Analysis and Implementation of Hierarchical Mutually Recursive First-Class Modules

vorgelegt von
Diplom-Informatikerin
Judith Rohloff
aus Lutherstadt-Wittenberg

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Tag der wissenschaftlichen Aussprache: 2. Juli 2015

Berlin 2015

D 83

Abstract

This thesis presents a novel way to introduce mutually recursive first-class modules into a call-by-value programming language. In our approach, all modules are accessed through dot-notation without any additional syntax for mutually recursive modules. In contrast to other call-by-value languages, mutually recursive modules can be declared without the use of module composition or a self variable. This simple mechanism is achieved through a dependency analysis. This analysis computes all necessary information, which, in other approaches, must be provided by the programmer.

The transformation based denotational semantics of a call-by-value language with first-class, hierarchical and recursive modules is presented. For this purpose, a small functional language—GLang—is defined. GLang uses the notation of Groups, as proposed by Pepper and Hofstedt in [31]. Groups merge dynamic data structures with aspects of modularisation and name binding in functional programming languages. Groups are first-class values, which capture recursive definitions, lexical scoping, hierarchical structuring of programs, and dynamically typed data structures in a single construction.

This thesis clarifies what problems occur in combining nested, recursive and first-class modules and shows how to solve these problems by a novel path resolution algorithm. This path resolution algorithm is the basis for a dependency analysis which determines the evaluation order for definitions. This evaluation order is used to transform a GLang expression into an intermediate representation. Each Group is transformed into a component which contains all definitions of this Group in dependency order. This ensures that Groups are kept as entities.

For the intermediate representation, an evaluation function is provided. The transformation and the evaluation function together define the call-by-value semantics of GLang. Moreover, the compilation into a call-by-value λ-calculus with records and recursive let-expressions is defined, and we discuss different possibilities to extend GLang.

Zusammenfassung

Diese Arbeit entwickelt eine neue Methode um zyklisch abhängige, first-class Module in eine call-by-value Sprache zu integrieren. Hierbei wird auf Module mittels dot-Notation zugegriffen und für zyklisch abhängige Module wird keine zusätzliche Syntax benötigt. Im Gegensatz zu anderen call-by-value Sprachen können hier zyklisch ahängige Module ohne Modulkomposition oder "self" Variablen definiert werden. Dieser einfache Mechanismus wird mittels einer Abhängigkeitsanalyse erreicht, welche alle Informationen, die sonst der Programmierer bereitstellen müsste, berechnet.

Es wird eine transformationelle Semantik für eine call-by-value Sprache mit hierarchischen, zyklisch abhängigen und first-class Modulen entwickelt. Dafür wird GLang, eine kleine funktionale Sprache, definiert. In GLang wird das Konzept der Gruppen [31] für die Definition von Modulen genutzt. Das Konzept der Gruppen kombiniert dynamische Datenstrukturen mit Aspekten der Modularisierung und der Namensbindung in funktionalen Sprachen. Gruppen sind first-class Values und vereinigen rekursive Definitionen, lexikalisches Scoping, hierarchische Strukturierung von Programmen und dynamisch typisierte Datenstrukturen in einem Konstrukt.

Die Probleme, die bei der Kombination von hierarchischen, zyklisch abhängigen und first-class Modulen entstehen, werden zunächst verdeutlicht. Diese Arbeit stellt im Anschluß eine Lösung dieser Probleme vor, welche auf einer neuen Pfadauflösung basiert. Die auf dieser Pfadauflösung basierende Abhängigkeitsanalyse berechnet die Auswertungsreihenfolge aller Definitionen. Diese Auswertungsreihenfolge wird anschließend genutzt um einen GLang-Ausdruck in eine Zwischenrepräsentation zu transformieren. Jede Gruppe wird dabei in eine Komponente transformiert, welche alle Definitionen dieser Gruppe in Abhängigkeitsreihenfolge enthält. Dies stellt sicher, dass Gruppen als Einheit erhalten bleiben.

Für die Zwischenrepräsentation wird eine Auswertungsfunktion definiert. Die Transformation zusammen mit der Auswertungsfunktion definieren die Semantik von GLang. Zusätzlich zur Semantik wird die Übersetzung in einen λ-Kalkül mit Records und rekursiven let-Ausdrücken entwickelt. Außerdem werden verschiedene Erweiterungen für GLang diskutiert.

Contents

Parts of this work are already published:

•	Principal author	Judith Rohloff
	Authors	Judith Rohloff and Florian Lorenzen
	Title	Call-by-value semantics for mutually recursive first-class modules
	Editors	Hans-Wolfgang Loidl and Ricardo Peña
	Book title	Trends in Functional Programming
	Series	Lecture Notes in Computer Science
	Volume	7829
	Year	2013
	Publisher	Springer Berlin Heidelberg

•	Principal author	Judith Rohloff
	Authors	Florian Lorenzen and Judith Rohloff
	Title	Gruppen: Ein Ansatz zur Vereinheitlichung von Namensbindung und Modularisierung in strikten funktionalen Programmiersprachen
	Editors	Herbert Kuchen, Tim A. Majchrzak, and Markus Müller-Olm
	Book title	Tagungsband 16. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS'11)
	Series	Arbeitsberichte des Instituts für Wirtschaftsinformatik, Westfälische Wilhelms-Universität Münster
	Number	132
	Year	2011

•	Principal author	Judith Rohloff
	Authors	Judith Rohloff and Florian Lorenzen
	Title	Gruppen: Ein Ansatz zur Vereinheitlichung von Namensbindung und Modularisierung in strikten funktionalen Programmiersprachen (Langfassung)
	Series	Forschungsberichte Fakultät IV – Elektrotechnik und Informatik, TU Berlin
	Number	2011-12
	Year	2011

1 Introduction

Modularization is an essential tool for managing the complexity of large-scale software. To be effective, module systems must provide features to organize code into fine-grained hierarchies without impeding reusability. The competing nature of these goals has given rise to several approaches for designing module systems. In this work, we introduce a module system for a language with first-class, nested, and mutually recursive modules.

Recent approaches [26, 37, 27] attempt to combine mutually recursive and first-class modules in a call-by-value language, but none of them achieves full support without additional syntax. In this work, we describe a way to give a call-by-value semantics to a language with higher-order functions and nested, first-class, mutually recursive modules where all definitions are referred to by dot-notation.

Our modules are based on the grouping construct proposed by Pepper and Hofstedt in [31]. Pepper and Hofstedt introduce the concept of Groups as a unified construct for bindings and data structures. A Group is a set of definitions. As first-class values they can also be used as records. They are different from records since all definitions are allowed to refer to each other. Pepper and Hofstedt treat Groups coalgebraically and regard them as objects implicitly defined by their observers.

In contrast to this semantic approach, we here focus on an efficient implementation by defining a transformational semantics and the necessary analysis for Groups. For this purpose, we define a small functional call-by-value language GLang, which is similar to an untyped λ-calculus extended by nested first-class modules.

The main purpose of this work is to find a call-by-value evaluation for GLang. As we analyze in detail in Sec. 2.5, this is a surprisingly difficult enterprise which requires a novel path resolution that is decidable for nested mutually recursive first-class modules. Based on this path resolution, we define special dependency analysis. This dependency analysis is used to transform GLang into an intermediate representation. In this intermediate representation the ordering is explicit, but there is still a hierarchy that must be handled in the evaluation.

Modules encapsulate closely related definitions, hence it is essential that they remain an entity especially during the evaluation and the code generation. Beside a better intuition for what happens during the evaluation of a program, it is essential for separate compilation that modules are treated as entities. Furthermore, we can reduce the amount of live variables during runtime to reduce spilling, which we show in Sec. 6.2.3. For these reasons, the transformation must ensure that modules remain entities in the intermediate representation. In the case of mutually recursive modules, this is not possible. Therefore, mutually recursive modules are combined into one entity.

The semantics of a GLang program is defined by the evaluation of the transformed program. Therefore, we define an evaluation function for this intermediate language.

Furthermore, we define a compilation into an untyped λ-calculus with recursive let-expressions and dynamic records. In this step, the hierarchy is dissolved. For this language an efficient implementation, using techniques known from the literature, is possible. In addition, we discuss some extensions for a better code reuse and show how they can be integrated into the given transformation and semantics.

1.1 Motivation

For large-scale software it is essential to split the code into small components—often called modules. With nested modules [4] a fine-grained module hierarchy is possible. These hierarchies provide separate nested name-spaces. These name-space hierarchies avoid name clashes, as the same identifiers are allowed to be used in different name-spaces. References to inner definitions are written as paths using dot-notation [7]. Nested modules provide an arbitrary nesting level; so the programmer can always split a module into smaller pieces.

Nearly all modern programming languages provide a flexible module hierarchy. For example Java packages [16], ML structures [24, 21], or Scala objects [27] can be nested. These nested modules are essential to divide large software projects into small manageable pieces.

The reusability of code can be improved by abstraction over modules. With parametric modules, adaptable code that is easy to reuse at different places of the architecture is possible. The combination of module hierarchies and these parametric modules form a flexible language to manage large-scale programs. We distinguish two different ways to introduce parametric modules. Either special functions for the module language are used or modules become first-class values. The most popular languages using the first way are the members of the ML family. In SML [24, 21] only first-order functions over modules—called *functors*—are allowed. There exist multiple approaches for extending ML modules with higher-order functors—realizing higher-order modules. For further information on the literature about higher-order functors and first-class modules see Sec. 1.2.2.

First-class modules increase the expressive power of modules, as [36] demonstrates by various good examples. For example, first-class modules enable runtime reconfiguration of the architecture. In recent years, there has been much work to integrate first-class modules into a variety of languages. There are two fundamentally different possibilities to introduce first-class modules into a language. In the first approach, especially for extensions of ML-like languages, first-class modules are special constructs with special functions. That is, a module can be packed as a value with a module type, holding information about exported definitions. The second possibility is to introduce modules as term level expressions. These first-class modules [33, 36, 34] can be arguments and results of functions. This allows parametric modules to be defined by the normal function abstraction such that no special construct is needed.

To gain the full flexibility of modular abstractions, we follow the second approach and (like in the Haskell approach [33]) we unify records, data structures, and modules in

```
structure Nat = struct
   datatype t = Zero | Succ of t | If of Bool.t * t * t
   fun eval (n:t):t = case n:t of
       Zero => n
     | Succ m => Succ (eval m):t
     | If b t e => case Bool.eval b:Bool.t of
                    True => eval t
                  | False => eval e
end
structure Bool = struct
   datatype t = True | False | Null of Nat.t
   fun eval (b:t):t = case b:t of
       True => True
     | False => False
     | Null m => case Nat.eval n : Nat.t of
                  Zero => True:t
                | Succ m => False:t
end
```

Listing 1.1: Russo's example for two mutually recursive structures [37].

one construct, called a *Group*. A hierarchy of modules is created by nesting module expressions.

In some situations, the natural decomposition of a system into modules introduces cycles. These cycles cannot be expressed in a hierarchical formalism without recursion. In module systems without mutually recursive modules, these mutually dependent parts must be combined into one module, which conflicts with the idea of decomposition and modular declaration [37]. We can sum up the situation by the slogan "hierarchical modules need recursion."

Russo gives a good example for the need of mutually recursive modules in [37]. He defines the evaluation of two mutually recursive data types `Bool` and `Nat` representing Boolean and natural number expressions. The conditional `If` is a natural number expression and the zero test `Null` is a Boolean expression. He gives the code of Listing 1.1 as the code one would like to write in ML.

The problem with this code are the forward references in the structure `Nat`. Here the type `Bool.t` as well as the eval function in the structure `Bool` are forward referenced, which is forbidden in a purely hierarchical module system.

In an ML like language, it is possible to combine both functions into one `let fun`. This is fine for a small program, but with big programs this leads to one big file, which is hard to manage. With recursive modules one can put each function into a separate module.

In recent years there have been multiple approaches to implement recursive modules. The most popular approach is called *Mixins*. Originally introduced for object-oriented languages, more and more languages use Mixins as modules. We will only summarize

```
trait Nat_Eval {
  def eval_bool: Bool => Bool
  def eval_nat: Nat => Nat = {
    case Zero => Zero
    case Succ(m) => Succ(eval_nat(m))
    case If(b,t,e) => eval_bool(b) match {
      case True => eval_nat(t)
      case False => eval_nat(e)
}}}
trait Bool_Eval {
  def eval_nat: Nat => Nat
  def eval_bool: Bool => Bool = {
    case True => True
    case False => False
    case Null(m) => eval_nat(m) match {
      case Zero => True
      case Succ(m) => False
}}}
trait Eval extends Nat_Eval with Bool_Eval {...}
```

Listing 1.2: Russo's example using Mixins in Scala.

the ideas here; for further information and references see Sec. 1.2.1. Mixin modules are modules with "holes" that can be combined by a gluing construct. Listing 1.2 gives the Mixin solution for Russo's example using Scala [29]. The two mutually recursive modules are defined in two separate traits, here used as Mixins. Both Mixins have the evaluation function of the other module as a deferred component—representing the holes. The trait `Eval` is the combination of both Mixins. The gluing is done by defining `Eval` as a subtrait of `Nat_Eval` and `Bool_Eval`.

This approach has two disadvantages. First, the two functions need to have different names. As the combined module is the union of both Mixins there would be a name clash otherwise. Second, if only one language is defined, the Mixin approach is not the best option, as both Mixins will only be used once—in the combined module `Eval`. So it would be enough to define each module just once and to allow that they access each other's members.

In hierarchical modules, the reference to other modules is usually written with the dot-notation by providing the path, e. g. `Bool.t`. Apart from Mixins, there is the path recursion [25] approach, in which all the nested definitions, in particular the mutually recursive ones, are accessed by dot-notation.

Path recursion is quite similar to the way in which object-oriented languages support recursive definitions across class boundaries. There the programmer selects the components and nested components of an object by dot-notation.

With respect to syntax, we agree with Garrigue and Nakata [26] that the dot-notation is more natural from the programmer's perspective than the composition of Mixin modules.

Russo [37] and Garrigue and Nakata [26] support mutually recursive modules using dot-notation and a "self variable." Here, we give only a short overview of path recursion, more details can be found in Sec. 1.2.1. Listing 1.3 shows the solution of the motivating example given by Russo.

In this example, one hierarchically organized module `Eval` is defined. The forward references to the `Bool` structure are done by referring to the self variable `X`. `EVAL` is a recursive signature defining the two datatypes `Nat` and `Bool`.

There is no need for forward declaration as in the Mixin approach. Furthermore, this approach enables nested name-spaces also for mutually recursive modules while Mixins create one huge name-space containing all the definitions. In this thesis, we follow the idea of path recursion. In contrast to the solutions of Russo and Garrigue and Nakata, there is no need for a self variable in our approach. In Sec. 2.3, we rephrase the motivating example using our language GLang.

To summarize, the modules in GLang are hierarchical first-class modules. Module abstraction is done by using normal functions and nesting is realized by nesting module expressions. Definitions are referred to by giving the path of the definition within the module hierarchy. Recursion across module boundaries is allowed and is realized simply by mutual reference among several definitions. We will introduce GLang in Chap. 2, but to give an impression of our modules and the way the programmer can write mutually recursive modules we give an example in Listing 1.4. In this example, the mutually recursive functions `even` and `odd` are spread over two modules `E` and `O`. This is the standard example used in many publications on mutually recursive modules. In our approach, the two functions are easily defined by selecting the other function. For example, `even` uses the function `odd` in module `O`, which is done by writing the path `O.odd`. One of the motivations for the development of GLang is to allow these forward references and to establish techniques to handle them correctly.

Evaluation

This thesis deals with a number of general problems for module systems that have been cast into a small example language GLang. As already mentioned, we require a call-by-value semantics, which causes a number of difficulties that need to be handled.

Although it is a matter of taste which kind of semantics one prefers, call-by-value languages are easier to combine with parallel programming and side effects. For this reason, it is desirable to give a call-by-value semantics to GLang, since in a call-by-value strategy the evaluation of a variable can be implemented in a simple and efficient way: A variable is always bound to a value before it is used. Therefore, its evaluation is a look-up in the current evaluation environment.

In Sec. 4.2, we introduce a context analysis that ensures that all variables in a program are defined. Consequently, the look-up of a variable can never fail and the evaluation function $\mathcal{E}$ for a variable is simply defined as:

$$\mathcal{E}[\![x]\!]\Gamma = \Gamma(x)$$

```
signature EVAL =
  rec(X: sig signature Bool: sig type t end end)
  sig structure Nat: sig
         datatype t = Zero | Succ of t
                    | If of X.Bool.t * t * t
      end
      structure Bool: sig
         datatype t = True | False | Null of Nat.t
         val eval: t -> t
      end
  end

structure Eval = rec(X:EVAL)
  struct structure Nat = struct
    datatype t = datatype X.Nat.t
    fun eval (n:t):t = case n:t of
        Zero => n
      | Succ m => Succ (eval m):t
      | If b t e => case X.Bool.eval b: X.Bool.t of
                      True => eval t
                    | False => eval e
  end
  structure Bool = struct
    datatype t = datatype X.Bool.t
    fun eval (b:t):t = case b:t of
        True => True
      | False => False
      | Null m => case Nat.eval m: Nat.t of
                    Zero => True:t
                  | Succ m => False:t
  end
end
```

Listing 1.3: Russo's example with a self variable `X` [37].

```
{ E = { even = λn. IF n==0 THEN true ELSE O.odd (n-1) }
  O = { odd = λn. IF n==0 THEN false ELSE E.even (n-1) }
}
```

Listing 1.4: Short example for mutually recursive modules in GLang.

To ensure that a variable is bound to a value before it is used, the definitions must be evaluated in dependency order. There are two possibilities for detecting this order. Either the programmer has to provide it or the compiler determines it by a dependency analysis.

Dot-notation and forward references provide a simple way to write mutually recursive modules. For example, the two mutually recursive modules `E` and `O` in Listing 1.4 are defined by the forward reference `O.odd`. This simple version does not allow the programmer to give the dependency order. Hence, we use a dependency analysis to determine the evaluation order. To determine the dependencies between definitions, all paths are resolved at compile time.

Nakata and Garrigue [26] have shown that path resolution is undecidable for nested, mutually recursive, higher-order modules. There are two possibilities to avoid the undecidability: Either one can restrict the expressive power of module abstraction (see Sec. 1.2.1), or one can restrict the path resolution. As our modules are first-class values, they are higher-order modules as well. To preserve them as first-class values, we reduce the power of the path resolution by splitting paths into static and dynamic parts.

According to MacQueen [21], a module is a collection of declarations (or definitions) and the evaluation of a declaration produces an environment. Consequently, evaluating a GLang module creates a mapping of selectors to values. A selection is the application of the mapping to the given selector. Nested modules are evaluated to nested mappings. During evaluation, we want to ensure that modules are evaluated as a whole without unnecessary interleaving. All definitions within a module should be evaluated one after the other. The only exceptions are mutually recursive modules, since they must be evaluated in a mixed way.

Avoiding unnecessary interleavings is important for two reasons: First, this is important for compiler construction reasons, as definitions of a module must be kept on the stack or in registers until the module is created. After the creation these values are no longer necessary, as they are available by selection. Second, independent modules should be handled independently, since this makes debugging much simpler.

1.2 Related Work

There are various approaches for flexible modularisation in statically and dynamically typed languages. Many languages provide a hierarchical module system and there are several approaches enabling modules to be mutually recursive. The most popular module system for functional languages is the one of SML [24, 21, 22]. SML has a core and a module language. The module language consists of signatures, structures, and functors. A signature is the type of a module, often called the interface. A structure is a collection of declarations holding type, structure, value, and exception bindings. So a structure is a module. A functor is a function mapping structures to structures and provides parametric modules. In the ML family there exist many different extensions of this module system.

Most module system approaches can be found in the area of the ML family, some are done for other statically typed languages, and only a few attempts are made for dynamically typed languages.

In the following, we have a closer look at different aspects of module systems closely related to our approach, especially concerning the aspects first-class modules and mutually recursive modules.

1.2.1 Recursive Modules

Disallowing mutually recursive modules often destroys the natural structure of a program [10, 25, 37]. If it is not possible to define mutually recursive modules within a language, some decompositions are not possible. There are already some functional languages supporting mutually recursive modules, e. g. Scala or various ML dialects. Most of these approaches use Mixin modules. All approaches enabling recursive modules can be split into two groups: either the two modules are defined separately and are composed in a second step, or mutual recursion is defined by forward references.

Recursion by Composition

Most languages introducing recursive modules by composition use Mixins. Mixins are originally introduced by Bracha and Cook [6] as a generalization of inheritance for object oriented languages. Inheritance is realized by composing Mixins. Many languages use the idea of Mixins for a flexible module system.

A Mixin is a collection of named components. They are either defined or deferred. The defined components are bound in a definition and the deferred represent “holes” in the Mixin. The composition of two Mixins, often called gluing, is a combined Mixin containing all the defined components of both. The deferred components are related to the definitions of the other Mixin with the same name. The resulting Mixin can contain deferred components as well. Mutually recursive modules are possible via the composition of Mixins .

We use the notation of Hirschowitz and Leroy [18] to give an example for two mutually recursive modules. In Listing 1.5, the functions `even` and `odd` are defined in two separate Mixin modules. Each module uses a deferred function. The Mixin `EO` is the composition of both modules and provides the mutually recursive functions `even` and `odd`.

Beside the composition, there are several operations on Mixins allowing late and early binding, deleting, and renaming of components. They provide a flexible module system for incremental programming. A Mixin is named and can therefore be utilized several times.

Ancona and Zucca [2] have formulated a module calculus based on Mixins called CMS. In this approach, there are two language layers—the module language represented by CMS and a core language. In CMS, a module consists of three sets of assignments: imports, exports and local definitions. An import assignment represents a deferred definition and maps it to an internal variable. The exports represent all definitions accessible from the outside and the local definitions are only visible within this module. All internal variables (imports and local definitions) must be distinct within one module.

```
mixin Even = mix
  ?val odd: int -> bool
  let even = λx. if x=0 then true else odd (x-1)
end.

mixin Odd = mix
  ?val even: int -> bool
  let odd = λx. if x=0 then false else odd (x-1)
end.

mixin EO = Even + Odd
```

Listing 1.5: Mutually recursive modules with Mixins using the notation of [18].

```
mixin EO_sum = mix
  ?val odd: int -> bool
  let even = λx. if x=0 then true else odd (x-1)
  ?val even: int -> bool
  let odd = λx. if x=0 then false else odd (x-1)
end.
```

Listing 1.6: Sum of two mutually recursive Mixins.

Two modules can be composed by a sum operation which yields the union of all definitions of both Mixins including all deferred components.

Renaming is realized by a reduct operation with which one can define two substitutions—one for the imports and one for the exports. A freeze operation binds imports and exports and identifies the deferred and defined components by name equality.

For example, the sum of the two Mixins `Even` and `Odd` in Listing 1.5 represents a Mixin equivalent to the Mixin `EO_sum` in Listing 1.6. The Mixin `EO_freeze` in Listing 1.7 is equivalent to a freeze of `EO_sum`.

In most Mixin approaches, the composition or gluing of two Mixins is a combination of sum and freeze. Additionally, there is a selection operator to access exported components. Ancona and Zucca have shown that ML-functors can be encoded in CMS. As a consequence, Mixins provide a powerful tool for mutually recursive parameterized

```
mixin EO_freeze = mix
  let even = λx. if x=0 then true else odd (x-1)
  let odd = λx. if x=0 then false else odd (x-1)
end.
```

Listing 1.7: Freeze of `EO_sum` from Listing 1.6.

modules.

Hirschowitz and Leroy [18] use CMS to introduce Mixin modules into the ML language. In contrast to the original CMS, they use CMS in a call-by-value setting. They use a dependency analysis to detect wrong cycles, where at least one definition is not a λ-abstraction. They store the dependency graph as type information of a module to check the program at compile time. We discuss this analysis in more detail in Sec. 1.2.3. Furthermore, they compile CMS into a λ-calculus with recursive let-bindings and records (see Sec. 1.2.4).

In Scala [29], traits are frequently used as Mixins [28]. A trait is a special abstract class and behaves similar to a Java interface. It can have abstract members representing deferred components. A new trait can extend another trait.

```
trait Even {
  def odd: Int => Boolean
  def even: Int => Boolean = x => if (x==0) true else odd(x-1)
  val test = even(4)
}

trait Odd {
  def even: Int => Boolean
  def odd: Int => Boolean = x => if (x==0) false else even(x-1)
}

trait EO extends Even with Odd {
  val test1 = even(5)
  val test2 = odd(5)
}
```

Listing 1.8: Scala Mixins example.

Listing 1.8 gives the example of even and odd using Scala traits. The two Mixins `Even` and `Odd` are defined as traits. In the former, the definition of `odd` is deferred and in the latter `even` is deferred. The composition `EO` of these two traits is done by the `with` construct. Such a composition is the combination of sum and freeze, as the deferred definitions are related to the defined definitions by name equality. The ordering of the traits can be important, as the latter may override definitions of the former. A concrete class of a trait is obtained by extending the trait and defining all deferred components. In the Scala world, traits are quite popular and they established a special programming pattern: the “cake pattern.”

Flatt and Felleisen introduce the language “units” in [12]. Units is a language-independent module language for dynamically and statically typed modules. Originally introduced for Scheme-like languages, it can also be used for the ML family. Units is a special module language allowing separate compilation, parameterized mutually recursive, and hierarchical modules. In [30], Owens and Flatt extend the approach of units. They introduce a typed language with modules and units. Modules are only allowed at the top-level and form entities for separate compilation. A module contains a sequence of

```
E_unit : unitT[odd: int -> bool, even: int -> bool] =
    unit import [odd: int -> bool]
         export [even: int -> bool]
      even = λn. ... odd (n-1)
O_unit : unitT[odd: int -> bool, even: int -> bool] =
    unit import [even: int -> bool]
         export [odd: int -> bool]
      odd = λn. ... even (n-1)

EO_unit : unitT[odd: int -> bool, even: int -> bool] =
   compound import []
            export [odd: int -> bool, even: int -> bool]
      link (E_unit): import [odd: int -> bool]
                     export [even: int -> bool]
           (O_unit): import [odd: int -> bool]
                     export [even: int -> bool]
      where (even <- even) (odd <- odd)
```

Listing 1.9: Example for recursive modules using units.

definitions given in evaluation order. They are not allowed to be mutually recursive. Dependencies to other modules are given explicitly in the body of the module by selection from these modules. Units are first-class modules and are quite similar to Mixins. We focus here on mutual recursion and describe of the first-class behavior in Sec. 1.2.2. In the definition of a unit all imports and exports are given explicitly providing compile-time information in the form of a signature. It is possible to define mutually recursive units by a compound similar to glue for Mixins. A compound can link two or more units forming a new unit. This new unit contains all definitions of the combined units in the given order, and in this order the definitions are evaluated. These definitions are allowed to be mutually recursive. In the case of a forward reference, it is checked whether the expression is a value or not. In the case of a value, the value is used, otherwise an error occurs. This is well suited for mutually recursive functions spread over module boundaries, as λ-abstractions are values. Some of those invalid forward references can be prevented if the inner definitions are re-sorted. The Racket language [13] provides a module system with units. In this implementation, the programmer has the possibility to give an order for all composed definitions, while these definitions are sorted automatically in our approach.

Listing 1.9 contains an example for two mutually recursive units defining the functions `even` and `odd`. In the definitions of the units, the deferred function is imported. The compound combines the two units into the unit `EO_unit`. In the linking section the two units are given together with the needed import and export declarations. The freezing is done via the `where` construct by defining the relations between imports and exports of the linked units.

```
structure EvenOdd = rec(X:EO) struct
  structure E = struct
    fun even (x:int):bool = if x=0 then true else X.O.odd (x-1)
  end
  structure O = struct
    fun odd (x:int):bool = if x=0 then false else E.even (x-1)
    val testO = odd 3
  end
end
```

Listing 1.10: Example for Russo's approach.

Path Recursion

A path is a qualified identifier and is used, e. g. in ML, to refer to nested modules. In contrast to the Mixins approach, in the path recursion [26] approach mutual recursion is not enabled by merging modules but by statically referring to definitions using the path of a definition.

In [37], Russo extends Mini-SML by recursive modules. As in all ML-languages, the evaluation order must be given by the programmer. All forward references are described via a "self variable" introduced at the beginning of the recursive structure. The self variable is only a trick for the type checker; at runtime the behavior is the same as without it. During evaluation, those forward references are assumed to be undefined. After evaluation, they are updated in the heap. In this approach, an exception is raised if the forward reference is used. Therefore, all occurrences of forward references must be under abstractions. The example given in Listing 1.10 illustrates the runtime behavior.

The structure `EvenOdd` contains the two mutually recursive structure `E` and `O`. As `even` is an abstraction, it will be evaluated to a closure. The same holds for `odd`. After the evaluation of `odd` both functions are bound to a closure-value and the call of `odd 3` can easily be evaluated. We change the code of this example in Listing 1.11 to get a program raising a runtime error.

The problem occurs in the evaluation of the right-hand side of `testE`, as `testE` is defined before `O`. The call of `even` leads to a call of `X.O.odd` in the closure-value bound to `even`. This is a use of a forward reference and raises an exception. This problem of runtime errors does not exist in our approach.

In [25], Garrigue and Nakata study applicative modules with polymorphic functors and recursion based on paths. Similar to Russo's approach, they introduce a self variable for structures. This variable is bound in all definitions of this structure. Using the dot-notation and this variable, it is possible to refer to any nested definition within this structure, regardless of the ordering. They define a type system for their modules and use a non-strict dynamic semantics.

In [26], Garrigue and Nakata formalize their approach using a small calculus for their module system of nested recursive modules. Figure 1.12 gives the syntax of this calculus.

```
structure EvenOdd = rec(X:EO) struct
  structure E = struct
    fun even (x:int):bool = if x=0 then true else X.O.odd (x-1)
    val testE = even 3
  end
  structure O = struct
    fun odd (x:int):bool = if x=0 then false else E.even (x-1)
    val testO = odd 3
  end
end
```

Listing 1.11: Problematic example for Russo's approach.

$$
\begin{array}{lcl}
e & ::= & \{\ (\mathtt{m} = e)^*\ \} \mid \lambda\,\mathtt{x}\ .\ e \mid p \\
p & ::= & \varepsilon \mid \mathtt{x} \mid p\ .\ \mathtt{m} \mid p\ p \\
P & ::= & \{\ (\mathtt{m} = e)^*\ \}
\end{array}
$$

Figure 1.12: Syntax of calculus for path recursion.

An expression e is either a module, an abstraction (functor), or a path. A path p is either the top-level module ε, a variable `x`, a selection, or an application of two paths. A program P is the top-level module. Garrigue and Nakata distinguish between selectors `m` and variables `x`. By dot-notation, a path can refer to a field at any level of nesting independently of the ordering. This enables recursive modules.

A path abbreviation is a definition of the form `m = p` representing a reference to another definition. They define path resolution as a rewriting of paths. The goal of this rewriting is to detect path equality. Therefore, a path pointing to a path abbreviation is replaced by this abbreviation and path applications are evaluated. They prove that path resolution is not decidable even for nested modules with first-order functors. They define different subsystems, for which path resolution is decidable, by reducing the power of expression of functions. In our approach, we need path resolution to detect dependencies between definitions. Instead of reducing the expressive power of the functions, we reduce the strength of the path resolution. For example, an application of paths is never reduced in GLang. As a consequence, our path resolution is decidable.

1.2.2 Parameterized Modules

A parameterized module—also called *generic* module—is a module with free variables. Similar to λ-abstractions, a parameterized module can be applied to some arguments creating a module instance. Parameterized modules support a better re-use of code. In SML parameterized modules are called *functors*. Functors in SML are first-order functions: They can only be declared at the top-level and cannot be applied to other functors; they are also not allowed to be recursive.

While first-order functors abstract over the name of another module, a higher-order functor [23, 17] abstracts over the name of another functor. Furthermore, it can yield a functor as return value.

Higher Order Functors

The SML of New Jersey compiler provides higher-order modules. Their semantics are given by MacQueen and Tofte in [23]. The current implementation of higher-order modules is described by Kuan and MacQueen in [20]. The ML context checker requires a static representation for structures. As nested structures provide different name-spaces, names are possibly ambiguous. To capture this problem, an entity variable is introduced to refer to type constructors, structures, and functors. A structure is represented as a map from entity variables to entities. In this approach, a structure is seen as a tree whose leaves are type constructors or functors and subtrees are substructures. The entity variables are represented by annotated edges. These trees are quite similar to the idea of our Group hierarchies (see Sec. 3.3). Furthermore, a compile time functor representation is needed, as functor application needs to be analyzed for type checking.

First-Class Modules

Approaches for first-class modules exist for multiple languages including Haskell [33], the ML family [36, 14], Scala [28], or Scheme [12].

Russo extends the ML module system with first-class modules in [36]. There is still a special module language, but structures can be packed as first-class values. Russo extends the core language with a special type `<S>` for packed structures and two expressions for packing and unpacking a structure. The type `<S>` is the type for a packed structure matching the signature `S`. The call-by-value semantics for `pack s as S` evaluates the structure `s` and packs it as a value of type `<S>`. The expression `open e as X:S in e'` is evaluated by evaluating `e` to a structure matching the signature `S`. The value is bound to the variable `X` in the expression `e'`.

In [33] a module system for Haskell is proposed where records and modules are joined into one concept. These record-modules are, as in our approach, first-class values. Inner modules are referred to by selection using dot-notation. As Haskell is statically typed, these records are typed as well. The type of every record must be given and provides information about all defined names of the module. A record consists of a set of possibly mutually recursive bindings. Valid selections are detectable by type inference for every expression. As Haskell has a non-strict semantics, it is not necessary to calculate the dependency order.

In Scala, objects are used as modules. As they are first-class values, modules are first-class values, too. In [27], a calculus for objects and classes (*vObj*) is proposed. In *vObj*, objects are first-class values. They can be abstracted using class templates and can be composed via Mixin composition. *vObj* can be seen as a base calculus for Scala since a subset of Scala maps to this calculus. Furthermore, the calculus can express most of the

ML module system. Odersky et al. show that structures can be represented by objects, functors by classes, using abstract class members and abstract types, and signatures by object types. The type system of *vObj* is undecidable and the evaluation left unspecified.

As already mentioned, the extended units approach [30] introduces units as first-class values. Consequently, units can be used to define nested modules and λ-abstractions can be used as functors. Inner definitions of units can only be accessed, when the unit is evaluated. The evaluation of a unit is enforced via the `invoke` expression. Only units without imports can be evaluated. To evaluate a unit, all inner definitions are evaluated in the given order. After the evaluation, all inner definitions are accessible through the given module identifier. As an example, we turn back to Listing 1.9. The functions `even` and `odd` in the unit `EO_unit` are not yet accessible. For example, one cannot write `EO_unit.even`. In the following line, the unit is evaluated and binds the result to `EO`; so afterwards the function `even` can be called through the path `EO.even`.

```
invoke EO_unit as EO:[odd: int -> bool, even: int -> bool]
```

In his PhD thesis [34], Reinke develops a functional call-by-value dynamic language which contains, among other contributions, frames. In particular, they are dynamically typed first-class values. Frames may contain a set of definitions and these definitions may be mutually recursive. Recursion over frame boundaries is not allowed and all names of other frames must be explicitly imported. These imports give the evaluation order for frames. Mutually recursive modules can only be implemented via functions where the relevant frames are parameters.

1.2.3 Dependency Analysis

Dependency analyses are a wide spread technique for optimization. It is only rarely used to sort program parts to enable call-by-value evaluation. In most call-by-value languages the programmer has to give the definitions in dependency order.

A notable exception is Opal [32]. In Opal, a module (called a structure) is divided into a signature and an implementation. The signature provides the interface of a structure and the implementation the definitions. An implementation is considered as a set of definitions, defining data structures, values and functions. The order of the declaration is not important [11]. According to the language report, all value definitions containing free variables are treated as functions, and are evaluated each time the defined operation is applied. The implementation of Opal uses an optimization based on a dependency analysis. All value definitions, i. e., definitions that are not of a function type, are ordered according to the analysis at compile time. Value definitions are evaluated during structure initialization. Programs containing mutually recursive value definitions are rejected at compile time.

In [5], Blume presents a dependency analysis for ML. This analysis is used in the compilation manager of SML of New Jersey. The analysis detects dependencies automatically. It takes a set of source files and produces a linear arrangement of these files, such that all variables are defined at the time when they are used. Blume proves that finding such an order is NP-complete for SML, as identifiers need not be unique at the top-level. The presented dependency analysis can neither handle mutually recursive source files, nor first-class structures.

As previously mentioned, Hirschowitz and Leroy use dependency information to detect forbidden cycles in [18]. Similar to our approach, all cycles may only consist of abstractions. Hirschowitz and Leroy do not define a dependency analysis but consider the dependency graph of a Mixin given in the signature type of the Mixin. The nodes of this graph are exported and imported components and the labeled edges represent the dependencies. A label can either be 0 or 1, where $X \xrightarrow{0} Y$ means that the term defining Y refers to the definition X. The label 0 indicates an edge, where at least one of the definitions is not an abstraction, while the label 1 represents a dependency between two abstractions. At compile time, the graphs of two composed Mixins are composed as well and this composed graph is checked for forbidden cycles—i. e., cycles, where at least one edge is labeled with 0. The dependency information must be given by the programmer.

1.2.4 Semantics by Transformation

The idea of defining language semantics and especially the semantics of module systems by translating into an intermediate language is well-established. We give two examples using this technique for recursive modules.

Rossberg and Dreyer define the semantics for their ML like language with Mixin modules (MixML) [35] by giving the operational semantics for an internal language combined with a transformation from MixML into this internal language. The internal language is an extension of System F_ω with records, single assignment references enabling recursive linking, and abstract types. They prove the type soundness of MixML by proving the soundness of the internal language together with type preservation of the translation.

Hirschowitz and Leroy compile the adapted CMS calculus [18] into a call-by-value λ-calculus with records and recursive let-bindings. Each Mixin is compiled into a record containing all exported components. All fields are abstracted over the input components on which they depend. To create this abstraction the dependency graph information given in the signature of the Mixin is used. The recursive let-construct is an extension of the common call-by-value construct, as it allows not only abstractions but function application on the right-hand side. Hirschowitz and Leroy show the type soundness of their CMS language in the same way as Rossberg and Dreyer.

2 Module Language

We define an example language GLang to demonstrate how to integrate nested, first-class, mutually recursive modules into a call-by-value language. The syntax of GLang is shown in Fig. 2.1. The usual bracketing and associativity rules of λ-calculus are employed. We assume an unbounded set x of variable names as well as primitive functions and notations for integers n, booleans b, and strings s.

GLang is a λ-calculus extended by hierarchical first-class modules, selection, and matching. As modules are first-class values, they can also be used as dynamic records. Following Blume, who also used this term for his hierarchical modules [4], we call these modules “Groups.”[1] Intuitively, a Group is a set of definitions. Notationally, definitions are enclosed by curly braces to emphasize that their order of appearance is irrelevant, similar to the enumeration of a set’s members. All definitions within a Group must have a unique identifier within this Group. Inner definitions of the same identifier shadow outer definitions.

A selection refers to a definition within a Group by the given identifier. There are two kinds of dynamic Group discriminators, exact matches and selector matches. The exact match returns true, iff the given Group defines exactly the set of selectors. The selector match checks whether the Group defines the given selector.

Let and recursive let-expressions can be easily expressed by Groups. A recursive let-expression of the shape

$$\textbf{letrec}\ \mathtt{x}_1\ \mathtt{=}\ \mathtt{e}_1\ \mathtt{...}\ \mathtt{x}_n\ \mathtt{=}\ \mathtt{e}_n\ \textbf{in}\ \mathtt{e}$$

[1]The term “group” has already been used much earlier in the COBOL language as the name for dynamic records.

$$\begin{array}{lcll}
e & ::= & \lambda\ x\ .\ e & \text{— Abstraction} \\
 & \mid & e\ e & \text{— Application} \\
 & \mid & \textbf{IF}\ e\ \textbf{THEN}\ e\ \textbf{ELSE}\ e & \text{— Conditionals} \\
 & \mid & n \mid b \mid s & \text{— Integer, Booleans, Strings} \\
 & \mid & x & \text{— Variable} \\
 & \mid & e\ .\ x & \text{— Selection} \\
 & \mid & e\ \textbf{DEFINES}\ \{\ x^*\ \} & \text{— Exact Match} \\
 & \mid & e\ \textbf{CONTAINS}\ x & \text{— Selector Match} \\
 & \mid & \{\ d^*\ \} & \text{— Group} \\
d & ::= & x\ \mathtt{=}\ e & \text{— Definition}
\end{array}$$

Figure 2.1: Syntax of GLang language.

```
List = {
  Nil = {}
  Cons = λx.λxs. { hd=x  tl=xs }
  head = λxs. xs.hd
  tail = λxs. xs.tl
  isNil = λxs . xs DEFINES {}
  length = λxs. IF isNil xs THEN 0 ELSE 1 + (length xs.tl)
  enum = λn. { enum = λi.
                 IF i==n THEN Nil ELSE Cons i (enum (i+1))
             }.enum 0
}
```

Listing 2.2: Lists in GLang.

is equivalent to this GLang expression:

`{ x`$_1$` = e`$_1$` ... x`$_n$` = e`$_n$`  _in  = e }._in`

We illustrate the language by some examples in the following sections. These examples shall illustrate that—in spite of the minimalistic design—GLang can express a wide variety of tasks. But certainly it would need additional syntactic sugar to make it into a “nice” programming language. However, GLang is not designed for practical programming, but for the study of certain concepts.

2.1 Introductory Example: Lists

Abstractions, applications and conditionals have their usual meaning. We focus on Groups and selections.

Listing 2.2 shows the well-known example of lists as a functional data structure.

The Group `List`[2] contains seven definitions `Nil`, `Cons`, `head`, `tail` , `isNil`, `length`, and `enum`. In this example the Group represents a module encapsulating definitions.

Both constructors, `Nil` and `Cons`, implement the list elements with Groups as well—`Nil` is the empty Group and `Cons` returns a Group containing the head `hd` and the rest `tl` of the list. In this case, Groups are used as data structures, which can be constructed and manipulated at runtime.

The functions `head` and `tail` select the first element and the rest of a list via the selection operator “`.`”. In this manner, these functions abstract the selectors `.hd` and `.tl`.

The discriminator `isNil` uses the match `DEFINES` and checks if the argument `xs` is a Group containing no definitions. The recursive function `length` uses this discriminator.

The function `enum` forms a list of numbers between `0` and `n`. This definition illustrates several aspects of our language design:

[2] In this thesis, we use the following convention for the identifier of a definition: The identifier of modules, functors, and data constructors always start with a capital letter. The identifier of functions, selectors, discriminators, constants, etc. start with a small letter.

```
Term = {
  Var = λx. { eval = λΓ. Γ.lookup x }
  Abs = λx.λt. { eval = λΓ. { var=x  body=t  env=Γ } }
  App = λt1.λt2. { eval = λΓ. apply (t1.eval Γ) (t2.eval Γ)
                   apply = λf.λa. f.body.eval (f.env.add f.var a) }
}
```

Listing 2.3: An eval-apply-interpreter in a functional-object-oriented style.

- All variables of the outer Group can be used on the right-hand side of a definition. In this example, these are `Nil` and `Cons`.
- Within the expression, an anonymous Group containing the definition `enum` is defined. We call a Group anonymous if it is not the top-level node of the syntax tree of a right-hand side of a definition.
- In the definition of `enum` of the anonymous Group, the variable `enum` is used. Because of the lexical scoping, the innermost definition is chosen. Thus, it is `enum` of the anonymous Group and not `List.enum`.
- On the right-hand side of `List.enum` the selector `.enum` is called. The anonymous Group and its selection are used as a local recursive "let-in" or "where" clause.

2.2 Functional-Object-Oriented Programming

Our second example shows how we can use Groups to program in an object-oriented manner. Listing 2.3 contains an eval-apply-interpreter with environments for the untyped λ-calculus. A term is either a variable `Var`, an abstraction `Abs` or an application `App`. These term constructors are quite similar to the "objects-as-closures" implementation [1], but here they are evaluated to Groups instead of closures. Those Groups contain an evaluation function `eval`, which uses an environment to compute the value of the term. In doing so, the evaluation functions of the subterms may also be used.

During the evaluation of a variable, its value is determined from the environment `Γ`. To evaluate an abstraction, a closure value (i. e. a Group) with the abstracted variable `var`, the function body `body`, and the current environment `env` must be built.

The implementation of the environment is shown in Listing 2.4. An environment is a list of records `{var=x val=v}` using the list implementation in Listing 2.2. An environment has the functions `add` to add a new binding and `lookup` to determine the value of a variable. The constructor `Env` creates an environment containing all bindings of `bdgs`.

In contrast to terms new environments must be created during evaluation. With `add` a new environment is created using the recursive function `Env`. The argument for the environment creation is a list which contains all previous bindings as well as the new binding `Bdg x v`.

```
Env = λbdgs. {
  Bdg = λx.λv. { var=x  val=v }
  add = λx.λv. Env (List.Cons (Bdg x v) bdgs)
  lookup = λx. { find = λbdgs. {
                           bdg = List.head bdgs
                           val = IF bdg.var==x
                                 THEN bdg.val
                                 ELSE find (List.tail bdgs)
                         }.val
               }.find bdgs
}
```

Listing 2.4: Environment for the eval-apply-interpreter.

As demonstrated by this example:

- Objects can be realized as Groups. Methods are definitions within a Group.
- Constructors of object oriented programming languages can be seen as functions returning Groups.
- Variables, abstracted by `λ` within a constructor play the role of fields. It is also possible to store them in definitions, making them accessible from outside by selection.
- A method call corresponds to the selection of a function.
- The binding rules for Groups enable methods to have access to the methods and fields of its enclosing object. A special reference like `this` in Java or `self` in Smalltalk does not exist.

2.3 Mutually Recursive Modules

The list example has already shown that definitions may be recursive. Mutually recursive definitions are allowed as well, even across Group borders. An example for mutually recursive Groups is given in Listing 2.5. As such situations pose the most complex challenge for the transformation, this program will be used as a running example throughout this work.

In this example, the two functions `E.even` and `O.odd` are defined. Jointly, they decide whether a given number is even or not. `O.odd` uses `E.even` and vice versa. Moreover these two functions are used by `E.is2even` and `O.is2odd`.

Another common example for mutually recursive modules are trees using a forest for handling the subtrees. A possible implementation of these trees is given in Listing 2.6. A tree node consists of a label and a subtree. A node is created via the constructor `Node`.

```
{ E = { even = λn. IF n==0 THEN true ELSE O.odd (n-1)
        is2even = even val
        val = 2 }
  O = { odd = λn. IF n==0 THEN false ELSE E.even (n-1)
        is2odd = odd 2 }
}
```

Listing 2.5: Mutually recursive Groups.

```
{ List = ...
  Tree = { Node = λa.λf. { label = a  subforest = f }
           Funs = { flatten = λtree. List.Cons tree.label
                                     (Forest.flatten tree.subforest)
                  }
         }
  Forest = { Add = List.Cons
             Empty = List.Nil
             flatten = λforest. List.map Tree.Funs.flatten forest
           }
}
```

Listing 2.6: Mutual recursion for trees.

A forest is either empty, or it is a list of trees. The two forest constructors just call the corresponding list constructors. Furthermore, we give a mutually recursive function `flatten`—collecting all labels of this tree in preorder. In the `Tree` Group, the function is defined in a nested Group `Funs`. Here the mutual recursion is not only crossing one module layer but two and the two functions are not in the same depth.

As another example, we present our solution for the motivating example for mutually recursive modules from Sec. 1.1. In this example, the evaluation of two mutually recursive data types is defined. Each function is defined in its own module. In our approach one can write the definitions of these two modules directly. Listing 2.7 contains the example using Groups.

As GLang is an untyped language, we have to use techniques from dynamic languages to encode data structures. The different variants of a sum type are distinct from each other by a tagging definition. The data types of `Nat` and `Bool` are encapsulated in a Group `T` each. Within `T`, the necessary constructors and discriminators are defined. The two `eval` functions can access each other by the corresponding path. As pattern matching is not supported by GLang, the discriminators are used instead. The check whether the evaluation of a conditional is `True` or `False` uses a forward reference to `T` in the Group `Bool`.

```
{ Eval = {
    Nat = { T = { Zero = { tag = 0 }
                  Succ = λm. { tag = 1  val = m }
                  If = λb.λt.λe.
                         { tag = 2  cond = b  then = t  else = e }
                  zero? = λt. t.tag==0
                  succ? = λt. t.tag==1
                  if? = λt. t.tag==2
                }
            eval = λe. IF T.zero? e THEN T.Zero
                       ELSE IF T.succ? e THEN T.Succ (eval e.val)
                       ELSE IF Bool.T.true? (Bool.eval e.cond)
                                                  THEN eval e.then
                       ELSE eval e.else
          }
    Bool = { T = { True = { tag = 0 }
                   False = { tag = 1 }
                   Null = λm. { tag = 2  val = m }
                   true? = λt. t.tag==0
                   false? = λt. t.tag==1
                   null? = λt. t.tag==2
                 }
             eval = λe. IF T.true? e THEN T.True
                        ELSE IF T.false? e THEN T.False
                        ELSE IF Nat.T.zero? (Nat.eval e.val) THEN T.True
                        ELSE T.False
           }
  }
}
```

Listing 2.7: Russo's example using Groups.

```
{ TreeSearch =
    λc. { findNext =
            λgC.λp. IF c.isEmpty gC THEN {}
                    ELSE { node = c.next gC
                           gCe = c.addAll node.subforest (c.rest gC)
                           res = IF p node
                                 THEN { n = node.label gC = gCe }
                                 ELSE (findNext gCe p) }.res
          findOne = λp.λt. (findNext (c.add t c.empty) p).n
          findAll = λp.λt. ...
        }

  Queue = { empty = List.Nil
            isEmpty = List.isNil
            add = λx.λq. IF isEmpty q THEN List.Cons x empty
                         ELSE List.Cons (next q) (add x (rest q))
            next = λq. q.hd
            rest = λq. q.tl
            addAll = λl.λq. IF isEmpty q THEN l
                            ELSE add (next q) (addAll (rest q) l)
          }

  DepthFirstSearch = TreeSearch SearchList
  BreadthFirstSearch = TreeSearch Queue
}
```

Listing 2.8: Tree search using first-class modules.

2.4 Parameterized Modules

In GLang, λ-abstraction is used for parameterized modules as Groups are first-class values. In the example in Listing 2.8, we define a flexible tree search algorithm parameterized by the container `c`.

Depending on the given container, the function `TreeSearch` provides different search algorithms. For example, in the case of a list, it is a depth-first, and in the case of a queue, it is a breadth-first algorithm. The containers must implement the interface of the algorithm. The search algorithm needs the check whether the container is empty (`isEmpty`) or not, must add a list of elements to the container (`addAll`) and needs to access the next element (`next`) and the rest of the container (`rest`). We could check if a given container matches this interface by the matching `CONTAINS`, but we omit this in the example. `TreeSearch` applied to a container returns a Group with different search functions to find the first or all elements in the tree matching a predicate. The Group within the function `findNext` is used to define local variables similar to a let-expression.

```
SearchList = { empty = List.Nil
               add = List.Cons
               next = List.head
               rest = List.tail
               isEmpty = List.isNil
               addAll = λl.λxs. IF isEmpty l THEN xs
                                 ELSE add (next l) (addAll (rest l) xs)
             }
```

Listing 2.9: New list Group without using any extensions.

```
{ ...
  SearchList = EXTEND ((List RENAME
                             (Nil AS empty) (isNil AS isEmpty)
                             (head AS next) (tail AS rest)
                             (Cons AS add))
                           ONLY empty isEmpty next rest add)
               BY { addAll =
                      λl.λxs. IF isEmpty l THEN xs
                               ELSE add (next l) (addAll (rest l) xs)
                  }
  ...
}
```

Listing 2.10: List Group modified for the search interface.

The given queue implementation uses a list and adds elements at the end of this list.[3] As the interface for the search algorithm is different from the List Group, we need a new Group `SearchList`. There are two possibilities to define the list container fulfilling the search interface. Either we define this Group by giving all the needed selectors by referencing to the Group `List` or we use the extensions to manipulate Groups as proposed by Pepper and Hofstedt in [31].

In Listing 2.9, we define `SearchList` without the extensions. Here all selectors used in the interface are defined by selecting the corresponding definition from `List`. The new selector `addAll` is defined through the new definitions.

The extensions proposed by Pepper and Hofstedt are useful for better code reusability. These extensions can be used to rename definitions and to add new definitions by extending the original Group. In Listing 2.10, the Group `SearchList` is defined using these extensions.

First of all, the definitions of the `List` Group are renamed. As we only need some of the definitions, we can leave out all other definitions by the restriction `ONLY`. The order of

[3]We use this simple version for brevity.

```
{ A = { x = B.x + y  y = g }
  g = B.x + B.y
  B = { x = 1  y = 3 }
}
```

Listing 2.11: Example for evaluation order.

these two steps does not matter, but in the restriction, the current name must be selected. Here, we restrict the Group by the renamed selectors. Furthermore, the new definition `addAll` must be added to this Group. We can do this by the `EXTEND` modification. Here we can use all definitions from the Group defined between `EXTEND` and `BY`. For example we can use `isEmpty` instead of `List.isNil`. The integration of these extensions is discussed in Chap. 7.

2.5 Towards an Evaluation Strategy

In a call-by-value language an evaluation order is necessary which respects the expectation that all variables are bound to a value before they are used. This evaluation order can be established in two ways: either the programmer defines the order explicitly or the compiler obtains it via a dependency analysis.

As explained in Sec. 1.1, mutually recursive modules with a simple forward reference force the compiler to determine the dependency order at compile-time.

A dependency analysis for first-class, hierarchical, and mutually recursive modules is not straightforward. Therefore, the way the analysis works is sketched in the next section and worked out in detail in Sec. 4.

As we combine all three features in our modules—hierarchies, mutual recursion, and the first-class property—it is a complex process to identify the evaluation order. Since the obstacles are not immediately obvious, we first discuss nested modules, then add mutual recursion, and finally we allow our modules to be first-class values.

2.5.1 Nested Modules

We start with nested modules, which enable building a hierarchy of Groups. Groups are only allowed as top-level expressions or as the right-hand side of a definition. Definitions are not allowed to be mutually recursive. It follows that within every Group, an order for all definitions can be given.

To identify the evaluation order for the example in Listing 2.11, we first have to evaluate the complete Group `B` with both definitions. Afterwards, the definition `g` can be evaluated, followed by the complete Group `A`.

For programs without first-class and mutually recursive modules, the order can be determined by regarding only the free variables of all right-hand sides. For the resulting dependency graph (see Sec. 3.6), the strongly connected components in topological sort

```
{ E = { even = λn. IF n==0 THEN true ELSE O.odd (n-1)
        is2even = even val
        val = 2 }
  O = { odd = λn. IF n==0 THEN false ELSE E.even (n-1)
        is2odd = odd 2 }
}
```

Listing 2.12: Mutually recursive Groups.

represent the order (see Sec. 3.8). The dependency graph for the current example is given in Fig. 2.13(a). An edge from a to b means a depends on b.

As demonstrated in the next section, this approach is insufficient when mutually recursive modules are allowed.

2.5.2 Mutually Recursive Modules

We now turn to the scenario where definitions are allowed to be mutually recursive. Especially, Groups can be mutually recursive across Group boundaries. As before, they are still allowed only as the direct right-hand side of a definition or at the top-level. As previously mentioned in Sec. 1.1, we forbid cycles in which at least one definition is not a function as is typical in most call-by-value languages. Even with this constraint, finding the evaluation order remains difficult. We demonstrate this using our running example. For quick reference, the code is repeated in Listing 2.12.

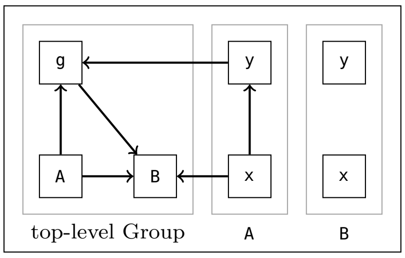
In a call-by-value semantics, recursion is only possible for λ-abstractions, since those definitions evaluate to closures. Therefore, it is not possible to build up a closure for the entire Group. This would only be possible, if the auxiliary definitions `is2even` and `is2odd` were not present. We have to find an order for all definitions within the two Groups `E` and `O`. For the given example, the call-by-value semantics requires that `even` and `odd` must be evaluated before `is2odd` and `is2even`. It is not possible to find this order by only regarding the free variables. The dependency graph for this example with free variables is shown in Fig. 2.13(b). There is a cycle between `E` and `O`, which is forbidden as both definitions are Groups and not functions. This contradicts the goal of allowing mutually recursive modules.

To calculate the correct evaluation order we look at dependencies across Group boundaries and also consider selection chains (see Sec. 3.2 for the precise definition). In this example, a possible evaluation order is:

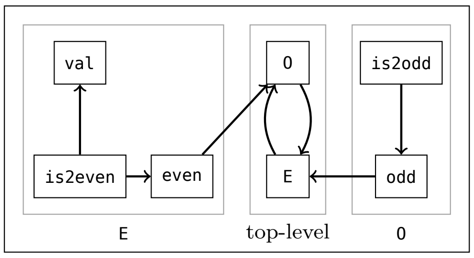
`E.val`, (`E.even`, `O.odd`), `E.is2even`, `E`, `O.is2odd`, `O`

By using the complete selection chain for the dependency graph, the hierarchy is broken and the mutually recursive Groups are handled simultaneously.

In [31], a flattening mechanism is proposed to find an evaluation order. There the hierarchy is broken and all definitions occur at the same level. The result of the flattening for the even-odd example is shown in Listing 2.14. Instead of just one identifier, the



(a) Dependency graph for the example in Listing 2.11.



(b) Dependency graph for the example in Listing 2.12.

Figure 2.13: Dependency graphs using free variables. An edge from a to b means a depends on b.

```
E.even    = λn. IF n==0 THEN true ELSE O.odd (n-1)
E.is2even = E.even E.val
E.val     = 2
O.odd     = λn. IF n==0 THEN false ELSE E.even (n-1)
O.is2odd  = O.odd 2
```

Listing 2.14: Example for flattening.

name on the left-hand-side of all definitions is the complete path in dot-notation. This name can be used as the identifying name of the definition. Furthermore, all variables must be replaced by the complete name within the Group, e. g. `even` in the definition `is2even` must be replaced by `E.even`. The name completion is necessary to ensure unique names after the flattening, as it is allowed to use the same selector in two Groups.

After flattening, we can build a dependency graph. But instead of just using the free variables, we now use the complete selection chain as it represents the name and corresponds to exactly one definition.

Flattening is possible for mutually recursive nested Groups. The technique is not always applicable to first-class values, as shown by the example in the next section.

2.5.3 An Undecidable Problem

Until now, Groups were only allowed as top-level expressions or as the right-hand side of a definition. If we allow Groups to be first-class values, i. e., if they are allowed at any position in an expression, then flattening is not possible anymore.

In the previous section, we have considered the complete selection chain for the dependencies. This mechanism is quite similar to path resolution, which is undecidable for nested mutually recursive modules as proven in [26]. We just present an informal explanation here.

In Listing 2.15, an artificial example is given. The function `AppList` creates a Group containing three functions. `listUntil` creates the list of elements resulting from the

```
...
AppList = λf.λp. { listUntil = λn. IF check n THEN List.Nil
                                   ELSE List.Cons n (listUntil (rest n))
                   check = p
                   next = f
                 }
...
dec2 = AppList (λx. x-2) (λx. x<0)
declist = (dec2.listUntil input)
second = IF !((dec2.check input) || (dec2.check (dec2.next input))
         THEN declist.tl.hd
         ELSE {}
...
```

Listing 2.15: A problematic example using `List` of Listing 2.2.

multiplied application of `f` to the input until the given predicate is fulfilled. `check` checks whether the input fulfills the predicate and `next` returns the result of the next application.

`dec2` creates an `AppList` Group with a function decrementing the element by 2 and a predicate that checks whether the input is less than 0. Consequently, `declist` describes the decreasing sequence of even or odd natural numbers starting with `input`. The definition `second` selects the second element of this list. If there is no second element, the empty Group—representing an error here—is returned.

To create any kind of dependency graph, we have to determine the dependencies by resolving all paths. Considering the definition of `second`, there is a dependency to `declist.tl.hd`. Path resolution would have to identify the intended definition. To explain the problem, we suggest different possible definitions for `input`. In the case of a simple definition like `input = 4`, it is imaginable that we can find the target of the reference `declist.tl.hd` by static analysis at compile-time, but what if the definition was:

```
g = λx. g (n + 1)
input = g 7
```

or the equivalent definition:[4]

```
input = (λx. (λg.λn. g (n+1)) (λz. (x x) z))
                 (λx.(λg.λn. g (n+1)) (λz. (x x) z)) 7
```

Both definitions do not terminate, which is obvious for the first version but not for the latter. In the extreme, the complete program must be evaluated to resolve the path.

Nakata et al. have shown in [26] that path resolution is undecidable for recursive nested modules and first-order functors. As Groups are first-class values, GLang contains not only first-order functors, but higher-order functors by virtue of λ-abstractions. Therefore,

[4]We obtain this expression by applying the call-by-value fixed point operator Z to the function `(λg.λn. g (n+1))` and unrolling it once.

we have to restrict GLang. Nakata et al. restrict their functions. We instead restrict the expressive power of the path resolution.

2.5.4 Solution

This restriction splits the path into a *static* and a *dynamic* part. The static part represents the dependencies and the dynamic part represents the selections. Selections can fail at runtime. This is quite common for selections from data structures, e. g. the selection of the head from an empty list.

We omit all applications and path abbreviations to identify the static path. Therefore, the path `declist.tl.hd` is not resolved completely. The static part is only the variable `declist` and the dynamic selection is `.tl.hd`.

Consequentially, there are some cases where we have to consider the selection chain and some cases where parts of the chain should not be considered. For every path we regard only the static part and ignore the dynamic part. The static part of a path is the variable and all selectors representing a definition within the current Group hierarchy (see Sec. 3.3). In this example, it is just the lexically visible variable `declist`, while `.tl.hd` is the dynamic part, as these names are only available after some steps of evaluation. The proper way to calculate the dependencies is described in Chap. 4.

Our dependency analysis uses this restricted path resolution to determine the evaluation order. The result of the dependency analysis is used to transform the whole input program into an intermediate representation with explicit ordering. This intermediate representation allows a definition of the semantics.

3 Basics

The basis for our dependency analysis is a novel path resolution algorithm. This algorithm identifies the dependencies between definitions within a program. In this section we give the necessary definitions and the path resolution algorithm itself.

In the following, we consider the concept of Groups combined with λ-expressions. For simplicity, we leave out matches and conditionals. They behave in all our definitions and functions like applications. The symbol c is used to represent all kinds of constants.

Fig. 3.1 contains the syntax for the sub-language that we consider in the function definitions in the rest of this work. But for enhanced readability the code given in our examples still uses the entire language.

We separate Groups from other expressions by splitting expressions into terms T and Groups G. In addition, we distinguish between a term definition, where the right-hand side is a term, and a definition of an inner Group, where the right-hand side is a Group-expression.

Furthermore, we annotate all expressions E by a unique index ℓ from the set of all labels Lab and use brackets for a clear scope of annotation. The mapping $\mathit{exp} : \mathit{Lab} \hookrightarrow G \cup T$ associates the expression to each label.

3.1 Notation

For all language definitions, the same symbol is used, both for the set and for the elements of this set. For example, an expression of the GLang language is E and the meta-level variable for a Group is G. For all other sets we define the set of meta-variables when we

$$
\begin{array}{rcll}
E & ::= & G \mid T & \\
T & ::= & [\lambda\ x\ .\ E]^{\ell} & \text{— Abstraction} \\
 & \mid & [E\ E]^{\ell} & \text{— Application} \\
 & \mid & c^{\ell} & \text{— Constant} \\
 & \mid & x^{\ell} & \text{— Variable} \\
 & \mid & [E\ .\ x]^{\ell} & \text{— Selection} \\
G & ::= & \{\ D^{*}\ \}^{\ell} & \text{— Group} \\
D & ::= & x = G & \text{— Definition of inner Group} \\
 & \mid & x = T & \text{— Term definition}
\end{array}
$$

Figure 3.1: Labeled and split syntax of the reduced subset of the GLang language.

introduce them. To distinguish between elements of a set we use indices and the prime symbol, e. g. ℓ and ℓ' are two elements of the set *Lab*.

The meta-variable representing a sequence of elements is represented by the overlined element variable. For example, a sequence of definitions is denoted by $\overline{D}$. We write the elements of a list as a sequence of elements separated by comma. We use $+\!\!+$ as concatenation of lists. Sets of elements are denoted by the symbol of the elements with a plural "s" appended—with one exception: sets of labels are denoted by *labs* and not by ℓs.

3.2 Path

The set of all identifiers is denoted by X and we use x, y, and z as meta-variables. A *selector* is an identifier with a prefix dot. A *selector chain* is a sequence of selectors. For a selector chain with length n we write $(\texttt{.}x_i)^{i\in 1..n}$ or the meta-variable sc. The set of all selection chains is denoted by Sc.

A *path* is a variable (identifier without a prefix dot) followed by a (possibly empty) selector chain. The set of all paths is denoted by *Path*. As a meta-variable for a path we use p. Furthermore we use $x(\texttt{.}x_i)^{i\in 1..n}$ to pattern match on the variable and all the selectors.

ε represents the empty path not containing any identifier, neither a selector nor a variable. A *subpath* is a path p_1 representing the prefix of another path p_2. We write the statement p_1 is a subpath of p_2 as $p_1 \sqsubseteq p_2$. The relation $\sqsubseteq$ is defined as:

- $\varepsilon \sqsubseteq p$ for any $p \in Path$
- $x(\texttt{.}x_i)^{i\in 1..n} \sqsubseteq x(\texttt{.}x_i)^{i\in 1..m}$ for $n \leq m$

3.3 Group Hierarchy

As Groups are first-class values, they are normal expressions. They can occur within any expression and as the right-hand side of a definition. Each time a Group is defined within a term—not at the right-hand side of a definition—a new path name-space is opened. As multiple Groups can occur within a normal term, multiple name-spaces are possible. There are two different kinds of Groups:

Top-level Group If a Group is not the right-hand side of a definition, we call it a *top-level Group* following Blume [4]. Blume's Groups are not allowed to be first-class values; therefore, there is only one top-level Group in each program. In contrast to this approach, there can be multiple top-level Groups within a GLang program. The complete path of a top-level Group is always the empty path ε.

Inner Group If a Group corresponds to the right-hand side of a definition, we call it an *inner Group*. The complete path from the current top-level Group to this inner

Group is represented by concatenating the selectors of the definitions pointing to this Group.

The splitting in G and T in our syntax clarifies the difference. All Groups that are also normal expressions E are top-level Groups and all others are inner Groups.

```
λx. { Foo = { A = {}¹  b = x }²
      bar = λy. { a = x  b = y }³
    }⁴
```

Listing 3.2: Example for multiple top-level Groups.

In the example in Listing 3.2, there are four Groups. Two of them are top-level Groups and two are inner Groups. The Groups 1 and 2 are inner Groups, as they are the right-hand sides of the definitions `Foo` and `A`. The complete path for these inner Groups are `Foo` and `Foo.A`. The Groups with labels 3 and 4 are top-level, as both form the body of an abstraction.

A *Group hierarchy* is built by a top-level Group and all its inner Groups. All top-level Groups occurring in any term-expression within this hierarchy are not part of this hierarchy but form their own. In the previous example, there are two top-level Groups, so there are also two Group hierarchies. The first hierarchy is the Group 4 and the inner Groups 1 and 2. The second hierarchy consists only of the top-level Group 3.

For every hierarchy, we define the inner and the outer-context. The *inner context* is the set of all visible selectors within this hierarchy and the *outer context* is the set of all variables defined above the top-level Group. Definitions from inner hierarchies are not part of the context, as they are only available after some steps of evaluation.

In the given example, the outer-context for the hierarchy with the top-level Group 4 is only the variable `x` bound in the abstraction.

The inner context are the four definitions: `Foo`, `bar`, `Foo.A`, and `Foo.b`. The definitions within the Group 3 are not accessible directly, hence they are not part of the context.

The complete path of a definition within a hierarchy is called the *name* of this definition. Within a Group hierarchy this is the selection chain of all selectors from the top-level Group to this definition ending with the identifier of the definition. We use the meta-variable N to indicate that the given path represents the name of a definition. The name of each definition is contained in the map $\mathit{name} : \mathit{Lab} \hookrightarrow \mathit{Path}$.

The name of a Group definition is always a subpath of all its inner definitions and the name of all top-level Groups is ε.

3.4 Free Variables and Paths

The dependency analysis must identify dependencies between definitions. In other languages, free variables indicate these dependencies. In GLang, the dependencies are identified via paths. In the following, we define how to identify these path dependencies.

$$
\begin{array}{lll}
\mathcal{F} : \; E \to \mathcal{P}(\mathit{Path}) & & \\
\hline
\mathcal{F}[\![\lambda x\,.\,E]\!] & = \mathcal{F}[\![E]\!] \ominus \{x\} & \\
\mathcal{F}[\![E_1\; E_2]\!] & = \mathcal{F}[\![E_1]\!] \cup \mathcal{F}[\![E_2]\!] & \\
\mathcal{F}[\![x(.x_i)^{i\in 1..m}]\!] & = \{x(.x_i)^{i\in 1..m}\} & (*) \\
\mathcal{F}[\![E\,.\,x]\!] & = \mathcal{F}[\![E]\!] & \\
\mathcal{F}[\![x]\!] & = \{x\} & \\
\mathcal{F}[\![c]\!] & = \emptyset & \\
\mathcal{F}[\![\{(x_i = E_i)^{i\in 1..m}\}]\!] & = \bigcup_{i\in 1..m} \mathcal{F}[\![E_i]\!] \ominus \{x_i^{i\in 1..m}\} &
\end{array}
$$

Figure 3.3: Function $\mathcal{F}$ for calculating free paths.

Every expression E of a program is mapped to four sets using the labels ℓ:

freePaths	$: \mathit{Lab} \hookrightarrow \mathcal{P}(\mathit{Path})$	set of free paths
availVars	$: \mathit{Lab} \hookrightarrow \mathcal{P}(X)$	set of Group-bound available variables
grpBndPaths	$: \mathit{Lab} \hookrightarrow \mathcal{P}(\mathit{Path})$	set of Group-bound free paths
outerVars	$: \mathit{Lab} \hookrightarrow \mathcal{P}(X)$	set of variables supplied by the outer-context

In the following, these mappings are described in detail.

3.4.1 Free Paths

Free paths extend the λ-calculus concept of free variables to paths. Paths are free iff the leading variable is free. The function $\mathcal{F}$ in Fig. 3.3 maps every expression to its free paths.

The calculation of classical free variables and our free paths differs only in equation (*) in Fig. 3.3. Here, the whole path $x(.x_i)^{i\in 1..m}$ is inserted instead of the variable x. Furthermore, the removal of bound variables must be extended to a removal of a path. We define the operation $\ominus : \mathcal{P}(\mathit{Path}) \times \mathcal{P}(X) \to \mathcal{P}(\mathit{Path})$:

$$ps \ominus xs = \{x(.x_i)^{i\in 1..n} \mid x(.x_i)^{i\in 1..n} \in ps \wedge x \notin xs\}$$

This extended removal of bound variables is used for variables bound in a λ-abstraction as well as for the selectors of a Group. In a Group, all selectors are bound in all definitions. So the free paths of a Group are all free paths of the right-hand sides without those paths where the variable is equal to one of the selectors.

The mapping *freePaths* is the composition $\mathit{freePaths} = \mathcal{F} \circ \mathit{exp}$.

3.4.2 Group-Bound Available Variables

Available variables of an expression E are variables introduced by the current context. We are especially interested in variables introduced by the current hierarchy. This set of variables is called *Group-bound available variables.* It is stored in the mapping *availVars*

$$\mathcal{A}^{\alpha} : \ \alpha \to \mathcal{P}(X) \to (Lab \hookrightarrow \mathcal{P}(X)) \qquad \text{for } \alpha \in \{E, T, G, D\}$$

$$\begin{array}{lll}
\mathcal{A}^{E}[\![T]\!]\ xs & = \mathcal{A}^{T}[\![T]\!]\ xs & \\
\mathcal{A}^{E}[\![G]\!]\ xs & = \mathcal{A}^{G}[\![G]\!]\ \emptyset & (\dagger) \\
\mathcal{A}^{T}[\![[\lambda x.E]^{\ell}]\!]\ xs & = \{\ell \hookrightarrow xs\} \cup (\mathcal{A}^{E}[\![E]\!]\ xs \setminus \{x\}) & \\
\mathcal{A}^{T}[\![[E_1\ E_2]^{\ell}]\!]\ xs & = \{\ell \hookrightarrow xs\} \cup \mathcal{A}^{E}[\![E_1]\!]\ xs \cup \mathcal{A}^{E}[\![E_2]\!]\ xs & \\
\mathcal{A}^{T}[\![[E.x]^{\ell}]\!]\ xs & = \{\ell \hookrightarrow xs\} \cup \mathcal{A}^{E}[\![E]\!]\ xs & \\
\mathcal{A}^{T}[\![x^{\ell}]\!]\ xs & = \{\ell \hookrightarrow xs\} & \\
\mathcal{A}^{T}[\![c^{\ell}]\!]\ xs & = \{\ell \hookrightarrow xs\} & \\
\mathcal{A}^{G}[\![\{D_i{}^{i\in 1..m}\}^{\ell}]\!]\ xs & = \{\ell \hookrightarrow xs\} \cup \left(\bigcup\limits_{i\in 1..m} \mathcal{A}^{D}[\![D_i]\!]\ (xs \cup \{x_i{}^{i\in 1..m}\}) \right) & \\
 & \quad \text{where} \quad D_i = (x_i = T_i) \text{ or } D_i = (x_i = G_i) & \\
\mathcal{A}^{D}[\![x = T]\!]\ xs & = \mathcal{A}^{T}[\![T]\!]\ xs & \\
\mathcal{A}^{D}[\![x = G]\!]\ xs & = \mathcal{A}^{G}[\![G]\!]\ xs & (\ddagger)
\end{array}$$

Figure 3.4: Calculation of Group-bound available variables.

```
{ ...
  K = { x = 1 }
  c = λa. { A = { h = a^1
                  f = [λh. [K.x + h]^3]^2
                }
            T = { m = [λy. y]^4 }
          }
... }
```

$$\begin{aligned}
\mathit{availVars}(1) &= \{\mathtt{A},\mathtt{h},\mathtt{f},\mathtt{T}\} \\
\mathit{availVars}(2) &= \{\mathtt{A},\mathtt{h},\mathtt{f},\mathtt{T}\} \\
\mathit{availVars}(3) &= \{\mathtt{A},\mathtt{f},\mathtt{T}\} \\
\mathit{availVars}(4) &= \{\mathtt{A},\mathtt{T},\mathtt{m}\}
\end{aligned}$$

Figure 3.5: Example for Group-bound available variables.

which is calculated by the family of functions $\mathcal{A}^E$, $\mathcal{A}^T$, $\mathcal{A}^G$, $\mathcal{A}^D$ of Fig. 3.4 (one for each nonterminal in the grammar of Fig. 3.1).

The second argument is the context xs of the current expression, which is a set of Group-bound available variables. The mapping *availVars* for a complete program E is defined as $\mathit{availVars} = \mathcal{A}^E[\![E]\!]\ \emptyset$ since the context is empty.

In equation (†), the set of Group-bound available variables is cleared, because at that point, a new hierarchy starts. In contrast, xs is not modified in equation (‡). The possibility to define the function $\mathcal{A}^\alpha$ in this simple form is one of the reasons to distinguish between Groups and other expressions in the syntax.

The example in Fig. 3.5 illustrates the mapping *availVars*. The Group-bound available variables for the expressions with labels 1–4 are bound within the hierarchy marked by $\underline{\{}$ and $\underline{\}}$. In the expressions 1 and 2, the same set of variables are bound. In expression 3, the variable `h` is not bound within the hierarchy as it is bound by the λ-abstraction. In expression 4, the definitions of `h` and `f` are not visible, but `m` is, as expression 4 is in another branch of this hierarchy. All four expressions can access the variables `K`, `c`

$$\mathcal{O}^{\alpha} :\ \alpha \to \mathcal{P}(X) \to \mathcal{P}(X) \to (Lab \hookrightarrow \mathcal{P}(X)) \qquad \text{for } \alpha \in \{E, T, G\}$$

$$\begin{aligned}
\mathcal{O}^{E}[\![T]\!]\ xs_o\ xs_i &= \mathcal{O}^{T}[\![T]\!]\ (xs_o \cup xs_i)\ \emptyset \\
\mathcal{O}^{E}[\![G]\!]\ xs_o\ xs_i &= \mathcal{O}^{G}[\![G]\!]\ xs_o\ xs_i \\
\mathcal{O}^{T}[\![[\lambda x\,.\,E]^{\ell}]\!]\ xs_o\ xs_i &= \{\ell \hookrightarrow xs_o\} \cup \mathcal{O}^{E}[\![E]\!]\ (xs_o \cup \{x\})\ xs_i \\
\mathcal{O}^{T}[\![[E_1\ E_2]^{\ell}]\!]\ xs_o\ xs_i &= \{\ell \hookrightarrow xs_o\} \cup \mathcal{O}^{E}[\![E_1]\!]\ xs_o\ xs_i\ \cup \mathcal{O}^{E}[\![E_2]\!]\ xs_o\ xs_i \\
\mathcal{O}^{T}[\![x^{\ell}]\!]\ xs_o\ xs_i &= \{\ell \hookrightarrow xs_o\} \\
\mathcal{O}^{T}[\![c^{\ell}]\!]\ xs_o\ xs_i &= \{\ell \hookrightarrow xs_o\} \\
\mathcal{O}^{G}[\![\{(x_j = E_j)^{j \in 1..n}\}^{\ell}]\!]\ xs_o\ xs_i &= \{\ell \hookrightarrow xs_o\} \cup \left(\bigcup_{i \in 1..m} \mathcal{O}^{E}[\![E_i]\!]\ xs'_o\ xs'_i \right) \\
&\qquad \text{where} \quad xs'_o = xs_o \setminus \{x_j{}^{j \in 1..n}\} \\
&\qquad\qquad\quad\ \ xs'_i = xs_i \cup \{x_j{}^{j \in 1..n}\}
\end{aligned}$$

Figure 3.6: Calculation of the outer-context.

and a, but the first two variables are bound in the outer-context and a is bound by a λ-abstraction.

3.4.3 Group-Bound Free Paths

In a normal dependency analysis, an expression depends on its free variables. As explained in Sec. 2.5, we have to use the selection chain to identify dependencies. Therefore, an expression depends on all its free paths. Since the dependency analysis is calculated separately for every hierarchy, only free paths within this hierarchy need to be considered. The set of all *Group-bound free paths* of an expression is the set of all free paths of an expression were the variable is a Group-bound available variable. The set is stored in the mapping *grpBndPaths* and can be calculated using the mappings *freePaths* and *availVars*:

$$\mathit{grpBndPaths}(\ell) = \{x(.x_i)^{i \in 1..n} \mid x \in \mathit{availVars}(\ell) \wedge x(.x_i)^{i \in 1..n} \in \mathit{freePaths}(\ell)\}$$

3.4.4 Outer-Context Variables

The outer-context is a set of variables. These are all variables defined above the current hierarchy, i. e., above the current top-level Group. The outer-context is necessary to detect undefined variables, as each Group hierarchy is analyzed separately. The mapping *outerVars* is calculated by the family of functions $\mathcal{O}^{\alpha}$ defined in Fig. 3.6.

There are two context sets used as arguments for the functions $\mathcal{O}^{\alpha}$. The first context is the set of all variables defined above the current hierarchy. The second set contains all variables defined within the current hierarchy.

For a term T, all variables are defined outside, i. e., the inner-context is empty. The outer-context contains all variables defined inside and outside the current hierarchy. In contrast, both context sets remain unchanged in the case of a Group. The variable

```
{ E = { even = [λn. IF n==0 THEN true ELSE O.odd (n-1)]^4
        is2even = [even val]^5
        val = [2]^6 }^2
  O = { odd = [λn. IF n==0 THEN false ELSE E.even (n-1)]^7
        is2odd = [odd 2]^8 }^3
}^1
```

Listing 3.7: Mutually recursive Groups with labels.

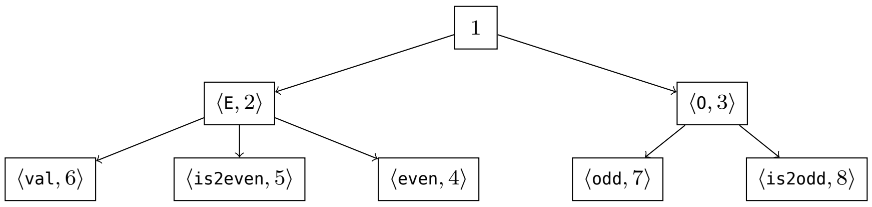


Figure 3.8: Group tree for the example in Listing 3.7.

of an abstraction is an outer-context variable for the inner expression. The variables of the definitions within a Group are inner-context variables, whereas variables in the outer-context are shadowed. Therefore, they are removed from the outer-context set.

3.5 Hierarchy Tree

Every Group hierarchy can be represented as a hierarchy tree. The root represents the top-level Group. The children of each vertex are the definitions of its associated Group.

As an example, the hierarchy tree for our running example of recursive Groups is shown in Fig. 3.8. The code, now augmented with labels, is given in Listing 3.7. For clarity, all unused labels are left out, so that only the right-hand sides of definitions and the top-level Group are augmented here with a label.

Every vertex represents a definition or a top-level Group. A vertex representing a definition is a tuple holding the identifier of the the label of this definition. As the label is unique, we can identify a vertex by its label.

The root of each Group tree is a special vertex as the top-level Group of a hierarchy is not a definition and so the vertex has no identifier but only a label; in this example the label is 1.

We define a tree relation $\longrightarrow$ which characterizes the dependencies between a Group and the inner definitions:

$$\begin{aligned}
\langle x, \ell\rangle \longrightarrow \langle x', \ell'\rangle &:\Leftrightarrow \mathit{exp}(\ell) = \{(y_i = E_i^{\ell_i})^{i\in 1..n}\} \wedge \exists j \in 1..n : \ell' = \ell_j \wedge x' = y_j \\
\langle \ell\rangle \longrightarrow \langle x', \ell'\rangle &:\Leftrightarrow \mathit{exp}(\ell) = \{(y_i = E_i^{\ell_i})^{i\in 1..n}\} \wedge \exists j \in 1..n : \ell' = \ell_j \wedge x' = y_j
\end{aligned}$$

3.6 Dependency Graph

A dependency analysis is always based on a dependency graph. The dependency graphs for GLang need to capture the dependencies between a Group and its definitions and the dependencies between all definitions. To distinguish between both kinds of dependencies, we define two different kinds of edges: `treeEdge` and `depEdge`. The definition of our dependency graph is given in Fig. 3.9. The hierarchy tree forms the basis for the graph. A vertex represents either a definition or a top-level Group. A definition $D = (x = E^{\ell})$ is represented by the identifier x of this definition and the label ℓ. The top-level Group is represented solely by its label. Elements of the set *Graph* are denoted by g. For vertices we use the meta-variable v and for edges we use e.

$$\begin{aligned}
\mathit{Graph} &:= \{\langle \mathit{vs}, \mathit{es}\rangle \mid \mathit{vs} \in \mathcal{P}(\mathit{Vert}) \wedge \mathit{es} \in \mathcal{P}(\mathit{Edge})\} \\
\mathit{Vert} &:= \{\langle x, \ell\rangle \mid x \in X \wedge \ell \in \mathit{Lab}\} \cup \{\langle \ell\rangle \mid \ell \in \mathit{Lab}\} \\
\mathit{Edge} &:= \{\texttt{treeEdge}\ v_1\ v_2 \mid v_1, v_2 \in \mathit{Vert}\} \cup \{\texttt{depEdge}\ v_1\ v_2\ \mathit{sc} \mid v_1, v_2 \in \mathit{Vert} \wedge \mathit{sc} \in \mathit{Sc}\}
\end{aligned}$$

Figure 3.9: Definition of dependency graphs.

We distinguish between tree and dependency edges, which enables a tree and a graph view. Therefore, this graph representation can also be used to represent a hierarchy tree.

To define the dependency analysis, we use the following observer functions for elements of the set *Graph*:

- eStart and eEnd: the start and the end of an edge.
- vLabel: the label of a vertex.
- parent: the parent—in the tree view—of a vertex through the label.
- leaves: the set of leaves of the tree.

Furthermore, we need a function subGraph, which calculates the subgraph containing only the given set of vertices and those edges, where the source and the destination are in that set:

$$\begin{aligned}
&\mathsf{subGraph} : \ \mathit{Graph} \to \mathcal{P}(\mathit{Vert}) \to \mathit{Graph} \\
&\mathsf{subGraph}\ \langle \mathit{vs}_1, \mathit{es}_1\rangle\ \mathit{vs} \ = \langle \mathit{vs}, \mathit{es}\rangle \\
&\quad \text{where} \quad \mathit{es} = \{e \in \mathit{es}_1 \mid \mathsf{eStart}\ e \in \mathit{vs} \wedge \mathsf{eEnd}\ e \in \mathit{vs}\}
\end{aligned}$$

In the tree view, such a subgraph is not a tree anymore but a forest—a set of trees. The function tops returns the set of all root vertices of this forest representing a Group. A root vertex has no ingoing tree edge. The original dependency graph has only one top vertex—the vertex representing the top-level Group of the Group hierarchy. In contrast, a subgraph obtained by removing this root vertex contains more than one top vertex.

$$\mathsf{tops} : \; \mathit{Graph} \to \mathcal{P}(\mathit{Vert})$$
$$\mathsf{tops}\ \langle vs, es\rangle = \{v \mid \nexists v_1 \in vs : \texttt{treeEdge}\ v_1\ v \in es \wedge \mathit{exp}(\mathsf{vLabel}\ v) \in G\}$$

3.7 Path Resolution

As mentioned in Sec. 2.5.3 we define a specific path resolution to determine the dependencies between definitions. This path resolution is decidable for GLang.

The path resolution resolves a free path $x(\texttt{.}x_i)^{i\in 1..n}$ of a definition $D = (y = E^\ell)$ using the hierarchy tree and the vertex $\langle y, \ell\rangle$. As a program can contain multiple Group hierarchies, there is a set of hierarchy trees representing a complete program. Each definition is part of only one tree. Therefore, the tree is implicitly given through the vertex.

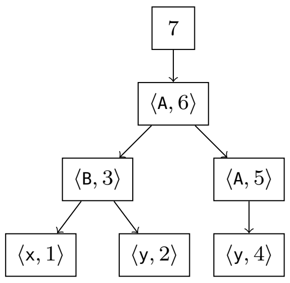
The resolution is done in two steps. The first one resolves the variable and the second one the selection chain. The variable is identified by an upward search starting in the vertex representing the definition D. The upward search is necessary to handle shadowing. The first occurrence of a definition representing the variable is identified and returned. If the variable is not even defined in the top-level Group, it is either not defined, or it is defined in the outer-context. For the path resolution this does not make a difference, as the variable is not defined within the current Group hierarchy in both cases. To represent an undefined variable, the upward search returns $\perp$. In the case of a Group-bound free path, the variable can always be resolved, i. e., the case of an undefined variable will never occur.

$$\mathsf{upwards} : \; \mathit{Vert} \to X \to (\mathit{Vert} \cup \{\perp\})$$
$$\mathsf{upwards}\ \langle y, \ell\rangle\ x = \begin{cases} \langle y_i, \ell_i\rangle\,, & \text{if } \exists j \in 1..n : y_j = x \\ & \quad \wedge\ \mathit{exp}(\ell) = \{(y_i = E_i^{\ell_i})^{i\in 1..n}\} \\ \mathsf{upwards}\ (\mathsf{parent}\ \ell)\ x, & \text{otherwise} \end{cases}$$
$$\mathsf{upwards}\ \langle \ell\rangle\ x \quad = \begin{cases} \langle y_i, \ell_i\rangle\,, & \text{if } \exists j \in 1..n : y_j = x \wedge \mathit{exp}(\ell) = \{(y_i = E_i^{\ell_i})^{i\in 1..n}\} \\ \perp, & \text{otherwise} \end{cases}$$

The downward search starts with the definition of the variable. If the selection chain is empty, the current vertex is returned. In the case of a Group definition, it is checked whether this Group defines the first selector. If so, the downward search continues with this definition. If there is no child matching this identifier or the end of the selection chain is reached, the current definition is returned. The downward search selects the innermost definition. This is the definition with the longest name representing the Group-bound free path.

```
{ A = { B = { x = [A.y]^1
              y = [B.x]^2]
            }^3
        A = { y = [2]^4 }^5
      }^6
}^7
```

(a) Listing.



(b) Hierarchy tree.

Figure 3.10: First example for path resolution.

$$\begin{aligned}
&\mathsf{downwards} : \ \mathit{Vert} \to \mathit{Sc} \to \mathit{Vert}\\
&\mathsf{downwards}\ \langle y, \ell\rangle\ (\texttt{.}x_i)^{i\in 1..n} =\\
&\qquad\begin{cases}
\langle y, \ell\rangle\,, & \text{if } n = 0\\
\mathsf{downwards}\ \langle y_i, \ell_i\rangle\ (\texttt{.}x_i)^{i\in 2..n}, & \text{if } \mathit{exp}(\ell) = \{(y_i = E_i^{\ell_i})^{i\in 1..n}\}\\
& \quad \wedge \exists j \in 1..n : y_j = x_1\\
\langle y, \ell\rangle\,, & \text{otherwise}
\end{cases}
\end{aligned}$$

The function resolve resolves the path for a vertex v and a free path $y(\texttt{.}y_i)^{i\in 1..n}$:

$$\begin{aligned}
&\mathsf{resolve} : \ \mathit{Vert} \to \mathit{Path} \to \mathit{Vert}\\
&\mathsf{resolve}\ v\ y(\texttt{.}y_i)^{i\in 1..n} = \mathsf{downwards}\ (\mathsf{upwards}\ v\ y)\ (\texttt{.}y_i)^{i\in 1..n}
\end{aligned}$$

We give two examples for the path resolution. Figure 3.10 shows the first example. To resolve the free path `A.y` in the vertex $\langle$x$, 1\rangle$, the free variable `A` is resolved by the function upwards. The vertex with label 1 represents a term definition. Therefore, the search continues with the parent $\langle$B$, 3\rangle$ of this vertex. In the Group `B` there is no definition `A`. The search continues in the vertex $\langle$A$, 6\rangle$. This Group has a definition `A`. The corresponding vertex is $\langle$A$, 5\rangle$. This is the innermost definition of `A`. The second step of the resolution starts in this vertex. The selection chain of the free path consists only of the selector `.y`. As there is a definition for `y`, the downward search continues with the definition $\langle$y$, 4\rangle$. In this step, the selection chain is empty. The path resolution returns the vertex $\langle$y$, 4\rangle$ representing the referenced definition.

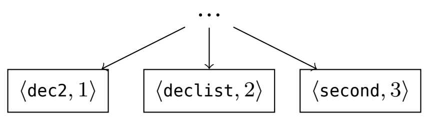
The free path `B.x` of the definition labeled with 2 is resolved to the vertex $\langle$x$, 1\rangle$.

As a second example, we return to the problematic example given in Sec. 2.5.3. In Fig. 3.11 the relevant part of the code is repeated and the corresponding extraction of the hierarchy tree is given. The upward step of the path resolution for the free path `declist.tl.hd` identifies the vertex $\langle$`declist`$, 2\rangle$. As this definition is a term definition, the downward step stops in this vertex and returns it. Thus, the selection chain `.tl.hd` is not resolved.

Based on this path resolution, we can define the "depends on" relation for definitions.

```
...
dec2 = [ AppList ... ]^1
declist = [ (dec2.listUntil input) ]^2
second = [ ... declist.tl.hd ... ]^3
...
```

(a) Listing.



(b) Group tree.

Figure 3.11: Path resolution for problematic example from Sec. 2.5.3.

We write $\langle x_1, \ell_1 \rangle \dashrightarrow \langle x_2, \ell_2 \rangle$, if $D_1 = (x_1 = E_1^{\ell_1})$ depends on $D_2 = (x_2 = E_2^{\ell_2})$. The relation $\dashrightarrow$ is defined as follows:

$$\begin{aligned} \langle x_1, \ell_1 \rangle \dashrightarrow \langle x_2, \ell_2 \rangle :\Leftrightarrow\ & \exists y(.y_i)^{i \in 1..n} \in \mathit{grpBndPaths}(\ell_1): \\ & \langle x_2, \ell_2 \rangle = \mathsf{resolve}\ \langle x_1, \ell_1 \rangle\ y(.y_i)^{i \in 1..n} \end{aligned}$$

A free path represents a dependency on a definition. As we allow shadowing of variables by inner definitions, an identifier is not unique. Within a Group hierarchy a path can either refer to an identifier bound by a λ-abstraction, to a definition bound in the outer-context, or to a definition within the current Group hierarchy. For the dependency analysis, only those paths bound in the current hierarchy are considered as we want to find an order for all definitions within this hierarchy. These are all Group-bound free paths.

We use the definition of “depends on” to sort all definitions within a Group hierarchy.

3.8 Strongly Connected Components

The dependency analysis results in a list of strongly connected components (SCCs) in topological sort. There are multiple ways to define SCCs, we follow [9]: The SCCs of a graph are the equivalence classes of the mutually reachable relation.

Two vertices v_1 and v_2 are reachable $v_1 \rightsquigarrow v_2$ iff there is an edge-path from v_1 to v_2. For a graph $g = \langle vs, es \rangle$ the relation $\rightsquigarrow$ can be defined as the transitive closure es^+. The mutually reachable relation $\leftrightsquigarrow$ can be defined as follows:

$$v_1 \leftrightsquigarrow v_2 :\Leftrightarrow \langle v_1, v_2 \rangle \in es^+ \wedge \langle v_2, v_1 \rangle \in es^+$$

An immediate consequence from this definition is that two graphs have the same mutually reachable relation iff they have the same transitive closure. Furthermore, it follows that these two graphs have the same set of SCCs.

A topological sort of a directed acyclic graph $g = \langle vs, es \rangle$ is a linear ordering of all its vertices vs such that if $v_1 \rightsquigarrow v_2$ then v_1 is before v_2. The graph for the topological sort of the SCCs is the graph $g_{\mathrm{SCC}} = \langle vs_{\mathrm{SCC}}, es_{\mathrm{SCC}} \rangle$:

- The set of vertices vs_{SCC} is the quotient set $vs/{\leftrightsquigarrow}$.
- For each edge $\langle v_1, v_2 \rangle \in es$ there is an edge $\langle [v_1], [v_2] \rangle \in es_{\mathrm{SCC}}$.

This graph is always acyclic. The topological sort is not unique as there is a degree of freedom in the ordering. The order for two definitions not depending on each other is not defined, so they can occur in any order in the topological sort.

We define a strongly connected component as an element of the set *Scc*:

$$Scc := \{\texttt{sccSingle}\ \ell \mid \ell \in Lab\} \cup \{\texttt{sccRec}\ labs \mid labs \subseteq Lab\}$$

Each component contains at least one definition. A definition is represented by its label. A component is either recursive or not; a non-recursive definition is represented by an `sccSingle` and a set of mutually recursive definitions is represented by an `sccRec`. We use *scc* as a variable denoting an element of the set *Scc*. The analysis returns a list of elements of *Scc* topologically sorted.

The set of labels of a component is determined by the function labels:

$$\begin{aligned}
&\mathsf{labels}: \ Scc \rightarrow \mathcal{P}(Lab)\\
&\mathsf{labels}\ (\texttt{sccSingle}\ \ell) = \{\ell\}\\
&\mathsf{labels}\ (\texttt{sccRec}\ labs)\ = labs
\end{aligned}$$

4 Dependency Analysis

To find the evaluation order for a GLang program, we have to perform a dependency analysis for this program. We need an evaluation order only for all term definitions. Every term T is evaluated as a whole. For this reason, we split the analysis and analyze every Group hierarchy separately. For each analysis there are three steps. First, the hierarchy tree must be built, then this tree is extended to a graph by adding the dependency edges, and, finally, the strongly connected components of this graph in topological sort are calculated.

4.1 Dependency Edges

The basis for the dependency graph is the hierarchy tree. For each element $v_1 \dashrightarrow v_2$ of the depends on relation, a dependency edge from v_1 to v_2 must be added into the tree to form a dependency graph. The resulting graph containing all tree edges as well as all dependency edges represents all dependencies. The tree edges are necessary as a Group depends on all its inner definitions. The time complexity for the algorithm to calculate the strongly connected components depends on the number of edges. We reduce the number of edges by removing all dependency edges starting in a Group. This optimized graph is used to calculate the SCCs. This optimization is correct, as the following proposition holds:

Proposition. *If $v_1 \dashrightarrow v_2$ then $v_1 \rightsquigarrow v_2$ is in the optimized dependency graph.*

Proof sketch. We prove this proposition by structural induction on $\mathit{exp}(\mathsf{vLabel}\ v_1)$.

- $\mathit{exp}(\mathsf{vLabel}\ v_1) \in T$:

  In this case, there is an edge from v_1 to v_2 in the optimized dependency graph, as the definition represented by v_1 is a term definition.

- $\mathit{exp}(\mathsf{vLabel}\ v_1) = \{(x_i = E_i^{\ell_i})^{i \in 1..n}\}$:

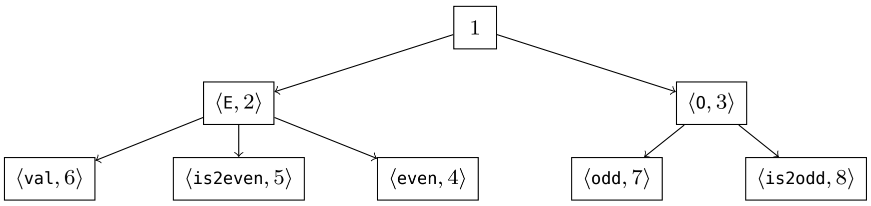
  From $v_1 \dashrightarrow v_2$ it follows:

  1. There is a path $p \in \mathit{grpBndPaths}(\mathsf{vLabel}\ v_1)$.
  2. $v_2 = \mathsf{resolve}\ v_1\ p$

  From the definition of the function $\mathcal{F}$ and (1.) it follows: $\exists j \in 1..n : p \in \mathit{grpBndPaths}(\ell_j)$. As a consequence of (2.) it holds: $\langle x_j, \ell_j \rangle \dashrightarrow v_2$. From the induction hypothesis it follows $\langle x_j, \ell_j \rangle \rightsquigarrow v_2$. As there is a tree edge from a Group vertex to all its inner definitions, there is an edge from v_1 to $\langle x_j, \ell_j \rangle$ and hence it holds: $v_1 \rightsquigarrow v_2$. □

```
{ E = { even = [λn. IF n==0 THEN true ELSE O.odd (n-1)]^4
        is2even = [even val]^5
        val = [2]^6 }^2
  O = { odd = [λn. IF n==0 THEN false ELSE E.even (n-1)]^7
        is2odd = [odd 2]^8 }^3
}^1
```

(a) Code.



(b) Group tree.

Figure 4.1: Repeated code and tree for running example defining `even` and `odd`.

As a consequence of this proposition, the transitive closure of the original and the optimized graph are the same and so are the SCCs (see Sec. 3.8).

To illustrate the algorithm calculating all dependency edges, we use the running example defining the `even` and the `odd` function. The code as well as the hierarchy tree are repeated in Fig. 4.1.

Instead of removing edges, our algorithm creates the optimized graph directly by inserting only those edges that start in a leaf of the hierarchy tree. In our example, these are all vertices with labels 4–8. First, the Group-bound free paths must be calculated. In Fig. 4.2, the three sets *freePaths*, *availVars*, and *grpBndPaths* are calculated for each term definition. In this example, the set of Group-bound free paths is identical to the set of free paths. This is not true in general, as variables from the outer-context may be used.

Label	*freePaths*	*availVars*	*grpBndPaths*
4	`O.odd`	`E`, `O`, `even`, `is2even`, `val`	`O.odd`
5	`even val`	`E`, `O`, `even`, `is2even`, `val`	`even`,`val`
6		`E`, `O`, `even`, `is2even`, `val`	
7	`E.even`	`E`, `O`, `odd`, `is2odd`	`E.even`
8	`odd`	`E`, `O`, `odd`, `is2odd`	`odd`

Figure 4.2: Mapping label to Group-bound free paths.

For each term definition D, the corresponding dependency edges are calculated. The set of all dependency edges for a term definition D results from the set *grpBndPaths*. For each path there is an edge from D to the vertex, the path is resolved to.

As an example, we take the term definition $\langle \texttt{is2even}, 5\rangle$. The set of Group-bound free paths is $\{\texttt{even}, \texttt{val}\}$. The path `even` is resolved to the definition $\langle \texttt{even}, 4\rangle$ and the path

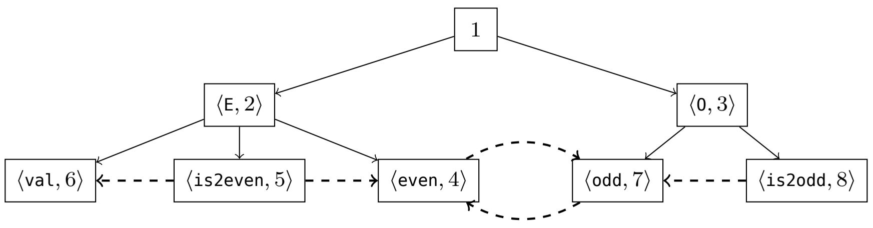


Figure 4.3: Dependency graph for the example in Listing 3.7 with dependency edges dashed.

`val` is resolved to the definition ⟨`val`, 6⟩. So for this term definition two dependency edges are added.

The resulting dependency graph for the running example is shown in Fig. 4.3. The dependency edges are dashed.

As already mentioned, there are two possibilities for the end of the downwards step. In the first case, the free path is found completely in the tree, thus, the free path refers to a definition within the current hierarchy. This allows breaking Group boundaries and is similar to the flattening approach (see 2.5.2). In the second case, the path is not resolved completely. The rest of the selection chain will be stored in the dependency edge for the context analysis (Sec. 4.2).

For the dependency graph, the strongly connected components are calculated via Sharir's algorithm [38].

In our running example, there is only a cycle between `even` and `odd`. Sharir's algorithm ($\mathsf{calcScc} : \mathit{Graph} \to \mathit{Scc}^*$) returns the following list of SCCs

$$\texttt{sccSingle}\ 6,\ \texttt{sccRec}\ \{4, 7\},\ \texttt{sccSingle}\ 5,\ \texttt{sccSingle}\ 2,$$
$$\texttt{sccSingle}\ 8,\ \texttt{sccSingle}\ 3,\ \texttt{sccSingle}\ 1\,.$$

This represents a correct evaluation order. All definitions are evaluated before they are used. The recursive SCCs are only allowed for abstractions. Programs containing non-allowed recursion are detected in the context analysis.

4.2 Context Analysis

Although GLang is a dynamic language, we use a context analysis to find potential errors in the given program as the given dependency analysis identifies such errors for free. Furthermore, we can assume in subsequent transformations that these errors cannot occur. The context analysis is also necessary to guarantee the termination of later phases. Our context analysis can identify the following three types of errors:

1. Undefined variables.
2. Wrong selections.

3. Forbidden recursion.

The context analysis is performed for each Group hierarchy.

4.2.1 Undefined Variables

To find undefined variables we have to check all leaves of a hierarchy tree. All free variables, which are not elements of the union of the available variables ($\mathit{availVars}$) and the outer-context ($\mathit{outerVars}$) are undefined. To determine the set of free variables, we can use $\mathit{freePaths}$ as a path is free iff the variable is free.

A program must fulfill for all dependency graphs g:

$$\forall(\langle x, \ell\rangle) \in (\mathsf{leaves}\ g) : \forall x(\texttt{.}x_i)^{i\in 1..n} \in \mathit{freePaths}(\ell) : x \in (\mathit{availVars}(\ell) \cup \mathit{outerVars}(\ell))$$

4.2.2 Invalid Selection

The algorithm for adding a dependency edge will lead to one of the following two scenarios: Either the path was found completely or there is no suitable child in the current vertex. In the latter case, there is a rest of the selection chain we could not find. For all those dependency edges with a rest of a selection chain, we can check whether the head of this edge (the expression the other one depends on) is syntactically a Group.[1] In this case the selection is impossible at runtime, so we can reject the program.

A program must fulfill for all dependency graphs g:

$$\text{For } g = \langle vs, es\rangle : \forall(\texttt{depEdge}\ v\ \ \langle x, \ell\rangle\ \ (\texttt{.}x_i)^{i\in 1..n}) \in es : n = 0 \vee \mathit{exp}(\ell) \notin G$$

In the Group in Listing 4.4 the definition of `b` depends on the definition `A` as there is the Group-bound free path `A.e`. There is no definition representing the selection `.e`. As the right-hand side of the definition of `A` is a Group, the selection of `e` will always fail at runtime.

```
{ A = { c = ...
        d = ... }
  b = A.e }
```

Listing 4.4: Example for a wrong selection.

4.2.3 Forbidden Recursion

As mentioned, recursive definitions are only allowed for λ-abstractions. All programs containing forbidden recursive definitions are rejected. To identify these programs, the strongly connected components for all dependency graphs are generated. If there is at least one recursive SCC containing not only abstractions this program is rejected. This

[1]This could be extended to abstractions, numbers, etc. But as GLang is dynamically typed we only regard Group-expressions.

is the most important error to be detected as the following step (see Sec. 4.4) in the analysis does not terminate for some programs containing forbidden mutual recursion.

A program must fulfill for all dependency graphs g:

$$\forall(\mathsf{sccRec}\ labs) \in (\mathsf{calcScc}\ g) : \forall \ell \in labs : \mathit{exp}(\ell) \text{ is a } \lambda\text{-abstraction}$$

4.3 Additional Dependency Edges

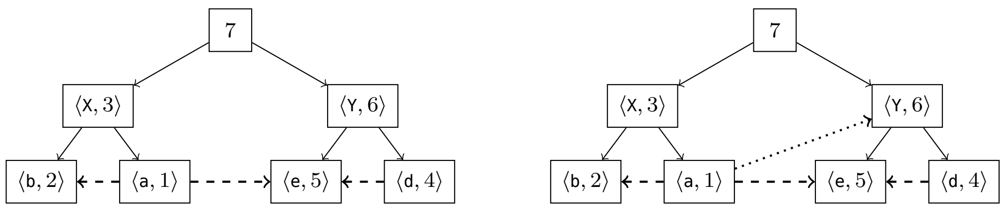
We want Groups to be evaluated in an interleaved fashion only if they are mutually recursive. If there are no dependencies between two definitions the ordering does not matter and the algorithm will return any order. The definition of the relation $\dashrightarrow$ does not include the dependencies on the Groups the definition selects from. Therefore, it can happen that the definitions of two Groups get mixed, although it would be possible to define an order for both Groups in isolation.

```
{ X = { a = [Y.d + e]^1
        b = 3^2 }^3
  Y = { e = 7^4
        d = e^5 }^6
}^7
```

Listing 4.5: Example for additional edges.

To illustrate this problem, we give an example of two Groups that are not mutually recursive in Listing 4.5. As the two Groups `X` and `Y` are not mutually recursive we can give an order for them: first `Y`, then `X`. The dependency graph using the relation $\dashrightarrow$ is shown in Fig. 4.6(a). As there are no dependencies between `X` and `Y`, the following order would be possible:

$\mathtt{sccSingle}\ 2,\ \mathtt{sccSingle}\ 5,\ \mathtt{sccSingle}\ 1,\ \mathtt{sccSingle}\ 3,\ \mathtt{sccSingle}\ 4,$
$\mathtt{sccSingle}\ 6,\ \mathtt{sccSingle}\ 7$



$$\begin{tikzcd}
 & & & 7 \arrow[dll] \arrow[drr] & & & \\
 & \langle X,3\rangle \arrow[dl] \arrow[dr] & & & & \langle Y,6\rangle \arrow[dl] \arrow[dr] & \\
\langle b,2\rangle & & \langle a,1\rangle \arrow[ll, dashed] \arrow[rr, dashed] & & \langle e,5\rangle & & \langle d,4\rangle \arrow[ll, dashed]
\end{tikzcd}$$

(a) Without additional edge ($\dashrightarrow$ relation).

$$\begin{tikzcd}
 & & & 7 \arrow[dll] \arrow[drr] & & & \\
 & \langle X,3\rangle \arrow[dl] \arrow[dr] & & & & \langle Y,6\rangle \arrow[dl] \arrow[dr] & \\
\langle b,2\rangle & & \langle a,1\rangle \arrow[ll, dashed] \arrow[rr, dashed] \arrow[urrr, dotted] & & \langle e,5\rangle & & \langle d,4\rangle \arrow[ll, dashed]
\end{tikzcd}$$

(b) With additional edge ($\cdots\blacktriangleright$ relation).

Figure 4.6: Dependency graph for the example in Listing 4.5.

In this order, the definitions of `X` and `Y` are mixed and therefore, the evaluation of both Groups will be interleaved. Another order is possible as well:

$$\texttt{sccSingle}\ 5,\ \texttt{sccSingle}\ 4,\ \texttt{sccSingle}\ 6,\ \texttt{sccSingle}\ 2,\ \texttt{sccSingle}\ 1,$$
$$\texttt{sccSingle}\ 3,\ \texttt{sccSingle}\ 7$$

This is the preferred order and we want to force it.

We want to schedule all Groups that the current Group depends on before the definitions of this Group. To achieve this, we extend the relation $\dashrightarrow$ to the relation *needs*. If D_1 needs D_2 we write $D_1 \mathrel{\cdots\!\blacktriangleright} D_2$.

$$\begin{aligned}\langle x_1, \ell_1\rangle \mathrel{\cdots\!\blacktriangleright} \langle x_2, \ell_2\rangle :\Leftrightarrow &(\langle x_1, \ell_1\rangle \dashrightarrow \langle x_2, \ell_2\rangle) \lor \\ &(\exists\, \langle x_3, \ell_3\rangle : \langle x_1, \ell_1\rangle \mathrel{\cdots\!\blacktriangleright} \langle x_3, \ell_3\rangle \land \langle x_2, \ell_2\rangle = \mathsf{parent}\ \ell_3 \\ &\land\ \mathit{name}(\ell_2) \not\sqsubseteq \ell_1)\end{aligned}$$

The relation $\mathrel{\cdots\!\blacktriangleright}$ is an extension of $\dashrightarrow$. Hence, if a definition $\langle x_1, \ell_1\rangle$ depends on a definition $\langle x_2, \ell_2\rangle$, $\langle x_1, \ell_1\rangle$ needs $\langle x_2, \ell_2\rangle$. Furthermore, if a definition $\langle x_1, \ell_1\rangle$ needs a definition $\langle x_2, \ell_2\rangle$, it also needs the parent of $\langle x_2, \ell_2\rangle$ unless this parent is an ancestor of $\langle x_1, \ell_1\rangle$. As $\varepsilon \sqsubseteq p$, there is no definition D, which needs the top-level Group of a hierarchy.

Based on the relation $\mathrel{\cdots\!\blacktriangleright}$, we can define mutual recursion of Groups. As for the SCCs based on the dependency relation, we have to consider the dependency of a Group on all its inner definitions as well, which is captured by the relation $\longrightarrow$. The relation $\mathrel{\overset{\cdots}{\longrightarrow}}$ is the union of both relations:

$$\langle x_1, \ell_1\rangle \mathrel{\overset{\cdots}{\longrightarrow}} \langle x_2, \ell_2\rangle :\Leftrightarrow \langle x_1, \ell_1\rangle \longrightarrow \langle x_2, \ell_2\rangle \lor \langle x_1, \ell_1\rangle \mathrel{\cdots\!\blacktriangleright} \langle x_2, \ell_2\rangle$$

Two Groups G_1 and G_2 are mutually recursive iff $G_1 \mathrel{\overset{\cdots}{\longrightarrow}}^+ G_2$ and $G_2 \mathrel{\overset{\cdots}{\longrightarrow}}^+ G_1$ where $\mathrel{\overset{\cdots}{\longrightarrow}}^+$ is the transitive closure of $\mathrel{\overset{\cdots}{\longrightarrow}}$.

We extend the dependency graph definition by a third kind of edge so we can distinguish between tree edges, dependency edges, and additional edges. The new set of all edges is defined as:

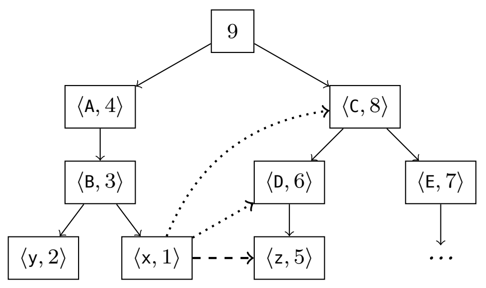
$$\begin{aligned}\mathit{Edge} :=\ &\{\texttt{treeEdge}\ v_1\ v_2 \mid v_1, v_2 \in \mathit{Vert}\} \\ \cup\ &\{\texttt{depEdge}\ v_1\ v_2\ \mathit{sc} \mid v_1, v_2 \in \mathit{Vert} \land \mathit{sc} \in \mathit{Sc}\} \\ \cup\ &\{\texttt{addEdge}\ v_1\ v_2 \mid v_1, v_2 \in \mathit{Vert}\}\end{aligned}$$

Similar to the normal dependency edges, we only add those additional edges starting in a leaf and we only add those not captured by a dependency edge. The dependency graph for the current example using the relation $\mathrel{\cdots\!\blacktriangleright}$ is presented in Fig. 4.6(b). The additional edges, resulting from the extended relation, are represented by dotted edges. In the graph, $\langle \mathtt{a}, 1\rangle$ does not only have an edge to the used term definition $\langle \mathtt{e}, 5\rangle$, but also to the surrounding Group `Y`.

The extended dependency graph still defines a partial order for the definitions and not a total order. Hence, the list of SCCs is not unique. For the example in Listing 4.5, the following two lists of SCCs are still possible:

```
{ A = { B = { x = C.D.z^1  y = 1^2 }^3 }^4
  C = { D = { z = 2^5 }^6
        E = { ... }^7
      }^8
}^9
```

(a) Listing.



(b) Dependency graph with additional edge.

Figure 4.7: Example for additional edges without mutually recursive Groups.

- `sccSingle` 5, `sccSingle` 4, `sccSingle` 6, `sccSingle` 2, `sccSingle` 1, `sccSingle` 3, `sccSingle` 7
- `sccSingle` 2, `sccSingle` 5, `sccSingle` 4, `sccSingle` 6, `sccSingle` 1, `sccSingle` 3, `sccSingle` 7

We know for each dependency graph:

$$G \dashrightarrow D \Rightarrow \exists G' : (G \dashrightarrow G' \wedge G' \longrightarrow D) \vee (G \longrightarrow^{+} D)$$

As a consequence, the following reordering for a corresponding list of SCCs is always possible: A definition D defined within a Group G' can be moved behind G in the given list of SCCs if Group G does not need Group G'. This allows to change the second list of SCCs from our example into the first list of SCCs. This property is used in the transformation to ensure that all definitions of a Group are combined into one component.

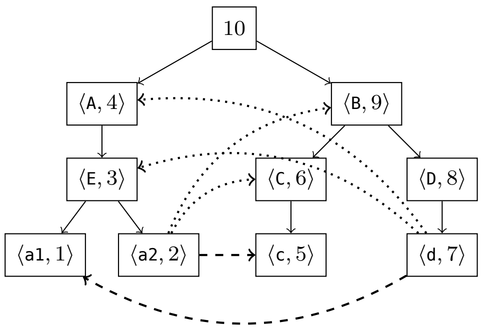
In the last example, there is only one surrounding Group. In the following, we consider mutually recursive Groups with more levels. We provide two examples with higher level distance—one without and one with mutually recursive Groups. In the example in Fig. 4.7 the definition $\langle \texttt{x}, 1\rangle$ depends on the definition $\langle \texttt{z}, 5\rangle$. The ellipsis in the Group `C.E` represents definitions without external dependencies. Therefore, the complete Group `C` can be evaluated before any of the definitions of Group `A`. As in the previous example, the additional edges achieve this. We add edges from $\langle \texttt{x}, 1\rangle$ to all surrounding Groups of `z` as shown in the dependency graph given in Fig. 4.7(b).

The necessity of the edge from $\langle \texttt{x}, 1\rangle$ to $\langle \texttt{C}, 8\rangle$ is clear as we want to force the complete Group `C` to be evaluated before $\langle \texttt{x}, 1\rangle$. Consequently, we have to add an edge to the surrounding Group with the shortest name. The need for the edges to the inner Groups will be motivated in the example given in Fig. 4.8. Here the two Groups `A` and `B` are mutually recursive, although the inner Groups `E`, `C` and `D` are not. We can give an evaluation order for these three Groups: `C`, `E`, `D`. The additional edges from $\langle \texttt{a2}, 2\rangle$ to $\langle \texttt{C}, 6\rangle$ and from $\langle \texttt{d}, 7\rangle$ to $\langle \texttt{E}, 3\rangle$ force this order.

We can summarize the behavior of the relation $\dashrightarrow$ as follows: A definition does not only need those definitions the Group-bound free paths resolve to but all surrounding Groups.

```
{ A = { E = { a1 = 1^{1}  a2 = B.C.c^{2} }^{3} }^{4}
  B = { C = { c = 2^{5} }^{6}
        D = { d = A.E.a1^{7} }^{8}
      }^{9}
}^{10}
```

(a) With mutually recursive Groups.



(b) With additional edge.

Figure 4.8: Example for additional edges with mutually recursive Groups.

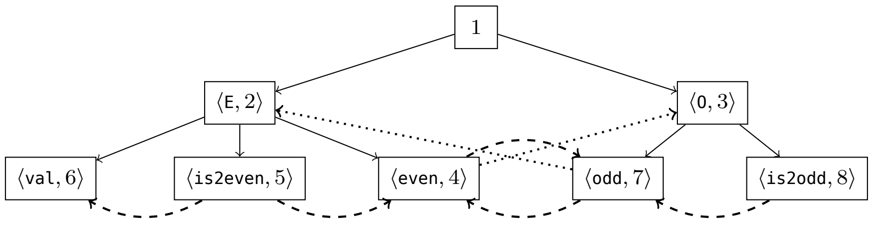


Figure 4.9: Dependency graph with additional edges for the example in Listing 3.7.

For each dependency edge, we can calculate the set of additional edges by the partial function additionals. The set of all additional edges for a graph is the union of all those sets.

$$\mathsf{additionals} : \; \mathit{Edge} \to \mathit{Graph} \to \mathcal{P}(\mathit{Edge})$$

$$\mathsf{additionals}\ (\mathtt{treeEdge}\ v_1\ v_2)\ g = \emptyset$$

$$\begin{aligned}
&\mathsf{additionals}\ e\ g = \\
&\qquad \begin{cases} \emptyset, & \text{if } \mathit{name}(\mathsf{vLabel}\ v) \sqsubseteq \mathit{name}(\mathsf{vLabel}\ v_1) \\ \{e_a\} \cup \mathsf{additionals}\ e_a\ g, & \text{otherwise} \end{cases} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{where} \quad \begin{aligned} e_a &= \mathsf{addEdge}\ v_1\ v \\ v &= \mathsf{parent}\ (\mathsf{vLabel}\ v_2) \\ v_1 &= \mathsf{eStart}\ e \\ v_2 &= \mathsf{eEnd}\ e \end{aligned}
\end{aligned}$$

In the running example, an edge from ⟨even, 4⟩ to ⟨O, 3⟩ and from ⟨odd, 7⟩ to ⟨E, 2⟩ must be added. The complete dependency graph is illustrated in Fig. 4.9.

The set of SCCs for an extended dependency graph are closely related to the set of SCCs of the original dependency graph. Each SCC of the extended graph is either an SCC of the original graph, or it is the union of two or more SCCs. There are two possibilities for a union of SCCs: Either the additional edges have introduced a new cycle, or an existing cycle is extended. If two Groups are mutually recursive although there are no mutually recursive functions, there is no cycle in the original graph but in the extended one. For example, in Fig. 4.8, the graph with additional edges has a new cycle containing the definitions with the labels: $\{9, 8, 7, 4, 3, 2, 9\}$. In the extended dependency graph mutually recursive Groups form a cycle. As a consequence, mutually recursive Groups are part of the same SCC. We want to define the evaluation order for these Groups. Therefore, these additional cycles are broken up by building hierarchical strongly connected components.

4.4 Hierarchical Strongly Connected Components

As mentioned previously, the additional edges lead to cycles, which have to be broken up into smaller components. For this purpose, we extend the definition of strongly connected components by a hierarchical SCC.

$$\mathit{HierScc} := \mathit{Scc} \cup \{\texttt{sccGrp}\ \mathit{labs}\ \overline{\mathit{scc}} \mid \mathit{labs} \subseteq \mathit{Lab} \wedge \overline{\mathit{scc}} \in \mathit{HierScc}^*\}$$

In this definition, we distinguish between two different kinds of recursive SCCs: an `sccRec` component contains only mutually recursive functions, i. e., allowed recursive definitions, and an `sccGrp` component representing mutually recursive Groups. An `sccGrp` component is defined by a set of all top Groups and a list of SCCs—an ordered list of all definitions of this SCC.

The function makeHierScc transforms a normal SCC into a hierarchical SCC.

$$\begin{aligned}
&\mathsf{makeHierScc} : \ \mathit{Graph} \to \mathit{Scc} \to \mathit{HierScc}\\
&\mathsf{makeHierScc}\ g\ (\texttt{sccSingle}\ \ell) = \texttt{sccSingle}\ \ell\\
&\mathsf{makeHierScc}\ g\ (\texttt{sccRec}\ \mathit{labs}) =\\
&\qquad\begin{cases}\texttt{sccRec}\ \mathit{labs}, & \text{if } \forall \ell \in \mathit{labs} : \mathit{exp}(\ell) \in T\\ \texttt{sccGrp}\ \mathit{labs}_{\mathit{tops}}\ \overline{\mathit{scc}}_2, & \text{otherwise}\end{cases}\\
&\qquad\quad \text{where} \quad \begin{aligned} g_1 &= \mathsf{subGraph}\ g\ \mathit{labs}\\ \mathit{labs}_{\mathit{tops}} &= \mathsf{tops}\ g_1\\ g_2 &= g_1 \setminus \{\texttt{addEdge}\ v_1\ v_2 \mid v_2 \in \mathit{labs}_{\mathit{tops}}\}\\ \overline{\mathit{scc}}_1 &= \mathsf{calcScc}\ g_2\\ \overline{\mathit{scc}}_2 &= \mathsf{map}\ (\mathsf{makeHierScc}\ g_2)\ \overline{\mathit{scc}}_1\end{aligned}
\end{aligned}$$

Only recursive SCCs containing at least one Group are transformed. The subgraph of the dependency graph containing the additional edges but only the vertices represented by this SCC is used. All additional edges ending in a top vertex are removed from this graph. For this graph, the list of SCCs is calculated. It is possible that there are recursive SCCs containing mutually recursive Groups within this list. Therefore, all elements are transformed with makeHierScc.

We prove that the time complexity of the function makeHierScc is in $O(n^3)$ in Sec. 4.4.1, where n represents the number of labels within a program.

To illustrate the creation of hierarchical SCCs, we return to our running example and

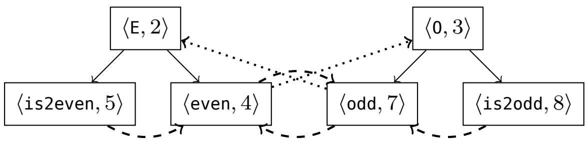


Figure 4.10: Subgraph for recursive SCC.

the extended dependency graph given in Fig. 4.9. The SCCs for the extended graph are:

sccSingle 6, sccRec $\{2, 3, 4, 5, 7, 8\}$, sccSingle 1

There is only on recursive SCC. The subgraph g_1 for this SCC is given in Fig. 4.10. The set of top vertices $labs_{tops}$ is $\{2, 3\}$. Both additional edges in this graph are ending in one of the top vertices, hence the graph g_2 contains no additional edges. The list of SCCs $\overline{scc}_1$ is:

sccRec $\{4, 7\}$, sccSingle 5, sccSingle 2, sccSingle 8, sccSingle 3

There is only one recursive SCC. All definitions of this SCC are term definitions. Hence the list $\overline{scc}_2$ is equal to $\overline{scc}_1$.

The complete list of hierarchical SCCs is:

```
sccSingle 6
sccGrp {2, 3}
    (sccRec {4, 7}, sccSingle 5, sccSingle 2, sccSingle 8, sccSingle 3)
sccSingle 1
```

4.4.1 Time Complexity

In the following, we analyze the worst-case time complexity for the calculation of the hierarchical SCCs. The time complexity for Sharir's algorithm is known to be in $O(n^2)$, where n is the number of vertices in the graph.

The time function for makeHierScc can be estimated by T_{hier} where n represents the number of labels within the SCC. Each SCC is split into k SCCs by the function calcScc. In the case that calcScc is not called, we assume $k = 0$.

The application of the map function can be expressed as a sum of all calls of the given function to all the elements of the list. All SCCs have a disjoint set of definitions. It follows: $n = \sum_{i=1}^{k} |scc_i|$. In the following we use the abbreviation n_i for $|scc_i|$. As each SCC contains at least one vertex, we know: $\forall i \in 1..k : 1 \leq n_i \leq n$.

$$T_{hier}(n, k) = \begin{cases} c_1, & \text{if } n = 1 \\ c_2, & \text{if } k = 0 \\ T_{\mathsf{subGraph}}(n) + T_{\mathsf{tops}}(n) + \\ \qquad T_{\mathsf{calcScc}}(n) + \sum_{i=1}^{k}(T_{hier}(n_i, k_i)), & \text{otherwise} \end{cases}$$

The calculations of the subgraph, the top vertices and the removal of the additional edges are all in $O(n^2)$. Furthermore, the concrete value of a constant does not matter in O-notation, hence we assume the same constant c for all constant values. We redefine the time function:

$$T_{hier}(n,k) = \begin{cases} c, & \text{if } n = 1 \lor k = 0 \\ cn^2 + \sum_{i=1}^{k}(T_{hier}(n_i,k_i)), & \text{otherwise} \end{cases}$$

For a worst case analysis, we first show that in each step the component is split into at least two components ($k \geq 2$) and afterwards we show that $k = 2$ is the worst case for k.

Theorem 4.4.1. *For an* (`sccRec` *labs*) *containing at least one Group the graph* g_2 *has at least two SCCs or the context analysis rejects the program.* g_2 *is defined as follows:*

$$\begin{aligned} g_2 &= g_1 \setminus \{\mathsf{addEdge}\ v_1\ v_2 \mid v_2 \in labs_{tops}\} \\ &\quad \textit{where} \quad g_1 = \mathsf{subGraph}\ g\ labs \\ &\quad\quad\quad\quad labs_{tops} = \mathsf{tops}\ g_1 \end{aligned}$$

Proof. To prove Thm. 4.4.1, we do not prove $A \lor B$, but $\neg A \to B$. In our case, A is “at least two SCCs” and B is “context analysis rejects program.”

We show that if all vertices of the graph g_2 are in one SCC, then there is a cycle containing only tree and dependency edges and at least one Group definition. As a consequence, the context analysis rejects this program, because the original dependency graph contains this cycle as well.

For this proof we define an edge path as a sequence of vertices ${v_i}^{i \in 1..n}$, where $n > 1$ and $\forall i \in 1..(n-1) : \exists e : \mathsf{eStart}\ e = v_i \land \mathsf{eEnd}\ e = v_{i-1}$. As abbreviation for edge paths we use $v_1 \mathrel{\cdots\twoheadrightarrow} v_n$ for an edge path possibly containing an additional edge and $v_1 \mathrel{\text{--}\twoheadrightarrow} v_n$ for an edge path without any additional edges. Additionally, we use the abbreviation $v_1 \mathrel{\cdots\twoheadrightarrow^*} v_n$, for a possibly empty edge path—a sequence of vertices of length 1.

Furthermore, we use the partial function `cycle`, which returns an edge path $v \mathrel{\text{--}\twoheadrightarrow} v$. This edge path is a cycle without an additional edge.

$$\begin{aligned}
&\mathsf{cycle} : \ v^* \to v^* \to v^* \\
&\mathsf{cycle}\ v_1\ v_1 \mathrel{\text{--}\twoheadrightarrow} v_1 = v_1 \mathrel{\text{--}\twoheadrightarrow} v_1 \\
&\mathsf{cycle}\ (v_1 \mathrel{\cdots\twoheadrightarrow^*} v_{n-1} \longrightarrow v_n)\ (v_n \mathrel{\text{--}\twoheadrightarrow} v_1) = \mathsf{cycle}\ (v_1 \mathrel{\cdots\twoheadrightarrow^*} v_{n-1})\ (v_{n-1} \longrightarrow v_n \mathrel{\text{--}\twoheadrightarrow} v_1) \\
&\mathsf{cycle}\ (v_1 \mathrel{\cdots\twoheadrightarrow^*} v_{n-1} \dashrightarrow v_n)\ (v_n \mathrel{\text{--}\twoheadrightarrow} v_1) = \mathsf{cycle}\ (v_1 \mathrel{\cdots\twoheadrightarrow^*} v_{n-1})\ (v_{n-1} \dashrightarrow v_n \mathrel{\text{--}\twoheadrightarrow} v_1) \\
&\mathsf{cycle}\ (v_1 \mathrel{\cdots\twoheadrightarrow^*} v_{n-1} \mathrel{\cdots\rightarrow} v_n)\ (v_n \mathrel{\text{--}\twoheadrightarrow} v_1) = \\
&\qquad\qquad \begin{cases} v \longrightarrow v_n \mathrel{\text{--}\twoheadrightarrow} v, & \text{if } v \in v_n \mathrel{\text{--}\twoheadrightarrow} v_1 \\ \mathsf{cycle}\ (v_1 \mathrel{\cdots\twoheadrightarrow} v)\ (v \longrightarrow v_n \mathrel{\text{--}\twoheadrightarrow} v_1), & \text{if } v \notin v_n \mathrel{\text{--}\twoheadrightarrow} v_1 \quad (*) \\ \qquad \text{where} \quad v_1 \mathrel{\cdots\twoheadrightarrow} v = \text{any edge path from } v_1 \text{ to } v & \end{cases} \\
&\quad \text{where} \quad v = \mathsf{parent}\ v_n
\end{aligned}$$

First, we explain the behavior of this partial function and afterwards, we show that it is well defined.

The partial function cycle takes two sequences of vertices. The first is a potentially empty edge path from vertex v_1 to vertex v_n. The second sequence forms an edge path from vertex v_n to v_1 without any additional edges. For all other cases the function is not defined. The function distinguishes between four cases:

- In the first case, the first edge path is empty. Here v_n must be v_1. The second argument is returned, as it is a cycle containing no additional edges. All other three cases differ in the last edge of the first edge path.
- In the case of a tree or a dependency edge, the second path is extended by this edge and the function continues with the rest of the edge path.
- In the case of an additional edge there are two different cases: If the parent of v_n is part of the second edge path, the function returns the cycle starting in the parent followed by the tree edge to v_n followed by the start of the second edge path up to the parent. If the parent of v_n is not part of the second edge path, the function cycle is called with any edge path from v_1 to the parent v. This path exists as all vertices are within one SCC.

To proof Thm. 4.4.1, we use the partial function cycle. To use this partial function, the two initial edge paths are needed. Furthermore, we have to prove that in all calls the arguments match one of the defined cases, that all cases are well defined and that the function terminates. For the proof, we show that for each call the two arguments fulfill:

1. The first vertex of the first edge path is equal to the last vertex of the second edge path.
2. The first vertex of the second edge path is equal to the last vertex of the first edge path.
3. The second edge path is not empty.
4. The second edge path has no additional edges.
5. All edges of both edge paths are within the graph g_2.

From this invariant it follows directly that one of the defined cases matches and that all cases are well defined always.

In the initial step of the construction we take any vertex $v_1 \in vs_{tops}$. All definitions are in one SCC, hence $\exists v_1 \twoheadrightarrow v_1$. The last edge in this edge path must be a dependency edge, as $v \in vs_{tops}$, there is no additional edge ending in v_1. We can split this edge path: $v_1 \twoheadrightarrow^* v_n \dashrightarrow v_1$. The initial call is cycle $(v_1 \twoheadrightarrow^* v_n)$ $(v_n \dashrightarrow v_1)$.

For these two arguments, the invariant holds. The first two calls within the function cycle fulfill this invariant.

We have a closer look at the last call. The last edge is an additional edge. A need edge always ends in a Group. Furthermore, in g_2 there is no need edge ending in a top vertex. Therefore, there must be a parent v of v_n within g_2. It follows that the tree edge $v \longrightarrow v_n$ is in g_2. As all vertices are within one SCC, there must be an edge path from v_1 to v.

Furthermore, we show that all cycles returned by the function contain only edges within g_2. In the first case, this follows directly from part 5 of the invariant. In the second case, there must be a parent v_n in g_2, and therefore, the tree edge is in the graph as well.

Additionally, we show, that each cycle contains at least one Group. There are two possibilities for a returned cycle. In the first case, this Group is v_1. As $v_1 \in vs_{tops}$ v_1 is a Group. In the second case, this Group is v. As v is a parent of another vertex it must be a Group.

This construction terminates, as the set of vertices is finite: As a consequence the condition (*) can only be satisfied finitely many times. At some point, parent v_n will return a vertex v that is already part of the second edge path and the function returns a cycle. In all cases, the construction terminates with a cycle not containing any additional edge. Therefore, the analysis either rejects the program or the graph g_2 is split into at least two SCCs. □

In each recursive call of the function makeHierScc the steps for the calculation is either constant or cn^2. From Thm. 4.4.1 it follows that in each call the graph is split into at least two SCCs. According to Lemma 4.4.2, the splitting of the graph into two SCCs with one containing a single definition and the other one the remainder is the worst case.

Lemma 4.4.2. *$\forall i \in 1..k : 1 \leq n_i$, and $k \geq 2$:*

$$\sum_{i=1}^{k} n_i^2 \leq ((\sum_{i=1}^{k} n_i) - 1)^2 + 1$$

Proof. We prove by induction on k.

Base case: $k = 2$

Proof by cases:

- $n_1 = 1$:

$$\sum_{i=1}^{2} n_i^2 = 1 + n_2^2 = (1 + n_2 - 1)^2 + 1$$

- $n_2 = 1$: similar to $n_1 = 1$
- $n_1 > 1, n_2 > 1$:

$$\begin{aligned}\sum_{i=1}^{2} n_i^2 &= n_1^2 + n_2^2 \\ &= n_1^2 + n_2^2 - 2n_1 + 2n_1 - 2n_2 + 2n_2 \\ &\leq n_1^2 + n_2^2 - 2n_1 + n_2n_1 - 2n_2 + n_1n_2 + 2 \\ &= n_1^2 + n_2^2 - 2n_1 + 2n_2n_1 - 2n_2 + 2 \\ &= (n_1 + n_2 - 1)^2 + 1\end{aligned}$$

Induction hypothesis for $k = t$ holds $\sum\limits_{i=1}^{k} n_i^2 \leq ((\sum\limits_{i=1}^{k} n_i) - 1)^2 + 1$:
Induction step $k = t + 1$:

$$\begin{aligned}
\sum_{i=1}^{t+1} n_i^2 &= n_{t+1}^2 + \sum_{i=1}^{t} n_i^2 \\
&\leq n_{t+1}^2 + (\sum_{i=1}^{t} n_i - 1)^2 + 1 && \text{(IH)} \\
&\leq n_{t+1}^2 + 2n_{t+1}(\sum_{i=1}^{t} n_i - 1) + (\sum_{i=1}^{t} n_i - 1)^2 + 1 \\
&= (n_{t+1} + \sum_{i=1}^{t} n_i - 1)^2 + 1 \\
&= (\sum_{i=1}^{t+1} n_i - 1)^2 + 1 \quad \square
\end{aligned}$$

We use Lemma 4.4.2 to define a the worst case time function independent of k. As $\sum_{i=1}^{k} n_i = n$ it holds:

$$\sum_{i=1}^{k} n_i^2 \leq (n-1)^2 + 1$$

And as c is always positive, it holds as well:

$$c\sum_{i=1}^{k} n_i^2 \leq c(n-1)^2 + c$$

In the case $n = 1$, the time function is constant, hence we can assume c to be the call of the time function with $n = 1$. The new time function for makeHierScc is:

$$T_{hier}(n) = \begin{cases} c, & \text{if } n = 1 \\ cn^2 + T_{hier}(n-1) + T_{hier}(1), & \text{otherwise} \end{cases}$$

This time function is equal to $c\sum_{i=1}^{n} i^2$. The sum is the square pyramid number, which is equal to $n(n+1)(2n+1)/6$. It follows that makeHierScc is in $O(n^3)$.

4.4.2 Complete Analysis

The evaluation order for a Group hierarchy is established by the function calcHierScc.

$$\begin{aligned}
&\mathsf{calcHierScc} : \ G \to \mathit{HierScc}^* \\
&\mathsf{calcHierScc}\ G = \\
&\qquad \begin{cases} \bot, & \text{if context analysis rejects } g \\ \mathsf{map}\ (\mathsf{makeHierScc}\ g_1)\ (\mathsf{calcScc}\ g_1), & \text{otherwise} \end{cases} \\
&\quad \text{where} \quad g_{tree} = \mathsf{calcTree}\ G \\
&\qquad\qquad g = \mathsf{calcDepGraph}\ g_{tree} \\
&\qquad\qquad \langle es, vs \rangle = g \\
&\qquad\qquad g_1 = \left\langle es \cup \bigcup_{e \in es} (\mathsf{additionals}\ e\ g), vs \right\rangle
\end{aligned}$$

First of all, the hierarchy tree is built up. Next, a dependency graph is created by inserting the dependency edges into this tree. For all correct programs, in the sense of the context analysis, a list of SCCs is calculated for the graph containing all additional edges. The function makeHierScc is applied to all elements of this list to build up a list of hierarchical SCCs. The results of the function calcHierScc for all Group hierarchies is stored in the mapping $\mathit{hierScc}$, which is used in the transformation described in the next chapter.

5 Transformation

In this chapter, we describe how the results of the special dependency analysis are used to transform the input program into an intermediate representation. This intermediate representation encodes the evaluation order for Groups explicitly. We define the intermediate representation in Fig. 5.1.

$$\begin{array}{lcl} I & ::= & C \mid S \\ S & ::= & \lambda\, x \,.\, I \mid x \mid I \,.\, x \mid \ldots \\ C & ::= & \texttt{DEF}\ p\ S \mid \texttt{RECDEF}\ (p\ S)+ \mid \texttt{SEQ}\ p\ C^* \end{array}$$

Figure 5.1: Abstract syntax with explicit dependency order.

We keep the splitting into terms and Groups: every expression is either a term S or a component C. In this syntax, Groups are represented by a sequence `SEQ` of definitions in dependency order. Every `SEQ` holds the name of the represented Group and the list of components representing the definitions. Mutually recursive Groups are combined in nested `SEQ` components.

A component is either a list of mutually recursive function definitions (`RECDEF`), a non-recursive definition (`DEF`), or a nested sequence (`SEQ`).

In the following, we define the transformation from the input program into the intermediate language.

All correct GLang programs are transformed into the intermediate representation. The transformation is based on the dependency order described in Chap. 4. Each Group hierarchy is transformed separately into a sequence containing all its definitions in dependency order.

The transformation expects as correct programs (in the sense of the context analysis of Sec. 4.2) as input and requires the mapping *hierScc*—the result of the dependency analysis—for the given program.

The intermediate representation is used to define the semantics of GLang. The transformation must fulfill three properties:

1. The transformed program must contain exactly the definitions of the original program.

2. The components must be sorted correctly. If a definition D_1 depends on a definition D_2, then D_2 must be represented by a component above the component representing D_1. The correct ordering ensures the correct access of variables and selectors.

$$\mathcal{T}^E : E \to I \qquad \mathcal{T}^G : G \to C \qquad \mathcal{T}^T : T \to S$$

$$\begin{array}{ll}
\mathcal{T}^E[\![T]\!] & = \mathcal{T}^T[\![T]\!] \\
\mathcal{T}^E[\![G]\!] & = \mathcal{T}^G[\![G]\!] \\
\mathcal{T}^T[\![\lambda x\,.\,E]\!] & = \lambda x\,.\,\mathcal{T}^E[\![E]\!] \\
\mathcal{T}^T[\![E_1\ E_2]\!] & = (\mathcal{T}^E[\![E_1]\!])\ (\mathcal{T}^E[\![E_2]\!]) \\
\mathcal{T}^T[\![E\,.\,x]\!] & = \mathcal{T}^E[\![E]\!]\,.\,x \\
\mathcal{T}^T[\![x]\!] & = x \\
\mathcal{T}^T[\![c]\!] & = c \\
\mathcal{T}^G[\![\texttt{\{}(x_i = E_i^{\ell_i})^{i\in 1..n}\texttt{\}}^{\ell}]\!] & = \mathtt{SEQ}\ \varepsilon\ (\mathsf{sortDefs}\ \mathit{hierScc}(\ell)\ \{\ell_i^{i\in 1..n}\}\ \emptyset)
\end{array}$$

Figure 5.2: Transformation into intermediate representation.

3. A Group that is not mutually recursive with other Groups is represented in only one `SEQ`. In the evaluation of a `SEQ` all inner components are evaluated in sequence. Consequently, the definitions of two Groups are only evaluated in an interleaved fashion in case they are mutually recursive.

The precise definitions of these properties are given in Sec. 5.2.

5.1 Transformation Function

Each GLang expression is transformed by the transformation functions $\mathcal{T}^E$, $\mathcal{T}^T$, and $\mathcal{T}^G$ defined in Fig. 5.2.

All terms are transformed into the corresponding term of the intermediate representation, where inner expressions are transformed as well. $\mathcal{T}^G$ transforms a Group hierarchy as a whole into a `SEQ` component. The name of this component is ε as this component represents the top-level Group of this hierarchy. The inner definitions of the top-level Group are ordered and transformed by the function sortDefs. The transformation is performed top-down considering the list of hierarchical SCCs given by the map *hierScc* (see Sec. 4.4.2). sortDefs keeps track of the definitions to be transformed and ordered in two disjoint sets. The first set $labs_d$ represents the inner definitions to be transformed and the second set $labs_r$ represents mutually recursive modules to be transformed together. The role of those two sets is discussed in more detail at a later stage. For the top-level Group the second set is empty and the first set represents all inner definitions.

The function sortDefs builds up the list of sub-components. We start with a short overview of the possible cases in the function sortDefs and discuss them in detail afterwards. If the set of sub-definitions—represented by the two sets of labels—is empty the result is the empty list.

If the sets of labels are not empty, the function findLast defined in Fig. 5.3 searches

$$\textsf{findLast} : \ Scc^* \to \mathcal{P}(Lab) \to \langle Scc^*, Scc\rangle$$

$$\textsf{findLast}\ {scc_i}^{i\in1..n}\ labs = \begin{cases} \left\langle {scc_i}^{i\in1..(n-1)}, scc_n\right\rangle, & \text{if } (\textsf{labels}\ scc_n)\cap labs \neq \emptyset \\ \textsf{findLast}\ {scc_i}^{i\in1..(n-1)}\ labs, & \text{otherwise}\end{cases}$$

Figure 5.3: Finding last element in list of SCCs.

$$\textsf{sortDefs} : \ Scc^* \to \mathcal{P}(Lab) \to C^*$$

$$\begin{array}{lrcll}
\textsf{sortDefs}\ \overline{scc}\ \emptyset & & = & \Diamond & \\
\textsf{sortDefs}\ \overline{scc}\ labs_d & & = & (\textsf{sortDefs}\ \overline{scc}_1\ labs_{d1}) +\!\!+\, C & (\dagger) \\
\quad\text{where} & labs_{d1} & = & labs_d \setminus (\textsf{labels}\ scc) & \\
 & C & = & \textsf{trsfScc}\ scc\ \overline{scc}_1 & (\S) \\
 & \langle \overline{scc}_1, scc\rangle & = & \textsf{findLast}\ \overline{scc}\ labs_d &
\end{array}$$

Figure 5.4: Simplified version of function sortDefs.

the last element scc in $\overline{scc}$ carrying a label contained in $labs_d$ or $labs_r$. $\overline{scc}$ is split: $\overline{scc} = \overline{scc}_1 +\!\!+\, scc +\!\!+\, \overline{scc}'$. $\overline{scc}_1$ is the prefix of the list $\overline{scc}$ ending one element before scc. The rest $\overline{scc}'$ is dropped. findLast implements the allowed reordering of SCCs described in Sec. 4.3.

Since the two sets $labs_d$ and $labs_r$ are always disjoint, two cases are possible: If there is at least one label of scc in $labs_d$, sortDefs has to transform an inner definition. If the label of scc is in $labs_r$, a mutually recursive Group must be transformed. The transformation of an scc is done by the function trsfScc.

We describe the details of both cases separately.

5.1.1 Without Recursive Groups

We start with explaining the transformation of Groups that are not mutually recursive with other Groups. Therefore, the list $\overline{scc}$ contains only `sccSingle` and `sccRec` and no `sccGrp` components and we can ignore the set $labs_r$. For convenience, we give a simplified version of the functions sortDefs and trsfScc in Fig. 5.4 and Fig. 5.5. In the simplified version of function sortDefs the last argument is omitted, since this set is only necessary for mutually recursive Groups.

All current inner definitions $labs_d$ must be transformed and ordered. The list is built up starting with the last component. Initially, the last component in the list $\overline{scc}$ which has a label contained in $labs_d$ is searched. The function findLast defined in Fig 5.3 selects the last element scc containing a label within $labs$ and returns the prefix of $\overline{scc}$ up to this scc as well. scc represents an inner definition, which is transformed by the function trsfScc (§ in Fig. 5.4).

Returning to our first example from Sec. 4.3 helps to illustrate the transformation. The

$$\mathsf{trsfScc} : \; Scc \to Scc^* \to C$$

$$\mathsf{trsfScc}\ scc\ \overline{scc} \quad = \begin{cases} \texttt{RECDEF}\ \big(\mathit{name}(\ell_i)\ \ \mathcal{T}^T[\![\mathit{exp}(\ell_i)]\!]\big)^{i\in 1..n}, & \text{if } scc = \texttt{sccRec}\ \{\ell_i^{i\in 1..n}\} \\ \texttt{DEF}\ \mathit{name}(\ell)\ \mathcal{T}^T[\![\mathit{exp}(\ell)]\!], & \text{if } scc = \texttt{sccSingle}\ \ell \wedge \mathit{exp}(\ell) \in T \\ \texttt{SEQ}\ \mathit{name}(\ell)\ (\mathsf{sortDefs}\ \overline{scc}\ \{\ell_i^{i\in 1..n}\}), & \text{if } scc = \texttt{sccSingle}\ \ell \\ & \quad \wedge\ \mathit{exp}(\ell) = \texttt{\{}(x_i \texttt{=} E_i^{\ell_i})^{i\in 1..n}\texttt{\}} \end{cases}$$

Figure 5.5: Simplified version of function trsfScc.

```
{ x = { a = [y.d + b]¹
        b = 3²  }³
  y = { e = 7⁴
        d = e⁵  }⁶
}⁷
```

Listing 5.6: Example for additional edges.

code is repeated in Listing 5.6.

The transformation function leads to:

$$\mathcal{T}^G[\![\texttt{\{ x = \{ ... \} y = \{ ... \} \}}]\!] = \texttt{SEQ}\ \varepsilon\ (\mathsf{sortDefs}\ \mathit{hierScc}(7)\ \{3,6\})$$

This leads to a call of sortDefs with the following arguments:

$$\overline{scc} = \texttt{sccSingle}\ 2,\ \texttt{sccSingle}\ 4,\ \texttt{sccSingle}\ 5,\ \texttt{sccSingle}\ 6,\ \texttt{sccSingle}\ 1,$$
$$\texttt{sccSingle}\ 3,\ \texttt{sccSingle}\ 7$$
$$labs_d = \{3,6\}$$

The set $\{3,6\}$ represents the inner definitions `x` and `y` of the top-level Group.

The last component in $\overline{scc}$ representing one of these two labels is `sccSingle` 3. trsfScc is called for this SCC with the following arguments:

$$scc = \texttt{sccSingle}\ 3$$
$$\overline{scc} = \texttt{sccSingle}\ 2,\ \texttt{sccSingle}\ 4,\ \texttt{sccSingle}\ 5,\ \texttt{sccSingle}\ 6,\ \texttt{sccSingle}\ 1$$

The list $\overline{scc}_1$ is the complete list up to this SCC.

The function trsfScc transforms an SCC into a C-expression. The first argument is the SCC to be transformed. The second one is the complete list $\overline{scc}$ up to scc. In the simplified version, there is no case for an `sccGrp`, as they are only necessary for mutually recursive Groups.

If scc is an `sccRec`, all definitions within this scc are functions. The C expression is a recursive component containing all transformed definitions of scc.

If *scc* is an `sccSingle`, the definition is either a Group or a term. In the case of a term, the component is transformed into `DEF` containing the name of the definition and the transformed term. In the case of a Group, the function returns, similar as for the top-level Group, a `SEQ`-expression for which the list of inner components is the ordered list of transformed inner definitions. This list is calculated via sortDefs using $\overline{scc}$ and the labels of the inner definitions of this Group.

In our current example, the label 3 represents a Group. Therefore, the function trsfScc returns a `SEQ`-component. The name of this `SEQ`-component is its name in the hierarchy retrieved from the map *name*. The inner definitions of this Group must be transformed and ordered, which is done by the recursive call of sortDefs. The arguments for this recursive call are:

$$\begin{aligned}\overline{scc} &= \texttt{sccSingle}\ 2,\ \texttt{sccSingle}\ 4,\ \texttt{sccSingle}\ 5,\ \texttt{sccSingle}\ 6,\ \texttt{sccSingle}\ 1\\ labs_d &= \{1,2\}\end{aligned}$$

The last component for this call is `sccSingle` 1. As the expression represented by 1 is a non-recursive term, trsfScc returns `DEF x.a (y.d + b)`. The recursive call of sortDefs (†) builds up the beginning of the list.

All definitions within the component *scc* are removed from the set $labs_d$. The set of definitions is determined by the function labels.

The list $\overline{scc}_1$ is the list $\overline{scc}$ up to the transformed component *scc*. If the set of labels is empty, the list is complete.

In our example, the remaining set of labels is $\{2\}$. So the next *scc* is `sccSingle` 2, which is transformed into `DEF x.b 3`. The next call of sortDefs is with an empty set of labels, and therefore, the list of children is complete. The result of the transformation of `sccSingle` 3 is:

```
SEQ x (DEF x.b 3, DEF x.a (y.d + b))
```

The transformed Group `x` is complete and without any other definitions in between.

The first recursive invocation of sortDefs has the following arguments:

$$\begin{aligned}\overline{scc} &= \texttt{sccSingle}\ 2,\ \texttt{sccSingle}\ 4,\ \texttt{sccSingle}\ 5,\ \texttt{sccSingle}\ 6,\ \texttt{sccSingle}\ 1\\ labs_d &= \{6\}\end{aligned}$$

The last SCC is `sccSingle` 6. The next call of sortDefs is for the following arguments:

$$\begin{aligned}\overline{scc} &= \texttt{sccSingle}\ 2,\ \texttt{sccSingle}\ 4,\ \texttt{sccSingle}\ 5\\ labs_d &= \{4,5\}\end{aligned}$$

For the transformation of an `sccSingle` representing a Group, only the inner definitions of this Group are ordered and transformed, hence SCCs within this list, which are not representing any subdefinition of the current Group, are ignored. For example, the `sccSingle` 2 within the list of SCCs is not transformed within the current call of sortDefs.

The complete intermediate representation for the example is given in Listing 5.7.

After the discussion of the transformation without mutually recursive Groups, we turn now to the transformation of a mutually recursive Group.

```
SEQ (SEQ y (DEF y.e 7, DEF y.d e),
     SEQ x (DEF x.b 3, DEF x.a (y.d + b)))
```

Listing 5.7: Example for additional edges.

5.1.2 Mutually Recursive Groups

The transformation must combine mutually recursive Groups. A set of mutually recursive Groups is combined within one `sccGrp` in the list of SCCs.

We have to extend the function **trsfScc** for this case:

$$
\begin{array}{l}
\mathsf{trsfScc}\ scc\ \overline{scc} \quad = \\
\quad \texttt{SEQ}\ \mathit{name}(\ell)\ (\mathsf{sortDefs}\ \overline{scc}_1\ labs_{d1}\ labs_{r1}),\ \text{ if } scc = \texttt{sccGrp}\ labs_{tops}\ {scc_j}^{j\in 1..m} \\
\qquad\qquad\qquad\qquad \text{where} \qquad \begin{aligned}
\texttt{sccSingle}\ \ell &= scc_m \\
\{(x_i = E_i^{\ell_i})^{i\in 1..n}\} &= \mathit{exp}(\ell) \\
\overline{scc}_1 &= \overline{scc} +\!\!+\ {scc_j}^{j\in 1..(m-1)} \\
labs_{d1} &= \{\ell_i^{i\in 1..n}\} \\
labs_{r1} &= labs_{tops} \setminus \{\ell\}
\end{aligned}
\end{array}
$$

The transformation creates one `SEQ`-component representing one of the top Groups of the `sccGrp`.

An `sccGrp` combines a set of mutually recursive Groups. The labels of the top Groups are given in the set $labs_{tops}$ and the order for theses Groups and their inner definitions are given by the list of inner SCCs (${scc_j}^{j\in 1..m}$). All top Groups are represented by an `sccSingle` in this list. The last SCC (scc_m) in this list represents always one of the top Groups. This SCC is transformed into a `SEQ`-component containing all other top Groups and all inner definitions of the Group represented by scc_m.

The remaining top Groups and the inner definitions are sorted based on a new list of SCCs. This new list starts with the old list of SCCs up to the current component and is extended by the list given in the `sccGrp`. The last component is already transformed, so it is dropped.

The other top Groups and the inner definitions of the Group represented by scc_m are combined in the list of sub-components of this `SEQ`. The definitions of all top Groups are stored in additional argument $labs_{r1}$. This set was empty in the simple case, as there were no mutually recursive Groups.

To show our line of thought we employ our running even-odd example. The arguments for the initial call of sortDefs are:

$$\begin{aligned}
\overline{scc} &= \texttt{sccSingle}\ 6,\\
&\quad\ \texttt{sccGrp}\ \{2,3\}\\
&\qquad\quad (\texttt{sccRec}\ \{4,7\},\ \texttt{sccSingle}\ 5,\ \texttt{sccSingle}\ 2,\ \texttt{sccSingle}\ 8,\ \texttt{sccSingle}\ 3)\\
&\quad\ \texttt{sccSingle}\ 1\\
labs_d &= \{2,3\}\\
labs_r &= \emptyset
\end{aligned}$$

The last SCC in $\overline{scc}$ representing one of the given labels, is the `sccGrp`. $labs_r$ is empty, so the second case of sortDefs applies. The values for all needed variables are:

$$\begin{aligned}
\overline{scc}_1 &= \texttt{sccSingle}\ 6\\
scc &= \texttt{sccGrp}\ \{2,3\}\\
&\qquad\quad (\texttt{sccRec}\ \{4,7\},\ \texttt{sccSingle}\ 5,\ \texttt{sccSingle}\ 2,\ \texttt{sccSingle}\ 8,\ \texttt{sccSingle}\ 3)\\
labs_{d1} &= \emptyset\\
C &= \mathsf{trsfScc}\ scc\ \overline{scc}_1\ \emptyset\ \emptyset
\end{aligned}$$

Both labels in $labs_d$ are within the `sccGrp`, hence in the recursive call of sortDefs both label sets are empty. Therefore, the result is a list of length one. The only component is the result of the transformation of the found SCC. The arguments for the call of trsfScc are:

$$\begin{aligned}
scc &= \texttt{sccGrp}\ \{2,3\}\\
&\qquad\quad (\texttt{sccRec}\ \{4,7\},\ \texttt{sccSingle}\ 5,\ \texttt{sccSingle}\ 2,\ \texttt{sccSingle}\ 8,\ \texttt{sccSingle}\ 3)\\
\overline{scc} &= \texttt{sccSingle}\ 6
\end{aligned}$$

The given SCC is an `sccGrp` and consequently, the new case matches. The needed arguments for this case are:

$$\begin{aligned}
\ell &= 3\\
\overline{scc}_1 &= \texttt{sccSingle}\ 6,\ \texttt{sccRec}\ \{4,7\},\ \texttt{sccSingle}\ 5,\ \texttt{sccSingle}\ 2,\ \texttt{sccSingle}\ 8\\
labs_{d1} &= \{7,8\}\\
labs_{r1} &= \{2\}
\end{aligned}$$

In our example, the last element scc_m in the inner list of SCCs is `sccSingle` 3—the recursive Group `0`. The list of sub-components is created by the call of sortDefs. All definitions of `0` are added to the set of inner definitions, hence the arguments for this call of sortDefs are:

$$\begin{aligned}
\overline{scc} &= \texttt{sccSingle}\ 6,\ \texttt{sccRec}\ \{4,7\},\ \texttt{sccSingle}\ 5,\ \texttt{sccSingle}\ 2,\ \texttt{sccSingle}\ 8\\
labs_d &= \{7,8\}\\
labs_r &= \{2\}
\end{aligned}$$

$$\mathsf{sortDefs} : \ \mathit{Scc}^* \rightarrow \mathcal{P}(\mathit{Lab}) \rightarrow \mathcal{P}(\mathit{Lab}) \rightarrow C^*$$

$$\begin{array}{l}
\mathsf{sortDefs}\ \overline{scc}\ \emptyset\ \emptyset = \Diamond \\[1ex]
\mathsf{sortDefs}\ \overline{scc}\ labs_d\ labs_r = \\
\qquad \begin{cases}
(\mathsf{sortDefs}\ \overline{scc}_1\ labs_{d1}\ labs_r) \mathbin{+\!\!+} C, & \text{if } (\mathsf{labels}\ scc) \cap labs_r = \emptyset \\
\quad \text{where} \quad \begin{aligned} labs_{d1} &= labs_d \setminus (\mathsf{labels}\ scc) \\ C &= \mathsf{trsfScc}\ scc\ \overline{scc}_1 \end{aligned} & \\[1ex]
\texttt{SEQ}\ \mathit{name}(\ell)\ (\mathsf{sortDefs}\ \overline{scc}_1\ labs_{d1}\ labs_{r1}), & \text{if } (\mathsf{labels}\ scc) \cap labs_d = \emptyset \\
\quad \text{where} \quad \begin{aligned} \texttt{sccSingle}\ \ell &= scc \\ \texttt{\{}(x_i \,\texttt{=}\, E_i^{\ell_i})^{i \in 1..n}\texttt{\}} &= \mathit{exp}(\ell) \\ labs_{d1} &= \{\ell_i^{i \in 1..n}\} \cup labs_d \\ labs_{r1} &= labs_r \setminus \{\ell\} \end{aligned} &
\end{cases} \\
\quad \text{where} \quad \langle \overline{scc}_1, scc \rangle = \mathsf{findLast}\ \overline{scc}\ (labs_d \cup labs_r)
\end{array}$$

Figure 5.8: Complete definition of the function sortDefs.

$$\mathsf{trsfScc} : \ \mathit{Scc} \rightarrow \mathit{Scc}^* \rightarrow C$$

$$\begin{array}{l}
\mathsf{trsfScc}\ scc\ \overline{scc} \quad = \\
\quad \begin{cases}
\texttt{RECDEF}\ (\mathit{name}(\ell_i)\ \ \mathcal{T}^T[\![\mathit{exp}(\ell_i)]\!])^{i \in 1..n}, & \text{if } scc = \texttt{sccRec}\ \{\ell_i^{i \in 1..n}\} \\[1ex]
\texttt{DEF}\ \mathit{name}(\ell)\ \mathcal{T}^T[\![\mathit{exp}(\ell)]\!], & \text{if } scc = \texttt{sccSingle}\ \ell \wedge \mathit{exp}(\ell) \in T \\[1ex]
\texttt{SEQ}\ \mathit{name}(\ell)\ (\mathsf{sortDefs}\ \overline{scc}\ \{\ell_i^{i \in 1..n}\}\ \emptyset), & \text{if } scc = \texttt{sccSingle}\ \ell \\
 & \quad \wedge\ \mathit{exp}(\ell) = \texttt{\{}(x_i \,\texttt{=}\, E_i^{\ell_i})^{i \in 1..n}\texttt{\}} \\[1ex]
\texttt{SEQ}\ \mathit{name}(\ell)\ (\mathsf{sortDefs}\ \overline{scc}_1\ labs_{d1}\ labs_{r1}), & \text{if } scc = \texttt{sccGrp}\ labs_{tops}\ {scc_j}^{j \in 1..m} \\
\qquad \qquad \text{where} \quad \begin{aligned} \texttt{sccSingle}\ \ell &= scc_m \\ \texttt{\{}(x_i \,\texttt{=}\, E_i^{\ell_i})^{i \in 1..n}\texttt{\}} &= \mathit{exp}(\ell) \\ \overline{scc}_1 &= \overline{scc} \mathbin{+\!\!+} {scc_j}^{j \in 1..(m-1)} \\ labs_{d1} &= \{\ell_i^{i \in 1..n}\} \\ labs_{r1} &= labs_{tops} \setminus \{\ell\} \end{aligned} &
\end{cases}
\end{array}$$

Figure 5.9: Complete function trsfScc.

The call of the function sortDefs with a non-empty set of recursive Groups differs to a certain degree from the simple version. We give the complete version in Fig. 5.8 and the complete function trsfScc is defined in Fig. 5.9.

Similar to the simple case, the last component must be found. But now it can either represent an element of the inner definition set or of the recursive Group set. In the first case, the component is transformed like in the previous section. The check whether it represents an inner definition or not changes compared to the simple version. The set of labels of the found SCC is calculated and the intersection with the set $labs_d$ must not be empty. To explain this check, we have to extend the definition of the function labels for the case of an `sccGrp` component:

$$\mathsf{labels}\ (\mathtt{sccGrp}\ labs\ \overline{scc_i}^{i \in 1..n}) = \bigcup_{i=1}^{n} \mathsf{labels}\ scc_i$$

In this case, the labels of all definitions captured in this SCC are returned. This is necessary as we have to remove all definitions of this SCC from the set $labs_d$ in the recursive call of sortDefs. Not all of these definitions are already added to the set $labs_d$. Therefore, we just check whether the intersection is not empty.

We return to our example. The last SCC is `sccSingle` 8—the definition `is2odd`. As it is an inner definition it is transformed by the function trsfScc. The function sortDefs is called recursively with the following arguments:

$$\begin{aligned} \overline{scc} &= \mathtt{sccSingle}\ 6,\ \mathtt{sccRec}\ \{4,7\},\ \mathtt{sccSingle}\ 5,\ \mathtt{sccSingle}\ 2 \\ labs_d &= \{7\} \\ labs_r &= \{2\} \end{aligned}$$

If the last SCC scc is related to a mutually recursive Group, this component must be an `sccSingle`, as all elements of the set $labs_r$ are top Groups of an `sccGrp`. This `sccSingle` is transformed into a `SEQ`-component with all other inner definitions and recursive components as inner definitions. As in the simple case, the list $\overline{scc_1}$ is the start of the list $\overline{scc}$ up to the current scc. The set $labs_d$ is extended by all definitions of this Group, and the Group is removed from the set $labs_r$.

In our example, the last SCC for the next call of sortDefs is `sccSingle` 2. $2 \in labs_r$, consequently the last SCC is a recursive Group and hence it is transformed into:

$$\mathtt{SEQ}\ \mathtt{E}\ (\mathsf{sortDefs}\ (\mathtt{sccSingle}\ 6,\ \mathtt{sccRec}\ \{4,7\},\ \mathtt{sccSingle}\ 5)\ \{4,5,6,7\}\ \emptyset)$$

The complete intermediate representation for the even-odd-example is given in Listing 5.10.

Here the definition `O.odd` is part of the component representing `E`. The two Groups `E` and `O` are mutually recursive, consequently one of them contains at least one definition of the other. Furthermore, the component representing `E` is part of the inner definitions of `O`. This is always the case for mutually recursive Groups.

```
SEQ (SEQ O (SEQ E (DEF E.val 2,
                   RECDEF (E.even (λx. ...), O.odd (λx. ...)),
                   DEF E.is2even (even val)),
            DEF O.is2odd (odd 2)))
```

Listing 5.10: Intermediate representation for even-odd-example.

5.1.3 Complexity

The transformation function is in $O(n^2 \log n)$ in worst case. This can be motivated as follows:

The functions sortDefs and trsfScc are defined in a way that each SCC is only transformed once. In each call of sortDefs either an `sccSingle` or an `sccRec` is transformed. The calculation of SCCs splits the set of vertices into disjoint sets, as a consequence the number of calls of sortDefs is bounded by the number of definitions n. Therefore, we can estimate the complexity for the transformation of a Group hierarchy with n definitions: n times the cost for one SCC. The most expensive operation in the transformation is to locate the next SCC. In the worst, case we have to look at each element in the list $\overline{scc}$ and have to calculate the intersection labels $scc_n \cap labs$. The intersection of two sets s_1, s_2 is in $O(|s_1| \log(|s_2|))$. The union of the two sets $labs_d$ and $labs_r$ is bounded by n as well. Consequently, the cost for the transformation of an SCC is in $O(n \log(n))$. It follows that the transformation is in $O(n^2 \log(n))$.

As the calculation of the hierarchical SCCs is in $O(n^3)$, this is an upper bound for the time complexity of the complete process of analysis and transformation.

5.2 Properties

As already mentioned, our transformation fulfills three properties to ensure the desired behavior of the semantics. We discuss these properties in more detail in the following. To define the properties we need to calculate the names of a Group as well as the names of a component. Both functions are defined in Fig. 5.11.

5.2.1 Structural Identity

The transformed program must be structurally identical to the original program. Therefore, all terms are transformed into terms of the same kind, e. g. abstractions are translated into abstractions, applications into applications, etc. Furthermore, all definitions of all Groups must be contained in the transformed version. We define a relation $I \multimapdotboth E$, in Fig. 5.12, pronounced "I represents E" that captures the structural identity. The transformation satisfies for all programs E, $\mathcal{T}^E[\![E]\!] \multimapdotboth E$.

The structural identity for terms is straightforward as there is the same set of term-expressions in both languages as established by the rules VAR, CONST, SEL, LAM, and APP.

$$\mathsf{names} : \ E \to \mathcal{P}(\mathit{Path})$$

$$\begin{aligned}
\mathsf{names}\ T &= \emptyset \\
\mathsf{names}\ \{(x_i = E_i^{\ell_i})^{i\in 1..n}\}^{\ell} &= \mathit{name}(\ell) \cup \bigcup_{i\in 1..n} \mathsf{names}\ E_i
\end{aligned}$$

$$\mathsf{names} : \ C \to \mathcal{P}(\mathit{Path})$$

$$\begin{aligned}
\mathsf{names}\ (\mathtt{DEF}\ N\ S) &= \{N\} \\
\mathsf{names}\ (\mathtt{RECDEF}\ (N_i\ S_i)^{i\in 1..n}) &= \{N_i^{i\in 1..n}\} \\
\mathsf{names}\ (\mathtt{SEQ}\ N\ C_i^{\ i\in 1..n}) &= N \cup \bigcup_{i\in 1..n} \mathsf{names}\ C_i
\end{aligned}$$

Figure 5.11: Definition of names for Groups and components.

$$\text{VAR}\ \frac{}{x \circ\!\!-\!\!\bullet\ x} \qquad \text{CONST}\ \frac{}{c \circ\!\!-\!\!\bullet\ c}\,c \qquad \text{SEL}\ \frac{I \circ\!\!-\!\!\bullet\ E}{I.x \circ\!\!-\!\!\bullet\ E.x}$$

$$\text{LAM}\ \frac{I \circ\!\!-\!\!\bullet\ E}{\lambda x.I \circ\!\!-\!\!\bullet\ \lambda x.E} \qquad \text{APP}\ \frac{I_1 \circ\!\!-\!\!\bullet\ E_1 \qquad I_2 \circ\!\!-\!\!\bullet\ E_2}{I_1\ I_2 \circ\!\!-\!\!\bullet\ E_1\ E_2}$$

$$\text{HIER}\ \frac{\begin{array}{cc} \mathsf{names}\ C = \mathsf{names}\ G & \mathsf{defs}\ C = \{(N_i \mapsto S_i)^{i\in 1..n}\} \\ \forall i \in 1..n : S_i \circ\!\!-\!\!\bullet\ T_i & \mathsf{defs}\ G = \{(N_i \mapsto T_i)^{i\in 1..n}\} \end{array}}{C \circ\!\!-\!\!\bullet\ G}$$

Figure 5.12: Structural identity for GLang and intermediate expressions.

$$\begin{aligned}
&\mathsf{defs}\ :\ C \to \mathcal{P}(Path \hookrightarrow S)\\
&\mathsf{defs}\ (\mathtt{SEQ}\ N\ C_i^{\ i\in 1..n}) &&= \bigcup_{i\in 1..n} (\mathsf{defs}\ \ C_i)\\
&\mathsf{defs}\ (\mathtt{DEF}\ N\ S) &&= \{N \mapsto S\}\\
&\mathsf{defs}\ (\mathtt{RECDEF}\ (N_i\ S_i)^{i\in 1..n}) &&= \{(N_i \mapsto S_i)^{i\in 1..n}\}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{defs}\ :\ G \to \mathcal{P}(Path \hookrightarrow T)\\
&\mathsf{defs}\ \mathtt{\{}(x_i = E_i^{\ell_i})^{i\in 1..n}\mathtt{\}} = \bigcup_{i\in 1..n} d_i\\
&\qquad\qquad \text{where}\quad d_i = \begin{cases} \{\mathit{name}(\ell_i) \mapsto T_i\}, & \text{if } E_i = T_i\\ \mathsf{defs}\ \ E_i, & \text{otherwise}\end{cases}
\end{aligned}$$

Figure 5.13: Definition of `defs` for Groups and components.

As Groups are transformed into `SEQ`-components, the structural identity requires all definitions of a Group hierarchy to be in that component. Therefore, the sets of all inner names of the Group hierarchy and the corresponding `SEQ` must be equal (first premise of rule HIER).

Furthermore, all inner terms must be represented by the corresponding transformed term. The functions `defs` for Groups and components, defined in Fig. 5.13, collect all term definitions in a map. They are used in the second and fourth premise of the rule HIER and their results are used to check whether the two terms corresponding to the same name are structurally identical (third premise).

The structural identity for terms follows directly from the definition of $\mathcal{T}^T$.

All definitions of a Group hierarchy are either represented by a `sccSingle` or a `sccRec`. Some of these SCCs are nested within a `sccGrp`. All SCCs within the given list are transformed exactly once. This makes sure that he first premise of HIER holds.

All SCCs representing a definition are transformed by applying the transformation function $\mathcal{T}^T$ to the right-hand-side of the definition. Therefore, the other three premises of HIER hold.

5.2.2 Complete Groups

Every Group, which is not mutually recursive with another Group is transformed into one `SEQ`. This means that exactly the names of this Group are contained in this `SEQ` component. Formally, we capture this property by the following definition:

$$\forall\mathtt{\{}D_i^{\ i\in 1..m}\mathtt{\}} \text{ in } E : \exists(\mathtt{SEQ}\ N\ C_i^{\ i\in 1..n}) \text{ in } \mathcal{T}^E[\![E]\!] : \bigcup_{i\in 1..m} (\mathsf{names}\ D_i) \subseteq \bigcup_{i\in 1..n} (\mathsf{names}\ C_i)$$

All `SEQ`-components must contain all inner definitions of the corresponding Group but can contain more, so the set of names of the Group is a subset of the names of the component.

In the case of a Group not mutually recursive with any other Group, the names of the corresponding component must be the same as the names of the Group.

$$\forall G \text{ in } E : \nexists G' \text{ in } E : G \text{ and } G' \text{ are mutually recursive Groups} \Rightarrow \exists C \text{ in } \mathcal{T}^E[\![E]\!] : \mathsf{names}\ C = \mathsf{names}\ G$$

We can see that the transformation fulfills this property as follows. For each Group there is an `sccSingle` in the list of SCCs. In the case of a mutually recursive Group this SCC is nested in an `sccGrp`. For all `sccSingle` ℓ, where $\mathit{exp}(\ell) \in G$, all inner definitions are added into the set $labs_d$. All elements of $labs_d$ are sorted and transformed by sortDefs. sortDefs terminates when all elements of $labs_d$ are transformed and added to the list of inner components. Consequently, all definitions of a Group are represented within the list of inner components or are nested within an inner component. Therefore, all names of the Group occur within the list of inner components.

For all top vertices of an `sccGrp` we know that the corresponding Groups are mutually recursive. Only the labels of top vertices are added into the set $labs_r$. As a consequence, an `sccSingle` ℓ, where $\mathit{exp}(\ell)$ represents a Group not mutually recursive with another Group, is always transformed by the call trsfScc (`sccSingle` ℓ) $\overline{scc}_1$. This call results in `SEQ` $\mathit{name}(\ell)$ (sortDefs $\overline{scc}$ $labs_d$ $\emptyset$), where the set $labs_d$ only contains the definition of the Group. Therefore, only the inner definitions of the Group are added to the list of inner components, hence the names of the Group and the names of the inner components are equal.

5.2.3 Correct Ordering

All components are ordered correctly with respect to their dependencies. That is, all definitions a `DEF` or a `RECDEF` depends on appear lexically before that `DEF` or `RECDEF`.

To precisely capture the notation of "lexically before" we assign position numbers to all `DEF` and `RECDEF` by a preorder traversal of a component and store those positions in the map pos mapping names to the position number.

In the Listing 5.14, the names of the components are subscripted with their position.

```
SEQ (SEQ A (DEF A.x₁ 3,
            SEQ A.B (DEF A.B.y₂ 4,
                     DEF A.B.z₃ (x + 4)),
            DEF A.f₄ A.B.z)
     DEF x₅ A.x)
```

Listing 5.14: Example with position subscripts.

For each Group hierarchy G the definitions are ordered correctly within the transformed component C iff:

For all definitions $D_1 := (x_1 = T_1^{\ell_1})$ within G and all definitions $D_2 := (x_2 = T_2^{\ell_2})$ D_1 depends on, we distinguish two cases:

```
{ X = { a = [Y.e + b]^1
        b = 3^2 }^3
  Y = { e = 7^4
        d = e^5 }^6
}^7
```

(a) GLang code.

```
sccSingle 2,
sccSingle 4,
sccSingle 5,
sccSingle 6,
sccSingle 1,
sccSingle 3,
sccSingle 7
```

(b) List of SCCs.

```
SEQ (SEQ Y
       (DEF Y.e 7,
        DEF Y.d e),
     SEQ X
       (DEF X.b 3,
        DEF X.a (Y.e + b)))
```

(c) Intermediate representation.

Figure 5.15: Example for reordering of definitions.

- if D_2 depends on D_1 both definitions must be within a `RECDEF` component, as they are mutually recursive:

$$pos(name(\ell_1)) = pos(name(\ell_2))$$

- if D_2 does not depend on D_1 the former must be defined before the later:

$$pos(name(\ell_1)) > pos(name(\ell_2))$$

The transformation is based on the dependency analysis. During this analysis, all definitions are ordered in the correct way. The transformation does not keep this order completely, as the transformation combines all definitions of a Group within its corresponding list of inner components. To illustrate this reordering, we return to the example of the transformation. The GLang expression, the list of SCCs, and the intermediate representation are given in Fig. 5.15.

For example, the definition with label 2 is the second element in the list of SCCs, but it is defined after the component representing the Group `Y`, although the SCC representing `Y` is the fourth element in the list. This reordering is correct, as motivated in Sec. 4.3. In our example, the Group `Y` does not need Group `X`, hence the definition with label 2 can be evaluated after `Y`.

The inner definitions are sorted in their dependency order. So the reordering for a Group not mutually recursive with another Group is correct.

Mutually recursive Groups are transformed together as they are handled by the set $labs_r$. All inner definitions of those mutually recursive Groups are ordered according to the order of the mutually recursive modules within the inner list of SCCs. Similar as for non-recursive Groups, the inner definitions defined before the current `sccGrp` are allowed to be reordered.

6 Interpretation and Compilation

The intermediate representation is used to define the semantics of the GLang language. Therefore we define an evaluation function for the intermediate representation. Furthermore, we compile the intermediate representation into a functional language with let-expressions and dynamic records.

6.1 Semantics for the Intermediate Representation

The result of the transformation is mapped to an interpretation by the evaluation function. We use the established techniques of denotational semantics [40, 15, 39] without going too much into detail.

According to [21], the evaluation of a declaration *dec* is the function: $[\![dec]\!] : \Gamma \to \Gamma$, where the first Γ holds the bindings for the free variables of dec and the result is the defined environment representing the bindings of the declaration. In GLang, the evaluation of a declaration results in only one binding, mapping the selector to the value.

In GLang, the scope of a definition is the complete Group containing all nested Groups. For the intermediate representation, we change this scope definition, as all definitions are sorted in dependency order. The scope of a component C is the rest of the current `SEQ`. So the evaluation of a definition uses the environments produced by the components defined lexically before this definition.

6.1.1 Value-Domains and Auxiliary Functions

We map every expression I to a value V from the following semantic domains:

$$\text{Values: } \mathsf{V} = \mathsf{C} + \mathsf{H} + \mathsf{F} \qquad \text{Groups: } \mathsf{H} = X \hookrightarrow \mathsf{V} \qquad \text{Functions: } \mathsf{F} = \mathsf{V} \to \mathsf{V}$$

The set of all values consists of constants C, Group values H, and function values F. Group values H are partial functions with a finite domain, mapping identifiers to values. Although the Group value can be regarded as an evaluation context which maps free variables of an expression to values, we use a special evaluation context Γ in order to avoid confusion.

The environment Γ maps all free variables to values. The operator $\lhd$ combines two environments Γ_1 and Γ_2 where the right mapping overrides definitions of the left one. For a syntactical distinction between the mapping within a Group value and within an environment, the environment mapping uses $\mapsto$ and the Group value uses $\multimap$.

Furthermore, we define three auxiliary functions for Group values: $\Mapsto$, $+$, and the projection $|$:

- The operation $\Mapsto$ constructs a Group value for a given name and value.

$$\Mapsto\colon\ \mathit{Path} \to \mathsf{V} \to (X \hookrightarrow \mathsf{V})$$

$$x(\texttt{.}x_i)^{i\in 1..n} \Mapsto v = \begin{cases} x \twoheadrightarrow (x_1(\texttt{.}x_i)^{i\in 2..n} \Mapsto v), & \text{if } n > 0 \\ x \twoheadrightarrow v, & \text{otherwise} \end{cases}$$

$$\varepsilon \Mapsto v = \emptyset$$

A Group value is a mapping of a selector to a value. In the evaluation, a name must be mapped to a value. For example, in the evaluation of a definition `DEF` N S the result is a Group value storing the result of the evaluation of S assigned to the name N. The result of $x(\texttt{.}x_i)^{i\in 1..n} \Mapsto v$ is a Group value with only one entry. This entry maps x to the Group value representing the inner selections. Each of the inner Group values contains only one entry as well. The last selector x_n is mapped to the value v. If the name is empty, the mapping is also empty.

- The operator $+$ combines two Group values.

$$+ :\ (X \hookrightarrow \mathsf{V}) \to (X \hookrightarrow \mathsf{V}) \to (X \hookrightarrow \mathsf{V})$$

$$\begin{aligned} \mathsf{H}_1 + \mathsf{H}_2 = \ & \{x \twoheadrightarrow (v_1 + v_2) \mid x \twoheadrightarrow v_1 \in \mathsf{H}_1 \wedge x \twoheadrightarrow v_2 \in \mathsf{H}_2\} \\ \cup\ & \{x \twoheadrightarrow v_1 \mid x \twoheadrightarrow v_1 \in \mathsf{H}_1 \wedge x \twoheadrightarrow v_2 \notin \mathsf{H}_2\} \\ \cup\ & \{x \twoheadrightarrow v_2 \mid x \twoheadrightarrow v_2 \in \mathsf{H}_2 \wedge x \twoheadrightarrow v_1 \notin \mathsf{H}_1\} \end{aligned}$$

When two Group values are combined, the resulting mapping is the union of both mappings, where those values mapped by the same selector are combined as well. The combination is only defined if both values are Group values. The evaluation of a transformed program combines only Group values.

- The projection $|$ creates an environment Γ for a name.

$$| :\ (X \hookrightarrow \mathsf{V}) \to \mathit{Path} \to (X \hookrightarrow \mathsf{V})$$

$$\mathsf{H}|_{x(\texttt{.}x_i)^{i\in 1..n}} = \Gamma' \lhd \begin{cases} v|_{x_1(\texttt{.}x_i)^{i\in 2..n}}, & \text{if } n > 0 \wedge x \twoheadrightarrow v \in \mathsf{H} \\ \emptyset, & \text{otherwise} \end{cases}$$
$$\text{where}\quad \Gamma' = \{y \mapsto v \mid y \twoheadrightarrow v \in \mathsf{H}\}$$

To handle the name-space we have to create environments out of a Group value for a name. This name specifies which definitions within this Group value are accessible. All selectors of the Group value are accessible (Γ'). If there is an identifier representing the variable of the path mapped to a Group value, this Group value is projected into the name-space of the selection chain.

$$\mathcal{E} :\ A \to \Gamma \to \mathsf{H} \to \mathsf{V}$$

$$\begin{array}{ll}
\mathcal{E}[\![\texttt{DEF}\ N\ S]\!]\ \Gamma\ \mathsf{H} & = N \mapsto (\mathcal{E}[\![S]\!]\ (\Gamma \lhd (\mathsf{H}|_N))\ \emptyset) \\
\mathcal{E}[\![\texttt{RECDEF}\ (N_i\ S_i)^{i\in 1..n}]\!]\ \Gamma\ \mathsf{H} & = v_1 + \cdots + v_n \\
\quad \text{where}\quad \Phi \left\langle \mathsf{F}_i^{\,i\in 1..n} \right\rangle & = \langle \mathcal{E}[\![S_i]\!]\ (\Gamma \lhd (\mathsf{H}'|_{N_i}))\ \emptyset \rangle^{\,i\in 1..n} \\
\qquad\qquad \text{where}\quad v'_i & = N_i \mapsto F_i \\
\qquad\qquad \mathsf{H}' & = \mathsf{H} + v'_1 + \cdots + v'_n \\
\quad \left\langle \mathsf{F}_i^{\,i\in 1..m} \right\rangle & = \mathsf{FIX}\ \Phi \\
\quad v_i & = N_i \mapsto F_i \\
\mathcal{E}[\![\texttt{SEQ}\ N\ C_i^{\,i\in 1..n}]\!]\ \Gamma\ \mathsf{H} & = N \mapsto \emptyset + v_1 + \cdots + v_n \\
\quad \text{where}\quad v_i & = \mathcal{E}[\![C_i]\!]\ \Gamma\ \mathsf{H}_i \\
\quad \mathsf{H}_i & = \mathsf{H}_{i-1} + v_{i-1} \\
\quad \mathsf{H}_1 & = \mathsf{H} \\
\mathcal{E}[\![E\,\texttt{.}x]\!]\ \Gamma\ \mathsf{H} & = \begin{cases} v, & \text{if } (x \twoheadrightarrow v) \in (\mathcal{E}[\![E]\!]\ \Gamma\ \mathsf{H}) \\ \text{ERROR}, & \text{otherwise} \end{cases} \\
\mathcal{E}[\![x]\!]\ \Gamma\ \mathsf{H} & = \Gamma(x) \\
\ldots &
\end{array}$$

Figure 6.1: Evaluation function.

6.1.2 Evaluation Function

Figure 6.1 shows the evaluation function for Groups and definitions. The evaluation function has three arguments: the expression to evaluate, a normal environment mapping variables to values, and a Group value H representing the current Group hierarchy. All definitions are evaluated to a Group value representing the name within the current Group hierarchy. The right-hand side of each definition is evaluated within a fresh (empty) hierarchy and an environment that contains the variables of the current scope. Consequently, the current environment is enriched by the projection of the current hierarchy into the scope of the definition. The projection adds all definitions of the current hierarchy that are visible for this definition. Recursive definitions are evaluated, as usual, via a fixpoint operator.

Groups are evaluated to Group values containing all inner definitions. The definitions are evaluated in evaluation order, i. e., in the order of the given list. The value of a subcomponent is added to the hierarchy value and the next subcomponent is evaluated. The result is the combination of all subcomponent values.

The result of a selection $E\,\texttt{.}x$ is either the value v, if x maps to v in the result of the evaluation of E or it is an error. Two possible errors exist: the value of E is not a Group or it does not contain an x.

A variable is evaluated by the look up of this variable in the environment. As the

```
{ x = { d = y.e }
  y = { d = e
        e = 7 }}
```

(a) GLang code.

```
SEQ ε (SEQ y (DEF y.e 7,
              DEF y.d e),
       SEQ x (DEF x.d y.e))
```

(b) Intermediate representation.

Figure 6.2: Example for evaluation function.

context analysis rejects all programs with undefined variables, the lookup is always successful.

The rest of the evaluation function is omitted as it is the usual definition with an environment and the hierarchy value is ignored.

6.1.3 Evaluation Example

For a better understanding of the evaluation function we give an example of two Groups which are not mutually recursive. The GLang expression and the intermediate representation are given in Fig. 6.2.

The evaluation of the intermediate code starts with:

$$\mathcal{E} \left[\!\!\left[\ \mathtt{SEQ}\ \varepsilon\ (\mathtt{SEQ}\ \mathsf{y}\ (\mathtt{DEF}\ \mathsf{y.e}\ 7, \mathtt{DEF}\ \mathsf{y.d}\ \mathsf{e}), \mathtt{SEQ}\ \mathsf{x}\ (\mathtt{DEF}\ \mathsf{x.d}\ \mathsf{y.e})) \ \right]\!\!\right]\ \emptyset\ \emptyset$$

This `SEQ` has two inner components. They must be evaluated one after the other resulting IN the values v_1 and v_2. So the next step is the calculation of v_1:

$$v_1 = \mathcal{E}[\![\mathtt{SEQ}\ \mathsf{y}\ (\mathtt{DEF}\ \mathsf{y.e}\ 7, \mathtt{DEF}\ \mathsf{y.d}\ \mathsf{e})]\!]\ \emptyset\ \emptyset$$

Both environments are empty. This expression is again a `SEQ` with two inner components. Next, those two components are evaluated, starting with the first:

$$\begin{aligned} v_1' &= \mathcal{E}[\![\mathtt{DEF}\ \mathsf{y.e}\ 7]\!]\ \emptyset\ \emptyset \\ &= \mathsf{y.e} \Mapsto \mathcal{E}[\![7]\!]\ \emptyset\ \emptyset \\ &= \{\mathsf{y} \rightarrowtriangle \{\mathsf{e} \rightarrowtriangle 7\}\} \end{aligned}$$

As both environments are empty, the term within this definition is evaluated in an empty environment. The name `y.e` is mapped to the result of this evaluation. The resulting Group value is the result of the evaluation of the `DEF`.

The second component is evaluated with this value added to the Group environment:

$$\begin{aligned} v_2' &= \mathcal{E}[\![\mathtt{DEF}\ \mathsf{y.d}\ \mathsf{e}]\!]\ \emptyset\ \{\mathsf{y} \rightarrowtriangle \{\mathsf{e} \rightarrowtriangle 7\}\} \\ &= \mathsf{y.d} \Mapsto \mathcal{E}[\![\mathsf{e}]\!]\ \{\mathsf{y} \rightarrowtriangle \{\mathsf{e} \rightarrowtriangle 7\}\}|_{\mathsf{y.d}}\ \emptyset \\ &= \mathsf{y.d} \Mapsto \mathcal{E}[\![\mathsf{e}]\!]\ \{\mathsf{y} \mapsto \{\mathsf{e} \rightarrowtriangle 7\}, \mathsf{e} \mapsto 7\}\ \emptyset \\ &= \mathsf{y.d} \Mapsto 7 \\ &= \{\mathsf{y} \rightarrowtriangle \{\mathsf{d} \rightarrowtriangle 7\}\} \end{aligned}$$

The term of this definition is evaluated in an environment containing the projection of the result v_1' into the name-space of this definition. This projection adds a mapping for `y` and `e` into the empty environment.

At this point both components of the `SEQ` with the name `y` are evaluated and the value for this `SEQ` can be created. The values v_1' and v_2' are added into an empty mapping for `y`.

$$\begin{aligned} v_1 &= (\mathsf{y} \Mapsto \emptyset) + \{\mathsf{y} \rightarrowtriangle \{\mathsf{e} \rightarrowtriangle 7\}\} + \{\mathsf{y} \rightarrowtriangle \{\mathsf{d} \rightarrowtriangle 7\}\} \\ &= \{\mathsf{y} \rightarrowtriangle \{\mathsf{e} \rightarrowtriangle 7, \mathsf{d} \rightarrowtriangle 7\}\} \end{aligned}$$

Next, the second component of the top-level `SEQ` is evaluated, with this value added to the Group environment:

$$v_2 = \mathcal{E}[\![\mathsf{SEQ\ x\ (DEF\ x.d\ y.e)}]\!]\ \emptyset\ \{\mathsf{y} \rightarrowtriangle \{\mathsf{e} \rightarrowtriangle 7, \mathsf{d} \rightarrowtriangle 7\}\}$$

The inner component of this `SEQ` is evaluated:

$$\begin{aligned} v_1'' &= \mathcal{E}[\![\mathsf{DEF\ x.d\ y.e}]\!]\ \emptyset\ \{\mathsf{y} \rightarrowtriangle \{\mathsf{e} \rightarrowtriangle 7, \mathsf{d} \rightarrowtriangle 7\}\} \\ &= \mathsf{x.d} \Mapsto \mathcal{E}[\![\mathsf{y.e}]\!]\ \{\mathsf{y} \mapsto \{\mathsf{e} \rightarrowtriangle 7, \mathsf{d} \rightarrowtriangle 7\}\}\ \emptyset \\ &= \mathsf{x.d} \Mapsto (\mathcal{E}[\![\mathsf{y}]\!]\ \{\mathsf{y} \mapsto \{\mathsf{e} \rightarrowtriangle 7, \mathsf{d} \rightarrowtriangle 7\}\}\ \emptyset)\,\mathsf{.e} \\ &= \mathsf{x.d} \Mapsto \{\mathsf{e} \rightarrowtriangle 7, \mathsf{d} \rightarrowtriangle 7\}\mathsf{.e} \\ &= \mathsf{x.d} \Mapsto 7 \\ &= \{\mathsf{x} \rightarrowtriangle \{\mathsf{d} \rightarrowtriangle 7\}\} \end{aligned}$$

The term in this definition is evaluated within the projection of the Group environment into the name-space of `x.d`. First,s the variable `y` is looked up in the environment and from the resulting Group value the item `e` is selected.

As this is the only component of `x` and all components of the top-level `SEQ` are evaluated, the complete value can be constructed:

$$\{\mathsf{y} \rightarrowtriangle \{\mathsf{e} \rightarrowtriangle 7, \mathsf{d} \rightarrowtriangle 7\}, \mathsf{x} \rightarrowtriangle \{\mathsf{d} \rightarrowtriangle 7\}\}$$

The complete evaluation is summarized in Fig. 6.3.

6.2 Code Generation

The evaluation of the intermediate representation needs in addition to the common environment a second environment. In this section, we give a compilation into a language with let-expressions and records. The syntax of the record language is shown in Fig. 6.4.

An R-expression is either a term U or a list of bindings followed by a record. The set of terms is equal to those defined in GLang and in the intermediate language. A recursive let-binding defines a set of mutually recursive functions. Records define a list of values, which can be selected through the given identifier. Within a record the other definitions are not visible.

$$
\begin{aligned}
&\mathcal{E}\left\llbracket\ \texttt{SEQ}\ \varepsilon\ (\texttt{SEQ y}\ (\texttt{DEF y.e 7}, \texttt{DEF y.d e}), \texttt{SEQ x}\ (\texttt{DEF x.d y.e}))\ \right\rrbracket\ \emptyset\ \emptyset\\
&\qquad v_1 = \mathcal{E}\llbracket \texttt{SEQ y}\ (\texttt{DEF y.e 7}, \texttt{DEF y.d e})\rrbracket\ \emptyset\ \emptyset\\
&\qquad\qquad v_1 = \mathcal{E}\llbracket \texttt{DEF y.e 7}\rrbracket\ \emptyset\ \emptyset\\
&\qquad\qquad\quad = \texttt{y.e} \Mapsto \mathcal{E}\llbracket 7\rrbracket\ \emptyset\ \emptyset\\
&\qquad\qquad\quad = \{\texttt{y} \mathrel{-\!\triangleright} \{\texttt{e} \mathrel{-\!\triangleright} 7\}\}\\
&\qquad\qquad v_2 = \mathcal{E}\llbracket \texttt{DEF y.d e}\rrbracket\ \emptyset\ \{\texttt{y} \mathrel{-\!\triangleright} \{\texttt{e} \mathrel{-\!\triangleright} 7\}\}\\
&\qquad\qquad\quad = \texttt{y.d} \Mapsto \mathcal{E}\llbracket \texttt{e}\rrbracket\ \{\texttt{y} \mathrel{-\!\triangleright} \{\texttt{e} \mathrel{-\!\triangleright} 7\}\}|_{\texttt{y.d}}\ \emptyset\\
&\qquad\qquad\quad = \texttt{y.d} \Mapsto \mathcal{E}\llbracket \texttt{e}\rrbracket\ \{\texttt{y} \mapsto \{\texttt{e} \mathrel{-\!\triangleright} 7\}, \texttt{e} \mapsto 7\}\ \emptyset\\
&\qquad\qquad\quad = \texttt{y.d} \Mapsto 7\\
&\qquad\qquad\quad = \{\texttt{y} \mathrel{-\!\triangleright} \{\texttt{d} \mathrel{-\!\triangleright} 7\}\}\\
&\qquad\quad = (\texttt{y} \Mapsto \emptyset) + \{\texttt{y} \mathrel{-\!\triangleright} \{\texttt{e} \mathrel{-\!\triangleright} 7\}\} + \{\texttt{y} \mathrel{-\!\triangleright} \{\texttt{d} \mathrel{-\!\triangleright} 7\}\}\\
&\qquad\quad = \{\texttt{y} \mathrel{-\!\triangleright} \{\texttt{e} \mathrel{-\!\triangleright} 7, \texttt{d} \mathrel{-\!\triangleright} 7\}\}\\
&\qquad v_2 = \mathcal{E}\llbracket \texttt{SEQ x}\ (\texttt{DEF x.d y.e})\rrbracket\ \emptyset\ \{\texttt{y} \mathrel{-\!\triangleright} \{\texttt{e} \mathrel{-\!\triangleright} 7, \texttt{d} \mathrel{-\!\triangleright} 7\}\}\\
&\qquad\qquad v_1 = \mathcal{E}\llbracket \texttt{DEF x.d y.e}\rrbracket\ \emptyset\ \{\texttt{y} \mathrel{-\!\triangleright} \{\texttt{e} \mathrel{-\!\triangleright} 7, \texttt{d} \mathrel{-\!\triangleright} 7\}\}\\
&\qquad\qquad\quad = \texttt{x.d} \Mapsto \mathcal{E}\llbracket \texttt{y.e}\rrbracket\ \{\texttt{y} \mapsto \{\texttt{e} \mathrel{-\!\triangleright} 7, \texttt{d} \mathrel{-\!\triangleright} 7\}\}\ \emptyset\\
&\qquad\qquad\quad = \texttt{x.d} \Mapsto (\mathcal{E}\llbracket \texttt{y}\rrbracket\ \{\texttt{y} \mapsto \{\texttt{e} \mathrel{-\!\triangleright} 7, \texttt{d} \mathrel{-\!\triangleright} 7\}\}\ \emptyset)\ \texttt{.e}\\
&\qquad\qquad\quad = \texttt{x.d} \Mapsto \{\texttt{e} \mathrel{-\!\triangleright} 7, \texttt{d} \mathrel{-\!\triangleright} 7\}\texttt{.e}\\
&\qquad\qquad\quad = \texttt{x.d} \Mapsto 7\\
&\qquad\qquad\quad = \{\texttt{x} \mathrel{-\!\triangleright} \{\texttt{d} \mathrel{-\!\triangleright} 7\}\}\\
&= \{\texttt{y} \mathrel{-\!\triangleright} \{\texttt{e} \mathrel{-\!\triangleright} 7, \texttt{d} \mathrel{-\!\triangleright} 7\}, \texttt{x} \mathrel{-\!\triangleright} \{\texttt{d} \mathrel{-\!\triangleright} 7\}\}
\end{aligned}
$$

Figure 6.3: Evaluation example.

The semantics of this language is straightforward and does not need a second environment. This language can be directly compiled into any dynamically typed language with records; for example into Lua [19].

For a correct translation, we do not only have to sort all definitions but also to rename all variables into the name of the definition the variable refers to. Furthermore we split this name into a static and a dynamic part. The static part is used as a normal variable and the dynamic part is the selection of elements from a dynamic record. As the sorting is already done for the intermediate representation we use the transformed program for the translation.

The code generation is split into two steps. In the first step, variables and selections

$$
\begin{array}{lcl}
R & ::= & U\\
 & | & B^{*}\ \texttt{<}\ (x\ \texttt{=}\ R)^{*}\ \texttt{>}\\
B & ::= & \texttt{LET}\ x\ \texttt{=}\ R\ \texttt{IN}\\
 & | & \texttt{LETREC}\ (x\ \texttt{=}\ \lambda\ x\ \texttt{.}\ R)^{+}\ \texttt{IN}\\
U & ::= & \lambda\ x\ \texttt{.}\ R\ \mid\ R\ R\ \mid\ R\ \texttt{.}\ x\ \mid\ \ldots
\end{array}
$$

Figure 6.4: Syntax of the record language.

are renamed; and in the second step, all definitions are compiled into let-bindings and records.

6.2.1 Renaming

The renaming deals with two tasks: All variables referring to a definition are exchanged with the complete path of this definition. Furthermore, each path is replaced by a string and a remaining selection. The first step is quite similar to the flattening mechanism described in Sec. 2.5.2. The second step is necessary because Groups are first-class values.

We explain the renaming by two examples first and give the definition of the function afterwards. We start with our running example. The result of the transformation is repeated in Listing 6.5.

```
SEQ (SEQ O (SEQ E (DEF E.val 2,
                   RECDEF (E.even (λx. ... O.odd ...),
                           O.odd (λx. ... E.even ...)),
                   DEF E.is2even (even val)),
            DEF O.is2odd (odd 2)))
```

Listing 6.5: Intermediate representation for even-odd-example.

The variables `even` and `val` in the definition `E.is2even` and the variable `odd` in the definition `is2odd` are only partial paths. These paths are replaced by the complete path—the name—of the definitions:

```
SEQ (SEQ O (SEQ E (DEF E.val 2,
                   RECDEF (E.even (λx. ... O.odd ...),
                           O.odd (λx. ... E.even ...)),
                   DEF E.is2even (E.even E.val)),
            DEF O.is2odd (O.odd 2)))
```

The second step introduces a variable for each definition and replaces all paths with this variable and the remaining selection. All paths in the current example can be replaced completely by such a variable. We decided to give the variables an identifier similar to the name of the definition. The function toIdent replaces the selection dot with the hash # forming a new identifier.[1]

$$\begin{aligned}&\mathsf{toIdent}: \ \mathit{Path} \to X\\ &\mathsf{toIdent}\ x(.x_i)^{i\in 1..n} = x\texttt{\#}x_1\texttt{\#}\ldots\texttt{\#}x_n\end{aligned}$$

After replacing all paths with the corresponding new identifier, the renaming is completed. The renamed code for our example is:

```
SEQ (SEQ O (SEQ E (DEF E.val 2,
```

[1]This is correct as long as the concrete syntax of GLang does not allow identifiers containing #.

```
                 RECDEF (E.even (λx. ... O#odd ...),
                         O.odd (λx. ... E#even ...)),
                 DEF E.is2even (E#even E#val)),
         DEF O.is2odd (O#odd 2)))
```

We return to the problematic example in Sec. 2.5.3. The relevant part of the intermediate representation for this example is given in Listing 6.6.

```
DEF declist (dec2.listUntil input),
DEF second  (... declist.tl.hd ...)
```

Listing 6.6: Example for additional edges.

In this example, all paths are already complete paths. The path `declist.tl.hd` refers to the definition `declist`. So the selection remains as it is.

The function splitPath splits a path into a new identifier and the remaining selection chain. The environment holds the mappings of paths to new identifiers. The longest sub-path the environment maps to a new identifier is replaced by this identifier.

$$\begin{aligned}&\mathsf{splitPath} : \ (\mathit{Path} \hookrightarrow X) \to \mathit{Path} \to \mathit{Path}\\ &\mathsf{splitPath}\ \Gamma\ x(.x_i)^{i\in 1..n} = y(.x_i)^{i\in k+1..n}\\ &\qquad \text{where} \quad (x(.x_i)^{i\in 1..k} \mapsto y) \in \Gamma \wedge \forall (p \mapsto y_2) \in \Gamma : x(.x_i)^{i\in 1..k} \not\sqsubseteq p\end{aligned}$$

To define the renaming function we need auxiliary functions similar to those given for the semantics in Sec. 6.1.1.

- The renaming function needs an environment mapping paths to new identifiers. The function ◀ adds new mappings to an environment. Unlike a normal combination function for environments, not only those mappings with the same path are overwritten but all paths one of the new paths is a sub-path of. This is necessary to cover lexical scoping correctly.

$$\blacktriangleleft : \ (\mathit{Path} \hookrightarrow X) \to (\mathit{Path} \hookrightarrow X) \to (\mathit{Path} \hookrightarrow X)$$

$$\begin{aligned}&\{(p_i \mapsto x_i)^{i\in 1..n}\} \blacktriangleleft \{(p'_j \mapsto y_j)^{j\in 1..m}\} =\\ &\qquad \{p_i \mapsto x_i \mid i \in 1..n \ \wedge \nexists j \in 1..m : p'_j \sqsubseteq p_i\} \cup \{(p_j \mapsto y_j)^{j\in 1..m}\}\end{aligned}$$

- Similar to the projection used in the evaluation function, the path projection $\big|$ returns a new environment for a definition. The information for the environment is held in a set of names. All names are mapped to the corresponding new identifier. Furthermore, we add the result of the projection of the names starting with x into the name-space of the definition.

$$\big| : \ \mathcal{P}(\mathit{Path}) \to \mathit{Path} \to (\mathit{Path} \hookrightarrow X)$$

$$\begin{aligned} Ns\big|_{x(.x_i)^{i\in 1..n}} &= \{N \mapsto \mathsf{toIdent}\ N \mid N \in Ns\} \blacktriangleleft \mathsf{project}\ scs\ (.x_i)^{i\in 1..n}\ x \\ &\qquad\qquad \text{where}\quad scs = \{(.y_i)^{i\in 1..m} \mid x(.y_i)^{i\in 1..m} \in Ns\} \end{aligned}$$

The function project calculates the projection of a set of selection chains into the name-space of a definition. The name of the definition is $p(.x_i)^{i\in 1..n}$. In all cases, all selection chains are mapped to the corresponding new identifier. In the case of an empty selection chain for the definition, the projection is complete. In the other case, the projection of all selection chains starting with x_1 into the name-space of the definition are added.

$$\mathsf{project} : \ \mathcal{P}(Sc) \to Sc \to Path \to (Path \hookrightarrow X)$$

$$\begin{aligned} &\mathsf{project}\ scs\ (.x_i)^{i\in 1..n}\ p = \\ &\qquad \{y_1(.y_i)^{i\in 2..m} \mapsto \mathsf{toIdent}\ (p(.y_i)^{i\in 1..m}) \mid (.y_i)^{i\in 1..m} \in scs\} \blacktriangleleft \Gamma \\ &\text{where}\quad \Gamma = \begin{cases} \emptyset, & \text{if } n = 0 \\ \Gamma = \mathsf{project}\ ps_1\ ((.x_i)^{i\in 2..n})\ p.x_1, & \text{otherwise} \\ \ \text{where}\quad ps_1 = \{(.y_i)^{i\in 2..m} \mid (.y_i)^{i\in 1..m} \in scs \wedge x_1 = y_1\} \end{cases} \end{aligned}$$

- The set of names used to create a new environment holds all defined names. The function addToNs adds all names of a C-expression to the given set of names.

$$\mathsf{addToNs} : \ \mathcal{P}(Path) \to C \to \mathcal{P}(Path)$$

$$\begin{aligned} &\mathsf{addToNs}\ Ns\ (\mathtt{DEF}\ N\ S) &&= \{N\} \cup Ns \\ &\mathsf{addToNs}\ Ns\ (\mathtt{RECDEF}\ (N_i\ S_i)^{i\in 1..n}) &&= \{N_i^{i\in 1..n}\} \cup Ns \\ &\mathsf{addToNs}\ Ns\ (\mathtt{SEQ}\ N\ C_i^{i\in 1..n}) &&= Ns \cup \{N\} \cup \bigcup_{i=1}^{n} \mathsf{addToNs}\ Ns\ C_i \end{aligned}$$

The renaming function is given in Fig. 6.7. All paths are replaced by the result of the function splitPath. In the case of a selection from an expression, the expression is renamed and the selection remains. Variables are replaced by the new identifier stored in the environment. An application is the application of the renamed function and the renamed argument. The body of an abstraction is renamed with an environment, where the variable of the abstraction is mapped to itself. The renaming of a SEQ-expression renames all inner components C_i. All defined names of a renamed component are added to the set of names to rename the next component. The right-hand side of a non-recursive definition is renamed using the current environment extended with the environment resulting from the projection of the set of names into the name-space of the current definition. Mutually recursive definitions can see each other, hence their names are added to the set of names. The right-hand sides are renamed with the extended environment resulting from the projection of Ns_1 into the definitions name-space N_i. The set of names and the projection is similar to the H environment in the evaluation function.

$$\mathcal{R} :\ I \to (\mathit{Path} \hookrightarrow X) \to \mathcal{P}(\mathit{Path}) \to I$$

$$\begin{array}{lll}
\mathcal{R}\llbracket x(.x_i)^{i\in 1..n}\rrbracket\ \Gamma\ \emptyset & = \mathsf{splitPath}\ \Gamma\ (x(.x_i)^{i\in 1..n}) & \\
\mathcal{R}\llbracket I(.x_i)^{i\in 1..n}\rrbracket\ \Gamma\ \emptyset & = (\mathcal{R}\llbracket I\rrbracket\ \Gamma\ \emptyset)\ (.x_i)^{i\in 1..n} & \\
\mathcal{R}\llbracket x\rrbracket\ \Gamma\ \emptyset & = \Gamma(x) & \\
\mathcal{R}\llbracket I_1\ I_2\rrbracket\ \Gamma\ \emptyset & = (\mathcal{R}\llbracket I_1\rrbracket\ \Gamma\ \emptyset)\ (\mathcal{R}\llbracket I_2\rrbracket\ \Gamma\ \emptyset) & \\
\mathcal{R}\llbracket \lambda x.I\rrbracket\ \Gamma\ \emptyset & = \lambda x.\mathcal{R}\llbracket I\rrbracket\ (\Gamma \blacktriangleleft (x \mapsto x)) & \\
\mathcal{R}\llbracket \mathtt{SEQ}\ N\ {C_i}^{i\in 1..n}\rrbracket\ \Gamma_0\ \mathit{Ns} & = \mathtt{SEQ}\ N\ {C'_i}^{i\in 1..n} & \\
 & \text{where}\quad C'_i = \mathcal{R}\llbracket C_i\rrbracket\ \Gamma\ \mathit{Ns}_{i-1} & \\
 & \qquad\quad \mathit{Ns}_i = \mathsf{addToNs}\ \mathit{Ns}_{i-1}\ C_i & \\
\mathcal{R}\llbracket \mathtt{DEF}\ N\ S\rrbracket\ \Gamma\ \mathit{Ns} & = \mathtt{DEF}\ N\ (\mathcal{R}\llbracket S\rrbracket\ (\Gamma \blacktriangleleft (\mathit{Ns}|_N))\ \emptyset) & \\
\mathcal{R}\llbracket \mathtt{RECDEF}\ {(N_i\ S_i)}^{i\in 1..n}\rrbracket\ \Gamma\ \mathit{Ns} & = \mathtt{RECDEF}\ {(N_i S'_i)}^{i\in 1..n} & \\
 & \text{where}\quad S'_i = \mathcal{R}\llbracket S_i\rrbracket\ (\Gamma \blacktriangleleft (\mathit{Ns}_1|_{N_i}))\ \emptyset & \\
 & \qquad\quad \mathit{Ns}_1 = \mathit{Ns} \cup \{N_i^{i\in 1..n}\} &
\end{array}$$

Figure 6.7: Renaming function.

6.2.2 Compilation

The compilation of the renamed intermediate representation is straightforward. The function definition is given in Fig. 6.8. A term is compiled into the equivalent term. A `SEQ`-expression representing a Group hierarchy is compiled into a sequence of let-bindings and the record holding all selectors of this Group. Each inner component of a `SEQ` is compiled into a sequence of let-bindings. The concatenation of all those bindings followed by a binding $bind_{seq}$ is the result of the compilation of a `SEQ` $N\ \overline{C}$. $bind_{seq}$ binds the name N to the record holding all selectors x_i of this Group.

A `DEF` is compiled into a let-binding and a `RECDEF` into a recursive let-binding. The body of all bindings is created afterwards and is either the next binding or a record resulting from the compilation of `SEQ` $\varepsilon\ \overline{C}$.

The result of the compilation function for our running example is given in Fig. 6.9.

6.2.3 Advantage of Additional Edges

In the following, we return our attention to the additional edges. We want to discuss one of their advantages.

The target language with let-expressions can be used to generate machine code. During machine code generation, we have to take care of the limited number of registers. During the register allocation, variables must be assigned to registers. Two live variables cannot be assigned the same register. Variables which cannot be assigned to a register must be spilled to memory.

$$\mathcal{C} :\ I \to R$$

$$\begin{aligned}
\mathcal{C}[\![I\,.x]\!] &= \mathcal{C}[\![I]\!]\texttt{.}x \\
\mathcal{C}[\![x]\!] &= x \\
\mathcal{C}[\![I_1\ I_2]\!] &= (\mathcal{C}[\![I_1]\!])\ (\mathcal{C}[\![I_2]\!]) \\
\mathcal{C}[\![\lambda x\,.I]\!] &= \lambda x\texttt{.}\mathcal{C}[\![I]\!] \\
\mathcal{C}[\![\texttt{SEQ}\ \varepsilon\ C_i^{\,i\in 1..n}]\!] &= (bind_1 \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} bind_n)\ \texttt{<}(x_j\ \texttt{=}\ x_j)^{j\in 1..m}\texttt{>} \\
\text{where}\qquad bind_i &= \mathsf{cToBind}\ C_i \\
\{x_j^{j\in 1..m}\} &= \{x \mid \exists i \in 1..n : x \in \mathsf{names}\ C_i\}
\end{aligned}$$

$$\mathsf{cToBind} :\ C \to B$$

$$\begin{aligned}
\mathsf{cToBind}\ (\texttt{SEQ}\ N\ C_i^{\,i\in 1..n}) &= bind_1 \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} bind_n \mathbin{+\!\!+} bind_{seq} \\
\text{where}\qquad bind_i &= \mathsf{cToBind}\ C_i \\
bind_{seq} &= \texttt{LET}\ (\mathsf{toIdent}\ N)\ \texttt{=}\ \texttt{<}(x_j\ \texttt{=}\ N\texttt{.}x_j)^{j\in 1..m}\texttt{>}\ \texttt{IN} \\
\{x_j^{j\in 1..m}\} &= \{x \mid \exists i \in 1..n : N\texttt{.}x \in \mathsf{names}\ C_i\} \\
\mathsf{cToBind}\ (\texttt{DEF}\ N\ S) &= \texttt{LET}\ (\mathsf{toIdent}\ N)\ \texttt{=}\ \mathcal{C}[\![S]\!]\ \texttt{IN} \\
\mathsf{cToBind}\ (\texttt{RECDEF}\ (N_i\ S_i)^{i\in 1..n}) &= \texttt{LETREC}\ ((\mathsf{toIdent}\ N_i)\ \texttt{=}\ \mathcal{C}[\![S]\!]_i)^{i\in 1..n}\ \texttt{IN}
\end{aligned}$$

Figure 6.8: Compilation function.

We follow the conjecture that for big programs the number of live variables should be reduced to avoid spilling. Normally a module encapsulates closely related definitions. Therefore, we assume it is better, if those definitions are evaluated directly one after the other. As a consequence, the values of the other definitions are still in a register and spilling can be avoided. Furthermore, the construction of a record needs all inner definitions and in the case that they are evaluated immediately before the record creation, spilling may not be necessary. The additional edges ensure that a Group, which is not mutually recursive with another Group, is transformed into a `SEQ` only containing the definitions of that Group. This `SEQ` is compiled into a sequence of let-expressions only defining inner definitions of this Group.

Furthermore, the introduction of additional edges helps to reduce the number of live variables and the live range of variables introduced for module definitions.

To explain how the additional edges help to reduce the amount of live variables we return to the example used to motivate the additional edges. The original code is repeated in Fig. 6.10(a) and the result of the compilation is given in Fig. 6.10(b).

The definitions of the Group `y` are defined immediately before the Group is constructed. After the creation of the record `y`, the definitions can either be accessed directly or by selection.

```
LET E#val = 2 IN
LETREC E#even = λx. ... O#odd ...
       O#odd = λx. ... E#even ... IN
LET E#is2even = E#even E#val IN
LET E = <val = E#val, even = E#even, is2even = E#is2even> IN
LET E#is2odd = O#odd 2 IN
LET O = <odd = O#odd, is2odd = O#is2odd> IN
<E = E, O = O>
```

Listing 6.9: Result of compilation for even-odd-example.

```
{ x = { a = y.d + b
        b = 3}
  y = { e = 7
        d = e}
}
```

(a) GLang code.

```
LET y#e = 7 IN
LET y#d = y#e IN
LET y = <e = y#e, d = y#d>
IN
LET x#b = 3 IN
LET x#a = y#d + x#b IN
LET x = <a = x#a, b = x#b>
IN
<y = y, x = x>
```

(b) With additional edges.

```
LET x#b = 3 IN
LET y#e = 7 IN
LET y#d = y#e IN
LET x#a = y#d + x#b IN
LET x = <a = x#a, b = x#b>
IN
LET y = <e = y#e, d = y#d>
IN
<y = y, x = x>
```

(c) Without additional edges.

Figure 6.10: Motivating example for additional edges.

Considering the expression `y#e + x#b` in the definition `x#a`, we know that `y` is defined completely before this definition. As a consequence, we can replace this variable by a selection resulting in a new line 6: `LET x#b = y.d + x#b`. This results in a dead variable `y#d` after the creation of `y`. For this code the maximum amount of live variables is four.

A possible code resulting from a transformation without additional edges is given in Fig. 6.10(c). Here, we use the unwanted evaluation order given in Sec. 4.3. The usage of the variable `y#d` can not be replaced by the selection as the record `y` is not yet created. In this code, the maximum number of live variables is five.

The dependency order could always schedule the wrong Group first, which increases the number of live variables for bigger programs a lot. The reduction of live variables leads to more selections.

To replace as many variables by a selection from a record as possible, we can easily change the renaming function. The only part we have to change is the auxiliary function addToNs. In particular, we change the case of a `SEQ`:

$$\mathsf{addToNs}\ Ns\ (\mathtt{SEQ}\ N\ C_i^{i\in 1..n}) = Ns \cup \{N\} \cup \\ \{N_i \mid N_i \in \bigcup_{i=1}^{n} \mathsf{addToNs}\ Ns\ C_i \land N \not\sqsubseteq N_i\}$$

In this new version, those names are removed which are inner definitions of the corresponding Group. Therefore, all selections starting with N are renamed into N followed by the selection.

With this new version of the function `addToNs` we replace as many variable accesses as possible by selection. This is obviously not always the best strategy, as selection is more expensive than variable access as long as spilling is not necessary. An inlining optimizer can easily replace those selections by the variable again. An evaluation of the best optimizing techniques is future work.

The narrowing of definitions to their usage in order to reduce the live ranges of variables and the reduction of live variables is normally done by optimizing scheduling techniques. As optimal schedules are NP-complete [8], they all work with heuristics. In the transformation we use the knowledge of the program structure as a heuristic. We are confident this heuristic is suitable to reduce spilling.

7 Extending GLang

In this Chapter we discuss the extension of GLang by two different examples: manipulating modules and import statements.

7.1 Module Manipulation

In Sec. 2.4 we used the extensions proposed by Pepper and Hofstedt define a Group `SearchList` fulfilling the interface required by the search algorithm. These extensions are useful for better code reusability. In this section, we discus how extensions of this kind can be integrated into GLang, the existing transformation, and the evaluation function.

All kinds of extensions are term-expressions. The syntax of our examples is given in Fig. 7.1.

With `RENAME` we can rename some selectors of a Group. The restrictions create a new Group with fewer selectors. `ONLY` creates a Group only containing the given selectors and `WITHOUT` removes all given selectors.

The inheritance modification allows to add definitions to a module. The expression between `EXTEND` and `BY` gives the module to be extended. The extension is a Group-expression giving the new Group definitions.

For all these extensions we can add an equivalent term in the intermediate representation as well as in the record language.

7.1.1 Dependency Analysis

The renaming and restriction extensions are straightforward to integrate. All three are terms, so for the dependency analysis only $\mathcal{F}$ (Fig. 7.2) and $\mathcal{A}^T$ (Fig. 7.3) must be extended.

The inheritance is a bit more complicated. In object oriented languages, the definitions of the first expression are visible in the extending expression. In GLang this is not easily possible as the dependency analysis needs to calculate the free names for all expressions.

$$\begin{array}{lcll} T & ::= & E\ \texttt{\textbf{RENAME}}\ (\texttt{(}\ x\ \texttt{\textbf{AS}}\ x\ \texttt{)})+ & \text{— Renaming} \\ & | & E\ \texttt{\textbf{ONLY}}\ x+ & \text{— positive Restriction} \\ & | & E\ \texttt{\textbf{WITHOUT}}\ x+ & \text{— negative Restriction} \\ & | & \texttt{\textbf{EXTEND}}\ E\ \texttt{\textbf{BY}}\ G & \text{— Inheritance} \end{array}$$

Figure 7.1: Extensions for GLang language.

$$\begin{array}{ll}
\mathcal{F}[\![E \;\textbf{RENAME}\; (x_i \;\textbf{AS}\; y_i)^{i\in 1..n}]\!] & = \mathcal{F}[\![E]\!] \\
\mathcal{F}[\![E \;\textbf{ONLY}\; x_i^{\;i\in 1..n}]\!] & = \mathcal{F}[\![E]\!] \\
\mathcal{F}[\![E \;\textbf{WITHOUT}\; x_i^{\;i\in 1..n}]\!] & = \mathcal{F}[\![E]\!] \\
\mathcal{F}[\![\textbf{EXTEND}\; E \;\textbf{BY}\; G]\!] & = \mathcal{F}[\![E]\!] \cup \mathcal{F}[\![G]\!]
\end{array}$$

Figure 7.2: Free names function for the extensions.

To illustrate the problem of free names, we give a small example in Listing 7.4. Here the result of `A` is extended by a definition `y`.

```
{ A = IF p THEN { x = 3 } ELSE { x = 5 }
  E = EXTEND A BY { y = x }
  x = E.y
}
```

Listing 7.4: Extension example.

To calculate the free names of `E`, the selectors of `A` must be calculated, as they are bound in `E`. This is not possible in all cases (see Sec. 2.5.3).

In the following we discuss two possible solutions: The definitions of the first expression are not visible, or the calculation of free names results in a superset.

If the definitions of the first expression are not visible in the second, the free names can easily be calculated by the union of the free names of both expressions. The example must be rewritten, as the definition `x` is not visible in the definition `y`. The correct program for our example is given in Listing 7.5. Here `x` must be selected from `A`.

```
{ A = IF p THEN { x = 3 } ELSE { x = 5 }
  E = EXTEND A BY { y = A.x }
  x = E.y
}
```

Listing 7.5: Correct version if the definitions of the extended Group are not visible.

The other possibility is a bit more complicated. We can not calculate the selectors of the first expression but we can calculate a superset of the free names by the union of the free names of both expressions. This union is a superset of the free names, as the free names of the extending Group-expression which are defined in the first expression are not free in the whole expression. In our example, `x` is in the set of free names of the

$$\begin{array}{ll}
\mathcal{A}^T[\![[E \;\textbf{RENAME}\; (x_i \;\textbf{AS}\; y_i)^{i\in 1..n}]^\ell]\!]\;\Gamma & = \{\ell \hookrightarrow \Gamma\} \cup (\mathcal{A}^E[\![E]\!]\;\Gamma) \\
\mathcal{A}^T[\![[E \;\textbf{ONLY}\; x_i^{\;i\in 1..n}]^\ell]\!]\;\Gamma & = \{\ell \hookrightarrow \Gamma\} \cup (\mathcal{A}^E[\![E]\!]\;\Gamma) \\
\mathcal{A}^T[\![[E \;\textbf{WITHOUT}\; x_i^{\;i\in 1..n}]^\ell]\!]\;\Gamma & = \{\ell \hookrightarrow \Gamma\} \cup (\mathcal{A}^E[\![E]\!]\;\Gamma) \\
\mathcal{A}^T[\![[\textbf{EXTEND}\; E \;\textbf{BY}\; G]^\ell]\!]\;\Gamma & = \{\ell \hookrightarrow \Gamma\} \cup (\mathcal{A}^E[\![E]\!]\;\Gamma) \cup (\mathcal{A}^E[\![G]\!]\;\Gamma)
\end{array}$$

Figure 7.3: Group-bound available variables for extensions.

$$\mathcal{E}[\![x]\!]\ \Gamma\ \mathsf{H} = \begin{cases} v, & \text{if } \Gamma(x) = v \\ \text{ERROR}, & \text{otherwise} \end{cases}$$

Figure 7.6: Variable access adaption in evaluation function.

extending expression, but as `x` is defined within `A`, it is not free in `E`. Hence, the correct set of free names for `E` is $\{\mathsf{A}\}$ but the union is $\{\mathsf{A}, \mathsf{x}\}$.

For the second solution we have to adapt the context analysis. The check, whether all variables are defined is not possible anymore, as the set of free paths is not exact. As GLang is a dynamic language this is no problem, but we have to check this in the evaluation functions instead. We skip this check and instead, we adapt the access of variables in the evaluation function as shown in Fig. 7.6. As undefined variables are possible, we must add the check whether a variable is defined or not in the evaluation function.

The problem with this solution is that correct programs are rejected by the context analysis during the check of allowed recursion. For example the program in Listing 7.4 is rejected, as the analysis detects a cycle between `E` and `x`. As `x` is defined by `A` it would not be free in an exact calculation of free names, and hence there would not be a cycle.

The ordering of the two expressions in the inheritance must not be calculated, as it is always the same—the first expression must be evaluated before the second. For this reason we do not have to insert any dependency edges for free names of the extending expression defined in the first expression. In the dependency analysis we do not have to adapt anything. We insert the dependency edges as described in Sec. 4.1.

The transformation for all extensions is straightforward: as all extensions are terms, they are transformed into the equivalent term in the intermediate representation.

7.1.2 Evaluation

The extension of the evaluation function is given in Fig. 7.7, where the two definitions for `EXTEND` arise from the two possible solutions.

As all four extensions are only defined for Groups, the evaluation will lead to an error if applied to some other value. In the case of renaming all definitions of the Group value are copied to the result by renaming the given selectors. The `ONLY` restriction only copies the given definitions and the `WITHOUT` restriction copies all except the given ones. In all cases non existing selectors are discarded.

For `EXTEND`, the first expression must be evaluated. If the definitions of the first expression are not visible in the extending expression, the second expression is evaluated in the given environment. In the other case, all definitions of the result of the evaluation of the first expression are inserted in the environment. The extending expression is evaluated in this new environment.

It is not possible to redefine a definition of the first expression by the extending Group. Therefore, the domains of both Group values must be disjoint.

$$\mathcal{E}[\![I \texttt{ RENAME } (x_i \texttt{ AS } y_i)^{i\in 1..n}]\!]\ \Gamma\ \mathsf{H} = \begin{cases} v_R, & \text{if } v_I = \{(z_j \twoheadrightarrow v_j)^{j\in 1..m}\} \\ \text{ERROR}, & \text{otherwise} \end{cases}$$

$$\begin{aligned} \text{where}\quad v_R &= \{y_i \twoheadrightarrow v_j \mid j \in 1..m \wedge i \in 1..n \wedge x_i = z_j\} \cup \\ &\quad \{z_j \twoheadrightarrow v_j \mid j \in 1..m \wedge x \notin \{x_i^{i\in 1..n}\}\} \\ v_I &= \mathcal{E}[\![I]\!]\ \Gamma\ \mathsf{H} \end{aligned}$$

$$\mathcal{E}[\![I \texttt{ ONLY } x_i^{i\in 1..n}]\!]\ \Gamma\ \mathsf{H} = \begin{cases} v_R, & \text{if } v_I = \{(z_j \twoheadrightarrow v_j)^{j\in 1..m}\} \\ \text{ERROR}, & \text{otherwise} \end{cases}$$

$$\begin{aligned} \text{where}\quad v_R &= \{z_j \twoheadrightarrow v_j \mid j \in 1..m \wedge z_j \in \{x_i^{i\in 1..n}\}\} \\ v_I &= \mathcal{E}[\![I]\!]\ \Gamma\ \mathsf{H} \end{aligned}$$

$$\mathcal{E}[\![I \texttt{ WITHOUT } x_i^{i\in 1..n}]\!]\ \Gamma\ \mathsf{H} = \begin{cases} v_R, & \text{if } v_I = \{(z_j \twoheadrightarrow v_j)^{j\in 1..m}\} \\ \text{ERROR}, & \text{otherwise} \end{cases}$$

$$\begin{aligned} \text{where}\quad v_R &= \{z_j \twoheadrightarrow v_j \mid j \in 1..m \wedge z_j \notin \{x_i^{i\in 1..n}\}\} \\ v_I &= \mathcal{E}[\![I]\!]\ \Gamma\ \mathsf{H} \end{aligned}$$

$$\mathcal{E}[\![\texttt{EXTEND } I \texttt{ BY } C]\!]\ \Gamma\ \mathsf{H} = \begin{cases} v_I + v_C, & \text{if } v_I = \{(z_j \twoheadrightarrow v_j)^{j\in 1..m}\} \wedge \\ & \qquad \mathrm{dom}(v_C) \cap \mathrm{dom}(v_I) = \emptyset \\ \text{ERROR}, & \text{otherwise} \end{cases}$$

$$\begin{aligned} \text{where}\quad v_I &= \mathcal{E}[\![I]\!]\ \Gamma\ \mathsf{H} \\ v_C &= \mathcal{E}[\![C]\!]\ \Gamma\ \mathsf{H} \end{aligned}$$

$$\mathcal{E}[\![\texttt{EXTEND } I \texttt{ BY } C]\!]\ \Gamma\ \mathsf{H} = \begin{cases} v_I + v_C, & \text{if } v_I = \{(z_j \twoheadrightarrow v_j)^{j\in 1..m}\} \wedge \\ & \qquad \mathrm{dom}(v_C) \cap \mathrm{dom}(v_I) = \emptyset \\ \text{ERROR}, & \text{otherwise} \end{cases}$$

$$\begin{aligned} \text{where}\quad v_I &= \mathcal{E}[\![I]\!]\ \Gamma\ \mathsf{H} \\ v_C &= \mathcal{E}[\![C]\!]\ \Gamma'\ \mathsf{H} \\ \Gamma' &= \Gamma \lhd \{x \mapsto v \mid x \twoheadrightarrow v \in v_I\} \end{aligned}$$

Figure 7.7: Evaluation function for extensions.

7.2 Import

Pepper and Hofstedt proposed two different kinds of imports in [31]: `USE` statements and `IMPORT` statements. Both import statements add definitions into the name-space of the current Group. The latter additionally forbids all other outer definitions. The imports we discuss in this section are introduced by the keyword `USE`.

Most module languages provide an import statement. There are two possibilities: qualified imports and wildcard imports. With a qualified import all definitions of the given module are accessible through a special identifier, with a wildcard import they are accessible directly. For example in the Scala language both cases exit:

```
import Module1.Module2
import Module1.Module3._
```

In the line 1, the module `Module2` is added to the scope, so all definitions of `Module2` are accessible through selections from `Module2`. In line 2, all definitions of module `Module3` are added to the scope.

Qualified imports are already possible in GLang as we can define an import by giving a path abbreviation:

```
{ Module2 = Module1.Module2 }
```

For convenience, we add an import statement to GLang and replace all import statements by the corresponding path abbreviations before the dependency analysis.

This import statement for our example is `USE Module1 ONLY Module2` (notation from Pepper and Hofstedt).

Using the restriction `ONLY` we can easily give a list of imports from one module. Furthermore, we can introduce new names for the imported modules through the renaming extension. Additionally, we can import single definitions from a module through the `ONLY` restriction.

As Groups are first-class values, it is possible to import from an expression evaluating to a Group value. In the following example the function `findOne` for a depth-first-search from Sec. 2.4 is imported:

```
USE TreeSearch SearchList ONLY findOne
```

Here we give the Group to import from by an expression.

We can summarize all possibilities for a qualified import by the following syntax extension:

$$D \quad ::= \quad \texttt{USE}\ E\ \texttt{ONLY}\ x+ \quad — \quad \text{qualified import}$$

All import statements of this kind are transformed by the source to source transformation $\mathcal{I}$:

$$\mathcal{I}:\ D \to D^+$$
$$\mathcal{I}[\![\texttt{USE}\ E\ \texttt{ONLY}\ x_i^{i\in 1..n}]\!] = (x_i \texttt{=} E\texttt{.}x_i)^{i\in 1..n}$$

Here for each imported definition a path abbreviation is inserted. To avoid the multiple evaluation of the expression E, it could be assigned to a fresh variable first.

Despite its simplicity, the extension of GLang by wildcard imports is difficult. The problem arises as modules are first-class values. To demonstrate the difficulties we look at the following import statement:

```
USE Module1.Module3
```

As modules are first class values, this would be a correct statement, as long as `Module3` evaluates to a Group. From the undecidability of path resolution [26] it follows, that we can not calculate the set of all selectors in general. Consequently, we can not transform this statement into a path abbreviation in all cases. Nevertheless, we can extend GLang by import statements of the second kind, but we have to restrict the possibilities.

The simplest possibility is a restriction similar to the one used in the path resolution: In an import statement only a path is allowed. This path must be resolvable to a definition, where the right-hand side is syntactically a Group. For all import statements, we can give a list of path abbreviations, as we know all selectors of the imported Group.

Similar to the qualified import, we extend GLang by a wildcard import:

$$D \quad ::= \quad \texttt{USE}\ p \quad \text{— restricted wildcard import}$$

These import statements can be transformed into qualified imports using the path resolution defined in Sec. 3.7.

$$\mathcal{I}:\ D \to D$$
$$\mathcal{I}[\![[\texttt{USE}\ p]^\ell]\!] = \begin{cases} \texttt{USE}\ p\ \texttt{ONLY}\ x_i^{\,i\in 1..n}, & \text{if } E_r = \{(x_i = E_i)^{i\in 1..n}\} \\ \text{ERROR}, & \text{otherwise} \end{cases}$$
$$\text{where} \quad E_r = \mathsf{resolve}\ \ell\ p$$

Import statements giving a path and no restriction are only allowed, if the given path can be resolved to a Group definition. All correct import statements can be transformed into qualified imports by naming all selectors of the Group.

As Groups are first-class values, it is desirable to have import statements not restricted to a path expression. For example, the following import statement imports the complete depth-first-search defined in Sec. 2.3 into the current Group:

```
USE TreeSearch SearchList
```

To integrate import statements of this kind, we need to determine the set of possible selectors the expression introduces. This is not possible in general. In Sec. 8.2 we discuss a possibility to integrate such kinds of imports without the `ONLY` restriction. But in principle we can say, the programmer should always know which definitions are used from an import statement, hence the `ONLY` restriction can be added easily.

8 Conclusion and Future Work

8.1 Conclusion

In this thesis we have demonstrated a novel way to introduce mutually recursive first-class modules into a call-by-value language. All modules are accessed through dot-notation without any additional syntax for mutually recursive modules. This especially means that neither special syntax for module composition, nor the introduction of a self variable is needed. This user-friendly mechanism is achieved through our dependency analysis, which computes all the necessary information. Hence, the programmer does not need to provide this information. The presented extensions as well as the first-class property of modules provide a flexible module system that allows multiple ways of code reuse.

The evaluation order for all definitions is determined by a novel dependency analysis. To detect the dependencies between definitions, we have defined a novel path resolution algorithm. Unlike other path resolutions, this restricted algorithm is decidable for mutually recursive first-class modules. To ensure the termination of the dependency analysis, we have defined a context analysis.

A transformation for GLang into an intermediate representation with explicit ordering is given. The transformation is based on the dependency analysis, which ensures that Groups are transformed into components that contain all definitions of the corresponding Group. For this intermediate representation, an evaluation function is presented. The evaluation function together with the transformation defines the semantics for GLang. Furthermore, a compilation into a simple functional language with dynamic records is defined. Additionally, the way to extend the module language with extensions for module manipulation is developed and some examples are laid out. Two import statements enabling path abbreviations are presented.

We have implemented the transformation as well as the evaluation function and the compilation into the functional language with dynamic records for the complete language including matches.

The described dependency analysis as well as the transformation can be easily extended for other term-expressions. Consequently, the proposed module system is nearly language-independent. Our approach provides an easy and flexible module system that can be used for other call-by-value languages.

8.2 Future Work

This thesis establishes the fundamentals for a module system with mutually recursive first-class modules in call-by-value languages. There are particularly three areas where future work could be carried out and open questions could to be answered: language extensions for GLang, extension of the path resolution algorithm, and code generation and optimization.

8.2.1 Language Extensions

In Chap. 7, we have presented several examples for module manipulation and have shown how these extensions can easily be integrated into the existing transformation. Similar extensions can be integrated the same way. An open question is the integration of late bindings used for Mixins. Here, a construct similar to `EXTEND` is needed that does not evaluate the left module first and allows to override definitions of this module by the following definitions. Especially the influence on the dependency analysis needs to be studied.

We discuss the `USE` statement proposed by Pepper and Hofstedt. We needed some restrictions for this import statement. In his Bachelor thesis [3], Benjamin Bisping implemented a control flow analysis to determine a superset of the set of selectors of a GLang-expression. The results of this control flow analysis could be used to establish an import statement without any restrictions.

The control flow analysis calculates a superset of the selectors. In a similar way, a subset of the selectors can be determined. In the simplest case, this subset is always empty. For each selector within the subset a path abbreviation is added. For all remaining selectors within the superset, a “possible” definition is added. In the case the path resolution resolves a path to such a possible definition, the path can possibly refer to another definition. Therefore, the path resolution does not terminate in a possible definition but tries to find another definition. Consequently, not only one dependency edge for each path is added, but a set of edges. This leads to more dependency edges than necessary and possibly to forbidden cycles containing possible definitions. Therefore, possibly correct programs would be rejected. In those cases, the programmer must give an `ONLY` restriction as described in Chap. 7.

8.2.2 Improve Path Resolution

We have restricted the path resolution by omitting application and variable dereferencing. Therefore, the program in Listing 8.1 is rejected, as this example defines a forbidden cycle: `O.odd` depends on `v`, `v` depends on `E`, `E` depends on `E.even`, and `E.even` depends on `O.odd`. A path resolution that examines variable dereferencing would resolve the dependency of `O.odd` to `E.even`. The cycle could be reduced into a cycle only containing `even` and `odd`. This is an allowed cycle as both definitions are λ-abstractions.

Although this looks pretty easy, we have omitted variable dereferencing, as the simple version of this path resolution would not terminate in all cases. An example for which the path resolution would not terminate is shown in Listing 8.2. This program would be rejected by the context analysis, but this happens after path resolution. Therefore, a mechanism to detect cycles in variable dereferencing is necessary to define a terminating path resolution.

```
{ E = { even = λn. IF n==0 THEN true ELSE O.odd (n-1)
        is2even = even val
        val = 2 }
  v = E
  O = { odd = λn. IF n==0 THEN false ELSE v.even (n-1)
        is2odd = odd 2 }
}
```

Listing 8.1: Limitation by path resolution.

```
{ a = b
  b = c
  c = a }
```

Listing 8.2: Termination problem in path resolution.

Although a completely unrestricted path resolution is undecidable, it would be possible to not only resolve variable dereferencing but to resolve some function applications. To ensure a decidable path resolution, the reducible applications must be restricted, similar to the solution of Garrigue and Nakata in [26].

The transformation as well as the code generation need not to be changed in the case of an extended path resolution. The definition of allowed recursion is based on the path resolution. Consequently, extending the path resolution will reduce the simplicity for the programmer. Especially in the case of function applications, it is not easy to know in advance which programs will be rejected and which can be resolved.

8.2.3 Compilation

In the current transformation, the complete program is translated as a whole. When extending the language with some mechanisms for hiding, we could use this information to identify independent parts that can be compiled separately.

Therefore, it is important to find a mechanism to hide outer definitions or to define on which outer definitions a Group depends on. A simple way would be to use a package mechanism similar to Java, but it would be nice to have it at every level, so a special mechanism would be quite useful.

In this thesis, we have developed a way to generate code for GLang. We translated GLang into a simple functional language with records and let-bindings. To generate efficient code for this language, a good representation for these records is needed. For this purpose, one should utilize the fact that these records are always created with a known size and are never extended later. As there are potentially multiple selections from those records, a low-cost look-up function is necessary.

Bibliography

[1] Harold Abelson, Gerald J. Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 2nd edition, 1996.

[2] Davide Ancona and Elena Zucca. A calculus of module systems. *J. Funct. Program.*, 12:91–132, March 2002.

[3] Benjamin Bisping. Implementing control flow analysis for first-class modules. Bachelor thesis, 2014.

[4] Matthias Blume. *Hierarchical Modularity and Intermodule Optimization*. PhD thesis, Princeton University, November 1997.

[5] Matthias Blume. Dependency analysis for Standard ML. *ACM Trans. Program. Lang. Syst.*, 21(4):790–812, July 1999.

[6] Gilad Bracha and William Cook. Mixin-based inheritance. *SIGPLAN Not.*, 25(10):303–311, September 1990.

[7] Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. In Manfred Broy and C. B. Jones, editors, *Proceedings IFIP TC2 working conference on programming concepts and methods*, pages 479–504. North-Holland, 1990.

[8] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47 – 57, 1981.

[9] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, 1989.

[10] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? *SIGPLAN Not.*, 34(5):50–63, May 1999.

[11] Klaus Didrich, Andreas Fett, Carola Gerke, Wolfgang Grieskamp, and Peter Pepper. Opal: Design and implementation of an algebraic programming language. In Jürg Gutknecht, editor, *Programming Languages and System Architectures*, volume 782 of *LNCS*, pages 228–244. Springer, March 1994.

[12] Matthew Flatt and Matthias Felleisen. Units: cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 236–248, New York, NY, USA, 1998. ACM.

[13] Matthew Flatt, Robert Bruce Findler, and PLT. *The Racket Guide*, 2014.

[14] Alain Frisch and Jacques Garrigue. First-class modules and composable signatures in Objective Caml 3.12. extended abstract for talk presented at the 2010 ACM SIGPLAN Workshop, 2010.

[15] Michael J. C. Gordon. *The Denotational Description of Programming Languages.* Springer-Verlag, 1979.

[16] James Gosling, Bill Joy, Guy L. Steele, Jr., Gilad Bracha, and Alex Buckley. *The Java® Language Specification – Java SE 8 Edition.* Oracle America, Inc., 2014.

[17] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 123–137, New York, NY, USA, 1994. ACM.

[18] Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. *ACM Trans. Program. Lang. Syst.*, 27:857–881, September 2005.

[19] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. *Lua 5.1 Reference Manual.* Lua.Org, 2006.

[20] George Kuan and David B. MacQueen. Engineering higher-order modules in SML/NJ. In *Proceedings of the 21st International Conference on Implementation and Application of Functional Languages*, IFL'09, pages 218–235, Berlin, Heidelberg, 2010. Springer-Verlag.

[21] David B. MacQueen. Modules for Standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 198–207, New York, NY, USA, 1984. ACM.

[22] David B. MacQueen. An implementation of Standard ML modules. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, LFP '88, pages 212–223, New York, NY, USA, 1988. ACM.

[23] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In Donald Sanella, editor, *Programming Languages and Systems — ESOP '94*, volume 788 of *Lecture Notes in Computer Science*, pages 409–423. Springer Berlin Heidelberg, 1994.

[24] Robin Milner, Mads Tofte, and David B. MacQueen. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA, 1997.

[25] Keiko Nakata and Jacques Garrigue. Recursive modules for programming. *SIGPLAN Not.*, 41:74–86, September 2006.

[26] Keiko Nakata and Jacques Garrigue. Path resolution for nested recursive modules. *Higher-Order and Symbolic Computation*, pages 1–31, May 2012.

[27] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proc. ECOOP'03*, Springer LNCS, 2003.

[28] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 41–57, New York, NY, USA, 2005. ACM.

[29] Martin et al. Odersky. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[30] Scott Owens and Matthew Flatt. From structures and functors to modules and units. *SIGPLAN Not.*, 41(9):87–98, September 2006.

[31] Peter Pepper and Petra Hofstedt. *Funktionale Programmierung – Sprachdesign und Programmiertechnik.* Springer, 2006.

[32] Peter Pepper and Florian Lorenzen. *The programming language Opal – 6th corrected edition.* TU Berlin, 2012.

[33] Simon L. Peyton Jones and Mark B. Shields. First class modules for Haskell. In *9th International Conference on Foundations of Object-Oriented Languages (FOOL 9), Portland, Oregon*, pages 28–40, January 2002.

[34] Claus Reinke. *Functions, Frames, and Interactions – completing a lambda-calculus-based purely functional language with respect to programming-in-the-large and interactions with runtime environments.* PhD thesis, Universität Kiel, 1997.

[35] Andreas Rossberg and Derek Dreyer. Mixin' up the ML module system. *ACM Trans. Program. Lang. Syst.*, 35(1):2:1–2:84, April 2013.

[36] Claudio V. Russo. First-class structures for Standard ML. *Nordic J. of Computing*, 7:348–374, December 2000.

[37] Claudio V. Russo. Recursive structures for Standard ML. *SIGPLAN Not.*, 36(10):50–61, October 2001.

[38] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.

[39] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* Computer Science Series. MIT Press, 1981.

[40] Robert D. Tennent. The denotational semantics of programming languages. *Commun. ACM*, 19(8):437–453, 1976.