Checking Semantic Requirements of Generic Algorithms

Holger Gast* and Christoph Schwarzweller

Wilhelm-Schickard-Institut für Informatik,
Universität Tübingen, Sand 13,
D-72076 Tübingen, Germany
{gast,schwarzw}@informatik.uni-tuebingen.de

Abstract. Algorithms in computer algebra lend themselves naturally to the software design method of *generic programming* in the sense of the C++ STL library [14]: They abstract over their concrete input domain in favour of the properties that the domain must have; when calling a generic algorithm, the instantiation domain must provide these properties.
We seek to design a language for generic programming; in particular, the compiler should be aware of the proof obligations associated with instantiation and should assist in proving them. Since properties will in general be formulated in a rich logic, we introduce symbolic adjectives as names for underlying formulae. The compiler then works on sets of adjectives as a representation for properties; relations between properties become declared implications between sets of adjectives.
We present a concise prototypical language Saga (Signatures and Adjectives for Generic Algorithms) that embodies this design principle and demonstrate its applicability by example.

1 Introduction

The term generic programming has been applied in such diverse fields of software engineering as purely functional languages [9], object-oriented programming [1, 15], module systems [2] and library design [14, 3]. The common theme is that the language objects can be parameterized to work uniformly on a variety of data types. Unlike conventional ML-style polymorphic functions, generic definitions can exploit type-specific properties, such as the data type constructors, the methods of a class or overloaded operators. In this paper, we explore the STL [14] notion of generic programming, where generic algorithms can abstract over types and associated operations. Instantiating an algorithm with given types is allowed only if all required operations can be provided and if they exhibit the required semantics.

In order to reason about the correctness of generic algorithms, the associated operations must be taken into account, that is, verification must be lifted

* The implementation in this work is an application of the first author's PhD project [7], which has been supported by the *Studienstiftung des deutschen Volkes*.

to deal with generic definitions [16]. This goal requires that the precise meaning of a program can be seen from the program text itself, not only from its translation. For the task to be feasible, the meaning of a program must be determined separately for each algorithm and the correctness of an algorithm instance must be clear from the correctness of the generic algorithm together with an argument showing that the desired instance is legal.

We address these demands by a language design called SAGA (Signatures and Adjectives for Generic Algorithms) in which we integrate a conventional imperative language with a calculus for symbolic deduction [17] of a data type's static properties. SAGA has the following capabilities, which distinguish it from the C++-based developments for error checking [18, section 2.5]:

- Calls to generic algorithms within generic algorithms can be checked once and for-all, without re-compilation at every call-site.
- Calls to generic algorithms can be shown to be legal by reference to their interface alone, without re-compilation of the definition.
- Legality checks include semantic knowledge about the operators involved.

The last point requires some form of language design decision, since including some expressive logic into the language renders compilation an undecidable problem. SAGA approaches this challenge by introducing *adjectives* as symbolic representations for associated formulae. Requirements of generic algorithms are sets of adjectives and legality checking becomes feasible. As the underlying formulae are not considered by the SAGA compiler, relationships between adjectives must be declared by *rules*, which are interpreted as implications between the meanings of adjectives.

In the remainder of this section, we give a short example and sketch the calculus of signatures and adjectives [17]. We describe SAGA in section 2 and give an extended example application to univariate polynomials in section 3. Section 4 points to related work and section 5 concludes.

1.1 Example: Euclidean GCD

The Euclidean GCD algorithm [13] works on rings `T` that provide division and modulus operations together with a suitable norm function to guarantee termination. These requirements can be captured in an adjective `euclidean_domain`:

```
Adjective euclidean_domain
for (T, +    : (T,T)->T,  *    : (T,T)->T,  zero : ()->T,
        div  : (T,T)->T,  mod  : (T,T)->T,  norm : (T)->Int)
means (all(x:T) norm(x) >= 0) /\
      (all(x,y:T) norm(x*y) >= norm(x)*norm(y)) /\
      (all(x,y:T) !(y EQ zero())
          => (x EQ div(x,y)*y + mod(x,y) /\
              (norm(mod(x,y)) == 0 \/ norm(mod(x,y)) < norm(y))));
```

With that adjective at hand, we can specify the interface of algorithm `egcd`; the ellipses '`...`' are part of SAGA and will be discussed in section 2.4.

$$\frac{P_2 \subseteq P_1}{R \vdash^{\text{calc}} P_1 \Longrightarrow P_2} \qquad \frac{R \vdash^{\text{calc}} P_1 \Longrightarrow P_2, \quad R \vdash^{\text{calc}} P_2 \Longrightarrow P_3}{R \vdash^{\text{calc}} P_1 \Longrightarrow P_3}$$

$$\frac{l \longrightarrow r \in R}{R \vdash^{\text{calc}} \sigma(l) \Longrightarrow \sigma(r)} \qquad \frac{R \vdash^{\text{calc}} P_1 \Longrightarrow P_2, \quad R \vdash^{\text{calc}} P_1 \Longrightarrow P_3}{R \vdash^{\text{calc}} P_1 \Longrightarrow P_2 \cup P_3}$$

Fig. 1. The calculus of Adjectives

```
Algorithm egcd
[ (T,...) with ring(T,...), euclidean_domain(T,...),
               equality_comparable(T,...), assignable(T,...) ]
( a_ : T; b_ : T ) return T
```

In order to instantiate `egcd` with the integers, we have to relate the known properties of type `Int` to the meaning of `euclidean_domain`. The following rule expresses this implication (incidentally with empty signature and premises):

```
Rules: for () { } ==> euclidean_domain(Int,+,*,int_0,/,%,abs);
```

With similar declarations for the remaining adjectives, we can call `egcd(42,15)`; the Saga compiler checks the legality of this call and computes the necessary instantiation of the generic algorithm. In general, rules can have both a signature and premises (see section 3).

1.2 The Calculus of Signatures and Adjectives

The calculus [17] describes a Horn-clause theory in which the adjectives play the role of predicates: Signatures specify operators with their corresponding domains and arities; adjectives give symbolic names to properties of sorts and operators (see section 2.2). Rules $P_1 \longrightarrow P_2$, where P_1 and P_2 are sets of adjectives, describe implications of the corresponding sets of properties. Note that rules can be proven correct, based on a formal definition of the meaning of adjectives. A given rule set R is the basis of the calculus presented in figure 1: A rule can be incorporated by applying an appropriate substitution σ for the domains and operators.

The calculus allows for a straightforward implementation by SLD-resolution: To show $P_1 \Longrightarrow P_2$ propagate the rules of R backwards starting with the adjectives contained in P_2 and eliminate adjectives from P_2 present in P_1 until P_2 is empty. Note that P_1 is not changed throughout the whole deduction.

2 The Language Saga

The design of Saga augments a largely standard, C/C++-like language with specific features for generic algorithms. As suggested in the introduction, generic programming is enabled by parameterizing algorithms over signatures and expressing the requirements on operators by means of adjectives. This two-level

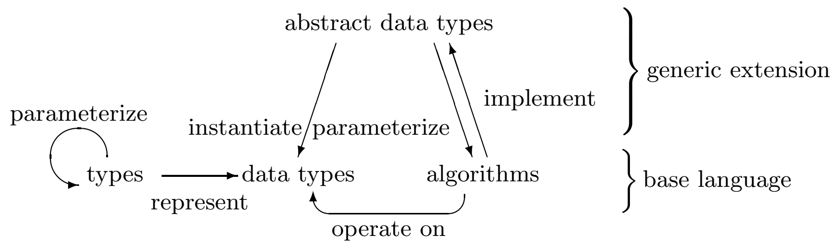

Fig. 2. Relation between Types, Data Types and Algorithms

design is sketched in figure 2: The algorithms work on data of basic types; these types can be parameterized, but they differ from algorithms in that their parameters are types, not signatures. This restriction keeps the design of the base language largely independent of the treatment of genericity, and we conjecture that other languages can be augmented by the SAGA-approach to genericity as well. Furthermore, restrictions on the instances can be given on an algorithm-to-algorithm basis rather than for each data type.

We describe the basic type system in section 2.1 and introduce the generic constructs in the subsequent subsections. Each of these constructs comes with a set of consistency checks, which together imply that all generated algorithm instances obey the basic type discipline. A declaration passing these checks is called *legal*. The link between the different constructs is the computation of instances (section 2.3): An adjective's parameters are transferred, perhaps through the application of rules (section 1.2), into the generic algorithm's signature instance. The inference rules for the type-checks are given informally, the full definitions can be found in [7].

2.1 The Basic Type System

The imperative base language is entirely conventional, with a C/C++-like notation for statements, and declarations ID : *type*. The basic type-language contains built-in types and application of user-defined constructors. Unlike other imperative languages, SAGA provides higher-order functions. To simplify parsing, we have adopted the convention that type-identifiers TID start with an upper-case letter, while all remaining identifiers ID start with a lower-case letter.

$$\begin{array}{rl} \textit{type} & ::= \textsc{tid} \mid \textbf{Int} \mid \textbf{Char} \mid \textbf{String} \mid \textbf{Bool} \mid \textbf{void} \\ & \mid\ \textbf{Array}\ \text{'('}\ \textit{type}\ \text{')'} \mid \textbf{Pair}\ \text{'('}\ \textit{type}\ ,\textit{type}\ \text{')'} \mid \textbf{Cell}\ \text{'('}\ \textit{type}\ \text{')'} \\ & \mid\ \&\textit{type} \mid \text{'('}\ \textit{typelist}\ \text{')'}\ \text{'}\rightarrow\text{'}\ \textit{type} \mid \textit{type}\ \text{'('}\ \textit{typelist}\ \text{')'} \\ \textit{typelist} & ::= \varepsilon \mid \textit{type}\ (\text{','}\ \textit{type}\ )^* \end{array} \tag{1}$$

Cells, arrays and pairs are heap-allocated and can be accessed by reference (type constructor `&`) via the operator `!`, array indexing and selectors `fst` and `snd`.

Our type-checking judgment $\Gamma; R; P \vdash^{\text{type}} e \longrightarrow i$ has standard components Γ, e and i, where Γ is a relation between identifiers and their declared types, e is an expression and i the translation to intermediate language. The contexts R and P account for the generic features of SAGA: They contain the valid rules and adjectives applicable in the basic calculus from section 1.2.

In the prototype implementation, there is no subtyping; a reference is accessed by operator `!`, and a defined type is cast to its representation type by `:*`. Overloading is permitted only in a form similar to that of the original Ada proposal [4, 5]: All possible type-correct interpretations of an expression are enumerated. If there is more than one interpretation, the expression is considered illegal.

We use a standard kinds-calculus [12, section 2.3] to ensure that type expressions are well-formed. The language of kinds is $\mathit{kind} ::= \; * \mid (\mathit{kind}^{+}) \rightarrow \mathit{kind}$, and we use the standard introduction and elimination rules for $\rightarrow$. Currently, abstraction in generic algorithms is over types, that is, all type variables have kind $*$.

2.2 Signatures, Adjectives and Rules

Signatures A signature is a sequence of type names and operators (with arities).

$$\begin{aligned} \mathit{signature} &::= \text{'('}\ \mathit{sig\text{-}elem\text{-}list}\ \text{')'} \\ \mathit{sig\text{-}elem\text{-}list} &::= \varepsilon \mid \mathit{sig\text{-}elem}\ (\text{','}\ (\mathit{sig\text{-}elem}\ ))^{*} \\ \mathit{sig\text{-}elem} &::= \text{TID} \mid \text{ID}\ \text{':'}\ \mathit{type} \end{aligned}$$

The type and operator identifiers in a signature must be unique, that is there is no overloading within one signature. This is consistent with mathematical usage where in every structure the operator names are unique, yet they may be reused in different structures. As a convention, we assume in the remainder of this paper that the type elements of a signature precede its operator elements.

A signature $S = (T_1, \ldots, T_n,\ x_1\!:t_1, \ldots, x_m\!:t_m)$ is *legal* if the types t_i of the operators are well-formed, i.e. they obey grammar (1) and the type names are either defined in the context or among the $T_1, \ldots, T_n$.

The signature S can be instantiated by replacing type names with types and operators with expressions. A *signature instance* S' of S is a sequence $(s_1, \ldots, s_n, e_1, \ldots, e_m)$ where $\{s_i\}_{i=1}^{n}$ are types and for $j = 1\,..\,m$, e_j is an expression of type $t_j[s_i/T_i]_{i=1}^{n}$. The corresponding judgment is $\Gamma; R; P \vdash^{\text{sinst}} S' \leq_S^{\sigma} S$.

A context Γ *enriched with* a signature $(T_1, \ldots, T_n,\ x_1\!:t_1, \ldots, x_m\!:t_m)$ yields a context $\Gamma' := \Gamma \cup \{T_i :: *\}_{i=1}^{n} \cup \{x_j : t_j\}_{j=1}^{m}$.

Adjectives An adjective is introduced by declaring its meaning as a formula, based on a given signature. For informal or preliminary definitions, the formula can be replaced by a text, interspersed with operator and type identifiers from the signature.

$$\begin{aligned} \mathit{adj\text{-}def} &::= \textbf{Adjective}\ \text{ID}\ \mathit{signature}\ \mathit{meaning} \\ \mathit{meaning} &::= \textbf{means}\ \mathit{formula} \mid \textbf{informal}\ (\text{STRINGLIT} \mid \text{TID} \mid \text{ID}\ )^{+} \end{aligned}$$

The order of elements in the *signature* is relevant, and we refer to them also as the adjective's *parameters*. An adjective definition with a signature $S = (T_1, \dots, T_n,\, x_1\colon t_1, \dots, x_m\colon t_m)$ is *legal* iff S is legal and the formula is legal in a context enriched with S.

An *adjective application* refers to a defined adjective by name, for instance to include its defined meaning as a requirement of a generic algorithm.

$$\textit{adj-apply} ::= \textsc{id}\ \text{'('}\ (\textit{type}\ \mid \textit{expr}\ )^{*}\text{')'}$$

An adjective application $a(s_1, \dots, s_{n'},\, e_1, \dots, e_{m'})$ is legal if a has been introduced with signature S and $(s_1, \dots, s_{n'},\, e_1, \dots, e_{m'})$ is a signature instance of S.

Regarding run-time type safety we observe that the type of an operator found as an adjective parameter is known to be an instance of the type included in the adjective's definition.

Rules Rules declare implications about sets of adjectives. For singleton sets, the curly braces may be dropped.

$$\begin{aligned} \textit{rule-decl} &::= \textbf{Rules}\ \textit{signature rule}\ (\text{','}\ \textit{rule}\ )^{*} \\ \textit{rule} &::= \text{'\{'}\ (\textit{adj-apply}\ )^{*}\text{'\}'}\ \Longrightarrow \text{'\{'}\ (\textit{adj-apply}\ )^{+}\text{'\}'} \end{aligned}$$

A rule declaration $\textbf{Rules}\ S\ \left(\{A_{ij}\}_{j=1}^{m_i} \Longrightarrow \{B_{ik}\}_{k=1}^{r_i}\right)_{i=1}^{n}$ is legal if the signature S is legal, and the adjective applications A_{ij} are legal in a context enriched by S. The adjective applications B_{ik} must be legal in a context enriched by S and premises $P = \{A_{ij}\}_{i=1}^{m_i}$. This treatment of the B_{ik} enables the use of (instances of) generic algorithms as operations on the right-hand side. The construction parallels the usual introduction of implication in natural deduction by discharging an assumption.

Again, the types of all operator parameters to adjectives are known to be instances of the declared types, such that instantiation information is propagated consistently through applications of substitution σ in figure 1.

2.3 Generic Algorithms

Defining Algorithms Algorithm definitions in SAGA feature a list of signatures and adjectives to account for their generic behaviour. Each signature can have a set of adjectives restricting the possible signature instances.

$$\begin{aligned} \textit{sig-params} &::= \text{'['}\ (\textit{sig} - \textit{param})^{*}\text{']'} \\ \textit{sig-param} &::= \textit{signature}\ \textbf{with}\ \textit{adj-apply}\ (\text{','}\ \textit{adj-apply}\ )^{*} \end{aligned}$$

Writing input type specifications as $(x_k : t_k)_{k=1}^{r}$, we arrive at the following shape of an algorithm definition. A definition

$$\begin{aligned} &\textbf{Algorithm}\ a\ \left[\left(S_i\ \textbf{with}\ (A_{ij})_{j=1}^{m_i}\right)_{i=1}^{n}\right] \\ &(x_k : t_k)_{k=1}^{r}\ \textbf{return}\ s \\ &\textbf{begin}\ B\ \textbf{end} \end{aligned}$$

is legal if the signatures S_i are legal and for each i, the adjective applications A_{ij} are legal in a context enriched with S_i; all the t_k and s must have kind $*$. Furthermore, B must type-check after enriching the context with all the S_i, and the assertions A_{ij}.

The input syntax provides for several signatures to enable overloaded names of imported operators. For a concise formulation of a generic algorithm's type, we assume w.l.o.g. that the operator names in the signatures $\{S_i\}_{i=1}^n$ are pairwise distinct: A legal algorithm can be rewritten (by name-mangling and overload resolution) so that it has a single signature $S = (S_1, \ldots, S_n)$ and a single set of adjectives $\{A_j\}_{j=1}^k$ where $k = \sum_{i=1}^n m_i$. We can then express the algorithm's interface by a qualified type [12, 20] where abstraction is over the signature and qualification captures the adjectives.

$$a : \forall S. \{A_j\}_{j=1}^k \Rightarrow (t_1, \ldots, t_r) \to s \tag{2}$$

Note that the qualification is about *static* properties of signature elements and never about dynamic properties of value parameters.

Calling Generic Algorithms The standard rule for function application is:

$$\frac{\begin{array}{l}\mathit{\Gamma}; R; P \vdash^{\text{type}} f : (t_1, \ldots, t_n) \to s \\ \text{for } i = 1 \,..\, n\text{: } \mathit{\Gamma}; R; P \vdash^{\text{type}} e_i : t_i\end{array}}{\mathit{\Gamma}; R; P \vdash^{\text{type}} f(e_1, \ldots, e_n) : s} \ (\to\text{-elim})$$

If f is a generic algorithm it has a qualified type (2). To use such an algorithm, its signature must be instantiated such that all predicates are satisfied; in other words, the qualified type must be eliminated. Towards that end, the intermediate language is enriched by a construct denoting instantiation.

$$\frac{\begin{array}{l}\mathit{\Gamma}; R; P \vdash^{\text{type}} f : \forall S. \{A_j\}_{j=1}^k \Rightarrow t \\ \mathit{\Gamma}; R; P \vdash^{\text{sinst}} S' \leq_S^\sigma S \\ R \vdash^{\text{calc}} P \Longrightarrow \{A_j\sigma\}_{j=1}^k\end{array}}{\mathit{\Gamma}; R; P \vdash^{\text{type}} f : t\sigma \longrightarrow f\langle S' \rangle} \ (\forall\text{-elim})$$

The rules ($\to$-elim) and ($\forall$-elim) can be implemented by unifying actual argument types with the formal parameter types and keeping the adjectives in a list of preconditions to be proven. As soon as all type parameters of an adjective are instantiated, it can be deduced from the context $R; P$. This step maps the adjective's name with the known parameters to the yet unknown parameters. All instantiations occurring at any point in this process are captured in the instance description $f\langle S' \rangle$.

In case the adjective names are structure names, for example if `ring(T,...)` denotes the ring properties of `T`, this procedure determines the operations of type `T`, perceived as a ring [6].

2.4 Default Extensions to Signatures

Writing down all signatures and adjective instances becomes very cumbersome, since for every adjective application, one ends up copying the adjective's signature to prove the application legal. Therefore SAGA allows an ellipsis '`...`' in both signatures and adjective applications. The missing operators are automatically inserted from the referenced adjectives' definitions, such that the legality checks are passed. Note that this operation is entirely syntactic and adjectives appear in internal form just as if the operators had been supplied by the programmer. For instance, we have an adjective `assignable`, which captures our convention about the type of the assignment operator:[1]

```
Adjective assignable
for (T, = : (&T,T)->&T )
informal "Operator" = " is an assignment on " T;
```

If an adjective application `assignable(T,...)` occurs in the context of a signature $S =$ `(T,...)`, the legality check requires the adjective's ellipses to be filled with a single operator of type `(&T,T)->&T`; it receives the default name `=` and is propagated to the ellipsis in S.

3 Univariate Polynomials

We shall now study a possible implementation of univariate polynomials over a coefficient domain `T`. For a full integration of polynomials into a library, three steps must be taken: To start, we define the basic arithmetic algorithms; we proceed by expressing the resulting algebraic properties in rules. With these provisions, we are able to use the defined polynomials as input to other generic algorithms.

Representation Type Polynomials are represented sparsely as sequences of monomials, ordered by ascending exponent. Since the current run-time system of SAGA does not provide linked lists, we choose an array for the sequences, that is, we have `Type Poly(T:*) = Array(<Int,T>)`.

Addition Out of the arithmetic operations, we treat addition in more detail. It is performed by adding coefficients with equal exponents; with the ordering by exponents, the algorithm's main component is a merge-loop. Inside the loop the sum of the coefficients is computed and if it does not equal 0, it is written into the result. These expressions access the operations and constants `+`, `zero`, and`!=` associated with the coefficient domain through the algorithm's signature. The signature is filled during the legality checks as described section 2.4.

[1] We treat assignment operator `=` by an adjective rather than generating it for every type. This decision derives from the application [7] of SAGA to STL concepts [3].

```
Algorithm poly_add
[(T,...) with ring(T,...),
              assignable(T,...), equality_comparable(T,...) ]
(x:Poly(T); y:Poly(T))
return Poly(T)
{ ... a=x:*; b=y:*; // convert to representation type
  t = snd(!(!a)[!i]) + snd(!(!a)[!j]);
  if (!t != zero()) { (!tmp)[!k] = mkpair(!e1,!t); ++k }
  ...
};
```

Algebraic Properties The remaining arithmetic operations can be defined for type `Poly` in a similar manner (note that division will require a field as the coefficient domain). We can then state the polynomials' algebraic properties by means of rules, for instance their ring structure. A subtlety arises because rules serve the double purpose of checking semantic implications and transferring instantiation information between adjectives. In order to *constructively* prove the ring properties, we have to reference `poly_add` as an operation, which again requires the technical adjectives `assignable` and `equality_comparable`:

```
Rules: for (T,...)
 { ring(T,...), equality_comparable(T,...), assignable(T,...) }
==> ring(Poly(T),poly_add,poly_sub,poly_mul,poly_zero,poly_one);
```

Whenever this rule is used in a deduction, requirements arise to give a constructive proof for the three premises; the computed instances are directly propagated to the conclusion's algorithmic operations, hence providing their instantiation.

Instantiating GCD We can use the Euclidean GCD algorithm from section 2 to compute the GCD of two univariate polynomials. In writing down a rule about `Poly` as a `euclidean_domain`, we must reference algorithm `poly_div`, which requires `T` to be a `field`.

```
Rules: for (T,...)
 { field(T,...), equality_comparable(T,...), assignable(T,...) }
==> euclidean_domain(Poly(T),poly_add,poly_mul,poly_zero,
                     poly_div,poly_mod,poly_degree);
```

Indeed, the above rule is only legal if the algorithm `poly_add` in the consequence can be instantiated legally. That algorithm requires a `ring`, while the rule's premise provides a `field`. Hence, the following rule is necessary:

```
Rules: for (T,...) field(T,...) ==> ring(T,...);
```

After these declarations, the polynomials over the rational numbers `Rat` provide a correct instantiation of `egcd`, given appropriate statements about the field-structure of `Rat` are also provided:

```
Algorithm rat_univariate_gcd (x:Poly(Rat); y:Poly(Rat))
return Poly(Rat)
begin return egcd(x,y) end
```

Saga checks that the call to `egcd` is legal and computes the necessary instances of all referenced generic algorithms. In contrast, `Poly(Int)` is no legal instantiation for `egcd`: The integers are not a field. Hence, if `x`, `y` are of type `Poly(Int)`, the call `egcd(x,y)` fails with the following error message, which points to the problem directly:

```
entailed <= adj[field](Int,x225,x226,x227,x228,x229,x230)
```

4 Related Work

Axiom [10] uses categories and inheritance to build up an algebraic hierarchy. Properties of the domains are given in the categories' documentation, but they are not part of the language itself. A particular domain or an implementation of a domain belongs to a category by assertion. Saga, in contrast, directly connects individual properties of operators with the algorithms themselves. Using rules describing implications of sets of properties then enables identification of implementations for legal instantiation without building up or extending a hierarchy.

Mizar [**?**] is a proof checker based on natural deduction. The Mizar language is designed as a formal counterpart of mathematical vernacular. It provides attributes similar to our adjectives and basic (algebraic) structures can be extended by attributes. Rings, for example, or even a single operator, can be further qualified as "commutative". With attributes, theorems can be stated and proven in a general setting: Only properties required for the proof appear in the theorem's statement [**?**]. This supports building up a library of mathematical knowledge because reuse of theorems is increased.

C++ treats templates by re-compiling the definitions [19]. The main tool to reason about correctness are the STL *concepts* [3], which group together required operators and their semantics. Siek has proposed *concept checks* and *concept archetypes* [18, section 2.5] as a limited form of compile-time checking; these checks are restricted to the existence of operators and do not carry information about their behaviour. We show by an example implementation [7] of `quicksort` [3, chapter 12], that Saga's adjectives can serve as a representation of concepts.

Haskell [11] provides a mechanism for bundling operators into type classes to give a clear semantics to overloading [21]. A type can be declared an instance of a class by providing implementations for the class's operators. An operator can be a member of at most one class, and the compiler ultimately maps operator names to their implementation via a lookup in the global instance table [8]. In other words, it is the operator's name which determines its required behaviour, not the algorithm in which the operator appears. This mismatch with Saga's concept of generic programming can be remedied technically by renaming operators, for instance replacing `+` with the more precise `ring_add` or `monoid_add`. Yet this renaming renders Haskell's subclass mechanism unusable and deprives the language of its counterpart of adjective implication.

5 Conclusion

We have presented a language design focussing on the precise specification of generic algorithms. It admits parameters for both types and their associated operations, which together form the algorithm's signature. Algorithm instantiation can be restricted by semantic conditions on parameters, expressed symbolically as adjectives with a defined meaning. The compiler then checks that the restrictions are obeyed in calls. At the same time, we can reason about the correctness of algorithms without referring to their instantiations in the compiled program: All proof-obligations about static properties are apparent from the program text and can be generated by citing the adjectives' definitions.

Saga is a prototype implementation of this design. We have demonstrated its applicability by an example, a sparse implementation of the univariate polynomials. More example material is covered in [7], where we represent STL concepts [3] by Saga's adjectives; concept checking can be conducted by Saga, including the semantic conditions missing in previous approaches [18, section 2.5].

The examples considered so far admit the following observation regarding the practical use of Saga: Due to the separate declaration of adjectives and their relationships, algorithm descriptions and collections of adjectives can be evolved in parallel. Whenever an adjective proves too coarse for the precise specification of an algorithm, it is split and the relation of the parts to the original is declared as a rule. By the same token, rules also enable cross-references between generic libraries with different terminology or background.

References

1. Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding Type Parameterization to the Java Language. In *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '97)*, volume 32 of *SIGPLAN Notices*, pages 49–65, Atlanta, Georgia, October 1997. ACM.
2. ANSI, editor. *The Programming Language Ada—Reference Manual.* Number 155 in Lecture Notes in Computer Science. Springer-Verlag, 1983.
3. Matthew H. Austern. *Generic Programming and the STL–using and extending the C^{++} Standard Template Library.* Addison-Wesley, 1st edition, 1998.
4. G.V. Cormack. An Algorithm for the Selection of Overloaded Functions in Ada. *ACM SIGPLAN Notices*, 16(2):48–51, 1982.
5. Harald Ganzinger and Knut Ripken. Operator Identification in ADA:Formal Specification,Complexity, and Concrete Implementation. *ACM SIGPLAN Notices*, (15):30–42, 1980.
6. 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,Universität Tübingen, November 2001.
7. Holger Gast. *Generating Type-Checkers in a Proof Theoretic Formulation.* PhD thesis, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, 2003. (forthcoming).

8. Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type Classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.
9. P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.
10. Richard D. Jenks and Robert S. Sutor. *AXIOM : the sientific computation system*. Springer-Verlag, New York u.a., 1992.
11. Hughes (ed.) Jones, Simon Peyton. Report on the Programming Language Haskell 98— A Non-strict,Purely Functional Language. `http://www.haskell.org/definition/haskell98-report.ps.gz`, February 1999.
12. Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–35, January 1995.
13. Donald E. Knuth. *The Art of Computer Programming*, volume 2 – Seminumerical Algorithms. Addison-Wesley, 3rd edition, 1997.
14. David R. Musser and Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 1996.
15. Martin Odersky, Enno Runne, and Philip Wadler. Two Ways to Bake Your Pizza – Translating Parameterised Types into Java. In M. Jazayeri, R. Loos, and D. Musser, editors, *Generic Programming '98*, volume 1766 of *Lecture Notes in Computer Science*, pages 114–132. Springer-Verlag, 2000.
16. Christoph Schwarzweller. *MIZAR verification of generic algebraic algorithms*. PhD thesis, Universität Tübingen, 1997.
17. Christoph Schwarzweller. Symbolic deduction in mathematical databases based on properties. In V. Sorge S. Colton, editor, *Proceedings of the Second International Workshop on the Role of Automated Deduction in Mathematics (RADM2002)*, Kopenhagen, Denmark, July 2002.
18. Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boots Graph Library*. C++ In-Depth Series. Addison-Wesley, Boston, 2003.
19. Bjarne Stroustrup. *The C^{++} programming language*. Addison-Wesley, Reading, Mass. u.a., 3rd edition, 1997.
20. Martin Sulzmann. *A General Framework for Hindley/Milner Type Systems with Constraints*. PhD thesis, Yale University, Department of Computer Science, May 2000.
21. Philip Wadler and Stephen Blott. How to Make ad-hoc Polymorphism Less ad-hoc. In J. Hughes, editor, *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 60–76, Austin, Texas, January 1989.