Language Engineering

domainSpecificLanguages :: TANotes

1 Introduction

This set of TA notes provides an explanation on the concepts of DSLs coming from a different pair of eyes than your lecturer, they will also be related to the different weeks of the worksheets. The relationship will be that the worksheet with a named section topic should be manageable by reading the TA notes with the same corresponding name. In this case these notes cover domain specific languages which means that by reading these notes it should be possible to do weeks 1 and 2 of the worksheets. Thank you for taking the time to read them, if you have any questions please feel free to ask any of the TA's in the lab sessions.

2 Programming Languages

There are two main branches of programming languages: general purpose programming languages and domain specific programming languages. General purpose languages (GPLs) are Turing-complete, meaning they can compute any function, but can also be uncontrolled. Domain specific languages (DSLs) on the other hand are specialised for a specific use, and may not be Turing-complete. GPLs can be split into imperative languages, where each computational step is clearly stated (e.g. C, Java, Python, etc.), and declarative. Declarative languages can either have a functional paradigm, such as Haskell, Ocaml, ML, etc. which are pure languages that compose functions together for computation, or logical paradigm, such as Prolog.

3 General Purpose Languages

Most programming languages that you have been using are most likely a general purpose language (GPL) which means that the language is Turing complete allowing you to code any possible program using it. This is a very useful property but can cause a wealth of features that don't just cause bloat but can be detrimental to your programming practice. An example of a language that you may have used but isn't general purpose would be $HTML$ as this allows for efficient creation of websites without allowing for unnecessary functions that could slow down the website or cause unknown bugs.

4 Domain Specific Languages

A domain specific language (DSL) is a language that isn't general purpose and thus not Turing complete, like $HTML$. This however brings about a new problem of scope of use, where $HTML$ is heavily used lots of support can be implemented for it like compilers, IDEs and other useful features. If you wanted to create a new domain specific language that has a smaller scope, you would still want all of these fun and useful features, but you don't have the time to create them all yourself. This causes a major issue and the way to get around that is by embedding you DSL into another language, like $Haskell$ for example. $Haskell$ has all of these good compiler optimizations, IDEs and other functionality that you can steal from and want to use, this saves a lot of time and effort. This brings us to the next issue, how do I actually embed a DSL into a GPL in the first place.

5 Embedding *DSL*s into *GPL*s

This is where the implementation of stuff begins, as there arises two methods of embedding (alhough later we will reveal a spectrum between the two). The two methods that this set of notes will talk about are *Deep* and *Shallow* embeddings which have varying advantages and disadvantages but follow a somewhat similar structure with distinct methodology between the two. *Deep* embedding is a method of creating a data structure in the *GPL* and deriving semantic functions from this data structure. *Shallow* embedding is a method of only using functions to derive a direct evaluation from combining the functions.

5.1 Designing a *DSL*

We will be thinking of a very basic *DSL* to be used as an example throughout the rest of these notes which allows you to view how the basics work without getting caught up in endless lines of code. This will present some problems of over simplicity not revealing the true potential of these methodologies but is a good starting basis. The context that we will look at is a class, a very basic data structure that helps to model a lecture theater or classroom by modeling the entities within it. This will be shown to be done using a *Deep* and *Shallow* embedding.

5.2 *Deep* Embedded *DSL*

To create this type of embedding inside haskell, we will first need to create a data structure that encompasses the model of the item of interest. For example in this case we start with a *Room* and add entities to it by appending them to the given room using the data structure.

```
data Class = Room
           | Lecturer Class
           | Student Class
           | Ghost Int Class
```

This data structure is created using **data** and is called *Class*. The first word to the right of the $=$ or $|$ is the data constructor which defines a possible version of the given data type. In this case we have a terminating (base) case of *Room* which is a lone data constructor and ends the structure. The next constructor *Lecturer* has another data type attached to it, in this case *Class* which allows for recursion of the data structure as it means 'I am a Lecturer and give me the rest of the Class'. Another constructor, *Student*, is then given which also takes in *Class*, but despite seeming similar, with an ability to recurse, the constructor difference means it could imply different things in different contexts. The final constructor *Ghost* can also recurse through the other entities using *Class* but also takes in an integer value which relates to its psycho-kinetic energy, P.K.E. (Ghostbusters).

We can then create a function that uses this data type and is easily created using pattern matching on the data constructors. This allows for a readable function, however if you end up with a large number of data constructors it can become very messy but there are ways around this problem, which are covered at a later part in this set of notes. We shall call this function *peopleInClass* and it will count the number of people present in the *Class*.

```
peopleInClass :: Class → Integer
peopleInClass Room          = 0
peopleInClass (Lecturer c)  = 1 + peopleInClass c
peopleInClass (Student c)   = 1 + peopleInClass c
peopleInClass (Ghost n c)   = 1 + peopleInClass c
```

The first thing to note about any function is its type and the type of *peopleInClass* is $Class \to Integer$, in this case that means that it takes the new data type we constructed and maps it to an integer value. We would like this integer value to be the number of people present in the class and thus when we pattern match we can work on this basis. The first data construct that we match to is *Room* which in our case is the base case and means that there are no more people in the given class, thus this output is relatively easy as the corresponding integer value also needs to be the base case at 0. Next we pattern match onto the next data constructor which is *Lecturer*, this implies that there is one person in the class and then we use the rest of the class and look for a function that has an integer output. As we want an output of integer the only outputs available to us are basic numbers, other prelude functions and functions that we have designed with given inputs. We have some unused input of the rest of the class, thinking about the output that we want we want, everyone in the $Class$, it would be sensible to use this value. Now we need to find a function that maps $Class$ to $Integer$ and the only function we have for that is $peopleInClass$ so this would be the best option for the remainder of the input. This results in the rest of the class being added up as well as adding 1 to the value for the $Lecturer$ that is in there. Next we will look at the $Student$ pattern match which in this case works the same as $Lecturer$ as they are treated the same when just counting up the number of people in a room. Lastly we need to pattern match on the final pattern of $Ghost$ and it will add 1 to the value and again we want an extra value but still need an $Integer$ output. We then have two possibilities, the same function that we used earlier as well as the P.K.E. of the ghost which despite being an $Integer$ isn't what we want and thus we use $peopleInClass$ with the rest of the class.

This means that we have ended up with a new data type $Class$ that encodes the information of a class, or at least the entities within it, and then have semantic functions that encode the outputs. An example can be shown below:

$$\begin{aligned}&exampleDeep = Student\ (Student\ (Ghost\ 3\ (Lecturer\ (Student\ (Room)))))\\&noOfPeople = peopleInClass\ exampleDeep\end{aligned}$$

5.3 *Shallow* **Embedded** *DSL*

This type of embedding uses only functions to map the required outputs using functions instead of constructors. The semantic that we want in this case is the number of people in the class and thus this can be calculated intuitively working from the base cases and thus base function.

$$\begin{aligned}&room :: Integer\\&room = 0\end{aligned}$$

The first base case is the room function which states that there is no one left in the room and is thus just a semantic output with no value, in this case equaling 0 to show there is no one left.

$$\begin{aligned}&lecturer :: Integer \to Integer\\&lecturer\ c = 1 + c\end{aligned}$$

The next function is mapping a $lecturer$ being in the room to the required output, this takes in an Int value being the rest of the room's semantic output and increments it by 1 as there is now another person in the room. This seems similar to the $Deep$ embedding above but the $Class$ data construct has been replaced with the Int value of the complete semantic output.

$$\begin{aligned}&student :: Integer \to Integer\\&student\ c = 1 + c\end{aligned}$$

This function works the same as the $lecturer$ function above by using the 1 to increment a value and then requiring an integer output also uses the input which is already in an integer format.

$$\begin{aligned}&ghost :: Integer \to Integer \to Integer\\&ghost\ n\ c = 1 + c\end{aligned}$$

The last function encodes the ghost entity into the output and takes in the value required, P.K.E., and the rest of the functions as the second input to then map the required output.

You can construct an instance of this by adding in the remainder of the functions as the input to the higher level functions like so:

$$exampleShallow = student\ (student\ (ghost\ 3\ (lecturer\ (student\ (room)))))$$

6 Extending an Embedding

This section will be extending the above $Deep$ and $Shallow$ embedding with various different extensions, namely more semantic outputs, more constructs and finding a middle ground between the two types of embeddings.

6.1 New Semantic Functions

First let's look at the much easier case of the *Deep* embeddings for new semantic functions. Let this function calculate the proportion of people that are lecturers in the room as a decimal value. This function relies on values from a different function at every stage of it's recursion to be able to compute, this is called dependent interpretation.

$$
\begin{aligned}
&\mathit{fInt} = \mathit{fromInteger}\\[2em]
&\mathit{decLec} :: \mathit{Class} \to \mathit{Float}\\
&\mathit{decLec}\ (\mathit{Room}) \qquad\ \ = 0\\
&\mathit{decLec}\ (\mathit{Lecturer}\ c) = (\mathit{decLec}\ c * \mathit{fInt}\ (\mathit{peopleInClass}\ c) + 1)\ /\ (\mathit{fInt}\ (\mathit{peopleInClass}\ c) + 1)\\
&\mathit{decLec}\ (\mathit{Student}\ c)\ \ = (\mathit{decLec}\ c * \mathit{fInt}\ (\mathit{peopleInClass}\ c) + 0)\ /\ (\mathit{fInt}\ (\mathit{peopleInClass}\ c) + 1)\\
&\mathit{decLec}\ (\mathit{Ghost}\ n\ c) = (\mathit{decLec}\ c * \mathit{fInt}\ (\mathit{peopleInClass}\ c) + 0)\ /\ (\mathit{fInt}\ (\mathit{peopleInClass}\ c) + 1)
\end{aligned}
$$

This function works using fancy mathematics in that to keep a running decimal total you can track the total number of people and the decimal, in our case both are tracked using the rest of the class. Then when a new input is added you scale up the decimal to get the total number, increment this value and then scale it back down using the new total number of people. Using a dependent interpretation where it uses the *peopleInClass* function to calculate the total number of people and then using this value and the previous number of lecturers, using the function *decLec*, it can calculate the new decimal.

To do this within a *Shallow* embedding we will need to update the above functions and thus we are forced to recreate them. This can of course be a massive hassle if this needs to be done repetitively, thus showing the advantage of a *Deep* embedding in this particular instance. We then also come across another big problem, the need for dependent interpretation. When looking at the above *Shallow* functions it only ever calculates the one semantic output forcing it to be independent interpretation, thus we need to find a way to get around this as it is impossible to do extra function calls in the *Shallow* embedding. The way around this is to use tuples, $(,)$, this allows for two semantic outputs to be calculated at the same time and both to be referenced by the other.

$$
\begin{aligned}
&\mathit{room} :: (\mathit{Integer}, \mathit{Float})\\
&\mathit{room} = (0, 0)\\[2em]
&\mathit{lecturer} :: (\mathit{Integer}, \mathit{Float}) \to (\mathit{Integer}, \mathit{Float})\\
&\mathit{lecturer}\ (\mathit{people}, \mathit{dec}) = (\mathit{people} + 1, (\mathit{dec} * \mathit{fInt}\ (\mathit{people}) + 1)\ /\ (\mathit{fInt}\ (\mathit{people}) + 1))\\[2em]
&\mathit{student} :: (\mathit{Integer}, \mathit{Float}) \to (\mathit{Integer}, \mathit{Float})\\
&\mathit{student}\ (\mathit{people}, \mathit{dec}) = (\mathit{people} + 1, (\mathit{dec} * \mathit{fInt}\ (\mathit{people}) + 0)\ /\ (\mathit{fInt}\ (\mathit{people}) + 1))\\[2em]
&\mathit{ghost} :: \mathit{Integer} \to (\mathit{Integer}, \mathit{Float}) \to (\mathit{Integer}, \mathit{Float})\\
&\mathit{ghost}\ n\ (\mathit{people}, \mathit{dec}) = (\mathit{people} + 1, (\mathit{dec} * \mathit{fInt}\ (\mathit{people}) + 0)\ /\ (\mathit{fInt}\ (\mathit{people}) + 1))
\end{aligned}
$$

These functions again look similar to the ones above with the first part of the tuple being exactly the same, the second part of the tuple now however encodes the proportion of lectures as a decimal value. To allow this function to use dependent interpretation it calls the value p, in this case, which is the current value of the number of people, and this can then be used within the new semantic. Thus the problem has been solved but now the output is forced to be all of the different possible semantics that the one semantic output you want needs in a tuple like structure, both helpful and annoying. It should also be noted that this isn't any less efficient as due to Haskell's innate laziness by calling *fst* or *snd* the other argument isn't calculate unless it needs to be.

6.2 New Data Construct

We will start off by looking at the easier case of *Shallow* embeddings when we are adding new data constructs, for this case we will be using the unextended versions of the functions and, in *Deep*s case, data structure. To do this in a shallow embedding all you need to do is simply introduce a new function that represents the new constructor. We will be adding teaching assistants to our class for this introduction and thus we simply need to introduce the new function:

```
ta :: Integer → Integer
ta c = 1 + c
```

This is all that needs to be done for the new data construct in the *Shallow* embedding, you simply need to create a new function that behaves exactly as the semantic function would want it to.

For *Deep* embeddings it becomes a lot harder however as this introduces the issue of going back and changing all of the preceding code that was written when an extension is required, we will thus rewrite all the data structure and functions below.

```
data Class = Room
           | Lecturer Class
           | Student Class
           | Ghost Int Class
           | TA Class


peopleInClass :: Class → Integer
peopleInClass Room         = 0
peopleInClass (Lecturer c) = 1 + peopleInClass c
peopleInClass (Student c)  = 1 + peopleInClass c
peopleInClass (Ghost n c)  = 1 + peopleInClass c
peopleInClass (TA c)       = 1 + peopleInClass c
```

These functions and data structure are very similar to the ones above but with a minor extension which shows how the advantage of the *Shallow* embeddings is in the introduction of new data constructs. This is due to being able to write new data constructs without changing any of the previous code.

7 The Middle Ground

This next section is going to be using a middle ground between *Shallow* and *Deep* embeddings that will attempt to gain the advantages of both of these constructs in one elegant embedding.

7.1 Core vs Everyday

A core language is minimal to the extreme which requires the least amount of data constructors while providing all of the functionality. They often encode some data constructs into the other data constructs provided, like basic identity functions may be a specific case of another data construct although this makes it less readable. Another approach is to create a larger scope data construct which may encode a much larger amount of things than the user may want making it difficult and dangerous to work with. An everyday language on the other hand is a language that is easy to use for the person actually using the language. This language has lots of superfluous data types that make life so much easier despite not being necessary as well as limiting it in a way to make it less likely to cause errors and bugs. This means that in general we want to work with an everyday language but would much prefer to design a core language, the question thus becomes how to get the best of both worlds.

7.2 Intermediate Interpretations

To get the best of both core and everyday as well as the best from *Shallow* and *Deep* ends up being the same method of intermediate interpretations. We will attempt to design one of these structures using the example that is given above. First we will construct a core language that should be succinct and easy to create semantic functions from:

```
data ClassCore = Room
               | Entity String Int ClassCore
```

This new data type limits itself to only one base case and one constructor, the base case is the same as before with the *Room* constructor base case. The other data construct encode every possible entity; it takes in the construct and then generates a *String* for the classifying name of the entity, an Integer for the P.K.E and the rest of the class as *ClassCore*. This will encode every possibility of the above language but in a very dangerous way, if used as it is it would be very easy to mistype the *String* or something else but then very hard to spot the mistake as it would happen after compiling. Next we will implement the wanted semantic functions of the new data type, this function will require much less pattern matching and thus easier to make.

```
peopleInClass :: ClassCore → Integer
peopleInClass (Room)         = 0
peopleInClass (Entity s n c) = 1 + peopleInClass c
```

This function works effectively the same as the one given above but requires a lot less work and seems much more readable. The true joy of this functionality though is that now it is really easy to add the above extension of a new semantic function.

```
decLec :: ClassCore → Float
decLec (Room) = 0
decLec (Entity s n c)
  | s ≡ "Lecturer" = (decLec c * fInt (peopleInClass c) + 1) / (fInt (peopleInClass c) + 1)
  | otherwise      = (decLec c * fInt (peopleInClass c) + 0) / (fInt (peopleInClass c) + 1)
```

This new functionality is much more easily introduced, with similar ease to the original *Deep* embedding which shows how we have encapsulated the ease of a *Deep* embedding within this core language. We would then like to introduce the everyday language into this new format so that all of the newly created functions in the core language work in the everyday language. To do this we have a few options, lets start with the first option where we create a new function that maps the easy to use data type to the new one.

```
alg2Core :: Class → ClassCore
alg2Core (Room)       = Room
alg2Core (Lecturer c) = Entity "Lecturer" 0 (alg2Core c)
alg2Core (Student c)  = Entity "Student" 0 (alg2Core c)
alg2Core (Ghost n c)  = Entity "Ghost" n (alg2Core c)
```

This new function maps the everyday language to the new core language, this means all the semantic functions can be written using the *ClassCore* data type but people can use *alg2Core* to map from the *Class* data type to it. While being useful we come across an issue when we try to introduce a new data construct, we arrive at the same problem as earlier where we have to change the *Class* data type and the *alg2Core* function. When we look at this we might realize that we are just mapping between one deep embedding and another which doesn't seem to be very useful as it just generates a slightly bizarre *Deep* embedding. How then are we going to introduce the advantages of the shallow embedding into this new construct. First we should probably ignore the data type *Class* as well as the function *alg2Core* that we just created as neither seemed to do the job we wanted. Instead we could try to model the everyday language using a *Shallow* embedding by modeling them all using functions.

```
room :: ClassCore
room = Room


student :: ClassCore → ClassCore
student c = Entity "Student" 0 c


lecturer :: ClassCore → ClassCore
lecturer c = Entity "Lecturer" 0 c
```

```
ghost :: Int → ClassCore → ClassCore
ghost n c = Entity "Ghost" n c
```

These new functions look very similar to the above ones for a shallow embedding but our target is now *ClassCore* instead of a specific semantic. Some advantages of this is any errors due to typing things in wrong should be caught at compile time and an extra auxiliary function isn't required after specifying the specific piece of data. The main advantage with this approach though is the joy of the advantage of the *Shallow* embedding where we can now really easily add in data constructs without any hassle what so ever. We can see this by introducing the above extra data construct without adjusting any of the previous code.

```
ta :: ClassCore → ClassCore
ta c = Entity "TA" 0 c
```

Thus finally we have reached the desired goal of combing both the advantages of a *Deep* embedding and a *Shallow* embedding in a half way house embedding. This approach helps to show that *Deep* and *Shallow* aren't a dichotomy but actually two ends of a gradual scale where by moving along it you can introduce lots of advantages of both the embeddings. It isn't all benefits for sitting in the middle as it can introduce more complexity into the making of the package by removing the complexity away from using it. It also has data constructs that classify a larger proportion than you want which is a generally bad thing however the use of a helper set of functions to specifically limit this makes it more manageable. So overall the intermediate interpretation enjoys benefits of both *Deep* and *Shallow* embeddings while letting you design it with a core language but let the users use an everyday language.

8 Conclusion

These notes have covered embedding domain specific languages, it's advantages, as well as different methods of doing so. Thank you for taking the time to read through this set of notes and I hope you found them helpful ♠. Below are some more examples of domain specific languages if you wanted a few more.

9 More Examples

9.1 TrianGraph

Another example and explanation is given in the mediocre report on triangle graphs (a brand new data structure). This report will hopefully be provided to you under the title "report1.pdf", if not please ask a TA (most likely Jamie) for a copy.

9.2 Sam's Quidditch Team Checker

This wonderful and amazing *Deep* embedding lets you check whether a real life Quidditch team follows the required rules. It has also been converted into a *Shallow* embedding, this shows why for some situations like this one where more functions are required a *Deep* embedding is more manageable.

```
data Gender = Male
            | Female
            | NonBinary
  deriving Eq


data Position = Keeper
              | Chaser
              | Seeker
              | Beater
  deriving Eq
  -- deep
data QuidditchTeam = End
                   | Player Position Gender QuidditchTeam


teamSize :: QuidditchTeam → Int
teamSize End            = 0
teamSize (Player p g qt) = 1 + teamSize qt


noGender :: Gender → QuidditchTeam → Int
noGender g End = 0
noGender g (Player p g′ qt)
  | g′ ≡ g     = 1 + noGender g qt
  | otherwise = noGender g qt


noPosition :: Position → QuidditchTeam → Int
noPosition p End = 0
noPosition p (Player p′ g qt)
  | p′ ≡ p     = 1 + noPosition p qt
  | otherwise = noPosition p qt


type SnitchON = Bool


validTeam :: SnitchON → QuidditchTeam → Bool
validTeam True t = (teamSize t ⩽ 7)
  ∧ (noPosition Seeker t ⩽ 1)
  ∧ (noPosition Beater t ⩽ 2)
  ∧ (noPosition Chaser t ⩽ 3)
  ∧ (noPosition Keeper t ⩽ 1)
  ∧ (noGender Male t ⩽ 4)
  ∧ (noGender Female t ⩽ 4)
  ∧ (noGender NonBinary t ⩽ 4)
validTeam False t = (teamSize t ⩽ 6)
  ∧ (noPosition Seeker t ⩽ 0)
  ∧ (noPosition Beater t ⩽ 2)
  ∧ (noPosition Chaser t ⩽ 3)
  ∧ (noPosition Keeper t ⩽ 1)
  ∧ (noGender Male t ⩽ 4)
  ∧ (noGender Female t ⩽ 4)
  ∧ (noGender NonBinary t ⩽ 4)


  -- shallow


type Male      = Int
type Female    = Int
type NonBinary = Int
type Keeper    = Int
type Chaser    = Int
type Seeker    = Int
type Beater    = Int
type TeamSize  = Int
```

```
end :: Bool → (Male, Female, NonBinary, Keeper, Chaser, Seeker, Beater, TeamSize, SnitchON, Bool)
end snitch = (0, 0, 0, 0, 0, 0, 0, 0, snitch, True)
```

please note as this DSL was designed initially as a deep embedding the shallow version is long and unmanageable (hats off to Sam for sticking with this pain and misery for so long). The reason it became so unmanagable is because it was designed to have lots of semantics so to embedd all of these takes much more work in shallow. If however the data structure were to be larger, or to need to be adapted then this shallow embedding would be a much better approach.

```
player :: Position → Gender
  → (Male, Female, NonBinary, Keeper, Chaser, Seeker, Beater, TeamSize, SnitchON, Bool)
  → (Male, Female, NonBinary, Keeper, Chaser, Seeker, Beater, TeamSize, SnitchON, Bool)
player Keeper Male    (m, f, nb, k, c, s, b, t, snitch, g)
  = ((m + 1), f, nb, (k + 1), c, s, b, (t + 1), snitch, (goodTeam (m, f, nb, k, c, s, b, t, snitch, g)))
player Keeper Female (m, f, nb, k, c, s, b, t, snitch, g)
  = (m, (f + 1), nb, (k + 1), c, s, b, (t + 1), snitch, (goodTeam (m, f, nb, k, c, s, b, t, snitch, g)))
player Keeper NonBinary (m, f, nb, k, c, s, b, t, snitch, g)
  = (m, f, (nb + 1), (k + 1), c, s, b, (t + 1), snitch, (goodTeam (m, f, nb, k, c, s, b, t, snitch, g)))
player Seeker Male    (m, f, nb, k, c, s, b, t, snitch, g)
  = ((m + 1), f, nb, k, c, (s + 1), b, (t + 1), snitch, (goodTeam (m, f, nb, k, c, s, b, t, snitch, g)))
player Seeker Female  (m, f, nb, k, c, s, b, t, snitch, g)
  = (m, (f + 1), nb, k, c, (s + 1), b, (t + 1), snitch, (goodTeam (m, f, nb, k, c, s, b, t, snitch, g)))
player Seeker NonBinary (m, f, nb, k, c, s, b, t, snitch, g)
  = (m, f, (nb + 1), k, c, (s + 1), b, (t + 1), snitch, (goodTeam (m, f, nb, k, c, s, b, t, snitch, g)))
player Chaser Male    (m, f, nb, k, c, s, b, t, snitch, g)
  = ((m + 1), f, nb, k, (c + 1), s, b, (t + 1), snitch, (goodTeam (m, f, nb, k, c, s, b, t, snitch, g)))
player Chaser Female (m, f, nb, k, c, s, b, t, snitch, g)
  = (m, (f + 1), nb, k, (c + 1), s, b, (t + 1), snitch, (goodTeam (m, f, nb, k, c, s, b, t, snitch, g)))
player Chaser NonBinary (m, f, nb, k, c, s, b, t, snitch, g)
  = (m, f, (nb + 1), k, (c + 1), s, b, (t + 1), snitch, (goodTeam (m, f, nb, k, c, s, b, t, snitch, g)))
player Beater Male    (m, f, nb, k, c, s, b, t, snitch, g)
  = ((m + 1), f, nb, k, c, s, (b + 1), (t + 1), snitch, (goodTeam (m, f, nb, k, c, s, b, t, snitch, g)))
player Beater Female  (m, f, nb, k, c, s, b, t, snitch, g)
  = (m, (f + 1), nb, k, c, s, (b + 1), (t + 1), snitch, (goodTeam (m, f, nb, k, c, s, b, t, snitch, g)))
player Beater NonBinary (m, f, nb, k, c, s, b, t, snitch, g)
  = (m, f, (nb + 1), k, c, s, (b + 1), (t + 1), snitch, (goodTeam (m, f, nb, k, c, s, b, t, snitch, g)))
```

```
goodTeam :: (Male, Female, NonBinary, Keeper, Chaser, Seeker, Beater, TeamSize, SnitchON, Bool)
               → Bool
goodTeam (m, f, nb, k, c, s, b, t, True, g) = (m ⩽ 4) ∧ (f ⩽ 4) ∧ (nb ⩽ 4) ∧ (k ⩽ 1) ∧ (c ⩽ 3)
                                              ∧ (s ⩽ 1) ∧ (b ⩽ 2)  ∧ (t ⩽ 7) ∧ g
goodTeam (m, f, nb, k, c, s, b, t, False, g) = (m ⩽ 4) ∧ (f ⩽ 4) ∧ (nb ⩽ 4) ∧ (k ⩽ 1) ∧ (c ⩽ 3)
                                               ∧ (s ⩽ 0) ∧ (b ⩽ 2)  ∧ (t ⩽ 6) ∧ g
```