

[Wei97] ROLAND WEISS. *ScmToCpp: a configureable, intelligent back end for* SUCHTHAT. Technical Report WSI 97-13, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, 1997.

[Win01] EMILY WINCH. *Heterogenous Lists of Named Objects.* Second Workshop on C++ Template Programming, Tampa Bay, Florida, USA, October 2001.

[WoAgUn99] MARIO WOLCZKO, OLE AGESEN, AND DAVID UNGAR. *Towards a Universal Implementation Substrate for Object-Oriented Languages.* Virtual Machine Workshop at [OOPSLA99], 1999.

[XML00] TIM BRAY, JEAN PAOLI, C. M. SPERBERG-MCQUEEN, AND EVE MALER (EDITORS). *Extensible Markup Language (XML) 1.0 (Second Edition).* W3C Recommendation, W3C XML Activity, October 2000. Available at [W3Tr].

[XMLA] WORLD WIDE WEB CONSORTIUM. *Extensible Markup Language (XML) Activity Statement.* Available at www.w3.org/XML/Activity.

[XMLL98] EVE MALER, AND STEVE DEROSE (EDITORS). *XML Linking Language (XLink).* W3C Working Draft, W3C XML Activity, March 1998. Available at [W3Tr].

[XMLN99] TIM BRAY, DAVE HOLLANDER, AND ANDREW LAYMAN. *Namespaces in XML.* World Wide Web Consortium, January 1999. Available at [W3Tr].

[XMLP98] EVE MALER, AND STEVE DEROSE (EDITORS). *XML Pointer Language (XPointer).* W3C Working Draft, W3C XML Activity, March 1998. Available at [W3Tr].

[XMLSD01] PAUL V. BIRON, AND ASHOK MALHOTRA (EDITORS). *XML Schema Part 2: Datatypes.* W3C Recommendation, May 2001. Available at [W3Tr].

[XMLSP01] DAVID C. FALLSIDE (EDITOR). *XML Schema Part 0: Primer.* W3C Recommendation, May 2001. Available at [W3Tr].

[XMLSS01] HENRY S. THOMPSON, DAVID BEECH, MURRAY MALONEY, AND NOAH MENDELSOHN (EDITORS). *XML Schema Part 1: Structures.* W3C Recommendation, May 2001. Available at [W3Tr].

[Zol01] LEOR ZOLMAN. *An STL Error Message Decryptor for Visual C++.* C/C++ Users Journal, July 2001.

Acknowledgements

Foremost, I want to thank Rüdiger Loos for his patient supervision of my work and all his stimulating suggestions and thoughts. Just as much I would like to express my great appreciation for the support of my colleagues Holger Gast, Albrecht Haug, Uwe Kreppel, Christoph Schwarzweller, and Volker Simonis. Christoph provided insights on specification and verification, Holger on type theory, and Uwe on algorithm selection. Furthermore, we had countless discussions on all aspects of generic programming. Volker always extended his ProgDOC system immediately in order to meet my particular demands.