Growing a Language: The GLOO Perspective

Markus Lumpe

Faculty of Information & Communication Technologies
Swinburne University of Technology
P.O. Box 218
Hawthorn, VIC 3122, Australia
mlumpe@swin.edu.au

Abstract. The design of programming languages is, in general, geared towards *accumulation* rather than *composition* of features. However, by adding an ever-increasing number of built-in abstractions, any programming language is eventually at risk to reach a critical mass at which it may become increasingly difficult for designers to maintain and for developers to use an evolving language appropriately. To tackle this *language design paradox*, we have developed GLOO, a small open-ended dynamic language, whose design philosophy aims at a unified approach in which program and language evolution result directly from the definition of *extensible domain sub-languages*. Surprisingly, these extensible domain sub-languages not only provide a framework to capture domain expertise, but also give rise to a powerful compositional model for language extension. To demonstrate the effectiveness of this approach, we develop the *Language of Namespaces and Traits* in this paper. We define this extensible domain sub-language as an aggregate of various forms of object-oriented language support. Using the *Language of Namespaces and Traits* as example, we show that GLOO's extension model plays a crucial role in achieving a flexible compositional approach for the design of readily-available and extensible programming abstractions.

1 Introduction

A major contributing factor for the success or failure of a software system is not only our understanding of the underlying problem domain, but also the choices of programming languages and their support in the target environment. This poses a particular challenge for language designers, who often have to choose between the features that a language has to provide and the ones that would make the language more versatile. A well-designed programming language can yield a creative medium for making programmers write good programs easily [19]. An overloaded language, on the other hand, may increase the likelihood of occurrences of awkward or lengthy formulations as developers find it more difficult to proactively organize their solutions within the framework provided by the language.

But how do we assess language features in practice? What are the practical means to implement, test, and incorporate new language abstractions into an existing programming language? In addition, the design of industrial-strength programming languages is, in general, geared towards *accumulation* rather than *composition* of language features [12]. However, by adding an ever-increasing number of built-in abstractions any

C. Pautasso and É. Tanter (Eds.): SC 2008, LNCS 4954, pp. 1–19, 2008.
© Springer-Verlag Berlin Heidelberg 2008

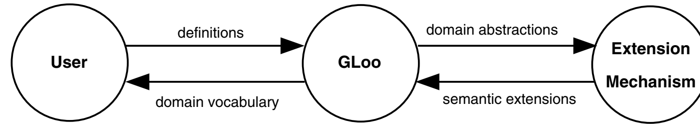

Fig. 1. The GLOO language extension model

programming language is eventually at risk to reach a critical mass at which it may become increasingly difficult to formally define, use, and maintain the language [25]. So, can we construct more effective means to "fine-tune" the level of abstraction provided by a programming language in order to avoid overwhelming both the programming language and the application programmer?

To study and experiment with different means of language support for software composition, we have developed GLOO [16, 15], a small open-ended dynamic pure functional language with a built-in extension mechanism to grow the language on demand. GLOO allows for both rapid language prototyping and the definition of readily available language abstractions. For example, we have defined the *Language of Java Services* [16] and the *Language of Traits* [15], two extensible domain sub-languages that provide an interface to incorporate existing Java software artifacts (i.e., Java classes and objects) and the concept of traits [24] into the GLOO framework. These *domain sub-languages* greatly benefit from the extension mechanism embedded in GLOO that is a key ingredient for the definition of arbitrary domain abstractions ranging from new data types to complex programming constructs in order to yield a *user-centric* view of the targeted problem domain.

The GLOO model for language extension induces two processes: *decomposition* and *abstraction*. The result of the decomposition phase is a set of *meta-level* domain abstractions that implement required core data types and their relationships in the problem domain, whereas the abstraction step yields a high-level and user-centric *domain vocabulary* to represent a desired specific aspect (or *view*) of the underlying problem domain [16]. The resulting domain sub-languages serve as fined-grained extensions to the GLOO language and can be viewed as subjects [21] or *compositional styles* [2] that encapsulate sets of first-class development artifacts to assist developers in solving problems in a given domain in a more efficient and convenient way. A schematic view of this technique is shown in Figure 1.

In this work our focus is on rapid prototyping of object- and class-based language extensions using *continuation-passing-style* (CPS). Rapid prototyping is a viable engineering technique to explore and validate desirable system characteristics of software products in a flexible and agile manner. Dynamic programming languages with their emphasis on developer productivity and software quality provide a good fit for the required programming approach. However, with the exception of Scheme or Smalltalk, languages are seldom used or designed to study programming language concepts in itself.

To explore how GLOO can be used to this purpose we define the *Language of Namespaces and Traits* in this paper. More precisely, we demonstrate how to specify the *Language of Namespaces and Traits* as aggregations of smaller domain sub-languages

using the concept of *mini parsers*. Mini parsers are *guarded continuations* that mimic the parsing process of the underlying syntactic categories. The names of these guarded continuations serve as keywords in the defined sub-language, whereas their bodies define *reusable parsing automata* for the corresponding associated keywords. We use the characterization of "reusable" to denote the fact that the guards of parsing automata are defined over equivalence classes of *permissible continuations*, which allows us to recombine mini parsers to accept new syntactic forms on demand.

The benefit of defining and evaluating language extensions this way is two-fold. First, the integration of new features into an existing language may impact the underlying language processor to an extent at which language experimentation becomes not feasible anymore. GLoo does not suffer from this problem, as its fine-grained scoping mechanisms provide us with the means to control the impact and visibility of different and, in general, orthogonal language features. Second, the notion of permissible continuations enables us to compose arbitrary mini parsers by locally defining corresponding equivalence classes, a technique that yields a natural approach for language composition.

The rest of this paper is organized as follows: in Section 2, we illustrate the main features of the GLoo programming model. In the Sections 3 and 4, we discuss the details of defining language extensions in GLoo. We present a model for the *Language of Namespaces and Traits* in Section 5. We proceed with a brief review of related work in Section 6 and conclude with a summary of our main observations in Section 7.

2 Abstraction Definition in GLoo

GLoo is a statically scoped language that uses *call-by-value* as default parameter passing mode. The core language of GLoo is based on $\lambda\mathcal{F}$ [14], a variant of the λ-calculus that combines the concepts of *dynamic name resolution*, *explicit namespaces*, and *foreign code gateway* in a single formal framework. Dynamic name resolution and explicit namespaces both are crucial for a software development approach enabling a controlled admission of new functionality into an existing software base [6]. The foreign code gateway, on the other hand, allows for an easy integration of *glue code* that is required to provide operational support for new language mechanisms [15].

There is however an important aspect to the semantics of name resolution that is unique to GLoo. GLoo permits for the occurrences of *unbound* names in expressions. Unbound names are placeholders for values that may be defined in the future. Unbound names are not to confused with free names that result in open expressions. In GLoo, every expression is closed, even the ones containing unbound names. An expression with occurrences of unbound names is subject to *incremental refinement* [16] that makes it possible for programmers to alter the meaning of an expression depending on the eventual existence of suitable declarations for unbound names. This concept is similar to the way content types are handled in Web-browsers. If a Web-browser defines a specific handler for a given content type, then this handler is invoked. Otherwise, the browser triggers the default behavior or may even ignore that content type altogether. In GLoo, every expression is evaluated with respect to an actual evaluation context to assign meaning to occurrences of unbound names. If that context does not define a

mapping for the occurrence of an unbound name, then that occurrence is substituted with the empty behavior.

The main programming entity in GLOO is a *specification unit*, defining a value or *component* that can be recombined with additional values or components exported by other specification units. GLOO specification units add basic data types, an import facility, term sequences, expression trees, computable binders, and a *Java gateway mechanism* to the core language. In essence, a GLOO specification unit declares a local scope for both the import of exported abstractions from other units and the definition of new abstractions that all contribute to the value exported by the current specification unit.

```
let
  load "OpenExtensibleImperativeClass.lf"
  load "LanguageOfNamespacesAndTraits.lf"
  load "System/Services.lf"

  StackNS = load "SampleNS.lf"
in
  Namespace TraceNS
    Import Stack of StackNS
       method top (\():: println "Calling top...";
                   super.top (||))
    endImport

    apply (use TReadInt of SampleNS) to Stack
  endNamespace
end
```

Listing 1. The namespace `TraceNS` in GLOO

Consider Listing 1 that depicts the use of the *Language of Namespaces and Traits*. The specification in this example defines the namespace `TraceNS`. This namespace exports one class, `Stack`, that is a refinement of class `Stack` imported from namespace `SampleNS`. In namespace `TraceNS` the class `Stack` is modified twice: we (i) define method `top` as an extension to class `Stack` and (ii) apply trait `TReadInt` to it. As a result, in namespace `TraceNS` we obtain a new version of class `Stack` that provides all inherited members defined by class `Stack` in namespace `SampleNS`, overrides method `top`, and guarantees a sound composition with trait `TReadInt`.

The structure of `TraceNS` is typically for programming abstractions defined at the *application level* in GLOO. Even though GLOO offers only a few basic constructs, these language elements suffice to define high-level programming features to build domain software artifacts easily. The core elements that enable this particular application level paradigm are *functions* and *function composition*. More precisely, the structure-giving elements are all functions (e.g., `Namespace`, `Import`, `of`, `method`, or `endImport`), whose names serve as domain-specific keywords, whereas the function bodies implement continuations that define the semantics and the corresponding verification rules of the modeled language features. By composing related *keyword functions* we obtain a desired new language element and assign it a concrete syntax and semantics. Moreover, these language elements remain *extensible* and can, therefore, be recombined to yield support for other language abstractions.

However, GLOO is also an open-ended programming language. Therefore, rather than providing a rich set of predefined operators and statements, GLOO only defines syntactic support for them. To give operators semantics or to add new language constructs like conditionals, loops, or assignment, one has to define *language extensions* to *borrow* appropriate and matching behavior from outside the language. This approach allows for an easy integration of new domain data types into the language and to fine-tune the existing language features to specific user needs.

```
%{ // auxiliary Java runtime representation of Cell data type
   class CellValue extends LiteralValue
   {
    private Value val;

    public Value get()            { return val; }
    public void set( Value v )    { val = v; }
    public CellValue( Value v )   { val = v; }
    public String toString()      { return val.toString(); }
   }
}%

let
 // gateway functions
 makeCell= %{ return new CellValue( aArg ); }%

 get= %{ Value c = EpsilonValue.EPS;
         try { c = ((CellValue)aArg).get(); }
         catch (FormException e) { Main.error( e.getMessage() ); }
         return c;                                                 }%

 set = %{ try { CellValue c = (CellValue)aArg.select("cell");
                Value  v = aArg.select( "value" );
                c.set( v ); }
          catch (FormException e) { Main.error( e.getMessage() ); }
          return EpsilonValue.EPS;                                 }%
in
 // exported Cell data type
 (| Cell = (\val:: (| get = (\()::get this),
                      set = (\nval::set (| cell = this, value = nval |)) |)
                    [(| this = makeCell val |)]) |)
end
```

Listing 2. A mutable `Cell` abstraction

One such language extension is `Cell`, as shown in Listing 2. The unit defining the `Cell` abstraction consists of three parts: (i) the definition of the auxiliary Java code that defines the (inner) Java class `CellValue`, (ii) the local declaration of the gateway functions `makeCell`, `get`, and `set`, and (iii) the definition of the exported function `Cell`, a data type constructor for mutable storage cells.

GLOO provides an explicit means to directly incorporate Java code into the scope of a specification unit in form of *gateway code*. Gateway code is enclosed in `%{...}%` and treated as a single token by the GLOO compiler. The GLOO compiler assembles the gateway code in a corresponding Java support class and emits appropriate linkage code to bridge between the GLOO and the Java world [16, 15]. In case of the `Cell` abstraction, the gateway code comprises of the definitions for `CellValue`, a class

derived from the GLoo runtime type `LiteralValue` to obtain a mutable container value type, the data type constructor `makeCell`, and the property methods `get` and `set`.

We define `Cell` as a wrapper function that returns a programming interface to `CellValue`. The body of `Cell` contains an expression of the form `a[b]`, called *context*, which denotes a term `a` whose meaning depends on the values defined in `b` if `a` contains free variables. In the case of `Cell`, the context

```
[(| this = makeCell val |)]
```

defines a binding for the variable `this`, which occurs free in the bodies of the functions bound to the exported names `get` and `set`. The effect of this context specification is that both functions share the common value `this` and since `this` is a mutable storage cell, we add *side effects* to the getter and setter.

The particular value of the `Cell` extension is that it adds a stateful programming abstraction to the GLOO language. As a pure functional language, GLOO does not possess any built-in support for assignment. However, certain approaches (e.g. object-oriented programming) are naturally *imperative* and allow or require operations to perform side effects on the state of other program entities [1]. It is one of the strengths of GLOO to accommodate *orthogonal* programming features while providing well-defined scopes within which these features are available and may impact each other. In the case of the `Cell` abstraction, the modeled assignment abstraction appears to be in fundamental conflict with GLOO's declarative programming model. However, the actual `Cell` object is implemented by means of a *read-only* literal value (i.e., an instance of class `CellValue`, a user-defined class derived from the GLOO runtime class `LiteralValue`) that hides its state-altering capability from clients of the `Cell` abstraction.

One of the rather subtle aspects in defining extensible software abstractions is *naming*. The choice of names can greatly effect our ability to recombine existing software artifacts as name clashes may occur. Also, sometimes we may not know the precise set of names to access individual features of software abstractions. For this reason, GLOO provides *computable binders*, expressions enclosed in the symbols { and }, that allow for both discovery and construction of names (i.e., labels of a namespace) at runtime.

```
let
 fix = (\F::h (| {FName} = h |))
        [(| h = (\FX::F (\Arg::(FX.{FName} FX) Arg)) |)]
in
 (\$Name:: (\FName:: (\F::fix (\{FName}::F))) (getIdString Name))
end
```

Listing 3. The recursive function builder `Rec`

A typical application of computable binders occurs in recursive functions. By design, GLOO does not provide a built-in support for the definition of recursive abstractions. We can, however, define a simple recursive function builder, as shown Listing 3. The recursive function builder `Rec` consists of two parts: (i) the local declaration of the

call-by-value fixed-point combinator `fix` and (ii) the exported definition of `Rec`, which constructs the proper recursive image of its argument function `F`. Both parts rely on computable binders. In the case of `Rec`, the computable binder { `FName` } enables *self-application* within the body of the recursive function being defined. For example, to build a recursive image of a function `F` we write `Rec self F`, which is an expression that yields a function in which `self` is the name of `F` in the function's body.

The feature that enables this particular technique is *delayed term evaluation* that enables us to defer the evaluation of arguments to functions until their value is actually being required [15]. Hence, when using delayed term evaluation, we have to ability to explicitly define *lazy evaluation* of function arguments. The need for delayed term evaluation arises, for example, from choice functions like conditionals in which the individual branches must not be evaluated before the corresponding guard evaluates to true. We use the symbol `$` to mark an expression delayed. Prefixing an expression `e` with the symbol `$` yields the *expression tree* of `e`. An expression tree comprises of the syntax tree of the denoted value and its lifetime *evaluation context history* to maintain static scoping.

Expression trees for identifiers are of particular interest, as they enable, in combination with incremental refinement, a *macro mechanism* [23]. Incremental refinement allows for *open* sub-expressions in both function bodies and actual function arguments. The idea behind this concept is that the target environment for the evaluation of a function may provide specific local bindings for occurrences of free names. For example, `fix` contains two free occurrences of `FName`. This enables `fix` to "adapt" to any function name. The purpose of `fix` is to generate the required repetitive structure of a recursive function, but the function name is not known at point of the declaration of `fix`. By placing `fix` underneath the binder `(\FName::(\F::fix ...))`, we capture `FName` and associate it with the name of the function being constructed to achieve the desired recursive linkage. The actual value of `FName` is the string denoted by the expression tree `Name`. We can use the gateway function `getIdString` to obtain the corresponding string representation.

3 Language Composition in GLOO

The definition of new language extensions does not occur in isolation. Programming idioms supported by languages like C# [18], Haskell [5], Java [3], Python [17], Perl [28], Self [26], Scheme [12, 23], Smalltalk [10], or Tcl [22] offer already a wealth of readily available and well-explored programming abstractions. However, only a few systems provide built-in support for *syntactic* and *semantic extensions* to add or experiment with new programming concepts.

GLOO's built-in *compositional* extension mechanism enables developers to amend the language through syntactic extensions, semantic extensions, or both. The main pillar of this compositional approach is the hypothesis that a language must reveal the need for additional features by removing pertinent weaknesses and restrictions [12]. Users supply definitions to model the problem domain to the GLOO compiler. The compiler maps the corresponding domain abstractions to associated representatives in Java support classes. The extension apparatus of GLOO translates those classes into executable

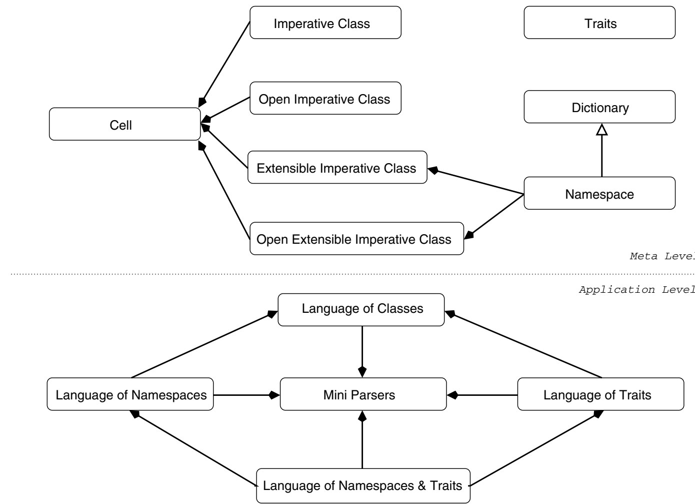

Fig. 2. A model for class-based programming features in GLOO

semantic extensions and loads them into the current GLOO runtime image [16]. Upon completion, the user-supplied definitions yield a domain vocabulary, which captures the modeled problem domain in a user-centric way and therefore facilitates program development for that domain.

Common to all GLOO domain abstractions is that they are composed from a *meta level*, a low-level layer that defines the behavior required to incorporate new domain abstractions into the GLOO runtime system, and an *application level*, a high-level layer that encapsulates the meta level and provides the application programmer with a user-centric domain vocabulary of the modeled domain. To illustrate this approach, consider Figure 2 that depicts the architecture of a set of class-based language extensions. These language extensions provide a Java-like programming model. At the meta level, we define objects, traits, classes, and dictionaries as *first-class* values. Furthermore and in order to obtain an imperative object model, all features use `Cell`. More precisely, *Imperative Class*, *Open Imperative Class*, *Extensible Imperative Class*, *Open Extensible Imperative Class*, *Dictionary*, and *Namespace* all encapsulate a `Cell` object and define an appropriate wrapper to achieve the desired corresponding imperative behavior.

The application level, on the other hand, is composed of *Mini Parsers* and the domain sub-languages *Language of Classes*, *Language of Traits*, *Language of Namespaces*, and *Language of Namespaces and Traits*. The intriguing aspect in the construction of the application level is that none of the defined sub-languages incorporates an object model directly. However, each language contains an occurrence of the unbound name `Class` that refers to a *class builder*. We bind the object model *late*, that is, rather than resolving occurrences of name `Class` within the defining scope, we use incremental refinement and provide a desired meaning in the importing scope. Consider again Listing 1. In this specification, we load the definitions for *Open Extensible Imperative Class* before we include the *Language of Namespaces and Traits* into the specification unit for

TraceNS. As a result, we bind Class within the scope of *Language of Namespaces and Traits* to the class builder defined by *Open Extensible Imperative Class* and obtain a suitable programming model for the definition of TraceNS.

4 Mini Parsers

For the construction of domain sub-languages at the application level we have developed the notion of *composable mini parsers*, which are first-class entities to capture the keywords of a specific syntactic category. Mini parsers provide an ambiguity-free specification format like *Parsing Expression Grammars* (PEGs) [9] to describe the syntactic structure of the underlying problem domain entities. We use continuation-passing-style to specify mini parsers. Moreover, as the mini parsers are defined in GLOO itself, we can avoid the integration of different tools and paradigms [11]. However, GLOO is not a compiler-compiler. Like Scheme [23], GLOO does presently not provide any support to alter or enrich its lexical syntax. As a consequence, we can only use identifiers as structure-giving elements when defining new language extensions.

<*Class*> ::= **'Class' 'super'** <*Class*> (<*Member*>)* **'endClass'**

<*Member*> ::= [**'static'**] **'var'** <*VariableName*> <*InitValue*>
| [**'static'**|**'protected'**] **'method'** <*MethodName*> <*Function*>

<*Object*> ::= **'new'** <*Class*> <*Initializer*>

Fig. 3. Syntax of the *Language of Classes*

```
method =
 (\AST::
    if (canProceed method AST SUPER_SEEN)
       (\$MethodId::
           isMethodUnique (||);
           (\Body::
              let
                // build new method specification
                method_class = methodLabel (||)
                newAST = (| AST,
                            {method_class} = (| (| AST->{method_class} |),
                                                {getIdString MethodId} = Body |) |)
              in
                 (\Cont:: Cont (| newAST, {MODIFIER} = DEFAULT |))
              end))
       (error "Illegal method declaration!"))
```

Listing 4. The mini parser **method** for the *Language of Classes*

Mini parsers define small parsing automata with one explicit state. This state serves as a guard for the body of the mini parser. Consider, for example, Figure 3 that shows the syntax of the *Language of Classes*. This domain sub-language requires a mini parser for the keyword **method**, as shown in Listing 4. After receiving the decorated syntax tree AST, a data structure that contains both attribute values and parser status information,

```
<Trait> ::= 'Trait' <TraitName>
              ( ('method' <MethodName> <Function> |
                 'requires' <MethodName> <MethodName> <String>)*
               |
                'join' <Trait> 'with' <Trait>
               |
                'refine' <Trait> ( 'alias' <MethodName> <MethodName> |
                                   'exclude' <MethodName>)*  )
             'endTrait'
```

Fig. 4. Syntax of the *Language of Traits*

we first evaluate the guard for **method**. This guard (i.e., `canProceed`) assures that the actual parser state recorded in `AST` matches `SUPER_SEEN`, which is an explicit state defined for the *Language of Classes* to indicate that we have successfully processed the super type specification for the current class. As *true*-continuation of this guard, we define a function, which consumes `MethodId`, an expression tree for the method identifier, a method body `Body`, and a class continuation `Cont`. In this function we also perform the required semantic checks and construct a new decorated syntax tree. We use *sequencing* (expressions separated by `;`) to compose the individual expressions. Please note that if the check in `isMethodUnique` fails, then the current program will terminate with an error message.

However, rather than just for one state, we define guards for mini parsers to accept an open set of *permissible continuations*. The set of permissible continuations forms an equivalence relation over parser states. By means of the predicate `canProceed`, this equivalence relation allows us to equate related explicit parser states across different domain sub-languages. More precisely, when analyzing the applicability of a mini parser, `canProceed` first compares the actual parser state recorded in `AST` with the target state defined by the mini parser. If both are the same, `canProceed` returns *true* immediately. Otherwise, `canProceed` checks whether the recorded state and the target state are considered equivalent for the current domain sub-language. If this case, `canProceed` returns *true* also, as desired. But if both tests fail, `canProceed` returns *false* and the current mini parser terminates with an error message.

The key mechanism to enable the reuse or composition of an existing mini parser for the definition of a new domain sub-language is our ability to dynamically alter the set of permissible continuations, as required for the definition of the *Language of Traits*. Traits provide a simple compositional approach to factor out common behavior and to easily integrate that behavior soundly into existing classes [24]. A possible way to capture the syntax of the *Language of Traits* is shown in Figure 4. We notice that the syntax for methods resembles closely the one used in the *Language of Classes*. Indeed, we can construct the *Language of Traits* as composition of the *Language of Classes* and trait-releated language elements. In particular, we can reuse the mini parser **method** and recover our previous development effort for the definition of the *Language of Traits*. To accomplish this, we need to declare **method** a permissible continuation for the keyword **Trait** in the *Language of Traits* using the following expression

```
addPermissibleState method IN_TRAIT
```

<*Namespace*> ::= **‘Namespace’** <*NamespaceName*>
(<*Class*> | **‘Import’** <*ClassName*> (<*Member*>)* **‘endImport’**)*
‘endNamespace’

<*Class*> ::= **‘Class’** <*ClassName*> **‘super’** <*Class*> (<*Member*>)* **‘endClass’**

<*ClassName*> ::= <*PlainClassName*> **‘of’** <*NamespaceName*>

<*Object*> ::= **‘new’** <*ClassName*> <*Initializer*>

Fig. 5. Syntax of the *Language of Namespaces*

The function `addPermissibleState` and its inverse function `removePermissibleState` can be used to enlarge or shrink the set of equivalent explicit parser states for a given mini parser. By adding the explicit state `IN_TRAIT` to the set of equivalent parser states for mini parser **method**, we enable, therefore, **method** to occur within a trait specification.

We can expect that some form of additional configuration is required to make a given mini parser meet the requirements of the new context in which it is being used. The *Language of Traits* is no exception. In the case to the *Language of Traits*, this additional configuration relates to the supported trait operations. A method declaration must not occur in a context other than a *new* trait declaration (i.e., neither in a *join* nor in a *refine* specification). Furthermore, methods are always *public*. No other modifier is permitted. But these criteria can easily be satisfied, as shown in Listing 5. To guarantee that **method** only occurs within the declaration of a new trait, we add an additional guard to **method**. This guard uses the `AST` flag `TRAIT_BUILD`, which records the current trait operation. The value `DEFAULT_TRAIT` means that we are about to define a new trait, whereas `NEW_TRAIT` states that we are currently building a new trait. The exclusion of additional modifiers is standard, as we have not declared them permissible continuations for the keyword `Trait`.

```
method = (\AST::
            if (AST.{TRAIT\_BUILD} == DEFAULT\_TRAIT)
               (method (| AST, {TRAIT\_BUILD} = NEW\_TRAIT |))
               (if (AST.{TRAIT\_BUILD} == NEW_TRAIT)
                   (method AST)
                   (error "Illegal method specification!")))
```

Listing 5. The mini parser **method** for the *Language of Traits*

There is an additional and noteworthy aspect related to the composition of mini parsers and domain sub-languages. The mini parsers **Class** and **Trait** can proceed in *any* state. This is due to the fact that both mini parsers represent the *root* symbol for their corresponding domain sub-language. Root mini parsers do not depend on any explicit state. However, to guarantee composability, root mini parsers take a decorated syntax tree as argument also. But this tree is, in general, empty. When exporting the mini parsers for root symbols, we bind, therefore, `AST` to the *empty value* [14].

5 Language Composition: The *Language of Namespaces and Traits*

We construct the *Language of Namespaces and Traits* in two phases. We first define the *Language of Namespaces* as an extension of the *Language of Classes*. In the second step, we use the *Language of Namespaces* and the *Language of Traits* to build an aggregate of both domain sub-languages to form the *Language of Namespaces and Traits*.

The *Language of Namespaces* is defined in the spirit of the classbox concept [4]. A namespace defines a packaging and scoping mechanism for controlling the visibility of extensions to portions of class-based systems. In particular, (i) a namespace defines an explicitly named scope within which classes, methods, and variables are defined and (ii) namespaces support the local refinement of imported classes by adding or modifying their features without affecting the originating namespace.

The syntax of the *Language of Namespaces* is shown in Figure 5. The *Language of Namespaces* defines an extension to the *Language of Classes*. More precisely, we import the *Language of Classes* into the local scope of the specification unit of the *Language of Namespaces* to provide a default meaning for imported mini parsers and compose it with the required namespace extensions. This language composition involves three major activities:[1]

- define the keywords **Namespace**, **Import**, **of**, **endImport**, and **endNamespace**,
- add namespace-related guards to **Class**, **method**, **var**, and **endClass**, and
- define a new format for the super class specification.

Our compositional approach to language extension reduces greatly the effort required to construct the new language elements. For example, we can reuse all structure-giving elements related to member declarations defined in the *Language of Classes* for the composition of the syntactic category of *class import* in the *Language of Namespaces*. We simply have to define appropriate guards to distinguish the context within which member declarations occur. We use the `AST` flag `EXTENSION` for this purpose. `EXTENSION` is an integer to count the number of methods or variables specified in the *import* mode. In case of a class declaration, this flag is simply ignored.

```
Class = (\AST::
            if (canProceed Class AST IN_NAMESPACE)
               (\$Classname::
                   Class (| AST,
                           { EXTENSION } = (-1),
                           { QID } = buildQualifiedName (getIdString Classname) |) )
               (error "Illegal class declaration!"))
```

Listing 6. The mini parser **Class** for the *Language of Namespaces*

To adapt imported mini parsers to the new requirements, we define appropriate wrapper functions. The wrapper for **Class** is shown in Listing 6. This wrapper guarantees that a class declaration can only occur within a namespace declaration. Furthermore, it also injects the need for an occurrence of a *class name* between the keywords **Class**

[1] In this presentation, we will only highlight the steps required to reuse imported mini parsers.

and **super**. Finally, the wrapper combines the received decorated syntax tree AST with the attributes EXTENSION and QID. The flag EXTENSION is set to -1 to denote the context *class declaration* and QID is set to the fully qualified class name of the class being parsed. The reader should note that the original mini parser **Class** (denoted by the name Class inside the wrapper) is not affected by the extra AST flags, as that mini parser is blind for those additions. The GLOO semantics guarantees, however, a proper forwarding of the additional information through the chain of continuations.

The wrapper for **endClass** (see Listing 7) exhibits a slightly more complex structure. It (i) checks whether the current parser state in AST is equivalent to SUPER_SEEN and (ii) verifies that **endClass** occurs within a class declaration. If both conditions are met, then we build a new class using the attribute values recorded in AST and add that class to the current namespace as a side effect of buildClass. At the end of the wrapper, we reset AST and return a continuation to accept further class and import declarations or **endNamespace** to close the current namespace specification.

```
endClass = (\AST::
              if ((canProceed endClass AST SUPER_SEEN) &&
                  (AST.{ EXTENSION } == (-1)))
                  (buildClass AST;
                   (\Cont:: Cont (| { STATE } = IN_NAMESPACE,
                                    { NAMESPACE } = AST->{ NAMESPACE },
                                    { THIS_NS } = AST.{ THIS_NS } |) ))
                 (error "Illegal class termination declaration!"))
```

Listing 7. The mini parser **endClass** for the *Language of Namespaces*

The wrappers for **method** and **var** implement a simple bookkeeping mechanism. If a **method** or **var** occurs within a class declaration (i.e., EXTENSION == (-1)), then we just forward the AST to the original mini parser. Otherwise, we increment the EXTENSION count by one and pass the updated decorated syntax tree AST to the original mini parser, as shown, for example, in Listing 8 for the mini parser **var**.

```
var = (\AST::
         if (AST.{ EXTENSION } == (-1))
            (var AST)
            (var (| AST, { EXTENSION } = AST.{ EXTENSION } + 1 |)))
```

Listing 8. The mini parser **var** for the *Language of Namespaces*

The final step in the construction of the *Language of Namespaces* is the aggregation of the *Language of Classes* with the newly defined namespace-related elements. This aggregation can be expressed by means of GLOO's *form extension operators* [14, 16]. Informally[2], we can write

Language of Namespaces =
Language of Classes $\oplus$ namespace extensions [*Language of Classes*]

[2] A detailed technical presentation of this step has been omitted in favor of readability.

<*Namespace*> ::= **‘Namespace’** <*NamespaceName*>
(<*Class*> |
‘Import’ <*CassName*> (<*Member*>)* **‘endImport’** |
<*Trait*>)*
‘endNamespace’

<*Class*> ::= **‘Class’** <*ClassName*> **‘super’** <*Class*> (<*Member*>)* **‘endClass’**

<*ClassName*> ::= <*PlainClassName*> **‘of’** <*NamespaceName*>

<*TraitApplication*> ::= **‘apply’** <*TraitSelection*> **‘to’** <*ClassName*>

<*TraitSelection*> ::= **‘use’** <*Trait*> **‘of’** <*NamespaceName*>

<*Object*> ::= **‘new’** <*ClassName*> <*Initializer*>

Fig. 6. Syntax of the *Language of Namespaces and Traits*

to denote the aggregation of the *Language of Namespaces* in which the operator $\oplus$ stands for GLOO’s *form binding operator*. More precisely, this aggregation yields a new domain sub-language that is composed of the *Language of Classes* and the newly defined namespace extensions and in what the original *Language of Classes* serves as a local context to resolve occurrence of references to reused mini parsers.

We are now ready to define the *Language of Namespaces and Traits*, whose syntax is shown in Figure 6. This language provides a classbox-like programming model with explicit class extensions in form of traits. The particular value of this language is two-fold. The *Language of Namespaces and Traits*, on one hand, provides us with the means to experiment with different forms of class extensions in one single framework. For example, when using local refinement, class extensions are integrated into its associated class immediately through an import declaration, whereas the definition of traits allows us to reuse or even export extension to classes to refine multiple classes simultaneously.

On the other hand, the *Language of Namespaces and Traits* also presents us with a means to reason about a suitable semantics for extensions to classes and their effects on classes and namespaces. The ability to locally refine classes in a namespace and also to apply traits to the very same classes poses a particular challenge. Both operations are rather orthogonal. When should we perform local refinement and when should we apply traits to classes? Experiments with different semantics definitions of the *Language of Namespaces and Traits* have shown that the most reliable way for the support of both features is to require traits to be applied first to classes. Using this approach we obtain a semantic model for the *Language of Namespaces and Traits* that is backwards compatible with the *Language of Namespaces*. Backwards compatibility enables phased software evolution and, therefore, allows developers to organize their solutions within the framework of the new language gradually.

The required configuration effort to compose the *Language of Namespaces* with the *Language of Traits* is rather minimal. We need to perform the following steps:

- establish the *Language of Namespaces* as local context for the *Language of Traits*,
- define the keyword **apply**, **use**, and **to**,
- define trait-related wrappers for **Class**, **Import**, **endNamespace**, and
- define namespace-related wrappers for **Trait** and **endTrait**.

The first step involves the construction of a proper *chaining* of defined wrappers for mini parsers in the *Language of Namespaces and Traits*. Fortunately, the order in which different wrappers for the same mini parsers occur is unimportant. Each wrapper has to be executed eventually. In the case of the *Language of Namespaces and Traits*, we define that wrappers defined in the *Language of Traits* have to be executed before wrappers for the *Language of Namespaces* are visited. Informally, this behavior is established by the following local context

Language of Traits [*Language of Namespaces*]

that states that all occurrences of names referring to imported mini parsers are now resolved with respect to the *Language of Namespaces*.

The next steps are dedicated to the definition of appropriate wrappers for reused mini parsers. The wrappers for both **Class** and **Import** make the corresponding mini parsers *trait-aware* as shown, for example, for **Class** in Listing 9. Again, the original mini parsers are blind for this extra information. However, the extended state information is needed to obtain the required behavior in the trait-aware mini parser for **method**.

```
Class = (\AST:: Class (| AST, { TRAIT_BUILD } = NEW_TRAIT |))
```

Listing 9. The mini parser **Class** for the *Language of Namespaces and Traits*

The purpose of the wrapper for **endNamespace** in the *Language of Namespaces and Traits* is to *finalize* all classes defined within the current namespace. In other words, we first build a new namespace according to the specification in the *Language of Namespaces* and then update this namespace by composing all classes in the current namespaces with applicable traits.

```
endNamespace = (\AST:: finalizeClasses AST.{ THIS_NS }
                                    (endNamespace AST) AST->trait_applications)
```

Listing 10. The mini parser **endNamespace** for the *Lang. of Namespaces and Traits*

The last two wrappers establish **Trait** and **endTrait** as permissible continuations within a namespace declaration. Moreover, the wrapper for **Trait** adds the required bookkeeping mechanism for members, whereas the wrapper for **endTrait** registers the newly defined trait with the current namespace.

```
Trait = (\AST::
           if (canProceed Trait AST IN_NAMESPACE)
              (Trait (| AST, { EXTENSION } = (-1) |))
              (error "Illegal trait specification!"))
```

Listing 11. The mini parser **Trait** for the *Language of Namespaces and Traits*

```
endTrait = (\AST::
              let
                 qid = buildQualifiedName AST.traitName
              in
                AST->{ NAMESPACE }.add qid (endTrait AST)
              end;
              (\Cont:: Cont (| AST,
                               { STATE } = IN_NAMESPACE,
                               { NAMESPACE } = AST->{ NAMESPACE },
                               { THIS_NS } = AST.{ THIS_NS } |)))
```

Listing 12. The mini parser **endTrait** for the *Language of Namespaces and Traits*

In the final phase, we build the *Language of Namespaces and Traits* again as an aggregation that is captured by the following term

Language of Namespaces and Traits =
Language of Namespaces $\oplus$ *Language of Traits* [*Language of Namespaces*] $\oplus$
new language elements [*Language of Traits* $\oplus$ *Language of Namespaces*]

In other words, the aggregation of the *Language of Namespaces and Traits* is a new domain sub-language composed from the *Language of Namespaces*, the *Language of Traits*, and the required new language elements whose mini parsers and wrappers start evaluation in the *Language of Traits*, pass through the *Language of Namespaces*, and terminate in the *Language of Classes*.

6 Related Work

Two systems that also provide support for language extension are Camlp5, a preprocessor and pretty-printer for OCaml [7], and Polyglot, an extensible compiler front-end for the Java programming language [20]. Camlp5 allows for the admission of new elements to and even the redefinition of the whole syntax of OCaml. Language extensions in Camlp5 are defined by means of extensible *grammar entries* that encapsulate stream parsers. Each grammar entry is associated with a corresponding scanner and grammar specification. Therefore, Camlp5 also permits the admission of new lexical elements that gives Camlp5 the flavor of a compiler-compiler. However, the scope of language extensions is restricted to the defining grammar entries. As a consequence, language extension by means of composing grammar entries is not possible.

Polyglot, on the other hand, aims at constructing extensible Java compilers. More precisely, Polyglot provides an extensible compiler for the Java base language. In the Polyglot framework, language extensions are defined as source-to-source compilers [20] that translate programs using language extensions to Java source code. For this purpose, Polyglot includes an *extensible parser generator* that enables one to specify language extensions as a set of changes to the Java base language. To assign language extensions a semantics or to alter the meaning of a given language element, Polyglot provides the AST `Node` interface and optionally allows for the refinement of existing AST node classes. Polyglot's AST rewriting mechanism guarantees a proper integration of defined language extensions in the Java base language.

Our technique of defining composable mini parsers is very similar to *parser combinators* [13], which are, in general, modeled using monads [5]. Unfortunately, not every programming feature can be modeled by monads [27]. The continuation-based approach presented here appears to not exhibit this problem even though several abstractions mimic monads. For example, `Cell` exhibits a semantics very similar to the `IO` monad, whereas the keyword continuations resemble the structure of the `STATE` monad.

Flatt et al. [8] have recently presented a comprehensive approach to incorporate object-oriented abstraction into Scheme in a purely compositional fashion. The techniques used by Flatt et al. are similar to the ones presented in this work, though the Scheme extensions are defined by means of macros [12, 23].

7 Conclusion

A major challenge in programming language design is to find the right balance between the features a new programming language has to provide and the ones that would make the new language more versatile. We advocate a compositional language design that allows for the definition of domain sub-languages, which provides support for the definition of a user-centric domain vocabulary for the underlying problem domain. This user-centric domain vocabulary can greatly simplify the comprehension, design, implementation, evolution, and reuse of readily available software artifacts. In this paper, we have illustrated how GLOO supports the compositional language design approach by defining a set of class-based language extensions. We have also demonstrated that *composable mini parsers* can be used as an enabling technology to seamlessly integrate domain sub-languages into GLOO.

Table 1. Matching object models

Domain Sub-Language	***Object Model***
Language of Classes	*Imperative Class*
Language of Traits	*Extensible Imperative Class*
Language of Namespaces	*Open Imperative Class*
Language of Namespaces and Traits	*Open Extensible Imperative Class*

In general, the most crucial ingredient in defining language support for object-oriented programming is the design of a suitable object model [1, 8, 26, 24, 4, 15]. However, the underlying object model for the language extensions presented in this work is secondary by design. We can assign each defined domain sub-language a specific object model, as shown in Table 1. But we are not required to do so, as each language is designed without targeting a particular object model. The framework provided by our object-oriented domain sub-languages is based on a compositional language design. We can, therefore, supply an appropriate and desired object model later, when all application-specific requirements are known.

GLOO provides a suitable environment for *language prototyping* in a very direct way. Unlike Scheme [23] that relies on *hygienic macros* for the definition of language

extensions, we can define language extensions in GLOO naturally in terms of functions and function composition. In addition, GLOO provides a scoping mechanism that allows for a fine-grained control of the visibility of language extensions. In Scheme, for example, language extensions have to be incorporated into the system at top level, which means that language extension have global impact.

References

1. Abadi, M., Cardelli, L.: A Theory of Objects. Springer, Heidelberg (1996)
2. Achermann, F.: Forms, Agents and Channels: Defining Composition Abstraction with Style. PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics (January 2002)
3. Arnold, K., Gosling, J.: The Java Programming Language. Addison-Wesley, Reading (1996)
4. Bergel, A., Ducasse, S., Nierstrasz, O., Wuyts, R.: Classboxes: Controlling Visibility of Class Extensions. Journal of Computer Languages, Systems & Structures 31(3-4), 107–126 (2005)
5. Bird, R.: Introduction to Functional Programming using Haskell, 2nd edn. Prentice Hall, Englewood Cliffs (1998)
6. Dami, L.: A Lambda-Calculus for Dynamic Binding. Theoretical Computer Science 192, 201–231 (1998)
7. de Rauglaudre, D.: Camlp5 - Reference Manual. Institut National de Recherche en Informatique et Automatique, Rocquencourt (January 2008)
8. Flatt, M., Findler, R.B., Felleisen, M.: Scheme with Classes, Mixins, and Traits. In: Kobayashi, N. (ed.) APLAS 2006. LNCS, vol. 4279, pp. 270–289. Springer, Heidelberg (2006)
9. Ford, B.: Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In: Proceedings of POPL 2004, pp. 111–122. ACM Press, New York (2004)
10. Goldberg, A., Robson, D.: Smalltalk-80: The Language. Addison-Wesley, Reading (1989)
11. Hughes, J.: Why Functional Programming Matters. Computer Journal 32(2), 98–107 (1989)
12. Kelsey, R., Clinger, W., Rees, J. (eds.): Revised^5 Report on the Algorithmic Language Scheme. ACM SIGPLAN Notices, 33(9) (September 1998)
13. Leijen, D., Meijer, E.: Parsec: Direct Style Monadic Parser Combinators for the Real World. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht (2001)
14. Lumpe, M.: A Lambda Calculus With Forms. In: Gschwind, T., Aßmann, U., Nierstrasz, O. (eds.) SC 2005. LNCS, vol. 3628, pp. 73–88. Springer, Heidelberg (2005)
15. Lumpe, M.: GLoo: A Framework for Modeling and Reasoning About Component-Oriented Language Abstractions. In: Gorton, I., Heineman, G.T., Crnković, I., Schmidt, H.W., Stafford, J.A., Szyperski, C.A., Wallnau, K. (eds.) CBSE 2006. LNCS, vol. 4063, pp. 17–32. Springer, Heidelberg (2006)
16. Lumpe, M.: Application = Components + GLoo. Electronic Notes in Theoretical Computer Science 182, 123–138 (2007)
17. Lutz, M.: Programming Python, 3rd edn. O'Reilly (2006)
18. Microsoft Corporation. C# Version 3.0 Specification. Microsoft Corporation, Redmond, WA (May 2006)
19. Mitchell, J.C.: Concepts in Programming Languages. Cambridge University Press, Cambridge (2003)
20. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An Extensible Compiler Framework for Java. In: Hedin, G. (ed.) CC 2003 and ETAPS 2003. LNCS, vol. 2622, pp. 138–152. Springer, Heidelberg (2003)

21. Ossher, H., Harrison, W., Budinsky, F., Simmonds, I.: Subject-Oriented Programming: Supporting Decentralized Development of Objects. In: Proceedings of the 7th IBM Conference on Object-Oriented Technology (July 1994)
22. Ousterhout, J.K.: Tcl and the Tk Toolkit. Addison-Wesley, Reading (1994)
23. PLT Scheme (2006), `http://www.plt-scheme.org`
24. Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.: Traits: Composable Units of Behavior. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, pp. 248–274. Springer, Heidelberg (2003)
25. Steele, G.L.: Growing a Language. Higher-Order and Symbolic Computation 12, 221–236 (1999)
26. Ungar, D., Smith, R.B.: SELF: The Power of Simplicity. In: Proceedings OOPSLA 1987. ACM SIGPLAN Notices, vol. 22, pp. 227–242 (December 1987)
27. Wadler, P.: Monads and composable continuations. List and Symbolic Computation 7, 39–56 (1993)
28. Wall, L., Christiansen, T., Orwant, J.: Programming Perl, 3rd edn. O'Reilly & Associates (July 2000)