ABSTRACT
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, these languages are seldom used to study programming language concepts. In this paper, we explore GLoo, a small open-ended dynamic programming language, whose design philosophy aims at an unified approach in which program and language evolution result directly from the definition of ”extensible domain sub-languages”. Surprisingly, these domain sub-languages do not only provide a high-level tool to capture domain expertise, but give also rise to a powerful compositional model for language extension. To demonstrate their effectiveness, we present the underlying concepts and illustrate their application by defining a GLoo model for object-oriented and class-based language extensions. We also demonstrate that the concept of composable parsers, in combination with open contexts, plays a crucial role in achieving the resulting flexibility and expressiveness.

Categories and Subject Descriptors
D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.3.2 [Language Classifications]: Extensible languages; D.3.4 [Processors]: Interpreters

Keywords
Language Prototyping, Language Extension, Incremental Refinement, Concept Validation

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 the programming languages and their support in the target environment. A suited programming language can provide a creative medium for making programmers write good programs easily [26], whereas an ill-suited one may trigger the use of awkward formulations due to unsuitable language constructs, and lengthy formulations due to an unsuitable level of abstractions at which behavior can be expressed. Fortunately, modern mainstream general-purpose programming languages have reached a level of maturity at which they can provide a fitting support for the development of most large-scale software systems. For example, C# [25] includes now among other elements extension methods, query expressions, and expression trees, new language constructs that greatly improve our ability to build complex, data-centric applications by uniformly representing heterogeneous data sets originating from, possibly remote, data sources (e.g., a relational database system) as collection types.

Nevertheless, there remain many challenges in programming language design. Language developers have to make difficult choices in order to find the right balance between the features a language has to provide and the ones that would make the language more versatile. 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 [17]. 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 to formally define, use, and maintain the language. So, can we construct more effective means to “fine-tune” the level of abstraction provided by a programming language in an attempt 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 [20,21], a small open-ended 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 [20] and the Language of Traits [21], 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 [30] into the GLOO framework. These
sublanguages greatly benefit from the extension mechanism embedded in GLoO, which allows for the definition of arbitrary domain abstractions ranging from new data types to complex programming constructs, allowing for a user-centric view to a specific problem 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 a programming 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.

In this work, our focus is on the specification of object- and class-based language extensions using continuation-passing-style (CPS). In particular, we demonstrate how to define the Language of Classes and the Language of Classboxes as compositions of mini parsers, that is, functions that mimic the parsing process of the underlying syntactic categories. The names of these functions serve as keywords in the defined sublanguage, whereas the bodies define weakly-specified parsing automata for the corresponding associated keywords. We use the characterization of “weakly-specified” to denote the fact that the parsing automata lack explicitly specified start and end states.

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 extend 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. Secondly, the notion weakly-specified parsers enables us to compose arbitrary parsers by locally defining equivalence classes of permissible continuations resulting in a natural approach for language composition.

The rest of this paper is organized as follows: in Section 2, we briefly illustrate the main features of the GLoO programming model. In Section 3 we discuss the details of defining language extensions in GLoO. We present a model for the Language of Classes and the Language of Classboxes in Section 4 and conclude this presentation in Section 5 with a summary of our main observations and outline some future activities to make GLoO more versatile and robust.

2. THE GLOO APPROACH

The major design goal of GLoO aims at a scalable, problem-oriented programming approach providing support not only for rapid language prototyping, but also for reasoning about the newly defined language abstractions [14, 20, 21]. In order to achieve the best possible match for the underlying problem domain, the GLoO approach induces a phased combination of two processes: decomposition and abstraction. The result of the decomposition phase is a set of meta-level abstractions that capture the core data types and their relationships in the problem domain, whereas the abstraction step yields a high-level user-centric domain vocabulary for the meta level that represents a specific aspect (or view) of the underlying problem domain [20]. The resulting domain sublanguages serve as fine-grained extensions to the GLoO language and can be viewed as subjects [27] (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.

GLoo’s built-in compositional extension mechanism enables developers to enlarge the language through syntactic extensions, semantic extensions, or both. A schematic view of defining language extensions in GLoO is shown in Figure 1. 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 [17]. Users supply definitions to model the problem domain to the GLoO compiler. The compiler distills the corresponding domain abstractions, which are hosted in so-called Java support classes. The extension apparatus of GLoO translates those classes into executable semantic extensions that are loaded into the current GLoO runtime image. 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.

2.1 Core Language Elements

GLoo is a pure functional programming language that offers a declarative programming paradigm favoring an exogenous control style [4] to separate the computational from the compositional aspects of defined software abstractions. The core of GLoO is the λF-calculus [19], a variant of the λ-calculus that combines dynamic binding, explicit namespaces [3], incremental refinement, and a foreign code gateway in a single formal framework. However, GLoO is also an open-ended language. That is, rather than providing a rich set of predefined operators and statements, these have to be supplied by the application programmer by means of GLoO’s extension mechanism, which enables one to borrow new, readily-available language constructs like conditionals, loops, or assignment from outside the language.

The main programming entity in GLoO is a specification unit, defining a value (or component), which can be recombined with additional values (or components) defined in other specification units. GLoO specification units add basic data types, an import facility, term sequences, a delayed
evaluation of terms, computable binders, and a Java gateway mechanism to the core language. In essence, a GLoo specification unit declares a local scope for both to import exported definitions from other units and to define new abstractions, which all contribute to the value (or component) exported by the current specification unit.

Every GLoo specification unit defines a top-level let-block, which may be preceded by some auxiliary glue code (enclosed in \%[and \%]) being required by the gateway functions defined within the GLoo unit. In the declaration section of the let-block we can import other units or define new gateway functions, plain GLoo functions, and constants, respectively. These named abstractions are then pooled in an explicit context serving as a local lookup environment to resolve occurrences of free variables in the body of the let-block.

To illustrate the basic GLoo concepts, consider Listing 1 that depicts the use of the user-defined Language of Classboxes in GLoo. A classbox defines a packaging and scoping mechanism for controlling the visibility of extensions to portions of class-based systems [7]. In particular, a classbox (i) defines an explicitly named scope within which classes, methods, and variables are defined and (ii) supports the local refinement of imported classes by adding and/or modifying their features without affecting the originating classbox.

```
let
  load "LanguageOfClassboxes.lf"
  load "System/Services.lf"

StackCB = load "StackCB.lf"

in
  Classbox TraceStackCB
  Import Stack of StackCB
    method top (\(): print "Calling top...";
               super.top (\()\)
  endImport
endClassbox
```

Listing 1: Classbox TraceStackCB in GLoo.

The corresponding classbox abstractions are defined in the unit LanguageOfClassboxes.lf, which is loaded into the scope of TraceStackCB first. In the next line we load the unit Services.lf that defines core programming abstractions like conditionals and IO-primitives. The remaining declarations define the classbox TraceStackCB by performing the following steps: (i) add a reference to classbox StackCB, (ii) open a fresh classbox, named TraceStackCB, (iii) import and extend class Stack from StackCB, and (iv) close the classbox TraceStackCB to perform the necessary checks and yield the associated classbox structure [7] of TraceStackCB.

The program structure shown in Listing 1 is typically for defining abstractions 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., Classbox, 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.

### 2.2 Gateway Functions

Whereas the application level employs a very high-level programming paradigm, the purpose of the meta level is to incorporate new domain data types into the GLoo runtime system. To achieve the most efficient representation of required domain data types, the meta level utilizes, therefore, rather low level features. The most prominent elements in the meta level are gateway functions [21]. Gateway functions provide an easy means to directly incorporate Java code into the scope of a GLoo specification unit. Gateway code is enclosed in the symbols %\{...\}% 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 [20, 21]. For example, each gateway function takes one argument, called \texttt{arg}, which contains the Java-equivalent of the GLoo value passed to the gateway function.

A typical scenario for using gateway functions is the definition of a mutable storage cell extension as shown in Listing 2. The particular value of this 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 and/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, respectively. In the case of the \texttt{Cell} abstraction, the modeled assignment abstraction appears to be in fundamental conflict with the declarative programming model. However, the actual \texttt{Cell} object is implemented by means of a read-only literal value (i.e., an instance of class \texttt{CellValue}, a user-defined class derived from the GLoo runtime class \texttt{LiteralValue}) that hides its state-altering capability from clients of the \texttt{Cell} abstraction and does not escape the defining scope.

The unit \texttt{Cell.lf} consists of three parts: (i) the definition of the auxiliary Java code that defines the (inner) Java class \texttt{CellValue}, (ii) the local declaration of the gateway functions \texttt{makeCell, get, and set}, and (iii) the definition of the exported function \texttt{Cell}, a data type constructor for mutable storage cells. \texttt{CellValue} is a Java class derived from the GLoo type \texttt{LiteralValue}, which defines the standard semantics of expressed values like \texttt{Integers, Booleans, or Strings}. However, by adding the method \texttt{set}, we obtain a mutable value type. In the final step, we define the wrapper function \texttt{Cell} that returns a programming interface to \texttt{CellValue}. The reader should note that the body of \texttt{Cell} contains an expression of the form \texttt{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 \( \text{Cell} \), the context

\[
( ( \text{Cell=} \text{aArg}. \text{select("cell")}) \\
| ( \text{Value=} \text{aArg}. \text{select("value")}) \\
| ( \text{c.set(\text{v})})
\]

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

In GLOoo functions are specified using the general form

\[
( ( Parameter:: \text{function-body})
\]

where the optional marker \( @ \) can be used to indicate that the actual argument is a namespace that will serve as an explicit lookup environment for the free occurrences of variables in the body of the function and where \( $ \) triggers lazy parameter passing. If a function does not take an argument, then we write \( () \) in place of the formal parameter. To invoke a function, we have to apply it to an argument, even if the function does not depend on one. For example, to invoke the parameter-less function \( f \), we write \( f () \) to call function \( f \) using the empty namespace \( () \). Finally, an expression enclosed in \( [1] \) and \( [1] \) denotes a namespace \([19]\). Namespaces in GLOoo behave like extendible records and define mappings from names to values. Namespaces can be extended with new bindings, composed with other namespaces, and refined using an explicit context.

### 2.3 Computable Binders and Delayed Terms

One of the rather subtle aspects in defining extendible software abstractions is naming. The choice of names can greatly affect our ability to recombine existing 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, GLOoo provides computable binders, expressions enclosed in the symbols \( \{ \text{and} \} \), that allow for both discovery and construction of names (i.e., labels of a namespace) at runtime. The expression denoting the label has to evaluate to a string. For example, if the value of the variable \( \text{FName} \) is "decorate", then the expression \( \{ \text{FName} \} \) evaluates to the name "decorate".

```plaintext
let
fix=($::f :\text{h} (\{\text{FName}=h\}))
[[[\{h=\text{h}\} \text{Fx}=\text{f} (\{\text{FName}=\text{Fx}\} \text{Fx} \text{Arg}))]])
in
($\text{Name}=:
(\{\text{FName}=:
(\{f::fix (\{\text{FName}=\text{f}\}) (\text{getidString Name})
end
```

Listing 3: The recursive function builder Rec.

A typical application of computable binders occurs in recursive functions. By design, GLOoo 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 \( \text{Rec} \) consists of two parts: (i) the local declaration of the call-by-value fixed-point combinator \( \text{fix} \), and (ii) the exported definition of \( \text{Rec} \), which constructs the proper recursive image of its argument function \( \text{F} \). Both parts rely on computable binders. In the case of \( \text{Rec} \), the computable binder \( \{ \text{FName} \} \) enables self-application within the body of the recursive function being defined. For example, to build a recursive image of a function \( \text{F} \), we can write \( \text{Rec self F} \), which yields a function in which \( \text{self} \) is the name of \( \text{F} \) in the function’s body.

A function that enables this particular technique is lazy parameter passing or delayed term evaluation. 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. The primary
use of expression trees in GLoo is to defer the evaluation of arguments to functions until their value is actually being required [21]. Hence, when using the lazy parameter mode, instead of passing the value of the argument its corresponding expression tree (with the current calling context as initial evaluation context history) is passed to the function.

Expression trees for identifiers are of particular interest, as they enable, in combination with incremental refinement [19], a macro mechanism [29]. Incremental refinement has the flavor of dynamic binding, but must not be mistaken for dynamic binding. Incremental refinement allows for open subexpressions 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:\(\mathrm{fix} \ldots\)), 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 result of (getIdString Name), where Name contains an expression tree of an identifier and getIdString is a gateway function that takes an expression tree and returns its corresponding string representation.

3. GROWING A LANGUAGE
The definition of new language extensions does not occur in isolation. Programming idioms supported by languages like C# [25], Haskell [9], Java [5], Python [24], Perl [32], Self [31], Scheme [17, 29], Smalltalk [15], or Tcl [28] 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 and experiment with new programming concepts. The most prominent approaches are Smalltalk and Scheme. In these systems, everything is available for modification without stopping to recompile and restart. For example, Flatt et al. [12] 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 [17, 29].

In order to illustrate the GLoo way, consider Figure 2, which depicts the architecture of a set of class-based language extensions. These language extensions provide a Java-like programming model that consists of abstractions to denote classes, objects, and classboxes. These elements are defined at two different levels of abstraction. At the meta level, we define classes, objects, and dictionaries as first-class values. Furthermore and in order to obtain an imperative object model, all features utilize Cell. More precisely, Imperative Class, Open Imperative Class, Dictionary, and Namespace all encapsulate a Cell object and define an appropriate wrapper to achieve the desired corresponding behavior. In addition, we observe that Namespace refines (or inherits from) Dictionary, Open Imperative Class replaces Imperative Class (i.e., Open Imperative Class implements the same set of abstractions, but with a different signatures), and Open Imperative Class uses a Namespace object to implement the required behavior.

The programming level, on the other hand, is composed of the Language of Classes and the Language of Classboxes. The Language of Classes uses Imperative Class to define a Java-like dialect for a class-based object-oriented language extension, whereas the Language of Classboxes uses Open Imperative Class, Namespace, and the Language of Classes to define a language extension that implements a Java-like classbox module concept [6]. The intriguing aspect of the construction of the Language of Classboxes is that we reuse the Language of Classes, but replace the underlying object model with that defined by Open Imperative Class.
3.1 Mini Parsers
The programming level in GLOO is composed from a set of composable mini parsers or continuations, each capturing the keywords of a specific syntactic category. These parser continuations are first-class entities and provide an ambiguity-free specification format like Parsing Expression Grammars (PEGs) [13] to describe the underlying problem domain. Moreover, as the mini parsers are also defined in GLOO itself, we can avoid the integration of different tools and paradigms [16].

Figure 3: Syntax of the Language of Classes.

There are two reasons for the reduced state space. First, the mini parsers denoted by the continuations do not require an explicit start and end state. The parser has already moved into the second state, when the body of the continuation is evaluated. Similarly, when the control is passed to the next continuation at the end, the parser does not need to remember the final state. Secondly, the continuations define tiny state machines with implicit states. These implicit states naturally coincide with the table-based parser states, but are embedded in the structure of the continuation and, therefore, not required to be made explicit.

The required explicit states give rise to the notion of permissible continuations, which captures the equivalence classes of mini parsers that are allowed in a given state:

A continuation is permissible in an explicit state n, iff either its first required explicit state is n or it does not rely on any explicit state at all.

In other words, the ability to compose two mini parsers depends only on their agreement on the first occurrence of an explicit state, if such a state exists. For example, in the case of the Language of Classes the set of permissible continuations of super contains the mini parsers for static, protected, var, method, and endClass, as these continuations contain a test for explicit state SUPER_SEEN. The notion of permissible continuations provides us with a great flexibility for the composition of a language from reusable mini parsers, while reducing the number of explicit states to a minimal set of required checkpoints in the resulting composite parser.

3.2 Mini Parser Composition

The Language of Classboxes, as shown in Figure 4, defines an extension of the Language of Classes. More precisely, we import the Language of Classes into the local scope of the specification unit for the Language of Classboxes and compose it with the required classbox-aware elements. The language composition involves five major activities: (i) re-define of the keywords Class and endClass to accommodate the open class object model, (ii) extend super to accept a ClassId, (iii) establish var and method as permissible continuations in both class and import declaration, (iv) alter new to receive a ClassId, and (v) define the keywords Classbox, endClassbox, Import, endImport, and of.

Figure 4: Syntax of the Language of Classboxes.

The Language of Classboxes has three explicit parser states: IN_CLASS and SUPER_SEEN, reused states that are induced by the Language of Classes, and the new state IN_CLASSBOX being used to guarantee that class and import declarations can only occur within a classbox declaration opened by the keyword Classbox.

An application of the Language of Classes is shown in Listing 4, which defines the classbox StackCB that exports a simple imperative Stack class. The structure of the class definition of class Stack corresponds to the one prescribed by the Language of Classes, except for the specification of the super class Root. In the classbox model each class is addressed by two components: the class name and the classbox name the class originates from. In the Language of Classboxes we use the keyword of to combine these two components in order to build a class identifier. Furthermore, the Language of Classboxes defines a Root namespace with one class: Object. Class Object serves as the root class for all classes. Unlike Java, we are always required to specify the root class when defining a fresh class within a classbox. The reader should note that this requirement is by design and could be relaxed by altering the precondition checks for the permissible continuations of the super class declaration.
4. THE IMPLEMENTATION

4.1 Decomposition: The Meta Level

4.1.1 Imperative Class

The most crucial ingredient in defining language support for object-oriented programming is the design of a suitable object model. Fortunately, this particular area is well understood and has been studied extensively [1,7,11,12,23,30,31].

The object-oriented programming approach is naturally imperative [1]. For this reason, we wish our object model to enable methods to perform side-effects on the state of objects. Furthermore, in order to enforce a strict encapsulation of an object’s state, the default visibility of both class and instance variables is private, whereas methods have public visibility by default. Finally, we represent classes and objects as first-class values and encode them as explicit namespaces that define an interface to their corresponding members.

The structure of the imperative class builder Class is shown in Listing 5. The meta level function Class takes a record with the following elements: (i) a set of class methods, (ii) a set of public methods, (iii) a set of protected methods, (iv) a set of initialized object-based state variables, (v) a set of class-based state variables, and (vi) a direct parent-class. When supplied with appropriate values, the function Class returns a namespace exporting all class-based features (denoted by class_image) and the function instance, which yields a fresh object with an unbound self-reference, and new, that returns a proper object, that is, the fixed-point of instance. The functions instance and new are similar to Cook and Palsberg’s notion of a generator and wrapper [11] and are a key ingredient in a delegation-based inheritance model [23,31].

The function instance\(^1\) (cf. Listing 6) creates a so-called intermediate object [23], an object with an unbound self-reference. This process (i) creates an intermediate object from its direct parent-class (denoted by super), (ii) merges this object with all methods defined within a new class itself, and (iii) exposes protected and public methods accordingly. The way inherited and newly defined methods are fused in the intermediate object is specified by the underlying inheritance model [23]. For example, in a Java-like inheritance model, newly defined methods are given precedence over inherited methods, hence achieving an overriding semantics.

\[\text{Listing 5: The structure of the class builder Class.}\]

\[\text{Listing 6: The function instance.}\]

\(^1\)The operators \#, \ and \(\rightarrow\) are namespace manipulators to merge, hide, and extract namespaces.
In our approach we enforce a clear separation of state and behavior. For this reason, all methods not only possess a reference to self, but also references to state, either at instance or class level. The state references are defined as free variables within method bodies. The proper binding of these free variables is achieved by evaluating the bodies of all methods in an especially created state context State in the function image, as shown in Listing 5. We use an open context argument (i.e., @State) to provide an appropriate lookup environment for both the instance and class variables.

### Listing 7: The function new.

The wrapper function new (cf. Listing 7) creates a new intermediate object by passing the constructor arguments to instance, binds self in the intermediate object by constructing the corresponding fixed-point, and exports the public features of this newly instantiated object by removing all private and protected features in its interface. It should be noted that both, new and instance, take the collection of constructor arguments, denoted by lArgs, as a parameter in order to properly initialize the corresponding instance variables with new values on demand.

```r
new = (\lArgs::
  {let
    self=Cell (|
      prototype=instance ( ()::self ( |) ) ) lArgs
    in
      self.set ( [ ( prototype->protected )#
         ( prototype->public |) ] )
      prototype->public
  end}
)
```

### Listing 8: The Imperative Class namespace.

Finally, the exported namespace of Imperative Class contains three elements: the class builder Class, the object constructor new that takes a class and constructor arguments and yields a fresh instance, and the root class Object to terminate the super class chain (cf. Listing 8).

#### 4.1.2 Dictionary and Namespace

The Dictionary defines a mutable mapping from names to values. It supports the standard features to insert and look up bindings. Furthermore, Dictionary also provides an in-place update of bindings and the merge of two dictionaries. The underlying data structure to enable this behavior is Cell. In a given Dictionary object all services share one and the same Cell, which is being created when a new Dictionary is instantiated.

Namespace is a refinement of Dictionary and defines a nested environment from ClassId’s to values. A ClassId consists of two parts: a namespace name and a class name. We use the namespace name to select the dictionary representing the namespace and the class name to store and look up the bound class. The implementation and the exported services are similar to those defined by Dictionary, except that Namespace also provides services to add existing and create fresh dictionaries. Furthermore, rather than a Cell object, Namespace uses a Dictionary object as shared instance.

#### 4.1.3 Open Imperative Class

The concept of open classes has been proposed as a means to enable class extensions [10]. The underlying idea of open classes is that one can add or replace behavior without editing a class directly. The most prominent example of a programming language that supports open classes is AspectJ [18]. However, the flexibility embodied in open classes is not tied to the aspect-oriented programming paradigm. For example, they also provide us with a means to explore an alternative implementation strategy for classboxes [7] in which a class hierarchy can contain hooks to incorporate new behavior into the hierarchy on demand at runtime.

Imperative Class yields a set of abstractions to construct classes adhering to Java semantics. This model hard-wires the inheritance relationship between classes. However, if we substitute the hard-coded super class links in all classes of a class hierarchy with reference-by-name hooks, then we obtain an open class hierarchy that allows for an easy incorporation of additional behavior into classes on demand [22]. In order to instantiate an open class hierarchy, we need to supply a class dictionary that maps class names (i.e., ClassId’s) to instances of classes. By using different class dictionaries, we can adapt a principle class hierarchy to instances that meet the requirements of a specific environment captured, for example, by a classbox.

```r
instance=
  (\NSDict::
    {let
      lArgs::
        ()
        NSDict .lookup super .instance NSDict Self lArgs
      parent=
    )
  ...}
```

### Listing 9: The open instance function.

Surprisingly, the required changes to our object model are minimal. Consider Listing 9 sketching the instance functions of Open Imperative Class. The instance function requires now an additional argument: NSDict, a namespace dictionary that maps class names to actual visible classes. In the instantiation of the super class, we use NSDict to look up its corresponding representative. Furthermore, to enable a dynamic lookup of the super class’ super classes we pass NSDict as an additional argument to the object builder instance of the super class. The result of this instantiation process is a proper super class object, whose capabilities are determined by the contents of the namespace dictionary NSDict, as desired.

The remaining changes affect the function new and the root


Listing 10: The open new function.

class Object. Both have to accommodate the additional namespace dictionary argument (cf. Listing 10 and Listing 11). Like in the instance function, new uses the namespace dictionary NSDict to dynamically look up the class to instantiate. This is required, as the namespace NSDict may also provide a refined version of the class for which an object is currently being created. Finally, we need to change the instance function of root class Object to consume the extra namespace argument.

Listing 11: The Open Imperative Class namespace.

4.2 Abstraction: The Programming Level

4.2.1 Language of Classes

The programming level of the Language of Classes is composed from seven mini parsers, resulting in the syntactic category Class. With the exception of Class, each mini parser takes one argument, named AST, that contains a decorated abstract syntax tree. More precisely than just passing the required attribute values to each mini parser, AST also encodes the parsing history, that is, the current explicit parser state and any additional status information like the visibility of class members to current explicit parser state, defining a checkpoint for the continuation.

The structure of the programming level of the Language of Classes is shown in Listing 12. In the declaration section we define the required parser-specific values: the explicit parser states, the visibility flags, the AST entries STATE and MODIFIER, auxiliary functions to assist in the definition of the mini parsers, and error messages. To avoid the potential clash with user-defined names, the AST entries are implemented by protected, respectively. The corresponding mini parsers are implemented by modifier, a function that when supplied with the appropriate parameters yields a continuation of static and protected, respectively. Please note that the structure of super, static, and protected is similar to the one of Class.

The purpose of Class is to initialize the abstract syntax tree AST with proper values, that is, set the explicit parser state to IN_CLASS and the visibility of class members to DEFAULT. The value DEFAULT means that methods are public, whereas state variables (both instance- and class-based) are private. The default visibility can be overridden with static and protected, respectively. The corresponding mini parsers are implemented by modifier, a function that when supplied with the appropriate parameters yields a continuation of static and protected, respectively. Please note that the structure of super, static, and protected is similar to the one of Class.

Listing 13: The continuation method.

Every new class declaration starts with the keyword Class. The corresponding mini parser is shown below:

```
method. It states that a method specification can only occur inside a class declaration after the super declaration. However, the continuation itself is blind for the notion of classes. The test simply states that method is a permissible continuation in the explicit state denoted by the free variable SUPER_SEEN. In the current context, the value of SUPER_SEEN is set to 2. In other words, if we want to reuse method in another context, we have to supply a binding for the free variable SUPER_SEEN and assign it a proper value in that context.

Following the initial test in method, we define a function, which consumes a method identifier MethodId (we use a delayed term argument to obtain a macro-like mechanism), a method body Body, and a class continuation Cont. In this function we also perform the required semantic checks and build the a new decorated syntax tree. We use sequencing (expressions separated by ;) to compose the individual expressions. Please note that if the check in isDefined fails, then the current program will terminate with an error message. Otherwise, its return value is ignored.

```
var =
  (
  \@AST::
   if (AST.(STATE) == SUPER_SEEN)
   \{ if (AST.(MODIFIER) == PROTECTED)
   \{ VarId :=
     let
       VarIdS = getIdString VarId
     in
     \{ InitVal :=
         // check uniqueness first
         isDefined (\{m = class_state , nid = Id\});
         isDefined (\{m = instance_state , nid = Id\});
       let
         // build new variable specification
         lab = if (AST.(MODIFIER) == DEFAULT)
            "instance_state" "class_state"
        else
            fixed_body_fnt := %if (aArg instanceof AbstractionValue ) return aArg;
            Label | Empty =
              new Label(EmptyIdentifier.EMPTY, -1);
            return
              new AbstractionValue(|Empty, aArg|);
            %
            newVal = if (AST.(MODIFIER) == STATIC)
              InitVal (fixed_body_fnt InitVal)
            else
              %newAST =
                ((AST., lab = ((AST., \{lab\}), \{Id\} = newVal)))
              in
              \{ Cont := Cont (\{newAST, \{MODIFIER\} = DEFAULT\})\}
            end
        end
      end
    end
  end
  (error e09)
  (error e07)
```

Listing 14: The continuation var.

The mini parser for var (cf. Listing 14) is structured similarly. The main difference is, however, the local function fixed_body_fnt. The purpose of this function is to construct an initializer for state variables, if no such initializer has been defined by the user. A gateway function is used because we need to construct an abstract syntax tree forconstitute the specification of the Language of Classboxes and compose it with the classbox-aware elements. This language composition involves four activities: (i) define the new keywords Classbox, Import, of, endImport, and endClassbox, (ii) establish var and methods as permissible continuations in both class and import declarations, (iii) add classbox-related guards to Class, super, method, var, and endClass, and (iv) substitute the underlying object model with that defined by Open Imperative Class.

The overall structure of the Language of Classboxes is shown in Listing 16. As in the case of the Language of Classes, we define the required explicit states, visibility flags, and AST-specific entries to record semantic attributes. The reader should note that we have to repeat some of the definitions of the Language of Classes. This is for documentation purposes only to make the "composibility" of mini parsers more explicit in this presentation.
add classbox-related guards where required. Consider, for example, Listing 18, which illustrates the necessary modifications to method. The function method can be used in both class and import declarations. To distinguish them, we use the AST flag EXTENSION, which is an integer to count the number of methods specified in the Import mode. In case of a class declaration, this flag is simply ignored. The reader should note that the original method is not affected by the extra AST flags but guarantees a proper forwarding to its continuation. Furthermore, we do not have to specify any STATE verification checks as they are performed in the original method function, as required.

```haskell
endClassbox=
| ( \AST::
  | if (AST.\{STATE\} == IN_CB)
  | \{ registerCB \{ name=AST.\{ THIS_CB \},
  | ns=AST->\{ NAMESPACE \} \}):
  | let
  | ns = AST->\{ NAMESPACE \}
  | in
  | \{ ns,
  | getClass = \{$Classname::
  | let
  | ClassnameS=AST.\{ THIS_CB \}+"::"+
  | \{ getIDString Classname \}
  | in
  | ns.lookup \{ buildQualifiedName ClassnameS \}
  | end \}
  | end \}
  | ( error "Illegal classbox termination declaration!" )
```

Listing 19: The continuation endClassbox.

The function endClassbox closes a classbox. Within the body of this function, we first check that this operation is permissible in the current context and then build the structure of the classbox using the semantic gateway function registerCB. The result of endClassbox is a namespace that defines entries for all classes (imported or new) defined in this classbox. In addition, we also export getClass, a classbox function to look up a class for a given class name defined in the receiver classbox.

5. CONCLUSION AND FUTURE WORK

A major challenge in programming language design is to find the right balance between 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 sublanguages that provide both a model of the underlying problem domain and a user-centric domain vocabulary to ease the comprehension, design, implementation, evolution, and reuse of readily available software artifacts.

In this paper, we have illustrated a model for extending G Luo, a small open-ended dynamic programming language, with class-based language features in a compositional manner. The extension mechanism offered by G Luo allows for the definition of domain sublanguages that provide both a model of the underlying problem domain and a user-centric domain vocabulary to ease the comprehension, design, implementation, evolution, and reuse of readily available soft-

```haskell
Listing 16: Structure of the Language of Classboxes.
```

```haskell
Listing 17: The keyword Classbox.
```

To create a new classbox, we apply the keyword function Classbox to a classbox identifier and an appropriate continuation. Inside the body of Classbox, we perform the required setup and pass the initialized decorated syntax tree AST to Cont. The permissible continuations at this point are Class, Import, and endClassbox, as desired.

```haskell
Listing 18: The guarded continuation method.
```

In order to be able to properly reuse the keyword functions, we need to establish them as permissible continuations and
ware artifacts. We have also demonstrated that composable mini parsers can be used as an enabling technology to seamlessly integrate domain sublanguages into GLoo.

GLoo is a very young language and several of its features have not yet fully matured. For example, the GLoo compiler performs only a limited number of compile-time checks to verify the correctness of a specification and most checks are performed at runtime. Therefore, we plan to add support for type annotations based on contractual specifications to GLoo that will enable the compiler to perform better compile-time checks and to insert runtime assertions into the code to enable invariance checks.

6. REFERENCES