A Domain-Specific Embedded Language Approach for the Scripting of Game Artificial Intelligence

Andrew Calleja
Department of Computer Science
University of Malta
acal010@um.edu.mt

Gordon J. Pace
Department of Computer Science
University of Malta
gordon.pace@um.edu.mt

Abstract

A computer game's artificial intelligence is often scripted using a domain-specific language designed specifically for the game. An approach to language design and prototyping, particularly appropriate to domain-specific ones, is that of embedding a language within a general purpose host language. In this paper, we present an approach to artificial intelligence scripting using embedded languages where the embedded language scripts become data objects in the host language enabling parametrized strategies and their manipulation through host language programs. We identify three classes of scripts: (i) fixed scripts written directly in the embedded language; (ii) adaptive scripts, dynamically generated and modified by the host language programs whilst reacting to the state of the game; and (iii) adaptive scripts which, by means of multi-tiered language frameworks, allow for different levels of abstraction in the specification of game artificial intelligence.

Index Terms

domain-specific embedded languages, games, scripting, artificial intelligence

1. Introduction

Ever since the first computers were created games have often been used as a test-bed for creating case-studies on how computers can emulate human thought and intelligence. To achieve this most games feature an artificial intelligence (AI) component which attempts to mimic human game-play behaviour. The most primitive forms of such AI use a form of fixed script which is run during a game session. These scripts are usually written in a domain-specific language (DSL) which is closely coupled to the game and often use a complex algorithm which attempts to capture all possible game-play strategies to reach a winning solution. This makes the AI quite rigid when compared to actual human intelligence. The aspect which makes human game-play behaviour so versatile is the ability to asses the information presented by the game and adapting to the situation at hand by selecting a possible strategy on-the-fly. Based on the strategy selected, the player then comes up with the sequence of actions to take which are supplied to the game using its input system. AI techniques such as trained neural networks or greedy algorithms exist which attempt to emulate this adaptive behaviour by generating fresh, adaptive scripts during a game session. Such processes however might demand too much computation-power or time in order to generate the required script and due to this often resort to an approximate best solution rather than an optimal solution.

In recent years a versatile technique has been developed where a DSL can be embedded into a host language. Such a domain-specific embedded language (DSEL) is not developed from scratch as is the case with normal DSLs, but rather a hosting language, which is usually a general purpose language, is adapted to supply the required syntax and semantics and also act as the embedded language's compiler. These DSELs allow us to quickly and efficiently create scripting languages which may be used to write scripts which encode simple and elaborate fixed AI strategies. However, the use of embedding brings about various other advantages which, by making possible the creation of adaptive scripts during a game-play session, allow our AI to become dynamic and enable it to act in a similar manner to how a human player behaves. This is possible since by virtue of embedding the AI scripts are first class objects of the host language thus allowing the latter to manipulate them. By querying the game state and selecting the appropriate strategy, the host language can act as a meta-language which generates the appropriate embedded language script. Various embedded language techniques have been developed lately for use with DSELs which we believe allow us to achieve this close emulation of a human player behaviour during game-play by a computer player.

Multi-tiered embedded language frameworks where languages which vary in level of abstraction are embedded within one another supply us with another advantage based on another human game-play behaviour, that of translating abstract strategies into concrete operations. It is possible by means of a simple translation process to expand high-level strategies written in an abstract language automatically into lower-level tactical plans of a less abstract nature and so on until the lowest level, that of an operational script is reached. We believe that embedded language frameworks would allow us to achieve this effect.

In this paper we briefly introduce language embedding and its use by the functional language Haskell [1] and proceed by showing how it may be used to create fixed and dynamic AI scripts for a Tetris-like game implemented in Haskell. Finally, we outline our current work on embedded language frameworks and exemplify their use by means of a turn-based strategy game.

2. Domain-Specific Embedded Languages

The notion of language embedding can be traced back to Landin's 1966 seminal paper "The Next 700 Programming Languages" [2]. Here Landin remarked that a language consists of a basic set of constructs and a number of ways in which to combine them but stressed the fact that the suitability of a language towards a domain is closely tied to the former rather than the latter. Thus, by identifying a number of basic constructs related to a domain, an appropriate DSL for the latter is formed. Landin then proceeded with building upon such a notion of a DSL by proposing DSELs as a natural step forward. To achieve this he suggested the creation of a general-purpose language which can be geared towards a particular domain by selecting a number of basic constructs. The first actual uses of language embedding were done via the functional language Lisp's [3] macro system. More recently, Hudak [4], [5] reintroduced the approach and made it popular as a viable methodology to develop programming languages for subsequent use in software development. DSLs offer the right amount of abstraction for software projects but creating a new language from scratch each time requires a considerable amount of work in terms of language design and tool development. Hudak thus suggested the use of an existing infrastructure of an existing language whose properties meet the requirements of the DSL. Using such a host language's mechanisms and tools, adapted to a particular domain it is possible to create the required DSEL with the added advantage of reducing costs, time and effort. The work required is only in adding the domain-specific functionality to the host language.

When selecting a host language for embedding, choosing the right language is important as both the positive and negative features of the host language are inherited by the embedded language. Functional languages are often selected as host languages due to their features which include pattern matching, lazy evaluation, module system, higher-order functions, strong typing, polymorphism and overloading which are useful for language embedding [6]. These combined features allow for a level of abstraction which enables the user to focus on the domain itself rather than having to consider a lot of implementation details which clutter domain-specific thoughts. Nowadays, the language of choice for embedding is Haskell since it has the features just mentioned while adding some of its own such as monads and a non-restrictive syntax. Languages related to various domains successfully embedded in this language, including: geometric constructions [7], [8], images [9], animations [10], hardware description [11], [12] and business processes [13].

Once a host language, in our case Haskell, is selected the domain-related abstractions are encoded by Haskell data types. These data types provide us with the syntax of our embedded language. Haskell's syntax allows us create an embedded language which is free from annotations which are not domain-related, something which often plagues other hosting languages, resulting in a look and feel of a separate language rather than an embedded one. The semantics of our language are provided by a number of functions in Haskell itself which act as the language's compilers or interpreters. By traversing the data structures created by the domain-related data types they attribute a meaning to such structures and return the required result of interpretation.

3. DSELs for Game Scripting

3.1. Fixed AI Scripting

The use of language embedding allows us to quickly create a domain-specific language for any game we

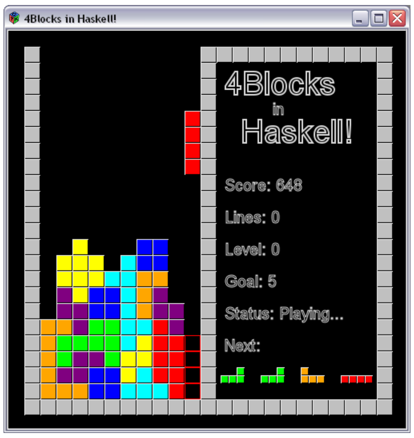

Figure 1. 4Blocks game implemented in Haskell.

might wish to script in order to provide it with a form of AI. Implementing a game scripting language is usually the case of determining the possible actions a player or any other entity can make during a game session and creating the appropriate commands accordingly as a data type of the host language. Using this embedded language we can then manually create elaborate scripts which the game can then process during a game session as a form of fixed or static AI.

Puzzle Game Case Study. As a case-study for static AI scripting we implemented a Tetris-like game in Haskell which we called 4Blocks (Figure 1). During a session of this game a continuous number of bricks appear on the upper side of the screen and fall downwards at a constant rate. The player performs a series of moves upon the brick such that it is positioned on the bottom of the game area with the objective of creating one or more complete lines. When a line is completed it disappears and any uncompleted lines above it shift downwards to replace it. Completing lines awards the player score points. Bonus points are obtained when the user completes more than one line at one go with a possible maximum of four lines at once. Other bonus points are given when the user forces the brick downwards themselves. As more lines are completed by the player they might reach the level's line-goal. When they do, the level is incremented causing the brick to fall faster. This allows the game to become more challenging as the player must move the brick in place faster in order to keep playing and earn more points.

There are a fixed number of axiomatic commands the player can perform upon the falling brick. These are:

- Shifting to the left/right,
- Rotating to the left/right,
- Performing a hard/soft drop,
- Performing no action at all.

These can be represented using a Haskell data type `Instruction` as follows:

```
data Instruction
  = ShiftLeft   | ShiftRight
  | RotateLeft  | RotateRight
  | HardDrop    | SoftDrop
  | NoAction
```

A number of such instructions combined in the correct way allow us to move the brick in the desired position. We have thus created another data type which we call `Program` defined as follows:

```
data Program a
  -- do nothing statement
  = Skip
  -- sequential composition
  | Program a :> Program a
  -- perform an instruction
  | Do a
  -- if-then-else statement
  | IfThenElse Pred (Program a) (Program a)
  -- repeat a program forever
  | Repeat (Program a)
  -- while statement
  | While Pred (Program a)
  -- perform program for this piece
  | ForThisPiece (Program a)
```

Using this data type as our language's syntax, objects of type `Program Instruction` form the static AI scripts for our 4Blocks game. The auxiliary type `Pred` consists of any form of predicate which can be used to make a decision. In our case it takes the form of a function which queries the game's state `Game` to return a Boolean value (`Game -> Bool`). A simple static script which can be written using this DSEL is as follows:

```
Do RotateRight :>
Do ShiftLeft :>
Do ShiftLeft :>
Do HardDrop
```

This script simply rotates the falling brick to the right, moves it two times to the left and then drops it to the bottom of the screen. A more elaborate script, such as:

```
ForThisPiece (
  While (brickHeightGreaterThan 5)
        (Do SoftDrop) :>
  Do ShiftLeft )
```

may be used to side-fit a brick into position. Using the `ForThisPiece` construct we are restricting the encapsulated code to the current brick only. The encapsulated code performs a soft drop until the brick's height is less then or equal to five block lengths at which point it then moves the brick to the side, thus completing the side-fit. More elaborate scripts can be created and composed together to script the game's desired AI.

The script handling component of the game, embodied in the `stepAI` function, takes the game state and the installed script (the `Maybe` type is used, since no script may be installed), and performs a single instruction along the installed script if one exists. If no script is currently installed, a script generator `thinkAI` (embodying the actual AI), is executed, to generate a new script.

```
type AI = Maybe (Program Instruction)

stepAI :: (Game, AI) -> (Instruction, AI)
stepAI (game, Nothing) = thinkAI game
stepAI (game, Just Skip) = skipAI game
stepAI (game, Just (prog1 :> prog2)) =
  sequenceAI (game, prog1, prog2)
...
```

This approach emulates human game-play since once the game-plan is known it is carried out mechanically a step at a time. As a simple example of interpretation see Figure 2. Here we see the effects of our first example script on an initial game state which consists of an empty game-area and the first brick.

3.2. Adaptive AI Scripting

When a human player thinks about where to position the current brick they do no usually think in terms of one unified script as our AI has been doing so far but rather they select one of a number of different strategies which they have come up with during the various times they have played the game. Once a strategy is selected they create their plan of action which is then carried out mechanically.

These concepts can be mapped to our AI allowing it to generate adaptive scripts based on the current game state. Our implementation for fixed AI scripts already handles the mechanical carrying-out of the AI script when there is an outstanding script. The only change we need to perform is upon `stepAI`'s case which implements the thinking phase (`thinkAI`). This case should no longer simply fetch a predefined script but rather it should select a strategy from a number of different strategies and then based on the selected strategy generate automatically the required script. We have thus defined `thinkAI` as follows:

```
thinkAI :: Game -> (Instruction, AI)
thinkAI game = (NoAction, Just prog)
  where prog = think game

think :: Game -> Program Instruction
think game
  | detectCompleteFourLines game
    = completeFourLinesStrategy game
  | detectMinimizeHeight game
    = minimizeHeightStrategy game
  ...
  | otherwise
    = lowestBestFitStrategy game
```

The `thinkAI` function makes use of an auxiliary function `think` which selects the relevant strategy and then generates the script accordingly. In order to do so we make use of a number of predicates, one for each strategy listed by priority, which query the game state to check whether the latter would benefit from selecting that particular strategy. In our case we have set the AI to pick a strategy from a number of arbitrary strategies which we believe can lead it into surviving for a good amount of time. It is however possible to come up with worse or better AI based on the selected strategies and their relative ordering. For example, for games such as 4Blocks, various other possible strategies exist which one might wish to include such as a strategy which positions the current brick while catering for the next one. In regards to ordering we have decided to give priority to our strategy which completes four lines since it awards the most points and reduces the well's height by four (the maximum possible). The next strategy attempts to minimize height by completing one to three lines if the well's height is above a fixed value such as twelve blocks. This strategy does not cater for creating holes in the well's construction when selecting the location where to place the brick since it attempts to clear lines in order to survive for a longer game session. And so on, until we reach the default strategy which simply attempts to find the lowest best fit for the current brick.

Each of the different predicates triggers the matching strategy's function. These functions act as program generators which generate the required script automatically based on a number of criteria which depend on the strategy itself. The general procedure followed by these strategy functions is that of first querying the game state for the possible number of brick fit locations. This is carried out by means of a number of predefined functions which are supplied to the AI's programmer in order to interface with the game state. The best location is then selected based on the criteria of the strategy which is usually a question of sorting the possible locations via their attributes such as their x-position or y-position, or by means of a fitness function carried out upon all the locations. In the latter case each location returns its fitness value which is then used to select the best position. Finally, the selected target location is used to generate the required script to reach it. To achieve this we make use of two types of specialized functions called parametrized objects and connection patterns. The first of these generate a script given a number of parameters as input while the latter

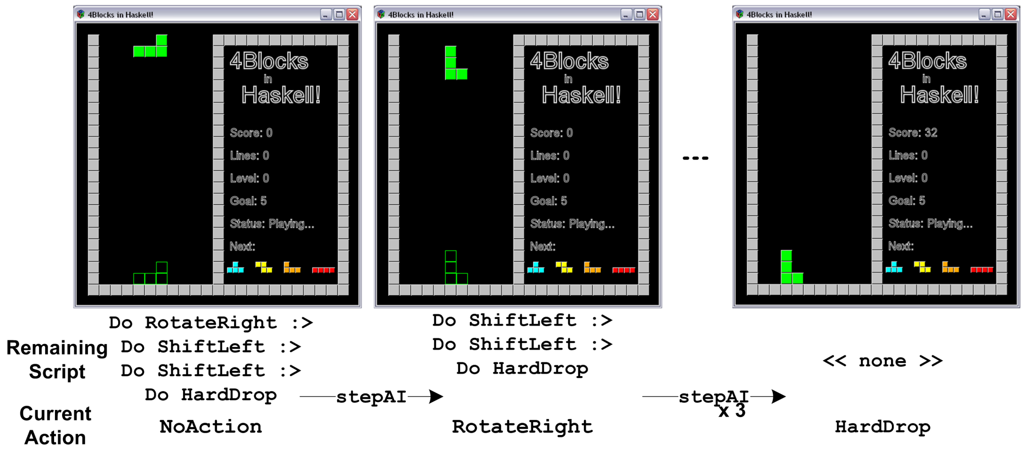

Figure 2. An example script and its interpretation by means of a number of `stepAI` function calls.

combine scripts together using a regular patterns. As an example of this approach we provide the code for the "complete four lines" strategy mentioned above. The predicate which triggers the selection of this strategy is defined as follows:

```
detectCompleteFourLines :: Game -> Bool
detectCompleteFourLines game
  = (length fourLines > 0)
  where
    fourLines = get4LinesPoss game
```

The function queries the game for the number of fits which complete four lines. It at least one is found this strategy is triggered. The strategy itself is defined by the following function:

```
completeFourLinesStrategy :: Game ->
                       Program Instruction
completeFourLinesStrategy game
  = generateNFScript game locX orient
  where
    fourLines = get4LinesPoss game
    ((locX,_),orient) = head $ sortBy
       (compare `on` (snd.fst)) fourLines
```

This time when the game state is queried for the number of fits which complete four lines these are sorted on the fit's y-position, thus ensuring that the lowest fit is selected. The fit's x-position and the brick's orientation at that position are saved and supplied to a parametrized object called `generateNFScript` which uses these values to generate the required script to perform a normal fit. The latter consists of rotating the brick to face the required orientation, moving it to the required x-position and then performing a hard drop to lock it into place.

Using this approach we were able to create the strategies shown earlier and depending on which one was triggered during a game session the adaptive AI generated the appropriate script. This was only rendered possible by means of the embedding process which allowed our game scripts to be first-class objects manipulated by Haskell which acted as a meta-language. We have tested the approach over a number of runs and we believe that that results obtained are comparable to that of an intermediate-level human player. We believe that introducing more strategies and improving our current strategies could allow us to reduce the negative results and improve our AI.

4. Extending the Approach: Multi-Tiered Language Embedding

Having more than one language at our disposal was advantageous as it has provided us with the right abstraction mechanisms to enable a two-tiered approach towards game scripting. For our future work we are considering whether allowing for further levels of abstraction, by means of various embedded languages embedded within one another, would provide us with further benefits. Using this approach, suggested by Claessen and Pace [14], the higher-level languages' syntax are implemented as a normal data type in Haskell but the structures of that latter type are not interpreted or compiled directly to Haskell which is an indirect host, but rather to the direct host language. Translation to Haskell is possible by means of a two-stepped translation: first to embedded hosting language and then the latter to the actual host language. This approach allows the creation of a multi-tier language framework where each language is at a level of abstraction higher than its host. Allowing for different levels of abstraction allows for faster programming in a specialized abstract domain which can be concretised by means of a simple function which takes the abstract program to a more concrete, realisable level.

Language frameworks may enable us to make use of higher levels of abstraction in our scripting languages. We believe that such frameworks would be able to model another aspect of how a person plays a game into our AI, that is the process of translating abstract thought into concrete action. A human player first thinks in a strategic high level and then breaks down his thoughts into tactical, more manageable ones. Finally each tactic is developed into a number of implementable operations. We might map this behaviour by developing a framework with three scripting languages: operational, tactical and strategical, where one is more abstract and high-level than the previous one. We can use the game state to create a script at a global level using the strategy-level DSEL. This script may then be used, by means of translation into a lower-level script, to come up with the right tactics which agree with the selected strategy. Finally, each tactical script is translated on a fine-grain scale into the actual operational script which is carried out within the game.

Puzzle Game Case Study As an initial attempt at this approach we have implemented two higher-order constructs called `RotateToOrientation` and `ShiftLaterally` within out DSEL for 4Blocks. Despite being included as part of the `Program` data type as follows:

```
data Program a
  = ...
  | RotateToOrientation Int
  | ShiftLaterally Int
```

we intend them to map to a number of lower-level `Program` rotations and shifts respectively as these are common actions most strategies make use of. For example, a normal fit is usually described by means of three sub-programs composed together: a number of rotations followed by a number of shifts laterally followed by a finalizing move such as a hard drop. A lower-level example script of this is:

```
(Do RotateRight :> Do RotateRight) :>
(Do ShiftLeft :> Do ShiftLeft) :>
Do HardDrop
```

This can be abstracted to the higher-level script:

```
(RotateOrientation 2) :>
(ShiftLaterally (−2)) :>
Do HardDrop
```

which can then perhaps be abstracted to the highest-level script `NormalFit` 2 (−2). Other possible high level constructs for 4Blocks are for example `SoftFit` and `SideFit`. These two are common idioms for 4Blocks that differ from `NormalFit` by moving softly downwards instead of hard-dropping and moving sideways before locking into place, respectively.

Introducing this language framework within our AI requires us to add a few intermediate steps. Our previous strategies must now generate high-level scripts in their thinking phase. Also, when we are carrying-out the scripts we must first translate and expand the current high-level construct into the equivalent low-level constructs which are understood by the game engine. Conceptually:

```
stepStratAI :: (Game, StratAI)
                 −> (Instruction, AI)
stepStratAI (game, Nothing) = thinkAI game
stepStratAI (game, Just (NormalFit o d))
  = (toOper.toTact) (NormalFit o d)
...
```

where `toTact` and `toOper` are the translation functions which allow us to translate a high level script of type `StratAI` into a low level one understood by the game engine.

Turn-based Strategy Game Case Study Despite being a good initial case study, puzzle games do not usually benefit from higher levels of abstraction as they are often tied to a low-level. As a better case study we are currently developing a turn-based strategy game which would allow us to make better use of language frameworks.

The game will set itself as a battle between two or more generals who wish to control the universe Risk-style. Each general initially controls a handful of planets in the universe. As the turns proceed the planets generate population and wealth. The general must conquer more and more planets thus increasing their wealth per turn until the final objective of controlling all the universe is attained. A particular turn consists of the deployment of resources to a planet to attack it or to reinforce it against attack. Each general also has a number of captains under their command. A captain is in charge of capturing an entire planet which in turn consists of a number of land masses or countries. Once a captain controls all the land masses on a planet they have conquered the planet for the general.

Our idea is to create an AI for this game which makes use of a two-tiered language framework consisting of a strategical/tactical language and an operational language. At the general's level the AI is strategical in nature and is concerned with attacking planets, deploying forces and so on. At a lower level, the captains are concerned with planning which land-mass to attack/defend in order to capture the planet and deploying the resources received from the general. We thus plan to have a hierarchical AI which generates the strategic script and then translates it into lower level scripts for the captains by means of a translation process. This will enable the AI to become tiered as suggested earlier.

5. Conclusion

In this paper we have seen how DSELs are an invaluable tool for the creation of game AI which models human game-play. They allow us to quickly create game scripting languages which can be used to write fixed scripts in a similar manner to how game AI is often written. However the main advantage proposed by our approach is that by making use of two languages at once, the domain-specific scripting language itself and the general purpose host language Haskell, we are able to generate adaptive AI scripts. This we have achieved by using Haskell to write a number of predefined game strategies which are able to automatically query the game state and generate the corresponding script automatically. We have also outlined how we tested this approach successfully using a puzzle game as a case study. Finally, we have proposed our future work on extending the approach to allow for a hierarchical form of AI which generates high-level scripts and expands them by an automatic translation process into low-level scripts. Using the same case study as before we have shown how this approach can work to some degree of success for a puzzle game. Our immediate next work is to attempt both the adaptive scripting approach and the language framework approach with another case study which consists of a turn-based strategy game outlined here.

References

[1] S. P. J. et al., Ed., *Haskell 98 Language and Libraries, the Revised Report.* Cambridge University Press, Apr. 2003.

[2] P. J. Landin, "The next 700 programming languages," *Communications of the ACM*, vol. 9, no. 3, pp. 157–166, March 1966.

[3] G. L. Steele, Jr., *Common LISP: the language (2nd ed.).* Newton, MA, USA: Digital Press, 1990.

[4] P. Hudak, "Building domain-specific embedded languages," *ACM Computing Surveys*, vol. 28, Jun. 1996.

[5] ——, "Modular domain specific languages and tools," in *in Proceedings of Fifth International Conference on Software Reuse*. IEEE Computer Society Press, 1998, pp. 134–142.

[6] S. N. Kamin, "Research on domain-specific embedded languages and program generators," in *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000.

[7] G. Caruana and G. Pace, "Embedded languages for origami-based geometry," in *Proceedings of Computer Science Annual Workshop*. Departments of Computer Science and AI, University of Malta, 2007.

[8] M. Grima and G. Pace, "An embedded geometrical language in Haskell: Construction, visualisation, proof," in *Proceedings of Computer Science Annual Workshop*. Departments of Computer Science and AI, University of Malta, 2007.

[9] C. Elliott, "Functional images," in *The Fun of Programming*, ser. "Cornerstones of Computing" series, J. Gibbons and O. de Moor, Eds. Palgrave, Mar. 2003.

[10] ——, "An embedded modeling language approach to interactive 3d and multimedia animation," *IEEE Transactions On Software Engineering*, vol. 25, no. 3, pp. 291–308, 1999.

[11] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: Hardware design in Haskell," in *In International Conference on Functional Programming*. ACM Press, 1998, pp. 174–184.

[12] G. Pace, "HeDLa: A strongly typed, component-based embedded hardware description language," in *Proceedings of Computer Science Annual Workshop*. Departments of Computer Science and AI, University of Malta, 2007.

[13] L. Micallef and G. Pace, "An embedded domain specific language to model, transform and quality assure business processes in business-driven development," in *Proceedings of the University of Malta Workshop in ICT (WICT'08)*, 2008.

[14] K. Claessen and G. Pace, "An embedded language framework for hardware compilation," in *In Designing Correct Circuits 02*, 2002.