

GLoo: A Framework for Modeling and Reasoning About Component-Oriented Language Abstractions

Markus Lumpe

Department of Computer Science
Iowa State University
Ames, IA 50011, USA
lumpe@cs.iastate.edu

Abstract. The most important contribution to the success or failure of a software project comes from the choice of the programming languages being used and their support in the target environment. The choice of a suitable implementation language is not a guarantor for success, but an unsuitable language may result in a long, error-prone, and costly implementation, often resulting in an unstable product. In this paper, we present GLoo, a framework for modeling and reasoning about *open-ended* language mechanisms for object- and component-oriented software development. At the heart of GLoo is a small *dynamic composition language* that provides abstractions to (i) define and/or import reusable software components, (ii) introduce new compositional language abstractions, and (iii) build executable and reusable component-oriented specifications. To demonstrate its flexibility and extensibility, we then present an encoding of the *traits* concept as an example of how to add support for a new and readily available language abstraction to the GLoo framework.

1 Introduction

Successful software systems have to abide by the *Laws of Software Evolution* [12], which require that software systems *must be continually adapted*, or else they become progressively less useful in a real-world environment. For this reason, software systems must be *extensible*, so that new behavior can be added without breaking the existing functionality, and *composable*, so that features can be recombined to reflect changing demands on their architecture and design.

By placing emphasis on reuse and evolution, component-oriented software technology has become the major approach to facilitate the development of modern, large-scale software systems [18,22,26]. However, component-oriented software development is in itself an inherently *dynamic* process in which we need to be able to deal with different component models, incorporate new composition techniques, and extend the means for specifying applications as compositions of reusable software components with new abstractions on demand [4]. Unfortunately, general-purpose programming languages are not suitable for this task, since they are not tailored to software composition [19]. As a consequence, when using a general-purpose programming language to specify applications as compositions of reusable software components one often has to use *awkward*

formulations due to unsuitable language constructs, and *lengthy formulations* due to an unsuitable level of abstraction at which compositional abstractions can be expressed.

Consider, for example, the composition of some orthogonal behavior originating from different sources, say different base classes. We can use *multiple inheritance* for this purpose. However, while multiple inheritance appears to provide an appropriate mechanism to express the desired functionality, “there is no good way to do it” [30]. Consequently, multiple inheritance has been mostly abandoned in modern language approaches in favor of *single inheritance*, which provides a more controllable way to build classes. Unfortunately, the lack of multiple inheritance often results in unsuitably structured class hierarchies when specifying the simultaneous support for totally orthogonal behavior. In addition, such class hierarchies may be hard to maintain due to frequent occurrences of code duplications in different places.

The component-based software development approach has emerged from the object-oriented approach, which has already shown a positive influence on software evolution and reuse. These aspects do, however, not depend on object-oriented techniques [22]. Moreover, the specification of applications as compositions of reusable components requires a language paradigm different from the one being used to define the components themselves. While object-oriented programming languages are well suited for implementing components, a specially-designed *composition language* is better for building applications as compositions of reusable software components [24].

We advocate a paradigm that combines the concepts of *dynamic binding*, *explicit namespaces*, and *incremental refinements*. Dynamic binding is a key element in a software development approach that, without affecting its previous behavior, allows for new functionality to be added to an existing piece of code [5]. On the other hand, explicit namespaces [3, 14] in concert with incremental refinements provide a suitable means to directly specify the sets of both *provided* and *required* services of components [14]. From a technical point of view, explicit namespaces serve as a lookup environment with *syntactic representations* to resolve occurrences of free variables in programs. However, the values bound in these namespaces may also contain occurrences of free variables. To resolve those, we can use incremental refinements that allow for a phased recombination of mappings in a namespace to new, updated values. The notion of incremental refinement is based on $\lambda\mathcal{F}$ -contexts [14]. A $\lambda\mathcal{F}$ -context is a term that is evaluated with respect to a local lookup environment. For example, the $\lambda\mathcal{F}$ -context $a[b]$ denotes a term a , whose meaning depends on the values defined in b , if a contains free variables. Thus, b denotes the requirements posed by the free variables of a on its environment [17].

In this work, we present a framework for modeling and reasoning about *open-ended* language mechanisms for object- and component-oriented software development. At the center of this framework is the small *dynamic composition language* GLoo designed in the spirit of PICCOLA [2, 13], which has already demonstrated the feasibility of a high-level composition language that provides component-based, compositional interfaces to services defined in a separate host language. However, PICCOLA is far from providing the ease and flexibility required to build reliable component-based applications due to a conceptual gap between the mechanisms offered by PICCOLA and the component-based methodology that it is supposed to support.

GLoo is essentially a *pure functional* language and therefore fosters a declarative style of programming. The core elements of GLoo are *first-class namespaces*, *methods*, and *variables*, but no predefined statements like conditionals, loops, and assignment. GLoo also provides built-in support for most operators found in Java or C#, but their semantics is *partially open*. That is, with the exception of the core integer operations addition, subtraction, multiplication, and division, for which GLoo provides a built-in implementation, all operators remain undefined and their actual semantics has to be provided by the application programmer. GLoo only specifies priority and associativity for operators, which cannot be changed.

One of the key innovations of GLoo with respect to PICCOLA is a built-in *gateway mechanism* to directly embed Java code into a GLoo specification. This mechanism is based to the $\lambda\mathcal{F}$ -concept of *abstract application* [14] and allows for code abstractions defined in both GLoo and Java to coexist in one specification unit.

The rest of this paper is organized as follows: in Section 2, we briefly describe the main features and design rationale of GLoo and discuss briefly related work in Section 3. We present the design and implementation of our encoding of the traits concept in GLoo in Section 4. We conclude this paper in Section 5 with a summary of the main observations and outline future directions in this area.

2 The GLoo Language

2.1 Design Rationale

A successful component-based software development approach not only needs to provide abstractions to represent different component models and composition techniques, but it has to provide also a systematic method for constructing large software systems [4]. Unfortunately, rather than high-level *plugging*, most existing component frameworks offer, in general, only low-level *wiring* techniques to combine components. We need, however, higher-level, scalable, and domain-specific *compositional mechanisms* that reflect the characteristics and constraints of the components being composed [2,24]. The ability to define these mechanisms will provide us with more effective means to do both to reason about the properties of composition and to enhance program comprehension by reducing the exposure of the underlying wiring mechanisms to the component engineer.

The design of GLoo targets a *problem-oriented* software development approach that provides a paradigm for both *programming in-the-small* and *programming in-the-large* [6]. More precisely, GLoo aims at a higher-level and *scalable* programming approach to encapsulate domain expertise that provides support for the definition of domain-specific abstractions enabling the instantiation, coordination, extension, and composition of components. These domain-specific abstractions can be defined in GLoo, Java, or both.

2.2 The Core Language

The core of GLoo is the $\lambda\mathcal{F}$ -calculus that combines the concepts of dynamic binding, explicit namespaces, and incremental refinement in one unifying framework. More

$F, G, H ::= ()$	<i>empty form</i>	$V ::= \mathcal{E}$	<i>empty value</i>
(X)	<i>form variable</i>	a	<i>abstract value</i>
$(F, l = V)$	<i>binding extension</i>	M	$\lambda\mathcal{F}$ – <i>value</i>
$(F \# G)$	<i>form extension</i>		
$(F \setminus G)$	<i>form restriction</i>	$M, N ::= F$	<i>form</i>
$(F \rightarrow l)$	<i>form dereference</i>	$M.l$	<i>projection</i>
$(F [G])$	<i>form context</i>	$(\setminus X :: M)$	<i>abstraction</i>
		$M N$	<i>application</i>
		$M[F]$	$\lambda\mathcal{F}$ – <i>context</i>

Fig. 1. GLoo-style syntax of the $\lambda\mathcal{F}$ -Calculus.

precisely, the $\lambda\mathcal{F}$ -calculus is a substitution-free variant of the λ -calculus in which variables are replaced by *forms* [15] and parameter passing is modeled by means of *explicit contexts* [1, 3]. Forms are first-class namespaces that provide a high-level and language-neutral abstraction to represent components, component interfaces, and their composition. Explicit contexts, on the other hand, serve as a syntactic representation that mimic λ -calculus substitutions, that is, they provide the means for a fine-grained and *keyword-based* parameter passing mechanism.

The design of the $\lambda\mathcal{F}$ -calculus, like Dami’s λN -calculus [5], tackles a problem that arises from the need to rely on the position and arity of parameters in mainstream programming languages. Requiring parameters to occur in a specific order, to have a specific arity, or both, imposes a specification format in which we are required to define programming abstractions that are characterized not by the parameters they effectively use, but by the parameters they *declare* [5]. However, in a framework especially designed for software composition this can hamper our ability to adapt existing software components to new requirements, because any form a parameter mismatch has to be resolved explicitly and, in general, manually.

The syntax of the $\lambda\mathcal{F}$ -calculus is given in Figure 1. The $\lambda\mathcal{F}$ -calculus is composed from the syntactic categories *forms*, *values*, and *terms*. Every form is derived from the empty form $(|)$. A form can be extended by adding a new mapping from a label to a value using *binding extension*, or by means of *form extension* that allows for a form to be extended with a set of mappings. The difference between these two extension mechanisms lies in the way the values \mathcal{E} and $(|)$ are handled. If we extend a form F with a binding $l = \mathcal{E}$ using binding extension, then the resulting form F' is equivalent to a form F'' that does not contain a binding for label l . In other words, binding extension can be used to *hide* existing mappings in a form. Form extension, on the other hand, is blind for bindings involving the values \mathcal{E} and $(|)$. That is, if the extending form contains bindings that map to those values, then these bindings do not contribute to the extension operation. For example, $(| (| l = a, m = b |) \# (| l = d, m = \mathcal{E}, n = c |) |)$ yields $(| l = d, m = b, n = c |)$. Form restriction can be considered the inverse to form extension, which can be used to remove bindings from a form. In combination, both form extension and form restriction play a crucial role in a fundamental concept for defining adaptable and extensible software abstractions [15]. Finally, form dereference

allows for a form-based interpretation of a value, whereas a form context $(| F [G] |)$ denotes a form F , whose meaning is refined by the local lookup environment G that may contain bindings for occurrences of free variables in F .

Forms and projections take the role of variables in terms. In a term, a form serves as an explicit namespace that allows for a computational model with *late binding* [14]. Abstraction and application correspond to their counterparts in the λ -calculus, whereas a $\lambda\mathcal{F}$ -context is the counterpart to a form context. In contrast to λ -calculus, however, the evaluation of an abstraction allows for an incremental refinement of its body. More precisely, the evaluation of an abstraction yields a *closure* that associates the current evaluation environment with the body of that abstraction. For example, if we evaluate the abstraction $(\backslash X :: M)$ using the form F as an evaluation environment, then the result is a closure $(\backslash X :: M[F])$ in which F serves as a local lookup environment for free occurrences of variables in M . The resulting closure can be subjected to further evaluations that will allow for additional lookup environments to be added. However, to retain a *static scoping* mechanism for occurrences of free variables in the body of an abstraction, the order in which these additional lookup environments are added is significant. For example, if we reevaluate the closure $(\backslash X :: M[F])$ in a new evaluation environment G , then we obtain a closure $(\backslash X :: (M[F])[G])$ in which G serves as an incremental refinement of $M[F]$. Bindings defined in F have precedence over the ones defined in G , but bindings in G may provide values to resolve free occurrences of variables in F and therefore allow for a local refinement of the meaning of M . Parameter passing works in a similar way. If we apply a value H as argument to the closure $(\backslash X :: (M[F])[G])$, then we have to evaluate $(M[F])[G]$ in an evaluation environment $(| X = H |)$, that is, we have to evaluate the term $((M[F])[G])(| X = H |)$. In other words, parameters are passed to functions using a keyword-based mechanism. For a complete definition of the evaluation rules, the interested reader is referred to [14].

2.3 GLoo Specification Units

From a technical point of view, component-oriented software development is best supported by an approach that favors a clear separation between computational and compositional entities [24]. This requirement is captured by the maxim

“Applications = Components + Scripts.” [24]

The definition of the GLoo language follows this maxim. A GLoo specification unit defines a value or *component* that can be recombined with values and/or components, which are defined in other specification units. In other words, a GLoo specification unit defines a single $\lambda\mathcal{F}$ -context that can locally define new abstractions or import definitions from other $\lambda\mathcal{F}$ -contexts in order to construct a new value or component.

GLoo specification units add support for basic data types, an import facility, term sequences, a *delayed evaluation* of terms, *computable* binders, and a Java gateway mechanism to the core language. These amendments solely serve to enrich the versatility of values, but do not change the underlying computational model of the $\lambda\mathcal{F}$ -calculus.

As a first example, consider the specification given in Listing 1. This script defines `IntRdWrClass`, a class that is composed from the class `IntClass` and the

```

1  let
2    Services = load "System/Services.lf"
3    load "Extensions/LanguageOfTraits.lf"
4
5    IntMetaClass = load "Classes/IntClass.lf"
6    TReadInt = load "Traits/TReadInt.lf"
7    TWriteInt = load "Traits/TWriteInt.lf"
8  in
9    IntMetaClass (trait "TRdWrInt" join TWriteInt with TReadInt)
10 end

```

Listing 1. The GLoO script `IntRdWrClass.lf`.

traits `TWriteInt` and `TReadInt`. The concepts of classes and traits [23] are not native to GLoO. GLoO is not an object-oriented programming language *per se*. However, by importing the units `LanguageOfTraits.lf` and `IntClass.lf` into the scope defined by `IntRdWrClass.lf`, this unit now provides support for the required object-oriented language features.

Every GLoO script defines a top-level *let-block* that contains a possibly empty set of *declarations* and a *single value*. The declarations are evaluated sequentially. Thus, the second declaration is evaluated in an environment in which the first binding is visible, and so on. There are five declarations in the script `IntRdWrClass.lf`. The integration of the core system services is defined in line 2. The unit `Services.lf` defines the basic programming abstractions for printing as well as IO, and provides also, for example, a standard implementation for conditionals. The declaration in line 3 extends the current scope with a *traits domain sublanguage* that provides support for defining, composing, and manipulating traits. The abstractions defined in the unit `LanguageOfTraits.lf` serve as *syntactic and semantic extensions* (i.e., keywords) to the GLoO language. Using this technique, we can focus on *what* an application offers (i.e., a programming approach supporting traits), without entering into the details of how it is implemented. The declarations in lines 5-7 introduce the components that we want to combine to the current scope. The reader should note that we do not acquire any explicit support for an *object model*. The object model is encapsulated in `IntClass.lf`. The details of the underlying object model of class `IntClass` do not pollute the declaration space of `IntRdWrClass.lf`. We know, however, that `IntMetaClass` is a function that may take a trait as argument.

The result of evaluating the unit `IntRdWrClass.lf` is a class `IntRdWrClass` that is composed from the class `IntClass` and the result of the composition of the traits `TWriteInt` and `TReadInt`, that is, the composite trait `TRdWrInt`. The underlying object-oriented programming abstractions guarantee the soundness of this composition. However, the details of the verification progress are encapsulated in the corresponding GLoO units and are not exposed to the current scope.

2.4 The Gateway Mechanism

The built-in gateway mechanism of GLoO provides an approach to directly incorporate Java code into the scope of a GLoO specification unit. The gateway mechanism can

```

let
  println = %{ System.out.println( aArg.toString() ); return aArg; }%

  eval = %{ // check for lazy value
            if( aArg instanceof LazyValue )
              // force evaluation of lazy values
              aArg = (((LazyValue)aArg).getValue()).evaluate( new EmptyForm() );

            return aArg; }%
in
  (| println = println, eval = eval |)
end

```

Listing 2. Selected abstractions provided by `Services.lf`.

be used for the specification of *glue code* to adapt components to fit actual compositional requirements [24], and to extend the GLoO language either by defining supported operators, adding new value types, or incorporating new and readily available programming abstractions. Gateway code is enclosed in `{...}%`, which is treated as a single token by the GLoO compiler. The Java code enclosed in `{...}%` is transformed into a static member function, which is emitted to a predefined runtime support class. The GLoO compiler uses `com.sun.tools.javac` to compile this runtime support class after all gateway specifications have been collected. If no errors are detected, then the generated runtime class is loaded into the GLoO system as a temporary runtime extension to support the evaluation of the top-level GLoO specification unit.

To illustrate the use of the gateway mechanism, consider Listing 2 that shows an extract of the specification unit `Services.lf`. This example illustrates how the functions `println` and `eval` can be defined in GLoO. The function `println` implements the output of the standard textual representation of each data type. It relies on the fact that all supported GLoO values have to override the `Object.toString()` method, so that a proper textual representation can be produced, if necessary. In addition, the reader should note that every gateway function takes one argument, named `aArg`, and has to return a value of a type that is a subtype of the GLoO type `Value`. For this reason, `println` returns its argument, which not only satisfies the protocol of gateway functions, but also allows applications of the function `println` to occur in positions, where its argument is required.

GLoO uses a *strict* evaluation model, that is, terms are evaluated as soon as they become bound to a variable or applied to a function. On the other hand, functions in GLoO are characterized by the arguments they use, not by the ones they define. Unfortunately, these competing aspects pose a serious conflict, because the strict evaluation model forces all arguments to a function to be evaluated before they are applied to it, even though the function may never use them. For this reason, GLoO also provides a special modifier (i.e., the symbol '\$') to explicitly mark a term *lazy*. The lazy modifier is, for example, crucial to the definition of choice statements, where the individual branches must not be evaluated before a corresponding guard evaluates to the value *true* (e.g., the *if-statement*).

```

let
  if = % { /* Java code defining the ternary procedure if-then-else */ } %
in
  (|
    if_1 = (\Bool:: if (| condition = Bool,
                      then = (\Arg:: eval Arg),
                      else = (\Arg:: (|)) |) ),
    if_2 = (\Bool:: if (| condition = Bool,
                      then = (\Then:: (\Else:: eval Then)),
                      else = (\Then:: (\Else:: eval Else)) |) )
  |)
end

```

Listing 3. Definition of conditionals in `Services.lf`.

The evaluation of a lazy term is *delayed* until its evaluation is explicitly triggered by a corresponding program abstraction. This is the purpose of the function `eval`. The `eval` function, as shown in Listing 2, taps directly into the `GLoo` evaluation machinery. More precisely, this function checks, whether its argument `aArg` needs to be evaluated or not by checking if it is an instance of type *LazyValue*. In such a case, `eval` forces the evaluation of `aArg` by calling its `evaluate` method using an empty evaluation environment (i.e., an empty form). This approach allows programmers to switch to a *lazy evaluation* model for selected arguments, and to determine when such arguments should be evaluated, if at all.

2.5 Support for the Definition of Language Abstractions

The specification shown in Listing 3 illustrates how conditionals can be defined in `GLoo`. In this example, we define an *if-statement* for both a single-armed and a two-armed version. The underlying semantics of the *if-statement* is implemented in the ternary gateway function `if`, whose visibility is restricted to the scope of the specification unit `Services.lf`. The functions `if_1` and `if_2` both define a wrapper for the local `if` function in order to implement the desired corresponding behavior of a single-armed and two-armed *if-statement*, respectively. More precisely, `if_1` and `if_2` both define appropriate *continuations* to consume the remaining arguments to a given *if-statement*. In the case of `if_1`, the continuations either force the evaluation of the next argument (i.e., the value `Bool` denotes *true*) or simply discard it (i.e., the value `Bool` denotes *false*). On the other hand, the continuations defined by `if_2` have to consume two arguments in turn, but evaluate only the one that corresponds to the truth value denoted by `Bool`. The reader should note that all arguments except `Bool` have to be marked lazy in order to prevent their premature evaluation, which could interfere with the commonly accepted conceptual model underlying the *if-statement*.

3 Related Work

Python [16], Perl [29], Ruby [27], Tcl [31], CLOS [25], Smalltalk [10], Self [28], or Scheme [7] are examples of programming languages in which programs can change

their structure as they run. These languages are commonly known as *dynamic programming languages* and *scripting languages* [21], respectively. However, the degree of dynamism varies between languages. For example, Smalltalk and Scheme are languages that permit simultaneously *syntactic* and *semantic extensions*, that is, everything is available for modification without stopping to recompile and restart. Python, Perl, JavaScript, and Self, on the other hand, support only semantic extensions either by *dynamic (re-)loading* of runtime modules, *runtime program construction*, or *copying and modifying prototypes*. However, program extensions defined in this way can only be expressed in either the language itself, C/C++, or a corresponding API. Extensions written in other languages may not be integrated as easily.

Since Smalltalk and Scheme both provide support for syntactic extensions, these languages are also examples of so-called *open-ended* programming languages. Open-ended languages allow for extending the language with new programming abstractions on demand. However, in the case of Smalltalk and Scheme, these extensions can only be defined in the very same language as the host language. An extreme example of an *open-ended* language is CDL [11], which is a programming language with an empty kernel. More precisely, CDL admits only one type, the *word*, which can be interpreted in the language only by means of *macros*. No other predefined concrete algorithms, types, or objects exist in the language. CDL provides, however, construction mechanisms for algorithms. Types and objects, on the other hand, cannot directly be expressed in the language. These have to be supplied by means of CDL's extensions mechanisms, which enable one to *borrow* new language constructs from outside the language. Language extensions are defined in *macro libraries* that serve as *semantic extensions* (CDL does not support *syntactic extensions*). In practice, these macro libraries are organized as standard API's capturing a specific application domain. As a result, programming in CDL is not much more cumbersome than programming in a mainstream and general-purpose programming language like C, Java, or C#.

4 A Model for Traits

In this section, we present a model of traits [23] as an example of how to add support for a new language abstraction to the GLoo framework.

Traits offer a simple compositional model for factoring out common behavior and for integrating it into classes in a manner consistent with inheritance-based class models [23]. Traits are essentially sets of related methods that serve as (i) a building block to construct classes, and (ii) a primary unit of code reuse. Reuse is a primary tenant in a component-oriented software development approach and it is, therefore, natural that we seek to explore the means for providing support for traits in the GLoo framework.

Unfortunately, to view a trait simply as set of methods is rather misleading, as traits require a rich supporting infrastructure in order to unfold their expressive power. Schärli et al. [23] characterize the properties of traits as follows:

- A trait exposes its behavior by a set of *provided* methods.
- A trait declares its dependencies by a set of *required* methods that serve as arguments to the provided behavior.

```

let
  read = (\():: Services.print "Input number: ";
          Services.stringToInt (Services.readString (|)))

  TReadIntProvides =
    (|
     readInt = (\():: (self (|)).setIntField (| aIntField = read (|) |))
    |)

  TReadIntRequires =
    (|
     readInt = (| setIntField = "Int -> Unit" |)
    |)

in
  trait "TReadInt" provides TReadIntProvides requires TReadIntRequires
end

```

Listing 4. Definition of trait TReadInt.

- Traits are stateless. A trait does not define any state variables, and its provided behavior never refers to state variables directly.
- Classes and traits can be composed to construct other classes or traits. Trait composition is commutative, but conflicts have to be resolved explicitly and manually.
- Trait composition does not affect the semantics of both classes and traits. Adding a trait to a class or trait is the same as defining the methods obtained from the trait directly in the class or trait.

In addition, to facilitate conflict resolution, Schärli et al. [23] propose two auxiliary operations: *method aliasing*, and *method exclusion*. These operations together with a suitable *flattening mechanism* for trait composition are required in a comprehensive approach that provides support for the traits concept in a class-based programming model.

The first two trait properties can easily be mapped to the concept of explicit namespaces. Unfortunately, trait composition, method aliasing, and method exclusion require an additional compile-time support that is not part of the GLoo framework by default. However, as we have shown in earlier work [20], object- and component-oriented abstractions can most easily be modeled if they are represented as *first-class entities*. We can use this approach to define a *meta-level* architecture that provides the means to differentiate the compositional aspects from the programming aspects of the traits concept.

4.1 Specifying Traits in GLoo

Programmatically, we define a trait as illustrated in Listing 4. The specification shown in Listing 4 defines a trait, called TReadInt, that defines one provided method readInt, and declares the method setIntField with the signature Int -> Unit as a required method of readInt. The focus of this work is on a suitable representation of traits, not on a type system for traits. It is, however, highly desirable to provide additional information regarding their compositional constraints for required methods of a trait. For this reason, we utilize the type syntax proposed by Fisher and Reppy [8] for a

```

let
  MetaTraits = load "Extensions/Traits.lf"
in
  (|
    trait = (\Name:: (\Cont:: Cont (| traitName = Name |))),
    provides = (\Args:: (\P:: (\Cont:: Cont (| Args, provides = P |))),
    requires = (\Args:: (\R:: MetaTraits.newTrait (| Args, requires = R |))),
    join = (\Args:: (\L:: (\Cont:: Cont (| Args, left = L |))),
    with = (\Args:: (\R:: MetaTraits.composeTraits (| Args, right = R |)))
  |)
end

```

Listing 5. Selected abstractions provided by `LanguageOfTraits.lf`.

statically typed calculus of traits, classes, and objects, but the type annotations are for documentation purposes only. A future model of traits in GLoo may also define a type verification process that takes these annotations to perform additional checks.

The general format of a trait specification follows the structure used to define the trait `TReadInt`. Within the local scope of a trait, we define the sets of provided and required methods. The set of provided methods is a form that maps the provided methods to their corresponding implementations. In method bodies, the term `(self (| |))` yields the current instance, and allows for *dynamic binding* of methods. Provided methods may also rely on some private behavior not exposed to clients. For example, the method `readInt` calls the private method `read` to fetch an integer value from the console.

The set of required methods is also represented by a form. However, each binding in that form maps to another form that records all methods, including their signatures, a given provided method depends on. This format is more verbose than the original specification of Schärli et al. [23], but it addresses a problem that can occur when defining the *exclusion of a method* in a trait. In such a case, we need to add the excluded method potentially to the set of required methods. In order to decide this question, we need to explore all remaining provided methods. Without the additional structure in our model, we have to extend the search to the source code of the provided methods, which may not be accessible anymore at the time of the search.

The required core abstractions to define traits are shown in Listing 5. The bindings defined in the unit `LanguageOfTraits.lf` serve as language extensions to an importing scope. The associated functions are defined in *continuation-passing style* (CPS) that mimics the parsing process of the corresponding syntactic categories. For example, `trait` is a function that takes a name of a trait and returns a function that consumes a continuation to construct a new trait. The continuation can be either the function `provides` to build a new trait or the function `join` to compose a trait with another trait. Both `provides` and `join` yield a function that takes a final continuation to actually perform the desired operation. In case of `provides`, we need to use the function `requires` that passes its arguments to meta level function `newTrait` to register a new trait with the meta level trait infrastructure. The function `join`, on the other hand, requires the function `with` that passes its arguments to `composeTraits`, a meta level function to construct a composite trait.

```

composeTraits =
(\Arg::
  let
    left = getTraitInfo Arg->left
    right = getTraitInfo Arg->right
  in
    Services.if_2
      (is_trait_composition_sound
        (| common = Services.intersection (| left = left->provides,
                                           right = right->provides |),
          left_origins = left->origins,
          right_origins = right->origins |))
      ($ let
        provides = (| (| left->provides |) # (| right->provides |) |)
        requires =
          filter_required
            (| required = (| (| left->requires |) # (| right->requires |) |),
              provided = provides |)
        origins = (| (| left->origins |) # (| right->origins |) |)
      in
        registerTrait (| traitName = Arg.traitName, provides = provides,
                       requires = requires, origins = origins |)
      end)
      ($ (Services.error "Conflicting trait methods encountered!"))
end)

```

Listing 6. Definition of method `composeTraits` in `Traits.lf`.

4.2 Operational Support for Traits

In order to add support for the traits concept to the GLoo framework, we represent traits at both a meta level and a programming level. The meta level defines the abstractions to (i) compose traits, (ii) alias methods, (iii) exclude methods, and (iv) encode metadata associated with traits. The programming level, on the other hand, provides a set of high-level abstractions to define and use traits. The programming level completely encapsulates the meta level and therefore shields the application programmer from the underlying complexity of the traits concept. Moreover, both the meta level and the programming level constitute a narrow-focused *domain-specific sublanguage* that enables us not only to use traits in GLoo specifications, but also to reason about the features and constraints of the traits concept.

The meta level support for traits is defined in the unit `Traits.lf` that defines an internal representation of traits called `MetaTrait`. A `MetaTrait` encodes the metadata associated with every trait. It records (i) the trait name, (ii) the set of provided methods, (iii) the set of required methods, and (iv) the *origins* of all provided methods. The latter is used for conflict resolution and enables us to check whether conflicting methods originate from the same trait in which case the conflict is resolved immediately.

The meta level also defines the functions `registerTrait`, `filterRequired`, and `is_trait_composition_sound` to name a few. These functions¹ are used to define the function `composeTraits` (cf., Listing 6), which takes two traits and builds their union, if possible. The purpose of `is_trait_composition_sound`

¹ A detailed presentation of these functions has been omitted due to a lack of space.

is to check that the provided methods of the two traits being composed are pairwise distinct. In order to verify this condition, we also pass the origins of both traits to `is_trait_composition_sound`. We acquire the origins of both traits by calling the function `getTraitInfo`, which returns the metadata associated with the corresponding trait. If the soundness test succeeds, then we are actually composing both traits by building (i.e., *flattening*) the new sets of provided and required methods, and the new joined origins. We pass these data together with the corresponding trait name to the meta-function `registerTrait`, which (i) registers the newly composed trait with the meta-level infrastructure, and (ii) returns the programming representation of it.

The unit `Traits.lf` also defines functions for method exclusion and method aliasing. These functions take a trait and a list of method names to either be excluded or aliased. The structure of these functions resembles the one of `composeTraits`. However, a detailed presentation of them has been omitted due to lack of space.

4.3 Using Traits

In the previous section, we have presented the core abstractions for trait construction and composition. In this section, we illustrate briefly how to compose classes and traits.

Trait composition is commutative, that is, the composition order is irrelevant. Conflicts must be explicitly resolved. However, Schärli et al. [23] define two additional criteria that must be also satisfied in the case of the composition of classes and traits:

- 1) “*Class methods take precedence over trait methods.*”
- 2) “*Trait methods take precedence over superclass methods.*”

In other words, the provided methods of a trait have to be *inserted* into a class between its inherited behavior and the behavior defined by that class. This is a logical consequence of the *flattening property* that requires that methods obtained from a trait must behave as if they were defined in the class directly [23].

To illustrate the composition of classes and traits, consider the GLoo specification unit shown in Listing 7. This unit defines a *meta-class* of the class `IntClass` that defines two public methods `getIntField` and `setIntField`, and a private instance variable `fIntField`. The general structure of a class definition is given by three abstractions: (i) an incremental modification, (ii) a generator, and (iii) a class wrapper [15]. In the case of class `IntClass`, these abstractions are `deltaIntClass`, `IntClassG`, and `IntClassW`, respectively. These abstractions not only define the required behavior of class `IntClass`, but also the underlying object model², which, in the case of `IntClass`, adheres to Java semantics.

We use the abstractions `deltaIntClass`, `IntClassG`, and `IntClassW` to construct a meta-class of class `IntClass`. This meta-class is actually a function that may take a trait as argument. To instantiate a class, we have to call this function using either a trait or an empty form as argument. The latter will simply instantiate the corresponding class, since we have applied an empty extension. If we, however, apply

² These two aspects should be specified in different scopes in order to separate the concerns and to raise the level of abstraction. We have proposed a solution for this problem in [15].

```

let
  fix = load "System/fix.lf"
in
  (\Trait::
    let
      IntClassBehavior =
        (|
          getIntField = (\():: State.fIntField ),
          setIntField = (\Args:: self (| State, fIntField = Args.aIntField |) )
        |)
      deltaIntClass =
        let
          pureTrait = Traits.pureTrait Trait
        in
          (Services.if_1
            (Services.not_empty pureTrait)
            ($ let
              allRequired =
                Traits.buildAllRequired (Traits.getTraitInfo Trait)->requires
            in
              (Services.if_1
                (Services.not_empty
                  (| (| allRequired |) \ (| IntClassBehavior |) |))
                ($ (Services.error "Composition incomplete!")))
              end));
            (| pureTrait # IntClassBehavior |)
          end
        IntClassState = (| fIntField = 0 |)
        deltaClass = (\State:: deltaIntClass)
        IntClassG =
          (\OArgs:: deltaClass
            (| OArgs # (| IntClassState \ (| fIntField = OArgs.fIntField |) |) |))
        IntClassW = (\OArgs:: (fix (| f=(\self:: IntClassG) |)) OArgs)
      in
        (| W=IntClassW, G=IntClassG |)
      end)
    end)
end

```

Listing 7. Definition of the (meta-)class `IntClass`.

a proper trait, then the behavior defined by this trait is composed with the behavior defined by the class in accordance with the criteria for the composition of classes and traits. First, we verify that the composition is closed, that is, all required trait methods are implemented by the class. Secondly, we merge the methods of both the trait and the class. By using form extension class methods are given precedence over trait methods. The result denotes a new incremental modification of the class that is obtained from the composition of the original incremental modification and the trait methods. The class generator finally merges any present inherited behavior with the new incremental behavior³ to create instances of the extended class.

5 Conclusion and Future Work

In this paper, we have presented GLoo, a framework for modeling and reasoning about component-oriented language abstractions. At the center of this framework is a small

³ The class `IntClass` does not inherit any behavior, hence this step is the identity function.

dynamic composition language that is based on the $\lambda\mathcal{F}$ -calculus [14]. The main tenants of the GLoO programming paradigm are dynamic binding, explicit namespaces [3], and incremental refinement. These concepts together with a built-in gateway mechanism to incorporate Java code directly into the scope of GLoO specification units provides us with the means for a problem-oriented software development approach.

To demonstrate how a domain-specific sublanguage can be defined, we have implemented the traits concept [23] in GLoO. The *language of traits* is defined as a readily available language abstraction that can be loaded on demand. The specific added value of this abstraction is a clear separation between compositional and programming aspects of traits, which facilitates both the construction and the composition of traits.

We have studied the encoding of classes, traits, and objects in GLoO. We need, however, also support for the integration of external software artifacts into the GLoO framework. We plan, therefore, to further explore the gateway concept in order to incorporate existing Java classes and components. Like the language of traits, we envision a narrow-focused domain sublanguage of classes and components that will allow application programmers to use existing Java abstraction as they were defined in GLoO directly.

Acknowledgements

We are deeply grateful to Andrew Cain for suggesting the name GLoO and Jean-Guy Schneider and the anonymous reviewers for their valuable comments and discussions.

References

1. Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lèvy. Explicit Substitutions. In *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages (FSE '90)*, pages 31–46, San Francisco, California, 1990. ACM.
2. Franz Achermann. *Forms, Agents and Channels: Defining Composition Abstraction with Style*. PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, January 2002.
3. Franz Achermann and Oscar Nierstrasz. Explicit Namespaces. In Jürg Gutknecht and Wolfgang Weck, editors, *Modular Programming Languages*, LNCS 1897, pages 77–89. Springer, September 2000.
4. Uwe Aßmann. *Invasive Software Composition*. Springer, 2003.
5. Laurent Dami. A Lambda-Calculus for Dynamic Binding. *Theoretical Computer Science*, 192:201–231, February 1998.
6. Frank DeRemer and Hans H. Kron. Programming in the Large versus Programming in the Small. *IEEE Transactions on Software Engineering*, SE-2(2):80–86, June 1976.
7. Kent Dybvig. *The Scheme Programming Language*. MIT Press, third edition, October 2003.
8. Kathleen Fisher and John Reppy. Statically typed traits. Technical Report TR-2003-13, University of Chicago, December 2003.
9. GLoO. <http://www.cs.iastate.edu/~lumpe/GLoo>.
10. Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley, September 1989.
11. Cornelis H.A. Koster and H.-M. Stahl. *Implementing Portable and Efficient Software in an Open-Ended Language*. Informatics Department, Nijmegen University, Nijmegen, The Netherlands, 1990.

12. M. M. Lehman, D. E. Perry, J. C. F. Ramil, W. M. Turski, and P. Wernik. Metrics and Laws of Software Evolution – The Nineties View. In *Proceedings of Fourth International Symposium on Software Metrics, Metrics 97*, pages 20–32, Albuquerque, New Mexico, November 1997. Also as chapter 17 in Eman, K. El, Madhavji, N. M. (Eds.), *Elements of Software Process Assessment and Improvement*, IEEE CS Press, Los Alamitos, CA, 1999.
13. Markus Lumpe. *A π -Calculus Based Approach to Software Composition*. PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, January 1999.
14. Markus Lumpe. A Lambda Calculus With Forms. In Thomas Gschwind, Uwe Aßmann, and Oscar Nierstrasz, editors, *Proceedings of the Fourth International Workshop on Software Composition*, LNCS 3628, pages 83–98, Edinburgh, Scotland, April 2005. Springer.
15. Markus Lumpe and Jean-Guy Schneider. A Form-based Metamodel for Software Composition. *Science of Computer Programming*, 56:59–78, April 2005.
16. Mark Lutz. *Programming Python*. O’Reilly & Associates, October 1996.
17. Oscar Nierstrasz and Franz Acherermann. A Calculus for Modeling Software Components. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Proceedings of First International Symposium on Formal Methods for Components and Objects (FMCO 2002)*, LNCS 2852, pages 339–360, Leiden, The Netherlands, 2003. Springer.
18. Oscar Nierstrasz and Laurent Dami. Component-Oriented Software Technology. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, pages 3–28. Prentice Hall, 1995.
19. Oscar Nierstrasz and Theo Dirk Meijler. Requirements for a Composition Language. In Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, LNCS 924, pages 147–161. Springer, 1995.
20. Oscar Nierstrasz, Jean-Guy Schneider, and Markus Lumpe. Formalizing Composable Software Systems – A Research Agenda. In *Proceedings the 1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems*, pages 271–282. Chapman & Hall, 1996.
21. John K. Ousterhout. Scripting: Higher Level Programming for the 21st Century. *IEEE Computer*, 31(3):23–30, March 1998.
22. Johannes Sametinger. *Software Engineering with Reusable Components*. Springer, 1997.
23. Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable Units of Behavior. In Luca Cardelli, editor, *Proceedings ECOOP 2003*, LNCS 2743, pages 248–274. Springer, July 2003.
24. Jean-Guy Schneider. *Components, Scripts, and Glue: A conceptual framework for software composition*. PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, October 1999.
25. Guy L. Steele. *Common Lisp the Language*. Digital Press, Thinking Machines, Inc., 2nd edition, 1990.
26. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, Second edition, 2002.
27. Dave Thomas. *Programming Ruby – The Pragmatic Programmers’ Guide*. The Pragmatic Bookshelf, LLC, second edition, 2005.
28. David Ungar and Randall B. Smith. SELF: The Power of Simplicity. In *Proceedings OOPSLA ’87*, volume 22 of *ACM SIGPLAN Notices*, pages 227–242, December 1987.
29. Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O’Reilly & Associates, Third edition, July 2000.
30. Peter Wegner. OOPSLA’87 Panel P2: Varieties of Inheritance. *SIGPLAN Not.*, 23(5):35–40, May 1988.
31. Brent B. Welch. *Practical Programming in Tcl and Tk*. Prentice Hall PTR, second edition, 1997.