DEPARTMENT OF COMPUTER SCIENCE

University of Malta

EMBEDDED SCRIPTING LANGUAGES FOR GAME ARTIFICIAL INTELLIGENCE

Andrew Calleja

August 2010

Supervisor: Prof. Gordon Pace

Submitted in partial fulfillment of the requirements for the degree of Master of Science

Declaration

Abstract

Scripting allows game-programmers to add content such as story-lines and interactions between the player and computer-controlled agents within the game world. Another reason why scripting is used within games is to allow the latter's developers to encode their artificial intelligence. Currently, various methods exist which allow us to introduce scripting within a game. One of these methods introduces scripting directly within the game engine and makes use of the game's implementation language itself to script the game. Other methods make use of an appropriate scripting language which is introduced specifically for scripting the game. Such a language may vary from being either domain-specific to a particular game only or generic for all scripting purposes. These methods excel in certain factors while performing poorly in others such as the time it takes to implement the language itself and whether the language is expressive enough to encode artificial intelligence.

In this thesis we propose and examine an alternative approach to such languages which makes use of a technique called language embedding. Language embedding is the process by means of which a language, usually a domain-specific language, is embedded within a general-purpose language. This approach allows us to quickly create and make use of a domain-specific embedded language for normal scripting. Apart from this it also permits us to use the general-purpose host language as a meta-language over the embedded language scripts. In this manner, these scripts become data objects in the host language, enabling parametrised strategies defined using the latter language to generate them and manipulate them. We have successfully made use of this method with two case-studies and also compared and contrasted our approach with the other scripting methods available based on a number of factors such as those mentioned above. This has allowed us to evaluate our approach qualitatively. Based on this, we conclude that our approach compares adequately with the other available approaches and supplies a useful alternative to them.

Acknowledgements

First and foremost, I would like to thank my supervisor Dr. Gordon Pace for giving me the chance to read a masters in gaming. I was lucky to find a tutor who likes games and appreciates their potential as much as I do. I would also like to thank him for the numerous meetings we spent discussing ideas and problems met during the course of this masters. His guidance, insight and support were invaluable assets in the shaping of this thesis.

Next I would like to thank my friends and office mates David Bailey, Alan Cassar, Kevin Falzon, Ingram Bondin and Andrew Gauci for our numerous lively discussions involving both research and leisure topics. Thanks for being both avid gamers and constructive critics. I would also like to thank the other researchers at University of Malta's Computer Science department Christian Colombo and Ruth Schembri for the discussions we shared. Thanks also goes to Dr. Adrian Francalanza for giving me the opportunity of delivering a number of talks at his club. As I learnt during this masters, part of being a good researcher is sharing your ideas with peers.

I would also like to thank my friends outside of University for putting up with me during the course of this thesis. A mention goes to the members of the group from my home island. You all know who you are. Thanks also to my friends in Malta, especially Reuben, Mark and Roberta, who have also helped me out numerous times.

A special mention goes to the folk at Haskell's IRC channel for being such a welcoming and supporting community.

Last but not least I would like to thank those most close to me for their unwavering support. Special thanks goes to Annaliz for sharing yet another thesis with me. I can never thank her enough. I would also like to thank her family for their help and support. Finally, I would like to say thanks to my family for all their patience, support and love. I surely could not have done it without their help.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Games have been an important area of study for computer science ever since the first computers were created. They have often been used as a test-bed for creating case studies on how computers can emulate human thought and intelligence. In fact, many games which feature a computer opponent frequently involve the use of some form of artificial intelligence (AI) in order to mimic human game-play behaviour. To achieve this, games often make use of a scripting language. This language is used to write scripts which model the intelligent behaviour that a game might require such as for example the actions of any non-player character or other in-game entity.

There are various traditional methods by means of which a scripting language may be introduced within a game system and they differ in their attributes. This means that the game developers often have to select the method which best suits their needs. The attributes are based on a number of basic factors which a particular method might lack or excel in. Since a programming language is being used to script the AI, the first set of factors are related to the former itself. Is the language easy to implement? Is the language easy to tailor or even update once it has been implemented or do these processes require considerable work as well? How easy is the scripting language to integrate in order for it to be actually used? The next set of factors relate to the method's target audience. Is the language employed within the method going to be used by the game programmers themselves for the specific purpose of creating the required AI or does it also require to be simple in order for non-programmers such as story-writers to be able to write the required plot? Thus the scripting language which a method makes use of might range from being universally general-purpose to wholly domain-specific. Finally, there are a number of features which are ideal to have within the method since they allow for a better experience whilst making use of the language itself. Is there a separation between the game logic and the scripts themselves such that changing the script does not imply a recompilation of the entire game? Does the method require scripts to be interpreted or compiled either at game compile-time or even in real-time? If we wish to allow third party programmers, such as fans and any company which might expand upon the game, to write extra features or extensions to the game once it has been completed, does the method we employ allow this? All these factors play a pivotal role in the selection of the required scripting method.

In recent years a versatile technique has been developed where a language pertaining to a particular domain, called a domain-specific language, can be embedded within a host language. Such a domain-specific embedded language is not developed from scratch as is the case with normal domain-specific languages, but rather a host language, which is usually a general-purpose language, is adapted to supply the syntax and semantics and thus act as the embedded language's compiler or interpreter. This body of work is motivated by this idea since we believe it allows us to suggest a new, alternative approach to game-scripting which scores adequately with respect to the basic factors mentioned earlier. In regards to ease of implementation, the embedded language approach enables us to quickly and efficiently create scripting languages without actually requiring us to develop traditional tools such as compilers and interpreters from the ground up. Moreover, altering the language in any way is also a very straightforward process. Integrating the language is also simple since we are essentially still programming in the host language. In fact, implementing the scripting language using the embedded approach means it is seamlessly integrated within the language used to program the game itself. Since we are developing a domain-specific language within a general-purpose language we can target both programmers and non-programmers as our audience. The embedded scripting language can be tailored to be completely domain-specific and easily understandable by non-programmers while programmers can use both simultaneously in order to write the required AI. Since the scripts written with the embedded scripting language are first-class objects of the host language, programmers can use the latter to manipulate them like any other data type. This gives rise to powerful script generators where the host language acts as a meta-language which queries the game state, takes certain clever decisions and then generates the appropriate embedded language scripts. Finally, the embedded approach maintains the separation of scripts from game logic, allows for scripts to be interpreted in real-time and can also permit third-party programmers to plug in their code if the right interface is provided. Hence we believe that the embedded approach provides a suitable alternative which is in par to other scripting methods.

1.2 Approach

The approach taken in this thesis is to first investigate current scripting methods in order to examine their attributes in light of the above factors. In this way we can extract the ideal attributes of a scripting method. Using these ideal attributes we shall guage qualitatively an alternative scripting method which we are proposing. This method applies the technique of language embedding just described to the domain of game scripting. We shall test our proposed method by first using it in order to create scripting languages for certain games and then by using the created languages themselves in order to encode the AI of the games. Once this is carried out we will evaluate the method itself by discussing its merits and drawbacks and by comparing and contrasting it with the other scripting methods. We shall do this by basing our discussion upon the derived ideal attributes of a scripting method.

1.3 Objectives

Our objectives revolve around the central aim of making use of language embedding within the domain of game scripting. These are:

- **Investigating traditional scripting methods** – Our first objective is to examine a number of common scripting methods with the main aim of extracting their strong points and pinpointing their weaknesses in regards to the factors mentioned above.

- **Examining language embedding and its related techniques** – Next we must examine in detail language embedding and the various techniques which have been created in order to improve this technique since we intend to port this approach to game scripting.

- **Using language embedding in order to create scripting languages** – Having obtained the necessary background information required, our next step is to actually implement a number of scripting languages. In order to do this we will first implement two games as case studies. The first game consists of a simple puzzle game while the second game is a more involved turn-based strategy game. We will make use of language embedding in order to implement scripting languages for these two games. Our aim here is to show how easy and straightforward it is to implement a scripting language using the embedding approach compared to the more traditional language-creation methods once the language is specified and designed.

- **Using the scripting languages to create AI scripts** – We then show the power of the embedded languages thus created by making use of them in order to create two forms of AI:
  - **Fixed AI** – Here we can make use of the embedded scripting language in order to write static scripts which form a fixed type of AI. This AI utilises a unified strategy embodied within a script. The script is run upon the game's state whereby an instruction or number of instructions are extracted in each interpretation step. These instructions are used to update the game state. Once the script is finished we simply retrieve a fresh copy of it and the process re-iterates until an end-game condition is met.
  - **Adaptive AI** – The embedding approach however allows us to improve upon this simplistic AI by enabling us to make use of the host language as well as the embedded scripting language as highlighted earlier. Using the host language we can manipulate the scripting language in such a way as to render our AI more dynamic by allowing it to become adaptive to the game's current state. We plan to do this in two ways. The first of these is to make use of a number of strategies which are detected based upon the game state. Each of these strategies encoded in the host language allows us to make use of embedded language techniques which may be used to generate domain-specific embedded language scripts on-the-fly, based upon the game state itself. The second method used to make our AI more adaptive is to allow it to react to certain events which might require us to change our current script and select a new strategy.
- **Exploring further advancements upon our technique** – We would also like to explore whether we can extend our scripting languages by introducing advantageous features such as concurrency within them. We intend to achieve this by adding message-passing as part of one of our scripting languages such that a number of scripts at different levels of abstraction and which thus can perform different actions can communicate with one another via messages. This allows the agents which they script to react to each other and work together in order to reach a common goal.
- **Evaluation of our approach** – Our final objective is to compare our approach to other traditional ones and evaluate qualitatively its merits and drawbacks.

1.4 Summary of Results

We have investigated the use of language embedding and successfully developed embedded scripting languages to script the aforementioned two games as case studies. The technique of language embedding provides a strong framework to quickly and easily create powerful scripting languages for use in game systems. However, in cases where traditional approaches are required, it may be used for quick language prototyping. We have also explored additional features available to our approach due to embedding itself such as adaptive on-the-fly program generation. Moreover, we have also successfully added concurrency via message-passing as an extra functionality to the scripting language of our second case study. Updating the language with extra features such as in the latter case is quite straightforward. Also, the language created via language embedding can be tailored for non-programmers by means of the domain-specific embedded language implemented and also for AI-programmers by means of the meta host language. The approach also allows scripts to be written apart from the game itself, enables run-time interpretation of scripts and finally also permits third-party programmers to plug in their own scripts if this is required.

In conclusion, our successful use of the embedded language approach within our two case studies and in light of the results achieved highlighted above, we believe that embedded scripting languages are a viable alternative approach to traditional scripting methods.

1.5 Document Structure

The rest of this document is divided into six additional chapters:

Chapter 2 introduces game scripting itself, its aims and advantages, and the different methods of introducing scripting within a game system along with their attributes.

Chapter 3 describes language embedding and domain-specific embedded languages in detail. We first discuss their aims, advantages and disadvantages, compare them to domain-specific libraries and discuss what the features of an adequate host language are. We then describe a number of example DSELs implemented for various domains apart from game scripting. Finally, we elaborate on a number of techniques which extend the embedding approach.

Chapter 4 presents our first case study, the puzzle game 4Blocks. Here we describe how a scripting language may be implemented for this game using the embedding approach and show how such a language may be used to write fixed scripts and adaptive scripts.

Chapter 5 presents our second case study, the turn-based strategy game Space Generals. Again we show how scripting may be introduced and use it to write fixed and adaptive scripts for this game. Here we also describe how advanced features such as concurrency may be added to the scripting language.

Chapter 6 discusses our results, evaluates them and compares qualitatively the embedded language approach to the traditional forms of language embedding. We also discuss the limitations of our approach and explore some work related to ours.

Chapter 7 presents a number of areas within our research which could use some future work, summarises our achievements and concludes with our final remarks.

Chapter 2

Game Scripting

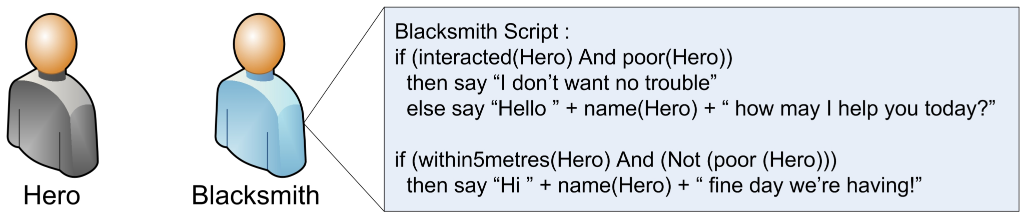
2.1 Introduction

The engineering of a game, even a simple one, is a project which requires work in many creative areas. At the core of the game lies a game engine which is often written by programmers in a general-purpose language such as C or C++. It provides a framework which gives life to the game by managing its game-play logic and various interactions from the outside world. Another area is that of graphics. Game graphics provide the player with visual simulations of the world the game is set in and can range from simple ASCII graphics to full-blown 3D immersive worlds. More often than not a game also requires background music and sound effects to create an even more immersive experience. Again these can range from simple computer-speaker beeps to full quality soundtracks involving an entire orchestra and a recording studio. Last but not least, you need content. What story does the game tell? What characters are in it? What do these characters do? How do they interact with the player or how does the player interact with them? This is the area of game scripting and is the focus of this chapter. We shall first discuss the relevance of scripts in gaming along with their main aims and advantages. We will then give an overview of a scripting engine and how it relates to the game engine. Next, we will discuss the merits and drawbacks of three common methods which are used to introduce such a scripting engine. Finally, we derive the ideal attributes of a scripting method and propose a fourth option. These attributes will be later on used to evaluate the fourth option and to compare it to the other three scripting methods.

2.2 Background

As any other perhaps more established medium, such as a book or a film does, a game tries to portray a story and create an entire experience. What a game does, however, which no other medium has been able to achieve so successfully so far is to involve the person interacting with it to a degree where the latter feels like they are actually within the fictitious world and can affect its behaviour. A game achieves this by various methods of which the most directly visible ones are the traditional ones found in other mediums such as sounds, music and graphics. Another method is that of allowing the possibility of direct input by means of commands through a peripheral such as a keyboard, mouse or other appropriate controller. Yet another method used which is often overlooked is that of scripting.

In more traditional settings a script is a set of static instructions performed by a person or other entity which is used to convey a message or something of interest. Scripts are often used in films or plays to dictate what the actors do and are used primarily with the aim of telling a story. In games, scripts are often used in a similar manner to program what entities in the fictitious world say or do and also how they react or behave



```
Blacksmith Script :
if (interacted(Hero) And poor(Hero))
  then say "I don't want no trouble"
  else say "Hello " + name(Hero) + " how may I help you today?"

if (within5metres(Hero) And (Not (poor (Hero)))
  then say "Hi " + name(Hero) + " fine day we're having!"
```

Figure 2.1: A hero interacting with a blacksmith in our example RPG. The blacksmith carries the script shown with him. Adapted from [68].

when a player interacts with them. They are thus essentially programs relating to a particular character or object which are executed statically (when instructed to) such as according to a storyline or dynamically (whilst reacting to a possible event) such as the player moving closer to them.

2.3 Aims and Advantages

The main aim behind scripts is to create stimulating content which allows the game to feel alive and believable [68]. For example, let us say we are playing a typical role-playing game (RPG) where you are a poor hero who wants to buy your first sword which will allow you to start questing. Since your clothes are in shambles, when you approach a non-player character (NPC) controlled by the game such as a blacksmith who could provide you with a sword, he immediately assumes you are going to try to steal from him which causes him to greet you with "I don't want no trouble here!". Later on, when you have established yourself, gained some gear and made further interactions with him he reveals his name and starts even greeting you by name when you are within five metres of his area. This is a minor example but it goes to show how much more realistic the blacksmith becomes this way.

In order to achieve all this, game scripting must be powerful enough to somehow allow us to encode some form of artificial intelligence within a character or other entity within a game. To do this we need a language which will allow us to describe the behaviour and possible reactions to interaction of each actor within the game. Such a scripting language, as it is often referred to, would allow us to write the required AI as a script which every actor within the game would carry around with them. The script would then interact appropriately with the game engine somehow allowing it to be run when required or triggered as a response to some event over the game's state [68]. Let us consider our former example again. Conceptually, the blacksmith carries a script with him as shown in Figure 2.1. The script is shown as being written in a high-level language similar to Java or C++ and behaves as described above. When the hero is in the same area as the blacksmith, his script is in scope and the game engine periodically interprets the code in the script (or runs it if it is pre-compiled) to see if something which the hero does triggers any one of the two conditional statements. Let us assume that the hero is within five metres of the blacksmith and is not poor anymore. When the script is interpreted the second conditional triggers causing the blacksmith to greet the hero. This script thus allows the blacksmith to attain an AI in the form of a guarded and opportunistic personality which is biased towards richer folk and weary of poorer folk who would be more inclined to steal from him.

Scripting brings another advantage, especially for gamers who are not part of the original development team. If the right application programming interface (API) is provided for the game it should be possible for these third-party programmers to write scripts for the game [68]. This gives rise to two forms of scripts known as **modifications** (or **add-ons**) and **macros**. Modifications (mods) and add-ons are scripts which modify or add more content to the game such as by altering the original storyline or by adding new missions, quests and objectives [68]. These usually lengthen the life of a game and if done with skill have allowed games, even old ones, to retain a cult-following up to this day. Macros (also known as **scripts** themselves)

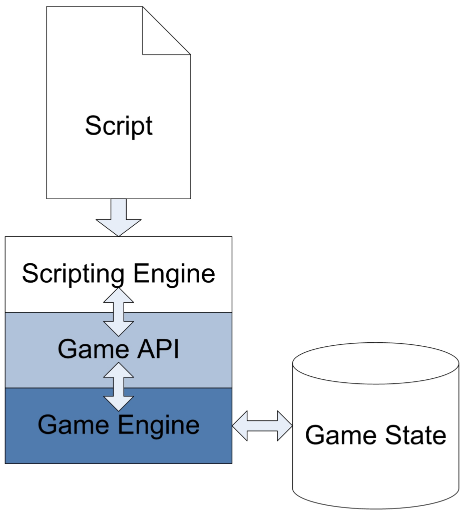


Figure 2.2: High level view of a Game Scripting System

are another form of script which automate certain tasks which could be done by a human player [68]. These are often used to automate menial tasks normally considered as important but which are repetitive such as exploring an area in a real-time strategy (RTS) game or levelling-up a skill in an RPG. However, if the API is powerful enough and the game itself allows it, these allow the player to write the AI for his character or characters within the game so that the AI can play instead of them. In certain games this has led to formal and informal competitions to find out who can write the AI which attains the highest score or finishes the game in record time.

2.4 Overview of a Game Scripting System

At a very high level a game's scripting system is implemented using the modules shown in Figure 2.2. At the lowest level we have the game engine. The game engine lies at the core of any game and is responsible for managing the game's logic as dictated by the game's design. It also interacts with and coordinates various other modules such as those dealing with the processing of input/output and with the loading of any content such as graphics, sounds and scripts [68]. Another module with which the game engine interacts is the game API. The game API is used to make calls to the game engine which the latter then uses to query or update the game's state. The API thus acts as a gateway with which the game engine interacts safely with the outside world [68]. Another module, known as the scripting engine, is able to communicate with the game engine via the API. This module is the one which handles scripts themselves and often acts as a form of interpreter over scripts. When a script needs to be run the scripting engine interprets it and extracts from it a number of API commands which are then carried out [68]. If we revisit our previous example and again assume that our hero is well-adjusted and in close proximity to the blacksmith, we see that once the script is executed by the scripting engine, the engine detects that it needs three API calls to see which conditional is satisfied: one to check the player's distance from the blacksmith, one to check the player's wealth and another one to check whether the player has attempted to interact with the blacksmith a few moments ago. Once the API returns the results the scripting engine is able to proceed with evaluating the required conditional branches each of which, in this case, require the blacksmith to say a message. Again this is done using the required calls to the API. As we shall see in the following section, implementing this system can be done in various ways which vary both in how closely coupled the three modules are and the type of language used for scripting itself amongst other factors.

2.5 Types of Game Scripting Systems

There are various ways in which game scripting may be introduced into a game following the above framework.

2.5.1 Game Engine Integration

An initial attempt at creating a game scripting system is one where the latter is integrated directly within the game engine [68]. Since the game engine and the scripting engine are so closely coupled, the language used is often the language with which the game engine was written with. Thus the scripts themselves take the form of API calls and are usually hard-coded within the game engine itself [68]. This approach brings with it the advantage that it allows the programmer who wrote the game engine to also write the scripting part for the game. Also, since there is no interpretation required, just calls to the API, this is by far the fastest approach when compared to other approaches which we shall discuss shortly.

Unfortunately, since the scripts themselves are essentially modules within the game engine, each time the game's content needs to be updated or modified, the game itself must be recompiled. This should not be the case as the scripts should essentially be content, just like any sound files or 3D meshes, which are loaded by the game only when required [68]. Furthermore, the fact that we have restricted ourself to game engine's language is itself a double-edged sword. Yes, it allows easier programming by the programmer, but in many games the author of the content is often not a technical person. More often then not they are writers who contribute artistically to the game in the form of lore or plots by writing script dialogue and actions. It is highly improbable that they know or understand the language with which the game engine was written in order to write the required scripts for the game. Due to this the game programmer must translate as closely as possible the writer's script and if this is not done well or without feedback from the writer themselves it could make or break the game [15]. Finally, it is not possible to allow the game to be expanded by means of mods or macros. The game is a closed world and apart from taking it apart and rewriting the modules which encode the scripts themselves it has no way of allowing any third party to alter it in order to, for example, update its AI or change its content [68].

2.5.2 General-Purpose Language Scripting

Another method with which scripting can be introduced within the game is by introducing a separate module which sits apart from the game engine. Such a module receives as input a script written in a general-purpose language which is often similar to C++ or Java. Acting as a lightweight virtual machine it then interprets or compiles the script into the required API calls which are then run by the game engine [68]. Many games make use of this solution as it solves most of the problems found in the approach described earlier. The first obvious advantage this method provides is that it imposes a separation between the scripting engine and the scripts themselves. If changes need to be made to the scripts this can be done at ease without having to edit the game engine itself. This also removes the need for constant compilation whenever a script needs to be changed [68]. Another advantage is that the general-purpose language used itself is often similar to the one used to program the engine. This allows the programmers of the engine to be more familiar with the language used and it enables them to write the scripts themselves if this is required. This approach also has the final benefit of allowing the writing of mods and add-ons with relative ease [68]. What is required is only one's familiarization with the language used for scripting. Often the language selected is a popular scripting one such as:

- **Python** – a general-purpose, object-oriented high-level language which is often used both for supporting other languages through scripting and for rapid prototyping of stand-alone applications. It tries to make use of English-like constructs whilst using white space as delimiter. Amongst its most notable features it has: an extensive, portable and cross-platform compatible standard library, graphics support, has interpreters for most operating systems, a dynamic type system, and memory management in the form of automatic garbage collection [27, 44]. Two popular games which are scripted using Python are the turn-based strategy (TBS) game Civilizations IV and the massively multiplayer online role-playing game (MMORPG) EVE Online.

- **Lua** – a lightweight general-purpose, multi-paradigm high-level language which is designed primarily for extending applications as a scripting language and for rapid prototyping. Lua makes use of a virtual machine such that the scripts written using it are first compiled into byte-code and then interpreted. Other special features are that it allows an easy way in which to extend its semantics (such as by adding classes), dynamic typing, simple syntax, automatic memory management in the form of garbage collection, powerful built-in data structures and object-oriented features [37, 27, 68]. Lua has been used for game scripting extensively. Games which make use of it are for example the MMORPG World of WarCraft and the RTS game Supreme Commander.

- **Ruby** – a general-purpose, object-oriented language which has been used for scripting in games. It is very portable and runs on most platforms. It comes in the form of an interpreter which translates the Ruby scripts in a similar fashion to Python. Since Ruby was designed in order to counteract Python's lack of certain object-oriented features it is by contrast purely object-oriented which means that all functions are actually class methods and that it has a class system. Other features are its dynamic typing, memory management including garbage collection, multithreading support, dynamic library loading and exception handling [27, 25, 68]. Since Ruby is the youngest language its use in scripting is limited up till now. A good example of Ruby's use however is in a game development kit called RPG Maker XP which allows one to design an RPG from scratch. Here Ruby is the main scripting language used.

As an example of general-purpose language scripting we will implement our running example involving our hero in Python (similarly for Lua and Ruby):

```
import api


def script(hero) :
    if api.interacted(hero) and api.poor(hero):
        api.say("I don't want no trouble")
    else :
        api.say("Hello " + api.name(hero) + " how may I help you today?")

    if api.within5metres(hero) and not api.poor(hero) :
       api.say("Hi " + api.name(hero) + " fine day we're having!")
```

The first line imports the game's API module which acts as a wrapper around the actual game engine API such that the functions `interacted`, `poor`, `say`, `name` and `within5metres` can be used to query or modify the game state. The rest of the code shows how we can use a Python function (by making use of the keyword `def`) and python's conditional `if` statement in order to write the required script.

Other game scripting languages exist of course but here we have represented three of the most prominently used in games. A general advantage of these languages is that they are fully-featured and well-implemented to suit most scripting needs and are therefore a popular approach to game scripting.

Unless it is developed from scratch, using an off-the-shelf general-purpose language saves time and cost building a custom scripting language. However, due to their generic nature off-the-shelf variants are often over-bloated with features which might not be required by a particular game. This might cause them to be quite slow when compared to other game scripting systems. Also they are technically inclined, in the sense that they are quite easier to understand by a programmer rather than a content designer. They thus still require the translation of scripts into actual code. Finally, integration is not straight forward to achieve and requires a good amount of work to link the scripting language to the game engine's API [68].

2.5.3 Proprietary Language Scripting

A third approach to scripting is by making use of a proprietary language designed specifically for the purpose of scripting the game or game engine (thus allowing it to be used with every game which makes use of the engine). Such a language is often not designed to be generic but domain-specific to the particular game or family of games making use of the engine [68]. This approach shares many advantages found in general-purpose scripting languages in that separation is still maintained between the game engine and scripting engine. This allows the scripts to be seen as content apart from the game engine itself hence removing the need for compilation whenever a script needs to be changed. Moreover, this allows for the possibility of allowing mods and add-ons to be written using some form of plug-in system. Another advantage is that the language may be completely customized and this allows the possibly of making it domain-specific. If written in an appropriate manner it could allow content designers to write the scripts themselves.

There are various forms of proprietary scripting languages. Some of them are command-based and consist of simple commands. Such languages often do not include more complex constructs such as loops and are thus very hard to use to implement complex scripts such as a character's AI [68]. Varanese [68] gives an example of such a language for a generic RPG game which is reproduced here:

```
MovePlayer 10, 20
PlayerTalk "Something is hidden in these bushes..."
PlayAnim SEARCH_BUSHES
PlayerTalk "It's the red sword!"
GetItem RED_SWORD
```

As can be noticed the language is very domain-specific and includes only commands related to the game such as moving the character to a location (`MovePlayer`) and obtaining items from the world (`GetItem`). These languages are thus easier to understand by non-programmers. On the other hand, other languages are almost similar in nature to general-purpose languages but restrict themselves to the game engine only in their scope. Such languages are better suited to encode AI by programmers. An example of such a language is UnrealScript[1] which makes part of the Unreal Engine[2] and which has been used for numerous games. It has been used to script new game content and create game events for such games. The language itself is very similar to Java in nature and is object-oriented. Unfortunately this makes it less attractive to game-designers. An example of UnrealScript code is the following:

```
class HelloWorld extends Mutator;

function PostBeginPlay()
{
  Super.PostBeginPlay(); // Run the super class function (Mutator.PostBeginPlay).
  Log("Hello World");    // Write our log message
}
```

This simple example outputs "Hello World" to the game's log file. It consists of a class which extends a more abstract `Mutator` class. This latter class defines an actor in the game world. What is important to

[1]http://udn.epicgames.com/Three/UnrealScriptReference.html (last visited August 2010)
[2]http://udn.epicgames.com/Three/WebHome.html (last visited August 2010)

notice here is that we have a variant of Java which is restricted to the character or objects acting in the game world.

Proprietary scripting languages are often ideal since they incorporate all the advantages of general scripting languages. Unfortunately, this comes at a cost which most game developers often see as outweighing the benefits, that of creating a language from the ground up. Designing a language is often not an easy task and requires a lot of work extracting what the language's needs are. Also, implementing the required tools such as compilers and interpreters takes a non-negligible amount of time even for the simplest of languages. Finally, fully optimising such tools is not an easy feat. Thus, developing such languages might only be feasible when the game engine is being planned to consist of a complete suite to be sold to and used by third party developers. In such cases it becomes more feasible to create a new language since it can act as one of the features in the game engine's selling-points.

2.6 An Alternative Approach

As we mentioned in Chapter 1, there are a number of factors which may be used as a means of selecting a scripting method over another. These factors allow us to deduce the ideal attributes of a scripting method:

- Ease of language implementation – the amount of work and time required to implement the method's scripting language itself.
- Ease of language tailoring – how easy it is to alter the method's language towards a particular user-base such as AI-programmers or script-writers (non-programmers).
- Ease of language updating – whether a significant amount of work is required in order to add new features to the method's language.
- Ease of language integration – how much effort is needed in order to integrate the method's scripting language within the game's implementation language in order to actually make use of.
- Programmers as target audience – whether programmers can use the method.
- Non-programmers as target audience – whether script-writers can use the method.
- Logic/content separation – the separation between the game engine and the scripts themselves such that the former does not require recompilation each time a script needs to be changed.
- Normal scripting – whether the method allows for content-scripting.
- AI scripting – whether the method allows for enough expressivity to script AI.
- Add-ons/mod – the method's support for third-party scripting.
- General-purpose language available – whether the power of a full general-purpose language is available.
- Requires interpretation/compilation – the need of a method to interpret or compile its script before making use of them.
- Meta-programming – whether the method by itself allows for the possibility of meta-programming.

The methods described above excel in some attributes while performing poorly in others.

Scripting using the game engine integration method does not make use of an explicit language. Thus, since we are in essence making use of the game's API itself, we forego the need to implement and integrate a language. This also means that the language is useful only for the game programmers since they already know the game's implementation language. However, content designers cannot work directly with the language and the separation of scripts from the game engine is virtually non-existent [68]. While it removes the need for interpretation or compilation, removing the scripting language also means that add-ons and macros cannot be added later as plug-ins without requiring a change in the game engine itself. So while this method is suitable for story-line scripting and AI scripting, it incurs various drawbacks in other factors due to its lack of a scripting language.

As a scripting method, general-purpose languages solve most of the problems of game engine integration. By making use of a script language they introduce the separation of scripts from the game engine, allowing for third party plug-ins and are expressive enough to ensure that programmers may use them to program both the story-line and AI [68]. They are, however, not for use by non-programmers. The third method discussed in this chapter, which makes use of proprietary scripting languages shares many of the advantages which general-purpose languages have over game engine integration while at the same time suffering most of its weaknesses. This is due to the fact that both methods make use of a scripting language. Proprietary scripting languages differ from general-purpose languages in the fact that the former vary in their expressive-power and simplicity according to the language designed. They are thus either intended for programmers or for story-writers, exclusively. However, the most crucial pitfall of both general-purpose language scripting and proprietary language scripting relates to the fact that they rely upon a language as the latter must be implemented and integrated. In fact, the creation of a new language is a problem which often causes many game creators to drift away from both of these approaches and instead make use of an off-the-shelf general-purpose language which is often bloated with unnecessary features [68].

In this body of work it is our interest to find a solution for the problems of the general-purpose scripting language approach and the proprietary scripting language approach with the hope that by combining them somehow we can take the best of both worlds. To achieve this we need a way to combine a proprietary language consisting of domain-specific constructs tied to a game with a general-purpose language which includes looping and conditional constructs. In this way such a language can be used both by story-writers in story-line scripting and programmers in AI scripting. Finally, we wish that if we have to implement and integrate such a language ourselves, these processes do not require such a huge effort as to be inadequate for use during game creation.

2.7 Conclusion

In the following chapter we shall discuss techniques which may be used to solve the problems discussed here. As we shall see later on in Chapter 4 such techniques could potentially allow us to create a domain-specific language for a game which is simple enough for story-line scripting. However, when the need arises and a game programmer is involved we can introduce and make use of general-purpose language constructs or other advanced language features which would allow us to write complex AI scripts. Moreover, these techniques allow us to create such a scripting language quickly and efficiently with the added benefit of giving us access to general-purpose language features by default due to the nature of the approach itself. Later on, after making use of our approach, we shall evaluate it using the above attributes as guidelines.

Chapter 3

Domain-Specific Embedded Languages

3.1 Introduction

A domain-specific embedded language (DSEL) in its simplest terms consists of a language pertaining to a particular domain, known as a domain-specific language (DSL), embedded in another host language, usually the latter being a general-purpose language. A more technical definition of a DSEL is that it consists of a number of data structures and operations on such structures related to a specific domain which are expressed in a general-purpose host language. Using this approach, the domain's required syntax is captured and embedded in the host language. Functions in the host language are then able to act as interpreters or compilers of these structures thus giving them the required semantics. An embedded language is thus formed which is able to express programs in the said domain using the host language as a meta-language. In this chapter we shall discuss DSELs in detail with a focus on how they were first introduced, their aims, their advantages and disadvantages, their comparison to their nearest equivalent, domain-specific libraries, and how their host languages are selected. We also describe a number of example DSELs with the aim of further discussing how these were implemented through a number of embedding techniques. Finally, we discuss the idea of interpretation and compilation of DSELs with a focus on language hierarchies.

3.2 Background

DSELs as they are known today were first conceived by Landin [41] in his seminal work "The Next 700 Programming Languages", back in 1966. In this paper Landin remarked that languages consist 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 basic set of constructs it has rather than the way in which they are combined. Therefore if one is able to identify a number of adequate basic constructs, one would be able to automatically obtain a full language which is specific to that particular domain, that is, a DSL. 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 constructs, thereby allowing a whole family of languages to be embedded within it.

Bentley [3] in 1986, returned to the idea of DSLs where he called them "little languages". Agreeing with Landin, he questioned whether DSLs require all the constructs and features often found in full-blown languages, such as loops or variable declarations. His definition of DSLs stated that they are problem-domain specialised and due to this, compared to general-purpose languages, lack certain features found in the latter. In the case of Bentley, as opposed to Landin, however, the idea of full language embedding was not considered. Bentley instead focused on the idea of an intermediate language. He argued that considerable costs, time and effort are saved if instead of having a compiler to object code for every language used, many languages which are perhaps domain-specific, compile into a more generic-language. As examples he used two DSLs for describing chemical structures and scatter-plot graphs which are compiled into a more generic but still domain-specific language targeted towards geometric pictures.

It was in 1996 however that Hudak [31] clearly defined what a DSEL is and made it a popular approach as a viable methodology to develop programming languages for subsequent use in software development. Hudak claimed that abstraction should be utilised as a tool when designing programming languages so as to produce languages tailored to particular domains. This in fact ties in with what Landin and Bentley had suggested earlier on. The difference between Landin's and Bentley's approach, and what Hudak [31, 33] suggested however was his innovative, simple and "lazy" approach, which subsequently lead to the development and widespread use of DSELs. While, Hudak thought, it was appropriate to use DSLs as they offer the right amount of abstraction, he did not want to deal with having to create a new language from scratch, as this often requires a considerable amount of work in terms of language design such as a domain-specific syntax and type-system, tool development related to the domain's semantics such as interpreters and compilers, other utility tools such as debuggers and profilers, and so on. Hudak thus suggested using an existing infrastructure of a well-designed and established language which is up to par with the required needs of the new language. Using such a host language's available language mechanisms and tools, adapted to a particular domain it is possible to create a DSEL, or a DSL embedded inside another language. This approach greatly facilitates the creation and use of DSLs in embedded form, and similarly to Bentley's intermediate language approach, has the added advantage of reducing costs, time and effort by not reinventing the proverbial wheel. Instead, the work required is only in adding the domain-specific functionality to the host language.

As Hudak [33] himself noted, the first actual uses of DSL embedding were done in one of the first general-purpose functional languages developed in the late 1950s by Turing award winner John McCarthy called Lisp [64]. Lisp's macro system is powerful enough to allow language embedding to be done with relative ease. For example, in Sandewall [57], the author discribes a number of small, useful embedded languages which are represented as Lisp data structures. Since they are normal data structures they can be manipulated by normal Lisp functions and can be interpreted and compiled using the tools available to Lisp. Hence the embedded language becomes a first-class object in the host language.

Nowadays, another functional language, Haskell [24, 34, 50, 66] has become one of the languages of choice when it comes to language embedding due to many of its features, such as: higher-order functions, polymorphism, lazy evaluation, type-classes, monads and non-restrictive syntax. This does not mean that such features are a requirement but rather valuable assets which greatly facilitate the process of language embedding [31, 33, 39]. Languages related to various domains which have been successfully embedded in this language include domains such as: geometric region analysis [31, 33, 36], geometric constructions [6, 26], images [21, 20, 23, 39, 40], animations [18, 19, 22], music composition [32, 34], financial contracts [38], query languages [42], hardware description languages [2, 4, 8, 10, 12, 51], testing [9], robotics [34], firewalls [58] and business processes [47]. Here it is worth noticing that while functional languages might be more amenable to language embedding and have been used successfully to achieve it, such as in this case for Haskell and other languages in the same paradigm such as Standard ML (see Kamin [39]), this technique is applicable to any high-level language of choice in any paradigm in varying degrees of success.

3.3 Aims

Traditionally, the design and implementation of a language, even a relatively small domain-specific language, is a costly and time consuming process which requires a lot of thought and effort from the language creator's side [31, 33, 39]. The language must be first designed by selecting the features it requires such as its syntax, evaluation method, type system and so on. This process is usually followed by implementation of basic tools in the form of interpreters and compilers, and other extra tools such as integrated-development environments and debuggers. Finally, a considerable amount of documentation is required, both technical and also on a user-level. Due to this, more often than not, despite the fact that domain-specific languages supply us with the "ultimate abstraction" [31, 33] which allows us to develop programs in the target domain quickly, when confronted with the question of which language to use to develop a system, we often end up picking a well-established and supported off-the-shelf general-purpose language, which caters for our needs as much as possible. In doing so we acquire the language's advantages and also prepare to suffer its disadvantages, one of the latter being its non-specialised nature. The main aim behind DSELs is to counter this problem by making DSLs more easily accessible and viable for use in the software development life-cycle [33]. To achieve this, we still have to pick carefully the language of choice for development. However, once selected, by means of language embedding, this language is used to develop a DSEL within itself which is geared towards the particular domain required by the software project. The process of DSL-creation using embedding is often quite straightforward to achieve when compared to the traditional approach highlighted earlier where the required DSL is created from scratch.

In the embedded language approach what is still required is figuring out the basic constructs of the required language. If, for example, we want a language which allows us to create circuits, we have to select its primitives such as wires, inputs, conjunction gates and negation gates, and express these in the host language, usually using appropriate data-structures and functions. Using this approach programs in the new language are simply first-class objects in the host language which have the "look and feel" of syntax [31]. The host language acts as a meta-language allowing us to make use of functionality of both the embedded language and the host language. Functions available to the host language can manipulate the programs written in the embedded language itself. Also, the semantics of the language, whose syntax is represented by the data-structure, can be obtained by creating the appropriate interpretative function upon it, in the host language. This can give rise to various interpretations for the same language in the form of embedded interpreters [31, 33] and embedded compilers [23, 42].

The implementation phase of a DSEL is also reduced since the embedded language inherits not only its host language's features but also its tools [31, 33, 39]. There is no need for the language creator to re-implement everything anew for the new language as what is already available for the host language becomes immediately available for use. This is the strong-point of this approach which greatly makes developing DSLs affordable both in terms of costs and time for language creation. Once the creator of the DSEL creates the required functionality in terms of the host language, the language can be put to use immediately.

Documenting the new language is also relatively easier since we are still in fact technically programming in the host language. If a language developer wants to extend the embedded language with a new construct, they can immediately do so using their knowledge and familiarity of the host language [42]. There is no need to go through a lot of documentation describing implementation details to find the correct spot of code to alter to add the required feature to the language. In this way technical documentation is greatly reduced to describing the added functionality provided by the embedded language. Finally, in regards to user documentation, DSELs as is the case with DSLs, require considerably less work in this area as well. Since the language pertains to specific domain and its users are usually well-acquainted with said domain, they apprehend the language with ease and learn how to use it quickly [31, 33].

3.4 Advantages and Disadvantages

Most of the advantages of DSELs stem as counterparts to the disadvantages found in DSLs. Some advantages of DSELs are as follows:

- **Creating a DSEL is usually much easier and faster than creating a DSL as there is no need to build everything from scratch** — We noted this earlier when we discussed the aims of DSELs.
- **The language obtained is usually as powerful as the equivalent built using traditional methods** — In fact, according to Kamin [39], one of the benefits of embedding is that one can be sure that the power attained by the embedded language is actually at least as much as that of its host language's.
- **DSELs allow the use of their host language if this is required** — The creation of a DSEL within a general-purpose host language does not mean that the latter cannot be used anymore. To the contrary, the host language's features and capabilities are still present and these can be used to manipulate the DSEL programs. This is often useful when the users of the DSEL are programmers themselves. It is also possible however, through host language capabilities, to restrict the use of the host language and allow only DSEL constructs to be in scope. The latter is useful when the users of the language are domain-experts and not programmers.
- **The embedded approach caters for the possibility of language evolution** — It is frequently the case that during language implementation, new constructs are added as needed. In fact, one might not achieve the required language at the first attempt and changes need to be done to the implementation. In the DSL case, managing changes requires a lot of work and is often done in inelegant ways [39]. It is in fact one of the main disadvantages of the approach. DSELs solve this problem by making DSLs more maintainable and more easily updateable [31, 33, 39]. Making changes in a DSEL is often a question of altering the implementation of the embedded language in the host language. This, more often than not, is rather straightforward. Hudak [31, 33] gives a good example of how a DSEL can evolve. He describes how in a joint project between Arpa, ONR and the Naval Surface Warfare Centre (NSWC) an experiment was carried out where DSELs were used in the area of geometric region analysis. A DSEL was created which was developed in three sequential stages. Each stage differed from the previous one by the amount of the domain captured by the language. The changes were quite easy to integrate with the previous code making DSELs indispensable in the project [31, 33].
- **When using DSELs it is possible to embed more than one language in the same host language** — It is thus possible to have languages from different domains communicating with one another using the meta-language as a form of common ground [33, 42]. As Leijen and Meijer [42] point out, this is a drastic improvement over DSLs due to the fact that for two separate DSLs to communicate with one another it is not usually a straightforward process. Moreover, there is the added benefit that since a common host language is involved, the embedded languages have a similar look-and-feel and can share the same tools. This drastically reduces the time required to familiarise oneself with the languages and the tools used to work with them.
- **The abstract nature and simple syntax of DSELs is an advantage they share in common with DSLs** – DSLs are sometimes targeted at non-programmers who are domain experts. The idea is that by means of their knowledge of the domain alone they can understand the programming language enough so as to perhaps being able to even program with it themselves with minimal training involved. In fact most of the constructs selected to be part of the DSL would be directly related to an object or action in the domain. DSELs can be easily designed and implemented with the aim of having a simple

syntax as well. In this way they share the advantage of being accessible to domain experts like DSLs are. As a simple example consider constructs such as `and2(x, y)`, `low` and `high`. A circuit designer can immediately realize that these constructs correspond to a two-input AND gate, a wire carrying a low-voltage value and a wire carrying a high-voltage value, respectively. The designer can create a new component such as a three-input AND gate easily using just the first construct as follows:

```
and3 (x, y, z) = ans
  where t   = and2 (x, y)
        ans = and2 (t, z)
```

A DSEL called Lava [4, 8, 10, 12] behaves in this spirit. As another example, Hudak [31, 33] observes that the highly comprehensible nature of DSELs can be noticed by the reactions of the non-programmers taking part in the NSWC experiment mentioned earlier, where people who did not know the host language Haskell could immediately understand the language related to geometric regions. Combining with this the fact that DSELs can be created and evolved quickly, we see how useful DSELs could be when used to develop DSLs while keeping domain experts involved in the process.

- **DSELs allow their users to prove properties about the domain** — Hudak [31, 33] argues that DSELs allow their users to prove properties about the domain in a very straightforward manner compared to other methods, thus making DSELs very useful for formal verification. The approach which can be taken is to reason directly in the domain semantics of the embedded language. Languages like Haskell have a simple algebraic semantics [30] which allows this to be achieved with ease. Using a set of established axioms in the domain it is then possible to prove more elaborate properties about the said domain. This was the successful approach taken in the NSWC experiment mentioned earlier.

Kamin [39] points out a number of problems with the DSEL approach. These are:

- **The possibility of non-optimal syntax** — Language embedding might give the syntax of the DSEL the feel of its host language instead of having a pure feel of a new, totally different language, as is the case of languages developed from scratch. This is in fact a positive and a negative aspect at the same time. On one side it allows experienced programmers in the host language to quickly become familiar with the language but as Kamin points out it might make programmers feel that the language is unnecessarily awkward due to the host language-based extraneous syntax which clutters the domain specific syntax. This might be a learning-curve disadvantage for the domain-experts who might need to utilise the language.

- **Error reporting** — When a program in the embedded language is compiled or interpreted using the host language's compiler or interpreter, the latter treat the program in the embedded language as any of its programs and therefore give out error messages designed for its own programs. Such errors are only understandable by programmers who know the inner workings of the host language. If the target audience consists of domain experts who are not programmers the messages generated are often not helpful and discourage the use of the embedded language. Moreover, there are cases where it is possible while programming in an embedded language to inadvertently create code which compiles or interprets correctly for the host language but which does not behave as intended as far as the embedded language is concerned [39].

3.5 Comparison with Domain-Specific Libraries

It is common practice in the development of domain-specific programs to make use of a general-purpose language to first create a library which pertains to the required domain. Such a domain-specific library is then put to use to create the required program. The program itself is written in the general-purpose language and calls are made to functions in the domain-specific library to achieve the domain-related effects required. At first glance this seems remarkably similar to the domain-specific embedded language method. In fact, both domain-specific embedded languages and domain-specific libraries have common attributes and thus it is quite hard to clearly define the dividing line between the two approaches.

The usual implementation of a DSEL takes the form of a library written in terms of the host language. This in itself makes it difficult to distinguish between a DSEL library and a domain-specific library as they appear the same to the host language which uses them. Moreover, their primary aims are the same as both library types try to represent as thoroughly as possible the required domain and to provide domain-specific functionality to a more generic language. Another feature which DSEL libraries and domain-specific libraries have in common is that both library types are written in the same general-purpose language which makes use of them. Subsequently, in order to be used they do not require their own proprietary compiler or interpreter, but rather make use of the tools of the language they were written with. Finally, both a DSEL and a domain-specific library can be used in conjunction with other DSELs and domain-specific libraries respectively since in both cases the language used to implement them is common.

Bentley [3] points out that one of the main differences between DSEL libraries and domain-specific libraries stems from the fact that domain-specific libraries are more useful when being incorporated into large programs while DSLs (and subsequently DSELs) are more natural to work with when creating stand-alone programs in the domain at hand. In fact, a domain-specific library requires thorough knowledge of the base language such that it can be integrated to be used to develop the intended program. On the other hand, a DSEL library can be used in such a way to give the illusion that the programmer is writing a program in a language that is distinct from its host language even if in reality this is not in fact so. This alone accounts for the fact that non-programmers are able to grasp the concepts behind an embedded language quickly while they would be at a loss if they are presented with a domain-specific library. This approach for the classification of DSELs is also taken by Pace [51] where he states that a DSEL is in fact a form of structured domain-specific library but which does not require the programmer to use it in conjunction with its host language. Just like a DSL, a DSEL allows separation of concerns, something which is harder to achieve with a domain-specific library. If for example we have a DSEL library which specialises on the creation of images, a programmer can use it to model the required images without thinking of how they will be actually represented. A function in the host language can then "interpret" or "compile" the model in the embedded language into a representation in the host language. There is a separation of *what* the model is from *how* it should be represented. This separation of modelling from representation is harder to achieve with domain-specific libraries since more often than not when using such libraries these concerns have to be intertwined to achieve the required result. This idea, discussed by Elliott and Hudak [18, 19, 22], can be adapted for other DSEL libraries and is thus a difference between the two library types.

Another difference which allows us to contrast DSEL libraries and domain-specific libraries is the variation one finds in how they represent the domain at hand. DSEL libraries tend to represent the domain by abstracting to its main notions. Even though the host-language is used to implement the DSEL, the DSEL library focuses on capturing and modelling the domain's core objects represented as structures and core actions represented as functions on such structures. Domain-specific libraries on the other hand tend to stay in the same level of abstraction as the language they are written in. In fact they do not try to reason in the domain-semantics but rather in the domain-semantics in context to the language being used to write the domain-specific library. Due to this fact it is more straightforward to write something in the related domain using the DSEL library rather than the domain-specific library as the latter has to achieve this through an indirect approach whereby its user has to first themselves work out how to achieve the required behaviour by combining the available functions. This leads us to another distinguishing feature between the two library types. Using what are known as combinator functions [2, 4, 8, 11, 50], represented as higher-order functions, the user of a DSEL library is able to pass functions as first-class parameters and create new functions by composing sub-functions. In this way a DSEL library's functionality can be extended on-the-fly according to the user's needs. This sort of flexibility is often not available to domain-specific libraries. New functionality is implemented either by creating an updated version of the library to replace the original one or by implementing a wrapper library which adds the extra features on top of the original library. DSEL libraries offer extensibility by nature if the language created captures the domain accurately: the combinator functions act as place holders for the assimilation of two or more basic pieces, which together form the new functionality.

3.6 Selecting a Host Language

Since the expressive power attainable by a DSEL is a factor which is mostly dependent on its host language selecting the correct one and in the correct paradigm is of crucial importance. Thus the first step in DSEL creation usually requires the language creator to go over in detail the features of the host language paradigms. In language embedding this is usually a case of selecting between the functional paradigm and the imperative paradigm. Features which are considered here are ones such as whether to use lazy evaluation normally found in the functional paradigm versus eager evaluation which is usually a component of the imperative paradigm, and whether higher-order functions are required since these are normally found in the functional paradigm but are not standard in the imperative paradigm. Once a paradigm is selected, a number of host language candidates in that particular paradigm are assessed to decide which language has features which best fit the criteria required by the embedded language. In the functional paradigm for example one of such features is whether to opt for a pure host language or an impure one since this dictates the purity of the embedded language itself. Functional language purity depends on whether the language at hand allows side-effects such as state and input/output inherently or not. Haskell, despite being a referentially transparent language and hence classified as pure, may still introduce such side-effects by means of monads [69]. Languages such as Scheme and Standard ML have a number of built-in side-effects such as assignments, exception handling and continuations which cause them to be impure functional languages. If language purity is recommended for example Haskell would be the better candidate.

As noted by Mernik et al. [46] in their study regarding a methodology for DSL development, in the case of using the embedding approach, more often than not researchers select functional languages. Languages in this paradigm have features, such as, pattern matching, lazy evaluation, module system, higher-order functions, strong typing, polymorphism and overloading which are useful for language embedding [39, 46]. The importance of these combined features, and hence the functional paradigm itself, is however the level of abstraction it offers its users. When used in the context of embedding, it is the abstraction offered by functional languages which allows the user to focus specifically on the domain itself rather than having to consider a lot of implementation details which clutter domain-specific thoughts. This is the primary reason why this paradigm is often selected in favour of others when selecting a host language.

One of the most prolific functional languages used for embedding is Haskell. As mentioned earlier, Haskell has been used as a host language for a considerable amount of DSELs related to various domains. According to various researchers [31, 33, 42, 51] it has many of the features just mentioned which make the process of embedding a much straightforward process. Hudak et al. [35] remark that probably Haskell's strongest features which lead it to be often used as a host language for embedding are its type classes, higher-order functions, infix syntax, over-loaded numeric literals, monads and lazy evaluation. Hudak [31, 33] remarks that despite the fact that languages with less features can be used, Haskell's features greatly facilitate the effort required for embedding. In the following section we will take a look at some of the DSELs which have been embedded in functional languages, most notably, Haskell.

3.7 Examples of Domain-Specific Embedded Languages

In this section we give a brief overview of various domains and their respective languages which have been embedded successfully in functional languages. As can be immediately noticed the domains which the languages represent are diverse, some remotely so. This in itself is indicative of the versatility of the embedded approach in capturing domain semantics by the use of abstraction. Moreover, the fact that most of these languages are embedded in Haskell gives evidence to the power of this language in the context of language embedding.

3.7.1 Geometric Region Analysis

In 1993 an experiment was carried out with the purpose of gauging the usefulness of prototyping in developing real-world applications in an efficient manner [36]. The experiment's main aim was to determine which languages maximise output in the form of executable code and minimise the effort and cost in terms of lines of code and development hours needed, respectively. Many candidate languages were chosen and used to develop a system's module which is concerned with the position of objects in relation to one another. Such a module can be considered a case of geometric region analysis since the domain concerns itself with the definition of regions and interactions of points in such regions.

The system developed by the team Haskell team consisted of a DSEL related to the domain at hand. Compared to the other solutions the Haskell one did remarkably well. Not only was the code delivered for the final solution the shortest in terms of lines of code but it also took much less time to deliver when compared to most other languages being compared. This not only shows the capacity of Haskell as a language but also the viability of the embedded approach for delivering solutions quickly and in a cost-effective manner. As noted by Hudak and Jones [31, 33, 36], this approach shows how DSELs produce maintainable and evolvable code since the latter was developed in three subsequence stages, each adding more functionality and making the system more general. According to Hudak and Jones, factors such as modularity and abstractions made possible by Haskell's simple syntax, the use of higher-order functions and list-manipulating primitives is what made the DSEL methodology using Haskell successful. The authors also remark that the use of formal methods to prove properties about the system representing the domain is very straightforward. Finally, it is worth noticing that due to the easily understood, high-level nature of the code produced in this way, the reviewing parties found it hard to distinguish between the latter and a top-level design with requirements specification. For example, the required engageability zone of the weapon system shown in Figure 3.1 was represented using a geometric region called an annulus described simply as [31, 33, 36]:

```
annulus :: Radius -> Radius -> Region
annulus r1 r2 = outside (circle r1) /\ circle r2
```

The actual definition of the annulus in the DSEL uses domain-specific, language constructs to define the region: `circle` creates a circular region given a radius, `/\` computes the intersection of two regions and `outside` computes the logical negation of a region. Using two radii as input and combining the constructs as shown the required region is formed. Notice how only domain-related words form part of the description leading to high understandability and ease of translation from specification to actual code.

3.7.2 Geometric Constructions

Grima and Pace [26] created a DSEL which deals with geometric constructions that can be drawn with a straight edge and a compass. By selecting and abstracting notions in this domain such as a point, a line and a circle, and creating a set of possible base constructions around such primitives, one is able to define complex geometric constructions. The primitives are modelled in Haskell as normal structures while the basic constructions consist of a number of Haskell functions which adequately manipulate these primitives.

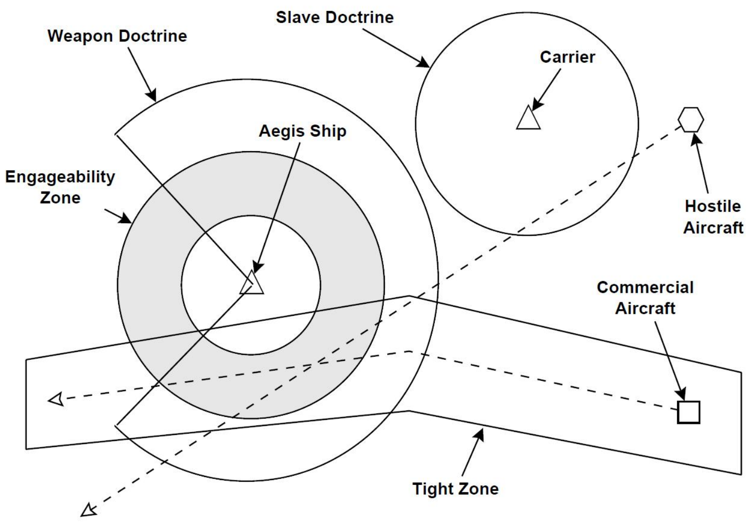


Figure 3.1: The engageability zone of a weapon system can be described by the geometric region which takes the form of an annulus as shown here. Reproduced from [36].

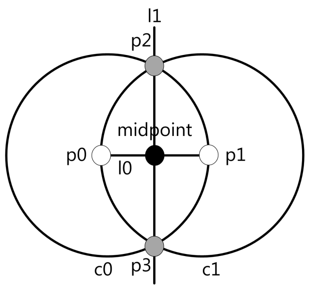


Figure 3.2: Example geometric construction which results in finding the midpoint between two points. Adapted from [26].

Creating a more elaborate construction is then a case of composing the basic constructions to form the required result. As an example of this DSEL consider the following code which is required to create the construction shown in Figure 3.2 that finds the midpoint between two points named `p0` and `p1` [26]:

```
midpoint (p0, p1) = midpoint
  where c0 = circle (p0, p1)
        c1 = circle (p1, p0)
        [p2, p3] = intersections (c0, c1)
        l0 = line (p0, p1)
        l1 = line (p2, p3)
        midpoint = intersection (l0, l1)
```

In this example we see the use of three domain-related components: lines, circles and points. The first two are created by means of their respective Haskell construction functions acting as DSEL constructs: `circle` and `line`. Points are created by functions (`intersection` and `intersections`) which detect the intersection points between domain objects. The `midpoint` program here can itself form part of other more complex constructions. Thus non-basic constructions can themselves become components in even more complex constructions hence forming modular constructions. Grima and Pace build upon such a DSEL to provide not only concretisations in the form of textual descriptions and 3D models, but also check the correctness of the constructions created by random testing and through formal verification using a theorem prover.

Another DSEL related to geometric constructions is OriDSEL, a DSEL developed by Caruana and Pace [6], this time focusing on Origami-based constructions. In this language, primitives are formed of points and lines (or folds) again modelled in the host language Haskell as structures. It is possible, using the available lines and points of a model to perform the main actions of folding and unfolding of paper-based plains. This allows the formation of new lines and points thus creating a new geometric construct. Line creation using folding is based on seven Origami axioms, while new points are created by the intersection of two lines and the non-deterministic choice of a point on a line done intuitively. All these are introduced in the DSEL as functions in Haskell. An important thing to notice about this language is that it also allows the formation of models from sub-models instead of just using the basic folds. This compositional nature is especially useful when building complex Origami models. The language provides concretisation features used to create a textual, HTML or animation-based description of the model so that it can be understood more easily by a series of easy-to-follow steps. Finally, the language also allows constraint checking of the models created since if certain constraints are not met the model cannot be concretised. An example of a simple construction which can be described with OriDSEL is the selection of the edges of a rectangular piece of paper, given that we know its four corners [6]:

```
fourSidedPaper :: (Point, Point, Point, Point) -> (Line, Line, Line, Line)
fourSidedPaper (nw,ne,se,sw) = (n,s,e,w)
  where n = foldThroughPoints (nw, ne)
        s = foldThroughPoints (sw, se)
        e = foldThroughPoints (se, ne)
        w = foldThroughPoints (sw, nw)
```

Here we see two of the language's primitives around which this DSEL's domain is built. These are the point and the line, represented by the types `Point` and `Line`, respectively. Four points representing the four corners of the rectangular piece of paper must be supplied as inputs to this construction. The steps of the construction itself consist of using this language's `foldThroughPoints` construct to carry out an axiomatic fold four times. By selecting the correct points per fold we obtain four new fold-lines which delimit the four-sided paper construction we want to achieve.

3.7.3 Images

Pan [21, 20, 23] by Elliott, is a DSEL embedded in Haskell which may be used for image synthesis. The main aim behind this DSEL is to allow its user to write programs which generate images where the latter are not riddled with implementation details not directly related to the image itself. To achieve this, Pan separates *what* an image is from *how* it is displayed. This separation of concerns allows one to write elegant and compact code which describes the images themselves. Pan can then use such programs to generate and concretise the actual image. In order to achieve this efficiently Pan is implemented as an embedded compiler [23] as we shall see later on in Section 3.9. Haskell's higher-order functions are useful here since in Pan images are seen as functions from a point in the Cartesian coordinate system to a type such as colour. Thus any image manipulation upon these images has to be a higher-order function. Polymorphic types, also available in Haskell, allow images to range over many domains such as colour and Boolean regions. The DSEL also supplies functionality which often makes part of image synthesis such as translating, scaling and rotating as combinator functions since they essentially combine smaller functions together to achieve the required functionality. By introducing the notion of time, a series of images can be formed which form an animation. This is the main idea which extends Pan into Fran, another language which we shall discuss shortly. Elliott [21, 20] also discusses how Pan can be used to reason about Boolean-valued regions as a form of region algebra. A simple, interesting example which shows the strength of Pan is the code required to create an infinite checked pattern shown in Figure 3.3. Such an image can be constructed as follows [21, 20]:

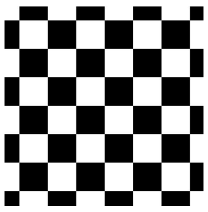


Figure 3.3: A chequered pattern produced from an example Pan program. Reproduced from [21, 20].

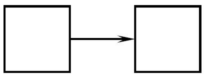


Figure 3.4: A simple flowchart which can be produced by an example FPIC program. Reproduced from [39].

```
checker :: Region
checker (x, y) = even (floor x + floor y)
```

As can be noticed, the description uses types and constructs related to the domain only. The type in this example, `Region`, consists of a function type from a point in the Cartesian plane to a Boolean and the functions `floor` and `even` calculate the floor of a value and check whether a value is even, respectively. Combined in the way shown above the code creates the chequered pattern effect.

Kamin's FPIC [39, 40] is a DSEL which also focuses on image creation. It is based on a Unix-based preprocessor for troff called pic. FPIC achieves what pic does without the need of preprocessing by its embedding into Standard ML. As a language it consists of a small number of primitive types which refer to points and pictures and a number of functions which comprise the language's functionality. Some of the functions construct pictures such as squares, circles or lines while others combine pictures into new pictures by superimposing a list of pictures or by placing two pictures next to each other in a horizontal or vertical fashion. A simple example of FPIC is presented in Kamin [39] where he describes the code required to compose two boxes of unit size sequentially connected by a right-directed arrow between them, as shown in Figure 3.4. The code used to achieve this image is as follows:

```
val sq = square 1.0;
val boxes = sq hseq harrow 0.5 1.0 hseq sq;
boxes;
```

Three of the FPIC's constructs are the functions `square`, `harrow` and `hseq`. The first two create a square and a horizontal arrow, respectively, while the latter sequentially composes two pictures together, one next to the other. Some of the basic, domain-related types around which the language is built are `Picture` and `Point`. It is possible, by combining more basic FPIC functions to create more complex functions which can be packaged in libraries specialised even more than the DSEL itself such as for example for plotting graphs or drawing pie-charts [40].

3.7.4 Animations

Another DSEL by Elliott et al., is Fran [18, 19, 22] (for Functional Reactive ANimation). Fran is based heavily on Pan and extends the latter by introducing the notion of time. This allows the creation of sequential images and hence an animation. In order to achieve this effect, since this language is embedded in Haskell, functional reactive programming [14, 19, 22, 65, 70] is used. Functional reactive programming is based on two abstract notions: that of behaviours that happen continuously over time and that of events which occur at discrete events in time. When used for Fran, these two notions allow the language to have values which act over time over subsequent images and also animations which react to an event. An example of a behaviour is a ball moving upwards and downwards according to Newtonian laws. Such a ball can also react to an event such as an impact with a floor or a ceiling.

A simple example program in Fran which displays both the behaviour and event aspects of an animation is the following:

```
pulsatingRedBlueBall = withColor (red ‘untilB‘ lbp -=> blue)
                                 (bigger (wiggleRange 0.5 1) circle)
```

The behaviour consists of a pulsating red ball whose size varies continuously with time. The colour of the pulsating ball changes to blue upon the event of a left mouse button press. The functions `withColor`, `bigger` and `wiggleRange` are functions related to different behaviours which have been combined here to achieve the pulsating ball effect: `withColor` sets a colour, `bigger` stretches a point to a shape and `wiggleRange` varies a value between two bounds. The functions `untilB` and `-=>` are constructs related to events: the first captures the occurrence of an event while the second is an event handler which allows us to respond to an event accordingly. This example was modelled in a single line using constructs which are domain-specific to animation and reactive programming. Like Pan, Fran is geared at describing the model and then allowing the representation to be handled elsewhere, that is, separation of the model from the presentation. Fran extends Pan's primitive notions to include 2D images, 3D geometry, geometric transforms and also the notions of reactive programming. Moreover, a number of functions, including some of the ones present in the example, are also provided as language constructs which manipulate these types to create the user's desired animation. According to Hudak [33] it is also possible to develop an algebra of animation using Fran. In fact, the algebra developed for Pan, that is, for images, can be shown to be extendible to include animations by means of Fran. Hudak [34], describes another language based on animations called FAL (for Functional Animation Language). FAL is a subset of Fran and can be considered as the same language since both are based on the notions of behaviours and events.

3.7.5 Financial Contracts

Peyton Jones et al. [38] have created and implemented a DSEL in Haskell which allows the composition and evaluation of financial contracts. Even though the number of contract types seem to be endless, one is able to come up with a number of basic contracts and a number of ways to combine them. Due to this fact, this DSEL is implemented as a combinator library with a basic set of primitives related to the basic contracts and a set of functions used to combine and compose such contracts into more elaborate ones. This allows an extensible amount of contracts to be described. Also, since a contract is based on a fixed set of primitives, even a complex contract can still be analysed, processed and evaluated in terms of its components. An example of a simple contract which can be described with this DSEL is one where its holder receives £100 at 1530GMT on January 1st 2010, which is written as follows:

```
simpleContract = scaleK 100 (get (truncate (date "1530GMT 1 Jan 2010") (one GBP)))
```

We see the use of five primitives here: `date` which is a constructor function which creates a date, `one` which creates a contract of one unit of a currency which lasts forever, `truncate` which makes sure that contract is not acquired later than a date, `get` which makes sure that a contract is acquired at the latest possible date and `scaleK` which multiplies the value of a contract a number of times. Combined together these primitives ensure the required behaviour of £100 being payable only at a particular date.

Abstraction plays an important role here. Since the language being used focuses specifically on the domain at hand, one can abstract away from the implementation details. This allows one description of a contract to be used for many automated tasks related to implementation such as concretisation, risk analysis and scheduling [38]. Moreover, since contracts depend on time, reactive programming's behaviours are used in the same way as they are used in Fran. This allows a number of formal proofs related to contracts to

Object	Edible	Inheritance	President
Rich people	False	False	True
Bean plants	True	False	False
CORBA	False	True	False
COM	False	False	False

Table 3.1: The `Rogerson` table upon which the example SQL query and its equivalent Haskell/DB query are performed [42].

be created and allows the processing of a contract to find its value at a given point in time. It is here that Haskell's use of lazy evaluation is put to use in the DSEL itself. Evaluating a contract by processing is an expensive process (computation-wise). Using laziness allows the focus on part of a contract at a time and also allows partial evaluation when there is no need to evaluate all the contract. According to the authors [38], it is possible to make the evaluation into processes faster by extend this DSEL with techniques such as observable sharing [11] and domain-specific compiling [23, 42]. Observable sharing relates to the fact that due to Haskell's referential transparency, one cannot distinguish between the contracts being evaluated. This causes unnecessary evaluation of the same objects since the DSEL cannot distinguish between them. The aforementioned technique makes this possible by making use of an implicit label for each contract. Embedded compilers on the other hand allow the translation of the DSEL into another language such as Haskell or C. This code can then be compiled using the latter language's compiler which creates much faster executable code. Each of these techniques shall be discussed later on in Sections 3.8.3 and 3.9 respectively.

3.7.6 Query Languages

Query languages such as SQL enable other languages such as VB or Java to query databases so as to retrieve, manipulate and store data. In fact SQL can be seen as a DSL which focuses on databases. Leijen and Meijer [42] show how it is possible to create a DSEL based on SQL and acquire a number of advantages in the process. The DSEL they create is called Haskell/DB and, as its name implies, it is embedded in Haskell. As an example of an SQL query which can be re-written in Haskell/DB, Leijen and Meijer [42] use the following SQL query:

```
SELECT r.Object
FROM Rogerson AS r
WHERE r.Edible = TRUE
```

which is carried out on Table 3.1 called `Rogerson`. In Haskell/DB the same query is written as:

```
do r <- table rogerson
   restrict (r!edible .==. constant True)
   project (object = r!object)
```

This query should return one row in its result set as only one row of the table satisfies it. As can be noticed using monadic do-notation the same exact query can be easily achieved. Constructs related to the domain such as `table` which declare which table is being used, `restrict` which selects the subset of rows with attributes satisfying a property (equivalent to the selection operator σ of relational databases) and `project` which selects a number of columns from the table instead of all of them (equivalent to the projection operator π of relation databases), are being used here. These are similar to the SQL keywords which achieve these effects, in this case the keywords `FROM`, `WHERE` and `SELECT`, respectively.

As Leijen and Meijer [42] point out, there are various problems when one uses SQL as an interface to a database. First of all, it is possible to write queries which are syntactically invalid or with incorrect types. This is because the language making use of SQL is not able to know the contents of the query as for the former the query is just a normal string. Another problem of this approach is that programmers have to learn two different languages altogether. Finally, there are various implementations of SQL, which might cause problems for people making use of the language as they have to learn the differences between the variants. By the use of Haskell/DB, Leijen and Meijer hope to mitigate these problems by obtaining a more transparent interface for programming with a database.

The first problem can be solved by a technique often used in domain-specific languages called phantom types [42]. Phantom types which restrict somewhat the types of the language so that the chances of errors due to syntax are reduced greatly. This allows the embedded language to become strongly-typed and allows one to write code which is checked during compile-time. Phantom types is a technique we shall discuss more in detail later on in Section 3.8.4. The second problem is solved by virtue of embedding itself. By embedding Haskell/DB in Haskell, one does not have to learn two completely different languages but rather one language and a domain-specific library. While opting for this approach still requires one to learn how to use the library, Haskell's non-restrictive syntax allowed the creators of Haskell/DB to adapt the DSEL's syntax such that it looks and feels like normal Haskell rather than a completely new language. Since the users are familiar with the syntax, they find it easier and natural to learn how to use the DSEL and what remains to be learnt is the domain-specific material only. This allows integration with Haskell to become seamless and with a much cleaner interface to database queries. This is in contrast to other languages where SQL is used where the separation between program code and query language code is more pronounced. The final problem related to the various SQL implementations available can be solved by making use of relational algebra as the basis of the DSEL instead of selecting one variant. This allows the DSEL to target different dialects, making it portable. To achieve this Haskell/DB makes use of a compiler embedded in Haskell which translates the DSEL-related code into the required SQL variant. Any variants can be added separately since the DSEL focuses specifically on the domain and abstracts away from the implementation details which can be added to or altered as required without affecting the actual language.

3.7.7 Music Composition

Haskore [32] and its simplified version MDL (for Music Description Language) [34] are two DSELs developed by Hudak, embedded in Haskell which are concerned with the domain of music composition. In both languages Hudak abstracts the primary notions of music such as notes and rests into data structures and creates two types of functions which work upon these structures. The first type of functions are related to transforming the structures themselves, such as, transposing the structures to be played earlier or later, and changing the tempo of the music, that is, the rate at which the music piece is played. The second type of functions combine different structures into one structure, that is, perform music composition by composing two or more music structures into one structure, thus forming one music piece. Examples of music composition functions are sequential composition and parallel composition where two music pieces are played sequentially or simultaneously, respectively. Hudak [32] gives a simple example of a music piece described in Haskore which is reproduced below:

```
funkGroove
  = let p1 = perc LowTom        qn []
        p2 = perc AcousticSnare en []
    in Tempo 3 (Instr "Drums" (cut 8 (repeatM
         ( (p1 :+: qnr :+: p2 :+: qnr :+: p2 :+:
             p1 :+: p1 :+: qnr :+: p2 :+: enr)
```

```
            :=: roll en (perc ClosedHiHat 2 []) )
        )))
```

The above code creates a simple music composition which can be played with percussion instruments. As can be noticed all the Haskore constructs made use of here, such as for example `perc`, `LowTom`, `AcousticSnare`, `ClosedHiHat` and `Tempo`, are related to the domain of music. The operators `:+:` and `:=:` are sequential composition and parallel composition, respectively.

The structures created, by themselves, however, are not music. A translation must occur which translates these structures into a concrete form such as a music file which can be played by a computer's sound card for example or printed into music notation to be played by a human performer. To achieve this, Hudak implements various interpreters embedded in Haskell itself which give semantics to the music structures and translate them into for example MIDI files [32, 34]. If music notation is required, the performance structure is translated accordingly in a similar manner by another set of appropriate functions. Finally, Haskore and MDL can be used to create an algebra of music. This allows one to prove various axioms which form the algebraic semantics at the base of the domain-specific abstractions used in these DSELs [34].

3.7.8 Robotics

The domain of robotics is based on the notions of behaviours which occur over spans of time such as velocity and reading from sensors, and discrete events such as when a robot bumps into a wall or detects that a light source has been switched on. Thus, by using reactive programming's notions of events and behaviours it is possible to write DSELs which focus on this domain. Two languages based on this approach are IRL [34] (for Imperative Robot Language) by Hudak and Frob [53, 54] (for Functional ROBotics) by Elliott et al. both of which are embedded in Haskell. Both of these languages follow the normal approach taken in most DSELs of separating the concerns of what and how. In IRL the main primitives of the language are based on what the robot can do, that is, actions such as moving forward and turning into a particular direction. In Frob the approach taken is similar but more elaborate by the fact that the robot is controlled by well-defined and understood control equations and primitives which combine these equations together into one system. These equations are however still tied to what the system targets to achieve rather than how it does so. The case of how the robot actually achieves this, is a separate concern which is related to implementation and so can be abstracted away from when one writes programs in both of the DSELs.

In IRL the robot's implementation is rendered by a simulation which gives meaning to a program in the DSEL by interpreting it. The interpretation uses graphics to represent the robot moving around a bounded flat surface as per instructions in the DSEL program. Since state is a factor here, monads, specifically the state monad, are put to use to monitor the state of the robot as it moves around the bounded world. Frob's implementation is more elaborate as it relates to a real-world implementation rather than a simple simulation. The DSEL code is interpreted and calls C++ code imported into Haskell via an interface library. This C++ code is understood by an actual robot system which acts accordingly.

Thus, once more, the key factor of abstraction allows the user of IRL and Frob to write highly understandable, simple-to-use, modular code which is not riddled with implementation intricacies of how the robot itself works such as obtaining sensor input or passing information to the effectors but rather of what to do and when. This makes the code shorter, more elegant and allows it to be applied to various implementations of robots without it being bound to a particular robot-model or any other form of implementation. Moreover, Haskell's type system and ad hoc polymorphism allows errors due to typing to be detected at compile-time, rather than during run-time. All this is inductive to producing code in a prototyping manner in a domain where normally time-taking, low-level programming is used. Furthermore, as with other DSELs, Frob can be used to formally reason about robots and allows for program transformation [53].

The following is an example of a program written in IRL adapted from [34]:

```
drawSquare
  = do penDown
       move
       turnRight
       move
```

This small program illustrates some of capabilities of the robot in IRL such as starting drawing mode (`penDown`), moving (`move`) and turning (`turnRight`). Here the robot moves in a path while drawing with its pen. As an example of Frob, we show an example adapted from Elliott et al. [54]:

```
goAhead r timeMax =
  forward 30 'untilB'
    (predicate (time > timeMax) .|.
     predicate (frontSonarB r < 20)) -=> stop
```

This time, the robot will move in a straight path forward at the rate of thirty centimetres per second (`forward`). By means of the construct `untilB` of reactive programming we are able to allow this behaviour to go on until an event occurs which interrupts the former somehow. The `predicate` construct generates an event when a condition is satisfied such as when the maximum time limit supplied is surpassed, or the front sensor detects that there is an object within twenty centimetres of the object. Using the combinator function `.|.` these two events are set to work in parallel using interleaving. If either of these events becomes true the robot is set to a new behaviour by means of the operator `-=>`, in this case stopping the robot at its current position (`stop`). As can be noticed by this example, this DSEL, similarly to IRL, is domain-focused on robots and uses only notions related to that particular domain only. This makes it more accessible and easier to work with when programming robots as it is high-level enough for anyone to use it without being an expert in the field. Despite this, it is not restrictive and is still expressive enough to allow the creation of elaborate descriptions without requiring the need to make use of the low-level code often associated with robot-programming.

3.7.9 Testing

QuickCheck [8, 9] is a tool developed by Claessen and Hughes related to the domain of testing which helps programmers writing code in Haskell to check that it works as intended by carrying out a number of random tests. It is divided into two sub-domains, one related to property specification and one concerning test data generation. The approach taken is that the programmer who wants to test a particular function, must first write a property in the form of another Haskell function which the former function must satisfy. The specification sub-component of the tool allows such properties to be written by providing a number of domain-related types such as `Property` and a number of domain-related operators such as extensional equality and implication in the form of combinators. Other constructs are provided to cover other functionality related to properties such as the classification of results obtained and the collection of such results in a histogram fashion for easier evaluation. Once the required property is specified, the actual testing is achieved by a call to QuickCheck's main function `quickCheck` which runs a number of random tests through the property and checks how many such tests are satisfied. The number of test cases to be generated can be supplied by the user and if any one of these fail a counter trace is supplied which aids in detecting what went wrong. The test cases themselves can be generated automatically by using a random approach but by means of QuickCheck's test data generation component it is also possible to fine-tune the random data generated such that it is distributed over the data type which the function receives. This can be achieved by means of a Haskell's type-class called `Arbitrary` and a domain-related type `Gen` which represents the generator for a particular type. A number of utility combinator functions are also present which for example instruct the test data generator to pick randomly between the constructors of a type, and also do so with varying degree of frequencies per constructor.

As a tool, QuickCheck threads the fine line between being a fully-fledged DSEL library or a normal domain-specific library. Despite the use of domain-related types such as `Property` and `Gen` there is no data type which acts as a DSEL's syntax on how domain-specific structures such as properties or generators are formed and composed. Combinator functions do exist as part of the library to facilitate the creation of properties and generators however the latter are actually defined by making use of Haskell's own functions and type-classes, respectively and cannot be used to form them on their own as is the case with other DSELs' combinators. In regards to semantics, QuickCheck's main function `quickCheck` is not a clear case of an interpretation function which acts over a domain structure to give it meaning. This function simply acts as a higher-order function which uses another function, the property, to test the required function a number of times. In a way, however, it is still providing a domain-specific interpretation for random testing but instead of receiving a domain-related structure it is making use of a normal Haskell function. Thus since the dividing line between QuickCheck and Haskell itself is not clearly defined, one might think that it is a normal library just for Haskell. However, the combinators supplied by both of QuickCheck's sub-domains can be seen as domain-specific language constructs which abstract away from implementation. In fact, they can and have been implemented for more programming languages since the property specification and test data generation domains captured by QuickCheck are applicable to all languages. By abstracting away from the implementation itself, as this could be any particular program in any language, and focusing on what a programmer needs to automatically test an application, QuickCheck's constructs allow the users to write properties and create generators for the code they want to validate. The only difference QuickCheck has when compared to other DSELs is that despite the generality of the constructs which are not really bound to any language, Haskell's functions are at the core of its domain thus causing it to be closely bound to the language rather than being generic. Thus properties written for programs in Haskell which must be applied to an implementation in another language would have to be translated to the equivalent of a Haskell's function accordingly but the meaning captured by the constructs is retained. If viewed in this manner QuickCheck may be acknowledged as an embedded language despite it not fitting the definition of the latter in all aspects.

Claessen and Hughes [8, 9] provide us with a good example of the use of QuickCheck's specification sub-language when they write the properties which the Haskell function `reverse` must satisfy. This function when supplied with a list, returns said list in reverse order. It should satisfy a number of properties, for example:

```
         reverse [x] = [x]
    reverse (xs++ys) = reverse ys++reverse xs
reverse (reverse xs) = xs
```

which are expressed in QuickCheck, respectively, as follows:

```
prop_RevUnit x = reverse [x] == [x]
prop_RevApp xs ys = reverse (xs++ys) == reverse ys ++ reverse xs
prop_RevRev xs = reverse (reverse xs) == xs
```

As can be noticed the properties are normal Haskell functions. By calling them through `quickCheck` we can run the `reverse` function such that the properties are tested with an arbitrary number of random inputs. In the case we express the second law differently as follows:

```
prop_RevApp xs ys = reverse (xs++ys) == reverse xs ++ reverse ys
```

we obtain a counterexample input which allows us to know where the property of the function fails.

We provide another example of QuickCheck by Claessen and Hughes [8, 9], this time making use of the test data generation sub-language to create generators for user-defined types. In this way we show certain domain-related constructs QuickCheck offers. An example of a user-defined type is the type `Colour` defined by:

```
data Colour = Red | Blue | Green
```

A simple generator for this type which picks from one of the colour constructors at random using a uniform distribution can be described in QuickCheck as follows:

```
instance Arbitrary Colour where
  arbitrary = oneof [return Red, return Blue, return Green]
```

Here we see the use of a Haskell type-class used by QuickCheck called `Arbitrary` which allows us to generate arbitrary elements of a user-defined type such that when the `arbitrary` function is used when a call to `quickCheck` is made it generates data of our user-defined types. The QuickCheck construct `oneof` allows the selection of one of the elements in a list passed to it as a parameter at random by making use of a uniform distribution. In this way we provide a simple data generator for our custom data type which allows the latter to be used with properties.

3.7.10 Business Processes

When developing business solutions, business analysts often create a high-level representation of the solution using a form of business process model. This model is used to communicate with the actual IT personnel in charge of developing the solution itself. The modelling technique used in such cases is in essence a domain-specific diagrammatic language used to communicate between two parties. Micallef and Pace [47] have proposed a DSEL hosted by Haskell called FuncBPML which captures the same domain semantics as one of these diagrammatic languages, namely IBM's WebSphere Business Modeler Advanced v6.0.2 [47]. The advantage gained by this approach is that models can be produced very quickly in just a few lines of code and in a high-level way which is abstract enough such that they can be understood not only by IT personnel but perhaps more importantly by business analysts. This is made possible by the DSEL's ability to focus on the behaviour which the business analysts are able to understand and abstract away from implementation-details. However, despite its high-level nature, the DSEL is able to go down to the implementation level by a process of translation which concretises its code into an implementation. If the mapping is done correctly for any required language of implementation, this allows not only interpretation but also allows its users to use the DSEL to analyse and transform the programs created. Another advantage of using the embedding approach is that models can be implicitly checked for quality assurance using three methods: Haskell's own type checking mechanism, the use of the aforementioned method of phantom types which allow the DSEL to have its own embedded type system inside of Haskell's and a number of specialised functions which make it possible to check models for correctness. A final advantage is the compositionality of the language made possible by means of a combinatorial approach taken which allows the creation of complex transforms using the smaller quality-assured transformations provided by the DSELs.

Apart from phantom types, FuncBPML makes use of various other techniques such as observable sharing, mentioned earlier, which allows the detection of loops and shared objects in business process models and tagging which allows composed processes to be packaged as one block. Another very useful technique is the use of connection patterns which are higher-order functions that allow the composition of functions together in a building block manner thus creating composite functions with one set of inputs and one set of outputs without referencing intermediate values between the composing functions. This technique is discussed later on in more detail in Section 3.8.6. Finally, parameterised objects are used which when supplied with a

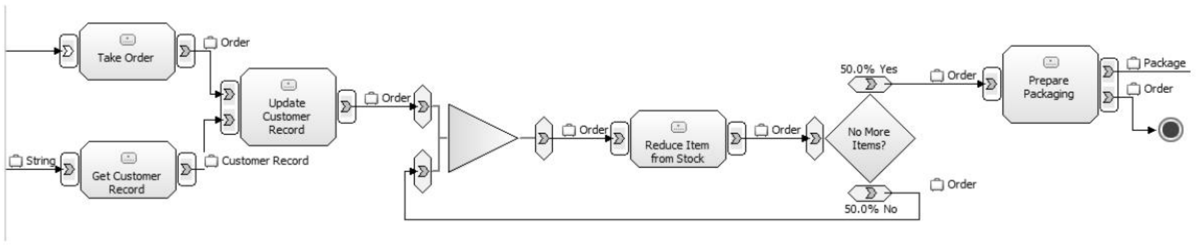


Figure 3.5: An example business process which handles orders constructed using IBM WebSphere Business Modeler Advanced v6.0.2. Reproduced from [47].

number of parameters allow the user to create repeated structures such as processes quickly without the need to define the same items each time. Parameterised objects are explained in more detail in Section 3.8.5.

As an example of the use of FuncBPML to describe business processes, Micallef and Pace [47] produce a diagram in IBM's WebSphere Business Modeler Advanced v6.0.2 which handles the processing of orders and proceed in describing it in FuncBPML. The diagram is shown in Figure 3.5. The process consists of taking a customer's order, updating their record after retrieving it, removing the items in the order from stock one after the other and finally packaging them to be given to the customer. The code to describe this diagram in FuncBPML is as follows:

```
pfOrderHandling = (tTakeOrder -|- tGetCustRec) ->>- tUpdateCustRec
                  ->>- soundCycle tReduceItemFromStock
                          ("No More Items?", (branchProp eNoMoreItems 0.5,
                                              branchProp eMoreItems 0.5))
                  ->>- tPreparePackaging ->>- (id -|- stop)
```

The constructs in the above example that begin with `t` refer to tasks. A task is a basic sub-component of a business process and is shown in Figure 3.5 as a round-edged rectangle. The `-|-` and `->>-` consist of connection patterns which are used to compose tasks either in parallel or in series, respectively. In this way, depending on the connection pattern used, these tasks are done either in unison or one after the other. This example also shows how this DSEL handles two important language notions: branching (`branchProp`) and looping (`soundCycle`), displayed in the diagram by means of a diamond and a triangle, respectively. Using these constructs the diagram in Figure 3.5 can be thus described using the short program above. This shows the versatility and ease-of-access of this language, especially for domain experts who wish to create business process diagrams rapidly and efficiently.

3.7.11 Firewalls

Writing the required number of access-list rules for a firewall is a tedious job which entails a lot of time and attention. Also the way to test the rules is often to put them into use and correct any faults as they arise. Other problems are related to the way the rules themselves are written. For every access-list rule the protocol used must be known in advance. The rules are often repetitive since no scoping is used. Ports which are not predefined by the router have to be entered using their port value instead of easier-to-remember names since there is no way to add them to the router. Moreover, there is no way to group services resulting in the network administrator having to write many similar rules. Finally, there is no way to assign a host name to an IP address, requiring the user to enter the full IP address each time it is used. These factors are problematic since they introduce the risk of errors. One can immediately notice these problems in the sample firewall rules written in a Cisco router's firewall language provided by Santos [58]:

```
access-list 101 permit tcp any any established
```

```
access-list 101 permit tcp any host 150.161.2.1 eq www
access-list 101 permit tcp any host 150.161.2.202 eq www
access-list 101 permit tcp any host 150.161.2.1 eq 143
access-list 101 permit udp any host 150.161.2.1 eq 143
access-list 101 permit tcp any host 150.161.2.202 eq 22
access-list 101 deny ip any any
```

In order to solve these problems, Santos [58] has created a Haskell-hosted DSEL which allows easier programming of access-lists by expanding upon the features available in firewall languages. The embedded language consists of a number of constructor functions such as `host/hosts`, `service` and `access-list` which act as language constructs and create network-based domain-specific objects mentioned earlier such as hosts, services and access-lists. As combinators, this embedded language makes use of Haskell's own lists to serve as a form of parallel combinator. The equivalent code for the above example in this language is as follows:

```
recife = host "150.161.2.1"
caruaru = host "150.161.2.202"
webservers = hosts [recife, caruaru]
ssh = service [tcp] [22] "ssh"
imap = service [tcp,udp] [143] "imap"
access-list 101
  = [permit established Connections From anyhost To anyhost,
     permit www Connections From anyhost To webservers,
     permit imap Connections From anyhost To recife,
     permit ssh Connections From anyhost To caruaru,
     deny ip Connections From anyhost To anyhost]
```

We gain a series of advantages here: hosts can be named and grouped, rules can be written which scope over an entire group, new services can be defined and rules about a particular service can be grouped. This results in compact code which removes a considerable amount of unnecessary repetition and is easier to read, understand, write and maintain, thus reducing the chances of errors creeping in.

By abstracting to the domain itself, rather than being restricted to any router-specific implementation this DSEL improves on the capabilities of any router's firewall language. The implementation-specific details come at a later stage where by means of interpretative functions the programs written in this embedded language are translated into code which is router specific. If a router is changed, the code written in the embedded language still applies, granted the function which translates to the new router's language exists or is created. Due to this and the fact that some usual DSEL features are missing, such as true combinators (rather than just Haskell's own lists) and the ability to encode repetitions, this DSEL acts as a form of macro language which expands firewall-independent descriptions into firewall-specific ones. Despite being a simple macro language, the use of interpretative functions coupled with Haskell's non-restrictive syntax gives the impression to the user that he is programming in a stand-alone DSL, while in fact they are writing a Haskell module and calling a Haskell function to interpret and translated it accordingly. It is also possible to interpret the code written in the embedded language in other ways such as extracting information from it thus allowing a better measure of the firewall's efficacy.

3.7.12 Hardware Description Languages

Hardware description languages (HDLs) are DSLs such as Verilog and VHDL which are used to describe and work with circuits. Lava [4, 8, 10, 12] is a HDL embedded in Haskell which focuses on the components

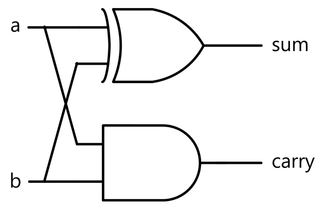


Figure 3.6: A half-adder. It is a circuit which receives two input signals and returns their sum and carry using a XOR gate (upper) and an AND gate (lower), respectively. Reproduced from [12].

in the circuit themselves, represented as functions. This allows higher-order functions to act as combinators which receive entire circuits as parameters and combine them together to form larger circuits. Such functions are called connection patterns since they are seen as functions which connect circuits together using certain known patterns such as serially or in parallel [4, 8]. Haskell's polymorphism allows a signal in Lava given as input to, or received as output from a circuit to be of any type depending on the required level of abstraction [8]. Combined with Haskell's type-classes, this allows a number of hierarchies the circuits can be described in [4, 8]. Type-classes used with a Haskell data structure also allow Lava to support overloaded functions which accept varying input and output structures [8, 12]. Laziness allows the conceptual creation of infinite circuits [8]. Finally, monads used in Lava allow concealed information plumbing [4, 8]. These features together allow Lava circuit descriptions to be interpreted in various ways of which some of the most important are simulation, symbolic interpretation, verification and translation into VHDL [4, 8, 12]. This approach not only removes the separation of concerns between structural and behavioural which plagued DSL-based HDLs [51], but also allows the creation of circuit generators, which supplied with a number of parameters are able to create the required structures in a compact and efficient manner. All of this is made possible by virtue of the embedding approach, which allows Lava to have access to Haskell's features.

Claessen [8, 11] created the DSEL technique of observable sharing for Lava in order to replace a previous method involving monads to solve the issue of detecting loops and shared components in circuits. As we shall see later on in more detail in Section 3.8.3, observable sharing is a non-conservative extension of Haskell which breaks referential transparency but which provides a more elegant solution than monads. Claessen and Pace [8, 10] have also hosted languages within Lava itself such that a framework is formed using this language as base. They have embedded an imperative language called Flash in Lava. Programs in Flash are compiled into Lava circuits using an embedded compiler inside Haskell itself. We shall discuss this in more detail in Section 3.9.

As an example of the use of Lava we describe a half-adder circuit from the language's tutorial [12]. A half-adder, represented by a function `halfAdd`, receives as input two signals and computes their sum and their carry which are returned as outputs, as shown in Figure 3.6. To describe such a circuit in Lava we use the following code:

```
halfAdd (a, b) = (sum, carry)
  where
    sum   = xor2 (a, b)
    carry = and2 (a, b)
```

The constructor functions which act as Lava constructs, `xor2` and `and2` construct a XOR gate and an AND gate and return a result depending on the values of the two input parameters supplied. These results are used as the outputs of the half-adder circuit itself. Note that Lava's modularity allows us to use this definition in other circuits such as a full-adder in a way which is similar to how circuits are often composed.

Wired [2] is a HDL embedded in Haskell which also deals with hardware description. It allows the programmer to describe circuits and explore designs at a wire-aware level in a comparable way to that of generating circuits at a higher level. As a HDL, this makes Wired different from Lava. Despite the fact that

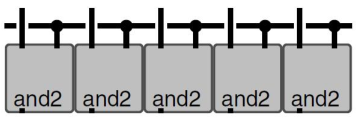


Figure 3.7: Layout of a 5-bit bit-multiplier. It consists of 5 repeated sub-circuits connected next to one another. Each sub-circuit consists of three components: an AND gate, a T-shaped wire junction and a wire pattern which consists of two wires overlapping each other without being connected. Reproduced from [2].

they share the same domain of hardware description, their focus is on different aspects of the latter.

One of the main differences between Lava and Wired, is that in the former the first class objects are the circuits while in the latter the focus is on the wires themselves. The primary reason for this approach is that wires are no longer negligible in circuits which must be implemented in very small scales. Their layout is important to maximise efficiency and performance, and minimise the use of resources. This language allows the users creating programs with it to use interpretative functions which check not only the functional aspects of the circuit, but also analyse non-functional ones such as space, power consumption and heat generated. Since these non-functional properties are concerned with lower levels of abstraction, and are more related to the actual physical implementation of a circuit, Wired usually works at a lower level of abstraction when compared to Lava. It is often the case that one starts writing his design in Lava, and then when the higher levels are complete, Wired is used to capture effects that apply only at a lower level.

Another notable difference between Lava and Wired is that the former is functional while the latter is relational. The reason for this is that in Wired when programs are interpreted for heat-generation or power consumption the direction of flow in the wires is immaterial and so is abstracted away from. In Lava things are different since the flow of current is important and so here the circuits are viewed as functions. Moreover, the use of functions lead to the creation of a number of connection patterns in Lava. Wired still makes use of connection patterns but these are simpler and can be seen as relations (represented as tiles) connecting components together. Combining tiles together forms the relations required and the circuit itself.

In Wired, a circuit is abstracted to a primitive called a description. A combination of two descriptions to form a composite one requires a combinator which describes how they are connected together to form a new one. Combinators such as `*=*` (below) and `*||*` (beside) exist which describe how the descriptions are positioned next to each other. These allow wires and other circuit components to be tiled near each other in a very straightforward manner. Another interesting aspect of Wired is that one can choose to omit fixing certain values such as wire length and then allow an instantiation engine to decide which value is best to give by detecting the current description surrounding said component. This engine achieves this by using fix-point iterations until no more changes occur in the descriptions. This allows Wired to make clever instantiation choices which result in well-positioned wires in a circuit. The technique used here, which we shall discuss in more detail in Section 3.8.7, is known as clever functions. The main advantage of clever functions is that they allow one to pass parameters to the function generating the circuit such that it can attempt to bias the instantiated circuit according to instructions received by the programmer. For example, if physical space is limited, the clever functions can attempt to minimize the maximum area utilised for the circuit. This approach was taken by Axelsson et al. [2] to study parallel prefix circuits and generate a number of candidate circuits by altering the biasing factors. The best candidates were comparable in performance to other well-known versions of such circuits.

We now show how an n-bit bit-multiplier circuit may be described in Wired as an example:

```
bitMult = row and_bitM
  where and_bitM = and2 *=* (cro *||* crT0)
```

An n-bit bit-multiplier consists of a number of repeated 1-bit bit-multipliers placed next to each other. Each of these bit-multipliers is made up of just one AND gate and two differently shaped wire connections: a

T-shaped wire junction and two crossing wires which do not connect. In the above Wired definition the 1-bit bit-multiplier is defined by `and_bitM`. As can be noticed we are using the aforementioned combinators `*=*` and `*||*` to describe how the AND gate and the two different wire connections are positioned relative to each other. Using a connection pattern defined by Wired, `row`, a number of these 1-bit bit-multipliers are placed next to each other in a regular structure which forms a row resulting in the required n-bit bit-multiplier circuit. The 5-bit version of the bit-multiplier is shown in Figure 3.7. The construct `row` is context sensitive: it is able to detect how many copies of its supplied circuit it should create by viewing its surrounding context. In this way we do not need to supply `row` with a predefined parameter for length but rather this value is automatically deduced upon instantiation of the circuit.

HeDLa [51] is a HDL embedded in Haskell which also deals with hardware description. Compared to Lava and Wired, HeDLa is neither functional nor behavioural in nature. Instead, it takes a component-based approach. Instructions in this language allow one to create, tag and compose circuit components made out of basic primitive components such as gates and then use these circuits as sub-components to form other, more complex circuits. Tagging is an important feature in this language as it allows for easier translation into VHDL and Verilog, it allows the user to view the circuits in a building-block manner and also allows the study of non-functional properties in a similar fashion to Wired.

HeDLa, like Lava, uses Haskell's type-classes to allow functions to take different inputs and output structures such as tuples and lists abstracted as a single input and output item, respectively. However, it differs from both Lava and Wired, by the fact that by means of Haskell's strong-typing HeDLa's components take both inputs and outputs of the circuit they represent as inputs. This approach is once again comparable to the structures employed within VHDL and Verilog. As is the case with Lava, HeDLa is also extensible to allow one to create circuits at different levels of abstraction by using different types, other than just Boolean signals. Finally, HeDLa also allows circuit interpretation akin to Lava's such as simulation, translation into VHDL and translation into model-checking code to aid in formal verification. What is especially useful in HeDLa's translation into VHDL is that unlike Lava the modular structure of the circuit is not lost by the translation.

HeDLa solves VHDL's problems concerning structural and behavioural descriptions problem by allowing the creation of new components with both descriptions available for use as required. This allows HeDLa to simulate them side-by-side as is often required to ensure that the structural description follows the behavioural one. By combining HeDLa with QuickCheck a complete set of tools is created which covers a circuit's development life-cycle. It is possible to specify a circuit using HeDLa's behavioural description, implement it using HeDLa's structural description and finally test the description using QuickCheck. Complementing the use of QuickCheck, HeDLa also allows another form of testing in the form of structural and behavioural observers. Finally, with the use of HeDLa's refinement over data it is possible to map the simulations using behavioural descriptions to simulations using structural descriptions thus allowing a multi-level refinement approach for testing the circuit descriptions with input data.

As an example of HeDLa code we show the description of a half-adder:

```
halfAdder = Circuit
  { name = "halfAdder"
  , inputs = ("a", "b")
  , outputs = ("s", "c")
  , description = use xor2 ("a", "b") "s"
                & use and2 ("a", "b") "c"
  }
```

While the same exact circuits are being created here as in Lava's example, HeDLa's language style is completely different from that of Lava or Wired. Each of the circuits is now described using a Haskell record. This record is constructed using a data constructor which has a tag related to the DSEL's domain, called `Circuit`. This tag thus acts as a keyword of the language. Each record defined is referenced by a function name. The former contains various information about the circuit such as its name, its inputs, its outputs and its description. Unlike Lava, the input and output wires are not parameters to a function or actual objects like in the case of Wired. Instead they consist of just structures such as tuples or lists of strings where each string refers to the name of a particular wire. The description of the circuit itself is achieved using the `use` keyword and `&` operator. The `use` keyword is able to create an actual circuit component such as a basic gate or a newly defined circuit. In the above circuit example we used this keyword to create AND and XOR gates. The `&` operator acts as a combinator.

SharpHDL [52], by Pace and Vella, is another HDL, this time embedded in the object-oriented programming language C#. C#'s features such as polymorphism, encapsulation and inheritance, derived from the object-oriented paradigm, allow a number of advantages. Polymorphism, for example, allows a circuit to be composed out of any other circuit as long as the latter are of the correct structure expected by the former. By means of encapsulation, circuits composing a circuit behave according to their functionality, without the latter knowing their inner-workings. Inheritance allows circuits to be viewed at various levels of specialisation. For example all objects created in SharpHDL are circuits, but these can be specialized into for example adder circuits or Fourier transform circuits. As a simple introductory example to the language, Pace and Vella [52] supply the SharpHDL code required to create a half-adder. This allows us to contrast and compare this DSEL's implementation of the half-adder with that of Lava and HeDLa. The code for the half-adder is as follows:

```
public class HalfAdder: Logic {
   public HalfAdder(Logic parent):base(parent){}
   protected override void DefineInterface(){...}
   public LogicWire gate(LogicWire in0, LogicWire in1, out LogicWire sum){...}

   public void gate_o(LogicWire in0, LogicWire in1,
                      ref LogicWire sum, ref LogicWire carryOut)
   {
       Wire[] input = {in0, in1};
       Wire[] output = {sum, carryOut};
       this.Connect(input, output);

       new And(this).gate_o(in0, in1, ref carryOut);
       new Xor(this).gate_o(in0, in1, ref sum);
   }
}
```

As can be noticed this DSEL's syntax is once more different from that of Lava or HeDLa. Contrasting with Lava's use of simple functions and HeDLa's use of records, in SharpHDL a circuit consists of an object class. Every circuit class has to implement a number of things in order to be integrated into and become usable by the DSEL as a circuit component.

As a language, SharpHDL's main aim is to describe the structure of large circuits. In fact in [52], this language was used to describe two well-known variations of the Fast Fourier Transform (FFT). As we saw in the half-adder example, its primitives are objects modelling notions related to circuits such as wires and logic circuits. However, it can also model abstract notions such as wires carrying complex numbers as is the case with FFT circuits. Finally, it also allows translation into a model checking language called SMV

[45] which allows verification of safety properties to be carried out on the circuits. Pace and Vella used this approach to verify that two forms of FFT are equivalent.

It is worthwhile to note how SharpHDL, due to its embedding in C#, looks and feels more like a domain-specific library rather than fully-fledged DSEL. The reason for this is C#'s syntax which is rather heavy and confining in nature. Compared to Haskell, for example, it constricts somewhat the way programs embedded inside it can be written since its notation keeps appearing in the embedded language. The fact that C# is being used is evident in the way the DSEL code has to be used to create a circuit description. The code written in SharpHDL is heavily influenced by C# syntax and it is evident that this DSEL is embedded in the latter. Haskell's syntax on the other hand allows a better measure of freedom, even if, if its user wishes, a similar notation to C# can be used. In fact, Lava, Wired and HeDLa show no such influences related to their host language. HeDLa's syntax is heavier than Lava's or Wired's simply because it was designed to be this way. Creating a DSEL which shows a high affinity to its host language is both advantageous and problematic. For C# programmers SharpHDL is easier to use and work with since it looks like C# applied to hardware description. For non-C# users or hardware designers who don't know C#, SharpHDL's syntax seems unnecessarily cumbersome and would lead to difficulties when writing the required descriptions. The latter are unable to identify with the language immediately and start using it since C#'s syntax is not suited for the semantics of the domain the DSEL wants to capture but rather adapted. This leads to a steeper learning curve or learning a number of things by rote which must be used each time. It is often for these reasons that functional languages such as Haskell are selected for the creation of DSELs instead of languages in other paradigms.

3.8 Techniques in Domain-Specific Embedded Languages

Ever since the concept of embedding languages was invented, a number of useful techniques have been developed and added to it in order to aid the developer creating the embedded language into producing a more complete and fully-fledged language. Most of the techniques were developed to solve certain issues involved with embedding which had previously restricted the number and quality of languages created. Other techniques do not solve any particular problem but rather have been used to add more functionality to the embedded language thus making it more versatile. The techniques are various and differ in their aims depending on the issue they wish to solve or the feature they wish to add. Since we are primarily making use of Haskell as our host language, in this section we shall present a number of techniques often utilised in this language:

- **Shallow and Deep Embedding** — This is one of the basic techniques since it is related to embedding itself. There are two possible types of embedding, shallow and deep. The difference between the two is tied to the way each of these models the domain being captured by the language.

- **Polymorphic Constructs and Default Value Generation** — Haskell's own type-classes can be used to add polymorphic constructs to the embedded language, allowing a construct to be overloaded when we require its use for different variations of the same case [35]. Type-classes can also be used to generate default values for constructs which might require them, especially in the case of constructs which act as object constructors for objects in the relevant domain.

- **Sharing and Loops** — Since our host language is referentially transparent, shared objects and loops cannot be intrinsically detected by it. We discuss various techniques such as explicit tagging [49], a monadic approach [4, 8, 11] and observable sharing [8, 11] as a means of introducing the side-effect of state to our host language and thus allow the detection of shared objects and loops.

- **Stronger Typing** — The techniques of phantom types [42, 56] and generalised algebraic data types [7, 28, 55] allow the embedded language to be more strongly-typed by embedding a domain-specific type system within the host language's type system itself. This ensures that the users of the language are less liable to the risk of introducing errors in the programs written.

- **Parameterised Objects** — Parameterised objects [8, 10, 60, 61, 62] allow the user to quickly create structures which contain repeated patterns by the use of simple generating functions. By supplying the required parameters the functions generate members of a family of similar structures thus allowing the introduction of a form of iterative construct in embedded languages.
- **Connection Patterns** — Also known as combinators, connection patterns [2, 4, 8, 10, 12, 47, 60, 62] are higher-order functions that receive other functions as input and combine them together to form more complex functions. In this way new functionality is introduced. When used in the context of DSELs connection patterns allow the combination of two different programs into one composite program.
- **Clever Functions** — Clever functions by Sheeran [60, 61, 62] expand the idea of parameterised objects by adding a degree of intelligence. Choices are made during generation which allow the programs generated to attempt to fit certain criteria as much as possible. The programs thus generated reflect the criteria passed as input to the clever function.

In each of the following sections we shall observe certain shortcomings met or useful features needed within embedded languages. We shall then discuss the relevant aforementioned technique which can help solve the problem or add the required feature in more detail and finally add the required enhancements to a hardware description embedded language which will serve as our running example.

3.8.1 Shallow and Deep Embedding

When embedding a language, the first choice to be taken refers to the type of embedding employed itself. There are two types of embedding: shallow and deep. The difference between the two is essentially related to the information they manage to store in their model of the domain at hand. Shallow embedding does not store any information in its model but rather a number of functions of the host language give the embedded language its syntax and semantics. In deep embedding the model makes use of a distinct syntax, created by means of a host language abstract data type which allows an instance of the latter type to represent the embedded language program. Semantics are attached to this syntax by means of a number of interpretative functions which give the structures different meanings.

The lack of an underlying syntax structure in shallow embedding causes this form of embedding to consist primarily of an evaluation strategy since the implementation of the language itself is an evaluation in terms of the host language. Moreover, since there is no information in the underlying model, shallow embedding is able to give the user only the final result of the program when the latter is interpreted. Any intermediate steps used to obtain the result are not observable. This also means that analysis of the model cannot be achieved by examining the syntax structure. Any attempts of program optimisation by the use of optimisation rules which alter the underlying structure so as to make a program more efficient but which do not change its meaning are not possible. Due to this, shallow embedding is often faster and easier to achieve but makes the language less useful as a good deal of valuable information is lost.

Embedded languages that employ a deeper form of embedding are able to use their syntax structure in various ways to obtain various meanings as required. Functions in the host language can receive these structures as input and provide us with various interpretations depending on the required semantics to be attributed to the syntax. Since interpretations are done on an underlying structure we can see the steps undertaken to achieve the final result. We can thus obtain a full trace if this is required. Also, analysis on the structure can be carried out and the program can be optimised by means of rules which can be used to make the program more efficient. It becomes trivial for example, using this form of embedding, to

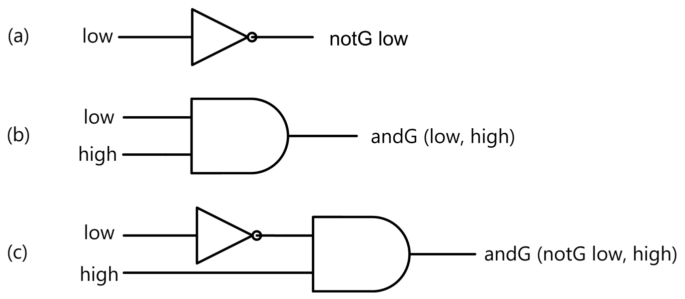


Figure 3.8: Three circuits which can be easily described using our DSEL which has been shallowly embedded in Haskell: (a) `notG low`, (b) `andG (low, high)`, and (c) `andG (notG low, high)`.

collapse away two NOT gates following one another in a structure or program representing the circuit if this is required. As we shall see shortly, compared to shallow embedding, initially deep embedding requires more work to achieve. However, the embedding formed outweighs the effort required since the embedded language describes the domain in an unambiguous way which can be interpreted in various ways as required using the same structure every time.

We shall compare and contrast these two forms of embedding by implementing a very simplified version of a HDL using both shallow and deep embedding. The embedded languages created are based on the notion of a wire. A wire itself can be either at a low voltage or at a high voltage. If using shallow embedding, these two notions are directly mapped to the equivalent Boolean-valued constant functions in Haskell. Thus:

```
low :: Bool
low = False

high :: Bool
high = True
```

The two functions `low` and `high` become syntax primitives in our shallow embedded language whose semantics are provided by Haskell's Boolean type which acts as an equivalent to low-valued or high-valued wires, respectively. We would also like to add two types of gates to our language which will allow us to construct our circuits. These are the NOT gate and the AND gate. Again, these can be mapped to their equivalents in Haskell, this time the Haskell function `not` which represents the Boolean negation operator and the Haskell operator `&&` which represents the Boolean conjunction operator.

```
notG :: Bool -> Bool
notG wire = not wire

andG :: (Bool, Bool) -> Bool
andG (wire1, wire2) = wire1 && wire2
```

The four constructs thus created: `low`, `high`, `notG` and `andG`, constitute our simple DSEL and can be used to write programs in this embedded language. As example programs consider the circuits shown in Figure 3.8. In Figure 3.8 (a) we have a simple NOT gate with a low input. This is expressed using our DSEL simply as `notG low`. Figure 3.8 (b) shows an AND gate with low-valued and high-valued input wires, written as `andG (low, high)`. Figure 3.8 (c) combines the two basic gates together receiving low-valued and high-valued input wires, described using the program `andG (notG low, high)`.

As a slightly more complex example, let us say we want to introduce an OR gate to our language without adding it directly to the embedded language. We can express the OR gate by using AND gates and NOT gates by De Morgan's Law as shown in Figure 3.9. This can be expressed as:

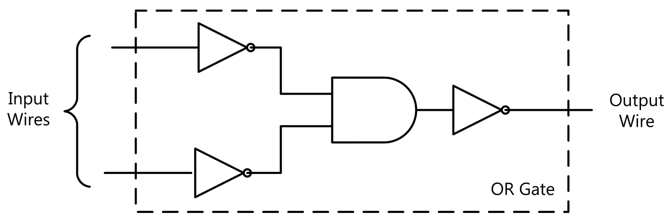


Figure 3.9: An OR Gate described in terms of an AND Gate and three NOT Gates by De Morgan's Law.

```
orG :: (Bool, Bool) -> Bool
orG (wire1, wire2) = notG (andG (notG wire1, notG wire2))
```

Here it is important to notice the observations made earlier about shallow embedding. The language is very easy to implement. However, it is tied to one particular interpretation only, the simulation of the circuit in terms of the Boolean data type in Haskell and the Boolean operators/functions for conjunction and negation also implemented in Haskell. The coupling is in fact too tight to allow different interpretations to be affected. If for example we wish to add another interpretation which counts the number of gates within the circuit we would have to re-implement the four constructs: `low`, `high`, `notG` and `andG` in order for the latter to make sense for such an interpretation. This is achieved as follows:

```
lowC = 0
highC = 0
notGC wire = 1 + wire
andGC (wire1, wire2) = 1 + wire1 + wire2
```

Moreover, all of the programs have to be re-written each time to reflect the new language. Finally, it is also worthwhile to notice the fact that we cannot examine how the answer of any particular interpretation was derived since we are only in fact calling functions with different arity combined together such that the final answer is obtained. By comparison, if we use deep embedding, using the aforementioned structure modelling the circuit, we obtain a way of viewing the interpretation in a step-wise manner which allows us to observe how the final result was derived.

As an example of deep embedding we shall re-implement the former HDL DSEL using this form of embedding. This will also form our running example which shall be used in the following sections to demonstrate the use of further techniques often utilised in DSELs. Since the basic notion around which our DSEL is built upon is that of a wire, in order to use deep embedding this notion is represented in Haskell as a normal data type as follows:

```
data Wire = High
          | Low
          | Not Wire
          | And Wire Wire
          | Input String
```

Each of the value constructors create different types of wires: high-valued wires, low-valued wires, NOT gates which receive one input, AND gates which receive two inputs and a final constructor which allows us to label the inputs of circuits by means of strings. The reason for this last constructor is a temporary solution to a problem which we face with interpretations, as we shall see shortly. This data type forms the core of our language and will model the circuits that can be created with our simple DSEL. As a simple

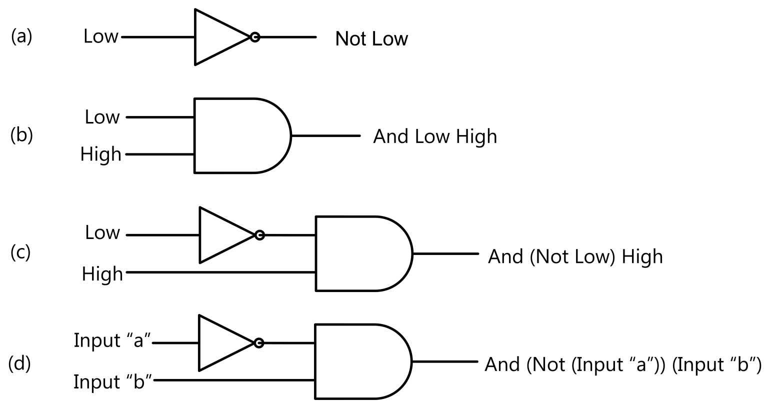


Figure 3.10: Four circuits which can be easily described using the `Wire` data type in our deeply embedded DSEL: (a) `Not Low`, (b) `And Low High`, (c) `And (Not Low) High`, and (d) `And (Not (Input "a")) (Input "b")`.

examples consider the circuits show in Figure 3.10, most of which we have met earlier in Figure 3.8. In Figure 3.10 (a) we have a simple NOT gate with a low-valued input expressed using the above data type as `Not Low`. Figure 3.10 (b) shows an AND gate with low-valued and high-valued inputs expressed as `And Low High`. Figure 3.10 (c) combines the two basic gates together with low-valued and high-valued inputs which is written as `And (Not Low) High`. In the last example, Figure 3.10 (d), the circuit expressed is exactly the same in structure to that of (c), however, this time the inputs are not fixed-valued wires but rather labelled ones. This is described as `And (Not (Input "a")) (Input "b")`. As can be deduced the two input wires are labelled by the strings "a" and "b" accordingly.

As a more complex example consider once more the definition of an OR gate circuit using De Morgan's Law:

```
orG :: (Wire, Wire) -> Wire
orG (wire1, wire2) = Not (And (Not wire1) (Not wire2))
```

As we highlighted earlier, in order to attach semantics to our constructs, interpretations are required which derive information from the underlying structure representing our circuit. Our simple DSEL library provides two interpretations which can be run upon circuits created using the DSEL itself. These are `countGates` and `simulate`. They have the following type signatures:

```
countGates :: Wire -> Int
simulate :: [(String, Bool)] -> Wire -> Maybe Bool
```

The first interpretation traverses an underlying circuit structure and returns the total number of gates found as output while the second interpretation traverses the same underlying structure and returns the result of the simulation. Both of the interpretative functions are implemented by using pattern matching and simple recursion which allow traversal of the circuit structure. Thus they are implemented as follows:

```
countGates :: Wire -> Int
countGates Low          = 0
countGates High         = 0
countGates (Input s)    = 0
countGates (Not w)      = 1 + countGates w
countGates (And w1 w2) = 1 + countGates w1 + countGates w2

simulate :: [(String, Bool)] -> Wire -> Maybe Bool
simulate _ Low = Just False
simulate _ High = Just True
simulate xs (Input name) = lookup name xs
simulate xs (Not w) =
  case (simulate xs w) of
    Nothing  -> Nothing
    Just ans -> Just (not ans)
simulate xs (And w1 w2) =
  case (simulate xs w1, simulate xs w2) of
    (Nothing, _      )     -> Nothing
    (_      , Nothing)     -> Nothing
    (Just ans1, Just ans2) -> Just (ans1 && ans2)
```

As can be noticed, two inputs are required to simulate a circuit: an association list of strings and corresponding Boolean values, used to lookup a labelled-input value, and the wire being simulated itself. This is the reason why earlier the `Input` value constructor was introduced into our DSEL. The circuits which a user of our DSEL may create can take an unbounded number of inputs which inputs can be also be arranged in various structures such as nested tuples or lists. The same reasoning applies to the outputs as these can also be structured in a number of ways. Thus, in order to simulate the gates of such a variety of circuits, we would have to provide a variation of each interpretative function for every possible input and output type. This is not something we wish since catering for all variations of input and output types is almost impossible. Using the `Input` value constructor we shall overcome this limitation for the time being allowing us to perform interpretations given any input structure to our circuits. We will also assume that our circuits are restricted to only one output. This is a minor restriction since it is possible to interpret circuits of any output type by simulating each of the outputs wires separately. As we shall see later on in Section 3.8.2, overloading can be used as a more elegant solution to this problem.

As an example of the use of interpretation we shall use the OR circuit we discussed earlier. If we attempt to count the number of gates for this circuit, the result returned should be a total of four gates. In the example below we pass two labelled wires, labelled with the strings “x” and “y” as inputs to the OR gate circuit and then count its gates by passing it to the `countGates` interpretative function as follows:

```
> countGates (orG (Input "x", Input "y"))
4
```

As expected the result received is 4 gates. Similarly, we can also simulate this circuit by supplying it with two inputs labelled “x” and “y” and passing it to the `simulate` interpretative function. The Boolean logic value of the inputs “x” and “y” are stored in the association list which is also supplied to the `simulate` function. If “x” is associated with true and “y” is associated with false, the answer we would expect is true, given that an OR gate’s output is always true if any of the inputs is true. Thus:

```
> simulate [("x", True), ("y", False)] (orG (Input "x", Input "y"))
True
```

In this section we have seen how a simple DSEL can be implemented in Haskell using shallow and deep embedding. Compared to a full-blown DSL this is relatively easier and can be carried out in less time. Even if we employ deep embedding for our DSEL, language interpretation is straightforward and can be easily achieved by means of a function written in Haskell which interprets the underlying structure representing our domain. Compared to shallow embedding it requires a bit more of initial work to obtain the type which accurately models the domain. However, once this is achieved any amount of interpretations can be made available since the type abstracts from any implementation details related to the interpretations. Due to this, various languages are often implemented using this form of embedding rather than the former one. In fact, most of the languages described earlier employ deep embedding.

3.8.2 Polymorphic Constructs and Default Value Generation

Haskell's type-class are invaluable tools for enhancing embedding languages. Two techniques which this language feature provides are polymorphic constructs and the generation of default values, both of which increase the language's accessibility and versatility.

Polymorphic Constructs One of the easiest techniques employed to improve a DSEL is to make a number of its constructs polymorphic, or overloaded over a number of types. Polymorphic constructs are thus basically Haskell functions which by means of type-classes are overloaded to work with a particular class of types and subsequently restricted to be used which such types only. By allowing a family of types to be used with a particular function we allow it to be used in more than one case. This approach also makes the polymorphic function feel more like a language construct enabling a better look-and-feel for our DSEL.

In the former version of `simulate` and `countGates` functions a temporary workaround was used which required us to inflate our language with an extra value constructor and also required the users of the DSEL to manually pass an association list of values. This was cumbersome and quite unsafe as nothing could prevent the user of the constructs from making mistakes such as omitting an entry for a label. Moreover, we had no apparent overloading over the output types and resorted to interpreting each output wire of a circuit separately as if the outputs come from totally different circuits. By adding polymorphic constructs we were able to overload the gate-counting and simulation constructs allowing the latter to cater for circuits which take different kinds of input types and return different kind of output types. In this way, we were able to simulate or count the gates of circuits which receive or return, for example, lists which have no upper-bound size or nested tuples with a fair amount of nesting, as inputs and outputs. Finally, polymorphic constructs allow for a clearer and easier-to-use syntax which resembles that of a fully-fledge language rather than a group of Haskell functions. As an example to illustrate the effect that polymorphic constructs have on a DSEL, compare the previous calling convention to the `simulate` function on a number of example circuits: `neg` (NOT gate), `and2` (AND gate) and `mux` (multiplexer circuit):

```
simulate [("x", True)] (neg (Input "x"))
simulate [("x", True), ("y", False)] (and2 (Input "x", Input "y"))
simulate [("s", True), ("a", False), ("b", True)]
                                          (mux (Input "s", (Input "a", Input "b")))
```

to one using polymorphism:

```
simulate neg True
simulate and2 (True, False)
simulate mux (True, (False, True))
```

In order to introduce polymorphism to our language constructs we must first add an auxiliary data type to our language implementation which we call `Structure`. A structure can represent either a simple wire or a structure which consists of a list of structures. By this first, basic type of structure and the second, recursive type definition we allow nested wire structures of varying depth to be created. We shall use such structures to model the input and output wires of our circuits which may consist of nested tuples and lists of wires. The `Structure` data type is defined as follows:

```
data Structure
```

```
  = Simple Wire
  | Structure [Structure]
```

By itself, the `Structure` is not enough to allow us to create polymorphic constructs. We also require a type-class which we call `Structurable`. This type-class will be used to convert from any type to a structure and vice versa by the use of its two constituent functions `toStruct` and `fromStruct`, respectively. If we create an instance of this type-class for a particular type, the latter becomes part of this class and we can create a structure of that type. The `Structurable` type-class is defined as follows:

```
class Structurable a where
  toStruct :: a -> Structure
  fromStruct :: Structure -> a
```

The functions of our language which we wish to make polymorphic over input and output structures must be given this type-class's context. In doing so we restrict them to a particular class of types which implement the required functions mentioned above only. These functions are required by the interpretation functions such as `simulate` and `countGates` to adequately manipulate the input and output structures.

Since our aim is to make the interpretative functions polymorphic over structures of wires, the first instance which implements this type-class is that for the `Wire` data type. A structurable wire is defined as follows:

```
instance Structurable Wire where
  toStruct wire = Simple wire
  fromStruct (Simple wire) = wire
```

In this way we have a means of converting between an object of type `Wire` to one of type `Structure` and vice versa. However, the inputs and output of circuits may consist of more than one wire arranged in tuples which themselves can be nested by other tuples of wires. By creating an instance of the `Structurable` type-class for such tuples, the ad hoc polymorphism supplied by this class allows us to provide a way to translate such tuples into and from their corresponding wire structures in terms of the `Structure` data type. For example, the instance of the `Structurable` type-class for a tuple with two components is written as follows:

```
instance (Structurable a, Structurable b) => Structurable (a,b) where
  toStruct (a,b) = Structure [toStruct a, toStruct b]
  fromStruct (Structure [a, b]) = (fromStruct a, fromStruct b)
```

Similarly for tuples with three or more components.

The next step is to create the overloaded versions of our interpretative constructs by re-defining their respective functions in terms of the above functions allowing them to become polymorphic. The `simulate` construct attains the following type signature and definition (we shall discuss `countGates` later on):

```
simulate :: (Structurable a, Structurable b) => (a -> b) -> a -> b
simulate circ wires = (fromStruct.simulateStructure.toStruct.circ) wires
```

The first parameter of the function is the circuit itself abstracted to just a function with two polymorphic type variables `a` and `b` where the former is for the input wire(s) while the latter is for the output wire(s). As long as the types selected for these type variables, such as lists or nested tuples, are instances of the type-class `Structurable` they can be used with this function. The second parameter of the simulation function consists of the input wires over which we wish the circuit to be simulated, also passed as input to the construct. These will be passed automatically to the circuit itself without the need for user intervention.

The definition of `simulate` shows how the new function makes use of the `Structure` type-class by its use of the `toStruct` and `fromStruct` functions. The function composed converts the circuit with compatible input signals into a structure of output wires, which structure of wires is simulated by the function `simulateStructure` and then returned back in the output form expressed by the circuit definition by converting the structure by means of `toStruct`. The function `simulateStructure` is able to handle recursively any structure of wires supplied. It is defined as follows:

```
simulateStructure :: Structure -> Structure
simulateStructure (Simple w) = Simple (simulateWire w)
simulateStructure (Structure xs) = Structure (map simulateStructure xs)
```

A structure which consists of a simple wire is simulated by simulating the wire itself using the `simulateWire` function and returning the simulated wire in a structure. The `simulateWire` is a function which interprets a `Wire` by giving it the semantics of simulation. It is recursively defined as follows:

```
simulateWire :: Wire -> Wire
simulateWire Low = Low
simulateWire High = High
simulateWire (Not wire) = answer
  where simWire = simulateWire wire
        answer =
          case simWire of
            Low  -> High
            High -> Low
simulateWire (And wire1 wire2) = answer
  where simWire1 = simulateWire wire1
        simWire2 = simulateWire wire2
        answer =
          case (simWire1, simWire2) of
            (Low, _)     -> Low
            (_, Low)     -> Low
            (High, High) -> High
```

The second function `simulateStructure`, is the recursive definition which states that to simulate a structure we map the function itself on all of its sub-structures first and return the result of simulation in another structure.

Having now updated `simulate` by rendering it polymorphic we can now compare it once more with its previous version by means of a call on the OR gate circuit. The previous version results in the following call:

```
simulate [("x", True), ("y", False)] (orG (Input "x", Input "y"))
```

By comparison the new `simulate` construct:

```
simulate orG (High, Low)
```

is more streamlined. A difference can be noticed in the syntax brought about by rendering the function polymorphic. The association list has been removed completely, as have the `Input` constructs. Also, we do not need to pass the input to the circuit. Moreover, note how before we had to pass the input values as Booleans instead of actual valued wires. This was done to ensure that the user did not supply other circuits as inputs. Now the inputs and outputs are of a type within our domain, `Wire`. The final result of this is a construct which has a look and feel which is nearer to a language construct rather than a function since it removes what for the DSEL user are a lot of apparently useless syntax rules which were found in the previous version.

Default Value Generation Type-classes have another useful purpose in regards to making an embedded language more natural and versatile. Some constructs in the language might require default input. This is often initialisation input such as a random generator seed or setting a component on the origin of the XY plane. Such input can be manually supplied by the user but this is often unnecessary and not recommended as it makes the syntax unnecessarily heavy for the user and also introduces a higher risk of errors. Ideally such input is generated and supplied automatically behind the scenes unless such functionality is explicitly required by the language users. The generation of such required values can be achieved by means of type-classes.

In our language we have done this when data is required to be generated as input to the circuit structures in order to allow these to be interpreted but is not essential or used by the interpretation itself. A case in point is our `countGates` construct. When using this interpretation, the value of the input which is supplied to the circuit is negligible, as long as it is a non-gate input which would affect the gate count. Therefore we need a way to generate a default value such as a low-valued wire or a high-valued wire which allows us to use this construct without the user having to pass input to the circuit themselves. To achieve this the construct must first be rendered polymorphic in a similar fashion to `simulate`. Thus the `countGates` acquires this signature and definition:

```
countGates :: (Structurable a, Structurable b) => (a -> b) -> Int
countGates circ = (sum.countGatesStructure.toStruct.circ) zero
```

The first and only input the function is now the circuit itself which is polymorphic and can take any type of input and return any type of output. In the definition, notice the use of the `toStruct` function of the `Structurable` type-class to convert the circuit with its inputs into a structure which we can then count the gates of using `countGatesStructure`. The latter function is analogous to `simulateStructure` but has a different interpretation. It also interprets the circuit using a recursive definition on the structure and makes use of a function similar to `simulateWire` called `countGatesWire` which is able to count the gates found in a circuit structure. The output to `countGatesStructure` is a flattened list of gate-counts, one per output wire which are added together to obtain the total number of gates present in the circuit.

There are two things one should note about `countGates`: its lack of use of `fromStruct` and its use of the `zero` construct. We do not use `fromStruct` to convert back to a structure which matches the structure of the output wires of the circuit (similar to `simulate`) since this is not required for this interpretation. The second difference, which for us is the most important in this case, is the `zero` construct which allows us to actually generate a default value. The `zero` construct is actually a polymorphic function which according to the circuit's input requirements creates the necessary amount of low-valued wires, arranged in an input structure which can be accepted by the circuit. To achieve this overloaded effect we add the `zero` function to our `Structurable` type-class as follows:

```
class Structurable a where
  ...
  zero :: a
```

Then we must add this function's implementation to the instances we created earlier. For `Wire`:

```
instance Structurable Wire where
  ...
  zero = low
```

This allows to use `zero` to generate a default value of type `Wire`. We have arbitrary selected our default value for a wire to be that it is set to a low value. In order to make the `zero` function overloaded for use with more kinds of circuits which might expect default inputs structured in different ways, we must add it to the other instances of this type-class. For example, we allow `zero` to work with tuples of two structurable types as follows:

```
instance (Structurable a, Structurable b) => Structurable (a,b) where
  ...
  zero = (zero,zero)
```

and similarly for larger tuples. Notice how when used with a tuple of two structurable types we call `zero` recursively. By means of the overloading facility provided by the type-class system the correct `zero` is called depending on the case at hand. As with the other two functions of this class, this allows `zero` to work with nested tuples as well. Since our wire is structurable this results in `zero` being able to generate low-valued wires arranged in the input structure required by the circuit which receives it as input. In this way default values are generated for use with our circuits when required without the need for the user to pass such values. This is especially useful with our `countGates` interpretation since when we use this construct we can simply call it on the circuit only even if valued-wires are required as input to allow the former work.

As an example of our use of the improved `countGates` we shall once more consider our OR gate circuit. To check the number of gates in the OR gate circuit we call `countGates` upon it as follows:

```
countGates orG
```

Compared to an equivalent call to the previous version:

```
countGates (orG (Low, Low))
```

the `countGates` construct has no unncessary inputs and thus appears much cleaner and understandable to the user of our DSEL.

3.8.3 Sharing and Loops

In this section we will first start by discussing referential transparency in general followed by its implications on our embedded language. We will also consider examples of structures utilised in our embedded language which suffer as a result of referential transparency, namely structures with shared objects and loops. Finally, we will consider a number of potential solutions such as explicit tagging with labels and integers, using monads and finally referential transparency. Each of these solutions solves the problem but might introduce certain disadvantages, some of which we need to remove using a subsequent solution. We compare and contrast these solutions and select the one which we believe is the best option for our example embedded language.

Referential Transparency Referential transparency is a property of mathematical functions which refers to the fact that no matter how many times a function is applied to its arguments and when such application is performed, given an input we always and invariably receive the same output. This makes the output of a function entirely dependent on the input it receives. Mathematical functions are thus considered as pure or side-effect free since nothing external to them apart from the input can affect their result. Thus, due to referential transparency, we can consider functions as stand-alone entities which do not affect the inner-workings of one another. Even if we compose together two or more functions we can reason about each one of them apart from the others. Also, we know that the composed function's output can be reasoned as being dependent only on its input as well. The implications of this latter fact is that any reference to the composing functions is lost in the process of composition: we have one function now and we cannot really tell where one function ends and another begins. As a simple example of this, let us take the function:

$$f(x) = x + 1 \tag{3.1}$$

A mathematician could easily define a second function, $g(x)$ in terms of $f(x)$ as follows:

$$g(x) = f(x) + 3x \tag{3.2}$$

Due to referential transparency they can substitute (3.1) in (3.2) safely knowing that the function's meaning and also its result is not compromised. Thus, $g(x)$ can be rewritten as:

$$g(x) = x + 1 + 3x \tag{3.3}$$

$$g(x) = 4x + 1 \tag{3.4}$$

As can be immediately noted, any references to $f(x)$ are lost in (3.4). There is no way by looking at (3.4) to realize that it was initially partially composed of $f(x)$. This "composition" component of $g(x)$ is lost and hence our reference to $f(x)$.

In pure functional programming languages such as Haskell, functions are analogous to mathematical functions. They are referentially transparent and are thus side-effect free. This means that such functions cannot have direct access to a global state or perform actions which break referential transparency such as IO. In other, impure languages, functions are allowed to process IO or have access to global variables which may return different outputs depending on external factors which affect it. For example, a function which echoes on screen a user's input depends entirely on what the user inputs. We cannot really know or predict its output as is the case of a function which is referentially transparent. Also a function which is dependent on a global state may have such state tempered with by another function which also has access to it. This might cause such a function to affect the other indirectly and we cannot know for sure what result is received. This makes reasoning about functions harder and there are more possibilities of mistakes in the meaning of the programs we write. By comparison, this is not the default behaviour of functions in Haskell. Despite the fact that such behaviour is possible by means of monads which encapsulate such unsafe computation, the default behaviour is that functions are stand-alone and are also directly affected by the input they receive only. As with mathematics, this means that any expression in Haskell can be replaced by its value without changing the meaning of the program. The consequences of this are two-fold. First of all, objects cannot be directly referred to, as they are merely values for Haskell. Secondly, it is not possible to detect loops and shared objects in the structures we might employ and try to evaluate in our embedded language, because we cannot differentiate between a node we have already visited and a node which we are visiting for the first time.

Object Sharing Haskell's referential transparency does not allow us to know when we are re-using the same object: once an object is created our reference to it is lost and we cannot refer to it again. We need to be able to create and maintain a reference to a constructed gate and re-use it later so as to refer to an object which has already been constructed. To achieve this we require a side-effect such as a state. Ideally, our constructors for our gates create the required references, save them in a state and then re-use such references as required. If such an approach is not taken when sharing is present copies of the same items might be created and the structure generated is in the shape of tree with repeated nodes instead of an equivalent graph with unique nodes.

As an example, let us consider the following DSEL program:

```
sharedCirc in0 = out0
  where not0 = Not in0
        out0 = And not0 not0
```

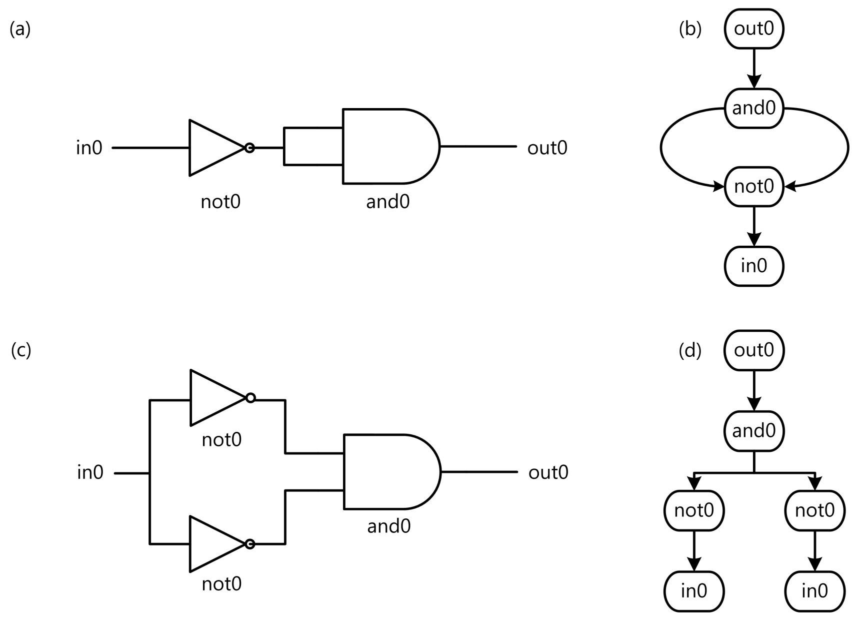


Figure 3.11: The circuit shown in (a) contains a shared NOT gate. Our referentially transparent constructors are unable to detect sharing and thus we obtain the tree shown in (d) corresponding to the circuit shown in (c). If we allow a side-effect such as state to record our references to the objects created and maintained, we can detect sharing and create a graph structure as seen in (b) which represents our circuit structure more accurately.

which we intend to use to construct the circuit shown in Figure 3.11 (a). Using our constructors the objects we create should form the structure show in Figure 3.11 (b). However, since the NOT gate is being shared within the circuit and our referentially transparent constructors are not able to detect this, this gate is created, along with its sub-tree, two times which results in the creation of the graph shown in Figure 3.11 (d). The circuit corresponding to this graph is the one shown in Figure 3.11 (c), which is not the one we intended in the first place since it contains three gates instead of two. As we shall see in this section the notion of state can be introduced within our constructors which enables us to create references. These references allow us to detect shared nodes within our structures resulting in the creation of graphs instead of trees when the need arises.

The problem of object sharing reappears when we attempt to interpret the structures created by our deeply embedded language. Once more, due to referential transparency, the interpretative functions are not able to detect the shared objects by themselves. In fact if a naïve function is used to traverse the graph structure the latter is unravelled as if it were a tree due to repeated visiting of shared nodes. Due to this the state holding the references must be used so as to check every time we are going to visit a node in our graph to check whether such a node has already been visited.

Let us consider the sharing example circuit `sharedCirc` introduced earlier once more. Calling `countGates` upon this circuit enables us to show how this referentially transparent interpretation encounters problems with shared objects. If we attempt to count the number of gates involved using the gate-counting function we get a gate count of three. The answer we obtained shows us that we are in fact simulating a circuit with three gates not two. The NOT gate is counted twice, once per output. This problem affects other possible interpretations. In fact had we written an interpretation function which concretises our description into VHDL for physical construction of the actual circuit, we would have ended up with a different circuit than originally intended. As we shall see later on in this section, by adding references to our gates and using them as part of our interpretative functions to record which nodes were visited during the traversal, it will be possible to write a version of `countGates` which is able to detect the NOT gate as an already visited node and so count it once only.

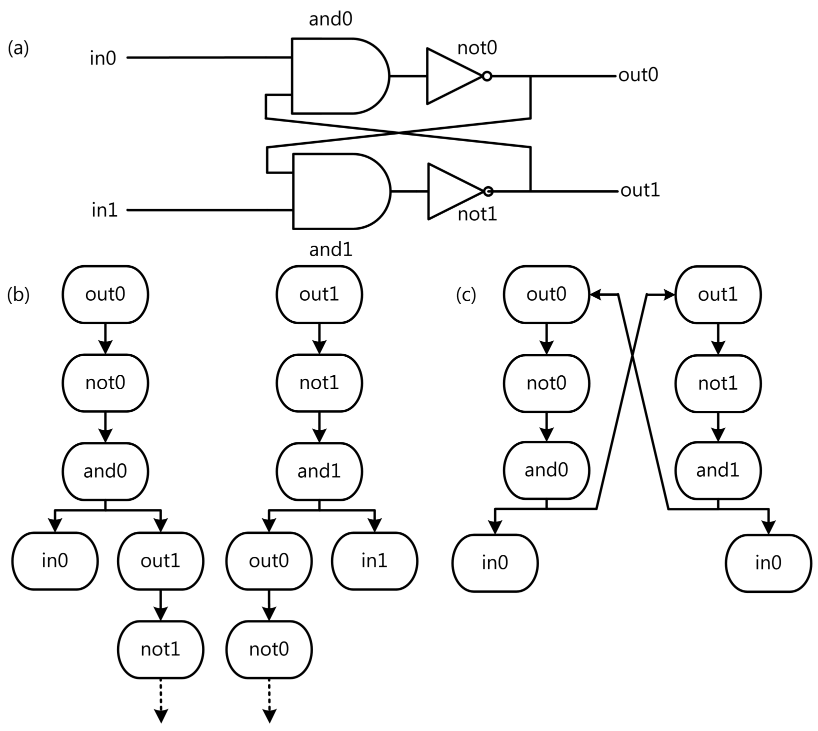


Figure 3.12: The circuit shown in (a) forms a NAND gate latch circuit with contains two loops. Since we cannot detect the start node of such loops due to referential transparency an infinite tree structures are created using our constructors as shown in (b). Adding a state we can store the references to such objects thus allowing us to form the finite graph shown in (c). This results in a more accurate representation of the circuit.

Loops Apart from shared objects, referential transparency causes another detection problem, that of detecting loops. Loops in our structures are paths which start from a node, traverse a part of the structure and end up in the same node. When we attempt to create such loops we cannot detect when we are trying to recreate a loop which has already been constructed before since we do not have a state against which we can check that the loop's items already exist. This results in the repeated creation of the contents of the loops and potentially causes the creation of infinite regular trees which do not correspond to the finite structure that we actually intended to create. Such finite structures with loops can be more accurately represented by cycles of nodes in graphs and so we should use this more accurate representation. To achieve this effect we can use the same method used for observing shared nodes, that is, to add the notion of references to our language. Using such references we can detect when we have already created the first node of a loop such that we stop at that point. In this way we can detect that the loop has been created and so proceed in creating the rest of our structure instead of being held in an infinite loop.

Consider as an example a circuit which contains two loops, shown in Figure 3.12 (a). This circuit produces what is known as a NAND gate latch which is often used in circuits to store data. As can be noticed in the diagram we feedback the output of each of these NAND gates to the other's input, in this way creating two loops. Such a circuit can be described in our language as follows:

```
loopCirc (in0, in1) = (out0, out1)
  where and0 = And in0 out1
        and1 = And in1 out0
        out1 = Not and1
        out0 = Not and0
```

Equational reasoning shows that the above circuit is actually being defined as follows:

```
loopCirc (in0, in1)
  = (out0, out1)
  = (Not and0, Not and1)
  = (Not (And in0 out1), Not (And in1 out0))
  = (Not (And in0 (Not and1)), Not (And in1 (Not and0)))
  = (Not (And in0 (Not (And in1 out0))), Not (And in1 (Not (And in0 out1))))
  = ...
```

Thus, attempting to construct such a circuit with our language results in the creation of an underlying structure which grows to infinity, as shown in Figure 3.12 (b). Here when we reach the NOT gates repeated structures are reconstructed unless we find a way to detect that these node has already been created. If we manage to do so we can halt the reconstructing process from carrying on a second or more times. By introducing a way of storing references to the gates already constructed we can detect that we are attempting to recreate the first gate of a loop of gates whose reference already exists. When this is detected we can make sure that the last item of the loop created points to the first item in the loop thus creating a cycle in the graph. In the case of our example this results in the finite graph shown in Figure 3.12 (c). As can be noticed this graph maps to the circuit itself more accurately than the tree does and is also a finite structure rather than an infinitely growing one. As we shall see in this section, this effect is best achieved by creating the references to the circuit components during the construction of such components themselves. When a new gate is being constructed we check the references to ensure that a gate does not exist before we actually try to create it.

As with graphs with shared nodes, evaluation of a finite graph with loops by means of an interpretative function is also problematic. Such functions have no way of detecting when a node has been visited or not during traversal and so end up in an infinite loop which attempts to re-evaluate the same cycle of the graph over and over again. The solution is once more the use of references. As the traversal function traverses the structure we store a list of visited references which gets updated every time we visit a new node. If a node has already been visited we do not visit it again. For example, if we are at a parent node with two children and we have already visited one of the children which ends up looping back to the parent node, we visit the other, unvisited node instead. In this way we ensure that the entire structure is traversed.

While sharing produced incorrect counts when we attempted to count the gates of such a circuit earlier, circuits with loops are even more problematic since they do not even return a value. The interpretative function is stuck evaluating a cycle until the memory available runs out as a stack overflow. In fact if we call functions such as `countGates` upon our example circuit containing a loop `loopCirc` our circuit with just four gates is interpreted as having an infinite number of gates. Moreover, since the structure is interpreted as infinite, `countGates` cannot finish its evaluation. Clearly, circuits with loops are possible in concretised circuits and so our language should not limit the users to circuits with no loops. Thus, state must be introduced once more to allow us to store the references of visited nodes and detect whether a node has been visited before or not. In this way we can detect loops and stop our traversal algorithm from being stuck in an infinite loop.

How does one introduce the notion of state into our referentially transparent host language Haskell which would allow us to create the reference we need to detect sharing and loops? In the following sections we shall discuss various techniques which make this possible, starting from the most easy to introduce and understand known as explicit tagging. This technique however, relies heavily on the user and has a higher potential for user-based errors. We thus replace it with another technique which involves the use of monads. While this approach is better since it is able to do some of the work for us it still suffers from certain drawbacks which affect user experience. Finally we introduce the last technique, that of observable sharing. This technique is incredibly useful since the user is absolved of most of the work which must be done when using the other techniques but as we shall see it has the non-negligible drawback of requiring a non-conservative extension to the host language.

Explicit Tagging The easiest and least invasive way to introduce state to our referentially transparent language is by threading it through the constructs which form our language such as constructors and interpretative functions. In this way we are able to create references for our objects and re-use them as required to detect shared objects and loops. This technique was first used by O'Donnell [49]. The references for the objects are created by labelling them with strings. In this section we discuss two variations of this technique, one where the users have to supply the strings themselves and manually ensure that there are no name collisions and a slight variation which uses a sequence of generated numbers as part of the labels to guarantee that the labels acting as references are unique.

Explicit tagging with labels Labelling is the simplest form of explicit tagging in which we add references to our objects by introducing labels into our embedded language. Such labels, which are manually supplied by the user while writing programs in the DSEL, are passed to the constructor functions such that they are tied to the objects created by the latter. In this way we give each of the created objects a reference which identifies them. One should notice that objects which are different are required to have different labels. However, it is possible, if required, to label two items with the same label. When this occurs we are assuming that these two objects are really the one and the same thing and not distinct. Thus the label is the discriminating factor which tells us whether two objects are different or not. Due to this the users have to be extra vigilant that they do not make mistakes when supplying the labels to the constructor functions. This is because this technique makes it impossible to know whether the user has in fact given the same label to two different components by mistake or on purpose. Using these labels it is possible to create nodes in structures which have references thus allowing us to detect sharing and loops.

To add labels to our language we need to alter our `Wire` data type by adding a new value constructor as follows:

```
data Wire = ...
          | Name String Wire
```

We shall use this value constructor to create our labelled wires.

As examples using the labelling method we show here how to rewrite the OR gate circuit `orG`, the shared object circuit `sharedCirc` and the looping circuit `loopCirc` mentioned earlier, using labelled wires with unique labels supplied manually:

```
orG (in0, in1) = out0
  where not0 = Name "G1" (Not in0)
        not1 = Name "G2" (Not in1)
        and0 = Name "G3" (And not0 not1)
        out0 = Name "G4" (Not and0)

sharedCirc in0 = out0
  where not0 = Name "G1" (Not in0)
        out0 = Name "G2" (And not0 not0)

loopCirc (in0, in1) = (out0, out1)
  where and0 = Name "G1" (And in0 out1)
        and1 = Name "G2" (And in1 out0)
        out1 = Name "G3" (Not and1)
```

```
      out0 = Name "G4" (Not and0)
```

Notice the repetitive use of the Name construct every time a new gate has to be constructed. This renders this technique quite heavy, syntax-wise for the user of our language. Also, as we mentioned earlier and highlighted by Claessen and Sands [11], despite the fact that this approach works it is inconvenient due to the fact that a programmer may easily give two components the same label by mistake. Moreover, there is nothing which makes sure that two components with the same tag actually refer to the same component. The fact that the programmer explicitly assumes that two components with the same tag are one and the same thing is not in reality a clean solution. These problems can be seen more clearly when we attempt to combine any of the two above circuits. All of these circuits have two gates which share the same label, "G1" and "G2", and so if combined together despite the fact that they are different gates for different circuits we are indicating otherwise here. This is a major problem since more often than not one would like to combine circuits together. O'Donnell [49] discusses several workarounds in order to partially mitigating this problem.

Labelling is also useful for interpretation. As we explained before, naïve functions cannot detect shared objects and loops on their own. To overcome this problem we add the notion of state to our functions. In this case we rewrite the interpretative functions such that they take an initial empty state which is filled with the references of visited nodes. In this way as we traverse the graph modelling our objects we add the references of the nodes which have been visited to the state. If the reference of the node we are about to visit is already in the state, we do not visit it again but attempt to visit its next sibling which has not yet been visited. In this way we ensure coverage of all the paths in the graph even if there are shared objects and loops. As an example of this we shall change the implementation of the countGates function. First we must create an auxiliary function which we shall call countGates'. This function receives as part of its input a list which acts as our state. This list, initially empty, becomes populated during traversal to included all the gates discovered in a circuit. The function is implemented as follows:

```
countGates' :: [String] -> Wire -> [String]
countGates' xs Low = xs
countGates' xs High = xs
countGates' xs (Not w) = countGates' xs w
countGates' xs (And w1 w2) = xs''
  where xs'  = countGates' xs w1
        xs'' = countGates' xs' w2
countGates' xs (Name s w)
  | s `elem` xs = xs
  | otherwise   = countGates' (xs ++ [s]) w
```

The previous function countGates is altered to call countGates' with an empty list and then count the total number of elements in the list of labels returned by the latter function:

```
countGates :: Wire -> Int
countGates w = (length . countGates' []) w
```

If we call the new version of countGates on the OR gate circuit orG which serves as our standard example we obtain the correct result of four gates. The example circuit which contains shared gates, sharedCirc now gives a correct count of two instead of three. Finally, the example circuit which contains a loop loopCirc not only finishes its evaluation but also returns the correct count of four gates.

Explicit tagging with integers Another form of explicit tagging which can be attempted similar to labelling, is by requiring the user of our embedded language to pass an integer value to each and every constructor used, such that for every component constructed, the constructor returns the constructed object and also an updated integer. In this way the component is given the integer passed to the constructing function as its label and also the latter updates the integer so that it is ready for the next component to be initialized. The programmer now only needs to set the initial value and depending on the number of components constructed, the system is able to automatically increment the integer value accordingly and so assign each component a new value.

To add sequential labels to our language we update the `Wire` data type as follows by adding an integer component to all the value constructors:

```
data Wire = Low Int
          | High Int
          | Inv Int Wire
          | And Int Wire Wire
```

Each of the wires are now labelled by their respective integer value. Also, constructor functions are now required in order to make use of a current integer value to create the required labelled wired and to return an updated integer value for use with any possible subsequent gates. Hence the following functions are required which replace the use of the former data constructors `Low`, `High`, `Not` and `And`:

```
low :: Int -> (Wire, Int)
low n = (Low n, n+1)

high :: Int -> (Wire, Int)
high n = (High n, n+1)

notG :: Wire -> Int -> (Wire, Int)
notG w n = (Not n w, n+1)

andG :: (Wire, Wire) -> Int -> (Wire, Int)
andG (w1, w2) n = (And n w1 w2, n+1)
```

For example, using these new language constructs the previous implementation of the OR gate circuit `orG` can be written as follows:

```
orG (in0, in1) n1 = (out0, n5)
  where (not0, n2) = notG in0 n1
        (not1, n3) = notG in1 n2
        (and0, n4) = andG (not0, not1) n3
        (out0, n5) = notG and0 n4
```

Similarly for `sharedCirc` and `loopCirc`. As can be seen the syntax introduced is still cumbersome for use and litters the domain-specific code with unnecessary information related to value-passing. Also, it also relies heavily on the user to ensure that such value-passing is done correctly despite the fact that the values are incremented automatically. Compared to explicit tagging with labels, objects making use of incrementing integers can now be composed in terms of non-basic sub-components, however, the user must still ensure that the integers passed between sub-components follow each other correctly. This approach also suffers from the drawback of us having to assume that two objects are the same because of their reference being the same.

Interpreting structures which use integer labels is virtually the same as interpreting the ones using normal labels. Using the integer values we are able to create references for the visited nodes which are stored in a state, this time a list of integers, as we traverse the graph structure. If, when we are going to visit a node, we detect that has already been visited by checking for the presence of its reference in the state we do not visit it again and skip it instead. This ensures that we visit shared nodes and run through a loop once only. In our case we again make use of an auxiliary function `countGates'` which is implemented exactly as before but which attains the following type signature to indicate the use of a list of integers as state:

```
countGates' :: [Int] -> Wire -> [Int]
```

The implementation and behaviour of the function `countGates` is equivalent to that for explicit tagging using labels.

Monadic Approach The threading around of an integer value as a state can be hidden by the use of a particular type of monad, specifically the state monad. This technique was first introduced in an earlier version of Lava by Bjesse et al. [4]. Using the state monad it is possible to incorporate integer plumbing in the embedded language such that it is done transparently from the user instead of them having to clutter their code with it and do it manually as in the case of explicit tagging with integers. The drawback of this approach is that our code now becomes monadic, requiring a change in syntax and an overhaul in the definition of our language constructs, both constructor functions and interpretative functions, to introduce monadic types. The embedded language's new syntax is by comparison very different to normal Haskell and we are more restricted in how things are written. Haskell's do-notation syntactic support does alleviate the problem somewhat by allowing us to write code which looks more similar to our original approach but the underlying mechanisms change substantially. The introduction of an implicit state causes our code to become seeming imperative rather than functional. Our code is still functional but we have encapsulated the impurity of state into a pure computation by means of a monad thus allowing us to make use of it in a pure language such as Haskell.

In our example DSEL, the first step required to apply the monadic approach is to alter the `Wire` data type in exactly the same way as was done with explicit tagging with integers. An integer component is introduced to all type constructors which will act as a label to the created wires. The introduction of the state monad is done on the constructor functions which wrap around these type constructors. The state monad consists of a Haskell data type `State s a` which encapsulates state-passing by means of a function called `runState` defined as follows:

```
runState :: s -> (a, s)
```

As can be deduced, this function receives the starting state and uses it to return as output the next state and possibly a new object which occurs as a result of the transition. In our case, for use with our constructor functions we need the type of the state to be of type `Int` and the output of a transition to be of our domain-related type, `Wire`, that is `runState :: Int -> (Wire, Int)`. Thus our constructor functions must be of type `State Int Wire` to make use of the state monad. In fact, they obtain the following signatures:

```
low :: State Int Wire
high :: State Int Wire
notG :: Wire -> State Int Wire
andG :: (Wire, Wire) -> State Int Wire
```

The functions themselves must now be written using the do-notation and by making use of monadic functions such as `put`, `get` and `return` which read the state, update the state and return the result of computation, respectively. Thus `low` and `andG` are defined as follows, with `high` and `notG` being defined in a similar manner:

```
low = do n <- get
         put (n+1)
         return (Low n)


andG (w1, w2) = do n <- get
                   put (n+1)
                   return (And n w1 w2)
```

Our example function `orG` is now written as follows:

```
orG (in0,in1) = do not0 <- notG in0
                   not1 <- notG in1
                   and0 <- andG (not0, not1)
                   out0 <- notG and0
                   return out0
```

Similarly for `sharedCirc` and `loopCirc`. There are two things to notice here. First of all, thanks to Haskell's type-inference engine we do not actually need to write the functions' type signatures thus allowing us to hide the use of monads from the user. Unfortunately, we cannot hide the compulsory use of do-notation as this is still more compact and easier to understand than true Haskell monadic code which this notation hides away. Secondly, circuits with loops, such as `loopCirc` require some extra work. The reason for this is related to the fact that in monadic code circular definitions are not allowed. A variable cannot be used before it is first defined as it is required when defining loops in circuits [8]. This is due to the explicit sequencing found when computations are encapsulated in the state monad. Claessen [8, 11] makes use of set of special combinators called loop combinators. These are in fact monadic fixpoint combinators which allow loops to be introduced in circuits.

As with the other previous techniques the interpretative functions must carry around a state which is updated with the labels of visited nodes such that the latter are not revisited again. To render such functions capable of receiving circuits which are written within the state monad the former must be adequately altered. As an example the `countGates` must be redefined as follows:

```
countGates :: State Int Wire -> Int
countGates w = (length . countGates' []) $ evalState w 0
```

where `countGates'` is the same as when defined for explicit tagging with integers and `evalState` is a function which receives a computation in the state monad and an initial state, in our case the label of value zero, and the returns the result of computation while throwing away the final state. The result of the computation consists of the constructed circuit which is being counted for gates. Using the monadic `countGates` on our rewritten example circuits `orG`, `sharedCirc` and `loopCirc` we obtain the same results as we received earlier.

Monads solve our problems of referential transparency and provide us with a way of automatically labelling our circuit objects and of performing value-passing transparently from the user. The problems with using monads are mostly related to the syntax they enforce on the user which might appear restrictive and which makes use of keywords (such as `do` and `return`) which are unrelated to the DSEL's domain. Finally, using monads, we are still assuming that two objects created with the same label are the same item. We will now introduce a final technique which can be used to overcome the problems of referential transparency while still not requiring us to have to come to terms with the disadvantages introduced by the previous solutions.

Observable Sharing The final solution proposed by Claessen and Sands [11], which they named observable sharing, was according to the authors, an idea initially thought of by O'Donnell and implemented in one of his earlier versions of a hardware description language called Hydra [49]. O'Donnell, however, saw this idea as something which negates the use of a functional language in the first place and so immediately discarded it without pursuing it further. The reason for his negative view on observable sharing is based on the fact that this technique breaks referentially transparent nature by the use of a non-conservative extension. Such an extension could affect other aspects of a language, thus effectively creating a completely different language with different semantics, features and properties. Claessen and Sands [11] however question how much this extension breaks Haskell and formally show that by using observable sharing's constrained references and equality test which is similar to pointer equality, almost nothing is lost. In fact, Claessen and Sands argue that observable sharing is the weakest possible non-conservative extension which allows us to detect sharing and looping without altering Haskell's semantics significantly. The weakest extension is desirable because it allows the creators of the embedded language making use of this extension to bypass the need to modify the host language's compiler to enable it [11].

In observable sharing, non-updateable references are employed whereby on construction of an object a reference to it is created behind the scenes. To create this reference an IO function call is used which gives us access to system's memory from within the host language in an unsafe environment, even if this is not normally recommended by the language due to its functional nature. Since we are exposing ourselves to the outside world, side-effects might appear which are outside of the host language's control. To stop this from occurring, once a reference is created in memory, it is never altered in any way. The most which can be done with such references in fact is dereferencing them to obtain the actual object or compare two references to check whether they are two different references to the same object or one same reference to an object. This ensures that despite the fact that an unsafe operation has been performed, since we are not altering the references at any time no unforeseen side-effect is liable to occur.

In order to introduce observable sharing to our example language the type of our language `Wire` and the constructor functions must be altered such that the references are created behind the scenes during object creation itself. Our basic `Wire` type must be changed as follows:

```
type Wire = Ref Wire'

data Wire' = Low
           | High
           | Not Wire
           | And Wire Wire
```

The type `Wire'` is thus effectively hidden away inside the type `Wire` which by means of the reference type constructor `Ref` now allows us to create non-updateable references to wires and gates. The constructors of our objects which act as language constructs must now create the references inside of them and so must be altered as well. The constructors `low` and `andG` are redefined as follows, with `high` and `notG` being defined similarly, respectively:

```
low :: Wire
low = ref Low

andG :: (Wire, Wire) -> Wire
andG (refwire1, refwire2) = ref (And refwire1 refwire2)
```

The `ref` function used in the above constructor functions is able to receive an object of any type and store it in a new mutable reference. In order to ensure that the new references being created are fresh it creates them within an unsafe IO operation. It is this operation which breaks Haskell's referential transparency causing observable sharing to be a non-conservative extension. It is worthwhile to notice how we can differentiate between two objects which are constructed using the same constructor function. If we create two high-valued wires, despite the fact that we are using the same function, since we are calling the reference-creating function `ref`, we are effectively creating different references to the same primitive. These references are distinct from one another since they are created in a non-referentially transparent way.

We can now rewrite our usual OR gate `orG` example using the above constructs as follows:

```
orG (in0, in1) = out0
  where out0 = notG and0
        and0 = andG (not0, not1)
        not0 = notG in0
        not1 = notG in1
```

Similarly for `sharedCirc` and `loopCirc`. As can be seen by means of the OR gate's definition, programming in an embedded language which makes use of observable sharing is equivalent to making use of a DSEL which does not cater for the problems introduced by referential transparency. All the problems introduced by the former techniques have been completely removed. The user no longer needs to add references manually as in the case of explicit tagging using labels or generate them automatically in sequence by passing around the state manually such as when making use of explicit tagging with integers. Also, we are not introducing new language constructs which are unrelated to the domain, such as the use of the naming construct in labelling or the constructs used when employing monads. Moreover, we do not enforce any rigid program writing rules or irrelevant syntax upon the user. The user makes use of Haskell's own light syntax rules which allow the language to have a look-and-feel of a proper domain-specific language. Thus observable sharing allows us to retain this important feature while at the same time allowing us to detect sharing and looping. Finally, it is possible to compose together components without worrying about reference name clashes.

The interpretative functions over programs which make use of references still require us to maintain a state. In order to traverse our program we store the references of the nodes we visit in an accumulating list of references. To find out if a node has already been visited we check whether its reference is already in the list of references and react in the usual manner. As our example we again make use of `countGates` which is altered for use with references by updating its helper function `countGates'` as follows:

```
countGates' :: [Wire] -> Wire -> [Wire]
countGates' xs refWire =
  case (deref refWire) of
    Low -> xs
    High -> xs
    (Not wire1) -> if (addedAlready)
                     then xs
                     else countGates' addCurrent wire1
    (And wire1 wire2) -> if (addedAlready)
                           then xs
                           else countGates' (countGates' addCurrent wire1) wire2
  where addCurrent = unionBy (<=>) xs [refWire]
        addedAlready = length xs == length addCurrent
```

It is important to notice the use of two functions related to references here: `deref` and `<=>`. The first of these dereferences a reference to a wire to obtain the wire itself while the second performs reference equality upon two references to check whether we are comparing the same reference to itself or comparing two different references. In this way the latter operator allows us to check whether a node has already been visited or not. By using a distinct reference which uniquely labels and identifies the object at hand, reference equality allows us to circumvent the disadvantages of the former approaches where two different objects with the same label were considered the same. These two operations on references are considered as safe as neither alters the reference in any way. The user of the DSEL calls the function `countGates'` by making a call to the function `countGates` which is defined in a similar way as defined when discussing explicit tagging using labels and integers. Calling `countGates` on the example functions defined earlier `orG`, `sharedCirc` and `loopCirc`, we obtain the same results as with the former three techniques we discussed.

Comparing explicit tagging and the monadic approach to observable sharing, we see that at the cost of a weak extension to Haskell, observable sharing allows the user to transparently create references to detect any shared objects or loops. While explicit tagging and monads are pure techniques since they do not actually break referential transparency, they are cumbersome to use for a non-experienced Haskell user. They are also both error prone since they introduce a number of ways in which the user can misuse the labelling language constructs provided. Finally, they also come at a cost in how one would describe things naturally and in a domain-specific way in Haskell. The observable sharing method however hides the need to keep references and while impure, it conservatively attempts to preserve referential transparency and at the same time allows the programmer to describe what they need in an elegant manner by enabling our DSEL to retain the look-and-feel of a true DSL.

3.8.4 Stronger Typing

The syntax of an embedded language is created by means of an appropriate data type within the host language. It is this data type which allows the domain-related abstraction to be achieved. In this way structures of the host language's data type become the programs of the DSEL. By virtue of embedding, the syntax of such programs is controlled by the host language's compiler which checks that the DSEL structures follow the type checking rules of the host language. Due to this, it is not possible to create syntactically incorrect embedded language programs as these are immediately captured by the host language's compiler. Languages such as Haskell which are statically-typed detect the embedded language's syntax errors as Haskell type errors upon program compilation and issue a number of error warnings. However, despite Haskell's strict controls on types and hence DSEL syntax, it is still possible to create programs which are incorrect for our embedded language. Such programs are created using structures which follow the embedded language's syntax and thus type check for the host language but are considered ill-typed for the embedded language as they make no sense within the realm of the domain we are trying to capture.

As a simple example, consider that we wish to expand our example language such that it is able to handle also wires with integer values. This would allow our language to be able to handle as primitives higher-level components than basic logic gates such as an integer multiplier and adder. To achieve this we will change our `Wire` data type to allow the use of low-valued and high-valued wires in the form of the Boolean constants `True` and `False` instead of the previous constructors `High` and `Low` respectively by introducing the constructor `ConstB`. In order to introduce integer-valued wires we add another similar constructor to `ConstB` which we call `ConstI`. To make use of these integer-valued wires we add a new circuit component which might form part of our circuits. This will be the binary integer adder represented by the data constructor `Add`. We also wish to include two other components `Equ` for equality of both integer-valued and Boolean-valued wires and `Ifte` (if-then-else) which depending on a Boolean-valued wire selects one of two branches which both together might be wires of either integer or Boolean type. Our `Wire` data type attains the following definition:

```
data Wire = ConstB Bool
          | ConstI Int
          | Not Wire
          | And Wire Wire
          | Add Wire Wire
          | Equ Wire Wire
          | Ifte Wire Wire Wire
```

The above data type results in the following language constructs implemented as Haskell functions as follows:

```
constB :: Bool -> Wire
constB b = ConstB b

constI :: Int -> Wire
constI i = ConstI i

notC :: Wire -> Wire
notC wire = Not wire

andC :: (Wire, Wire) -> Wire
andC (wire1, wire2) = And wire1 wire2

addC :: (Wire, Wire) -> Wire
addC (wire1, wire2) = Add wire1 wire2

equC :: (Wire, Wire) -> Wire
equC (wire1, wire2) = Equ wire1 wire2

ifteC :: (Wire, Wire, Wire) -> Wire
ifteC (wire1, wire2, wire3) = Ifte wire1 wire2 wire3
```

It is our wish that the `Not` and `And` components are tied to Boolean-valued wires only while the `Add` component is tied to integer-valued wires only. The `Equ` component and `Ifte` should be tied to both integer-valued and Boolean-valued wires since they can be defined on both of these types. In equality, if the first input is a wire of a certain type, the second wire should be of the same type. The wire returned by equality as output can be of Boolean type only, since it depends on whether the two values supplied to the component are equal or not. The `Ifte` component's branches and hence its output can be wires which are either all of Boolean type or all of integer type. We also need to restrict this gate's first input to be a wire of Boolean value which allows us to select the one of the two branches.

With the current definition of the `Wire` data type and the supplied constructing functions, however, no such type-related restrictions are imposed on the embedded language. Thus, despite the fact that our language's syntax restricts us to using a type related to our domain, that of a wire, we are not in any way restricting the values such a wire can carry as we would wish in the above cases. In fact any DSEL program written using this data type as its syntax can be considered untyped as far as the embedded language itself is concerned. These problems stem from the lack of an adequate type system for the embedded language which allows ill-typed programs to be written. These programs should not compile but since they are valid Haskell programs they pass the compilation stage. Some examples are:

```
wrong1 = notC (constI 5)               -- (i)
wrong2 = equC (constB True, constI 5)  -- (ii)
```

In the first example `(i)`, the circuit created attempts to find the negation of an integer value. Haskell has no problem accepting this program as a valid one. However, it is visibly incorrect for our embedded language since it does not follow our restriction on the NOT gate. In `(ii)`, we are attempting to equate a Boolean value with an integer value. Comparing two values of different types is not acceptable for us. However since for Haskell `constB True` and `constI 5` are both of type `Wire`, this program type checks and compiles.

An initial method one could attempt to use is to defer the process of type-checking the embedded language programs to the functions which interpret the latter's structures in order to attach semantics to them. As an example of this we make use of the `simulate` interpretation. Since our wires can carry either integers or Booleans we shall make use of the `Either a b` data type so as to be able to output both these types using `Either Int Bool`. The `simulate` function is thus defined as follows:

```
simulate :: Wire -> Either Int Bool
simulate (ConstB b) = Right b
simulate (ConstI i) = Left i
simulate (Not wire) =
  case (simulate wire) of
    Right i -> Right (not i)
simulate (And wire1 wire2) =
  case (simulate wire1, simulate wire2) of
    (Right i, Right j) -> Right (i && j)
simulate (Add wire1 wire2) =
  case (simulate wire1, simulate wire2) of
    (Left i, Left j) -> Left (i + j)
simulate (Equ wire1 wire2) =
  case (simulate wire1, simulate wire2) of
    (Right i, Right j) -> Right (i == j)
    (Left i , Left j)  -> Right (i == j)
    simulate (Ifte wire1 wire2 wire3) =
  case (simulate wire1) of
    (Right b) ->
      case (simulate wire2, simulate wire3) of
        (Right i, Right j) -> if b then Right i else Right j
        (Left i , Left  j) -> if b then Left  i else Left  j
```

As can be noticed the `simulate` function receives as input a wire and returns either a Boolean value or an integer value. Using this function we can now simulate the correctly-typed circuits. But what happens with circuits which have incorrect types? If we take as examples the above mentioned wrong circuits and two similar but correctly-typed circuits as follows:

```
correct1 = notC (constB True)           -- (i)
correct2 = equC (constI 5, constI 5)     -- (ii)
```

and try them out with the `simulate` function we get results which indicate that the calls to `simulate` which consist of correctly-typed circuits are simulated correctly. However, the wrongly-typed circuits are not captured by the case statements and so cause a run-time exception. While a possible solution to the problems of untyped DSELs, using case statements and pattern matching is not often an ideal solution since errors in the circuits are detected whilst the program is running rather than earlier at compile-time. We thus require a way to make our DSEL more strongly-typed and statically-typed in such a way as not to allow descriptions such as the ones above to be written by capturing them at compile-time. Thus, what we are looking for is a way to embed a type system for our DSEL in Haskell. Such an embedded type system would allow us to type check the programs written in the embedded language thus removing the risk of type errors.

Phantom Types Phantom types by Leijen and Meijer [42] is a technique which has been introduced and used specifically to allow the creation of strongly-typed DSELs which are checked for type errors at compile-time. By allowing us to embed domain-related types into Haskell's type system we can implement domain-specific type checking for the embedded language. This allows us to ensure that the programs written in the DSEL are type safe and correct by means of Haskell's own compiler instead of by means of the interpretative functions themselves. In this way compiling of the embedded language programs is no longer done at run-time as shown previously but instead at compile-time.

To achieve this we create an auxiliary parameterised data type which receives as input a polymorphic type. Despite requiring such a type as input, the newly created data type does not make use of the former thereby being completely independent from it. Thus the input type variable is often referred to as a phantom type since it appears on the left of a type definition but completely disappears from the right-hand side of the type declaration. The implications of this, is that by passing the appropriate type as a variable to the new phantom type definition we can create and impose a stricter type checking on our DSEL's own types. The values which are accepted and returned by the functions in our DSEL making use of this new type are properly restricted to the latter's supplied type thus allowing a higher measure of control. In fact, if used with constructor functions, phantom types allow the latter to return only correctly-typed objects. This enables us to write embedded language programs which have more restricted types and are thus type safe according to our language constraints.

We require two changes to our implementation in order to impose stricter type checking: the introduction of a new data type which would allow us to use the phantom type mechanism, and altering the constructor functions to make use of this new data type. Adding a phantom type is done simply by encapsulating our original `Wire` data type as follows:

```
data W a = W Wire
```

As can be noticed our new data type has a type variable `a` which means that it requires a data type to be supplied as input before it can be used. We shall make use of this supplied data type to enforce our type constraints on the `Wire` data type without actually using it. The constructor functions must now be altered such that they use this new type. The functions which create value-carrying wires will now use `W a` type where `a` is the `Int` data type for integer-carrying wires and the `Bool` data type for the Boolean-carrying wires. Thus:

```
constB :: Bool -> W Bool
constB b = W (ConstB b)


constI :: Int -> W Int
constI i = W (ConstI i)
```

The wires created by these functions are now restricted to the required respective types. The same approach is used for the circuit components by restricting the input and output types accordingly:

```
notC :: W Bool -> W Bool
notC (W wire) = W (Not wire)


andC :: (W Bool, W Bool) -> W Bool
andC (W wire1, W wire2) = W (And wire1 wire2)

addC :: (W Int, W Int) -> W Int
addC (W wire1, W wire2) = W (Add wire1 wire2)

equC :: (W a, W a) -> W Bool
equC (W wire1, W wire2) = W (Equ wire1 wire2)

ifteC :: (W Bool, W a, W a) -> W a
ifteC (W wire1, W wire2, W wire3) = W (Ifte wire1 wire2 wire3)
```

In this way any programs written in our embedded language using these become fully-typed as according to our needs. If we attempt to compile the former two wrong examples we now obtain compile-time errors. This shows us how phantom types allow us to successfully introduce type checking to our language thus making it more safe by stopping the user from attempting to create syntactically correct but ill-typed programs at compile-time before they are even run.

Another important advantage of phantom types is that they still allow Haskell's type inference engine to function properly as with the untyped version of the DSEL. This allows the users to decline from manually writing the types of their programs if they wish. In fact, Rhiger [56] states that the type system we embed in Haskell for the embedded language, although not a physical one which we manually implemented but rather simulated by means of phantom types, is type sound and complete. The host language accepts only domain-specific objects which can be represented by its type system, that is objects whose type can be inferred and hence successfully type check in the latter's type system. As an example which allows us to illustrate that the power of Haskell's type inference engine is not lost via phantom types we present the following circuit:

```
doubleEqual ((a, b), (x, y)) = equC (equC (a, b), equC (x, y))
```

This code creates a component that returns a true value if and only if the first tuple of inputs supplied are equal in value and second tuple of inputs supplied are equal in value as well regardless of each of the pair's types. Haskell can correctly infer this and also that the output wire is of Boolean type regardless of the input types. As a further example, consider the following simple circuit:

```
ifThenZeroElse (a, c) = ifteC (a, constI 0, c)
```

In this circuit we have a simple if-then-else component whose first branch is restricted to output the default value zero if the first, Boolean-typed wire supplied has the value true, and any other integer value supplied otherwise. Haskell detects that the true branch of the underlying branching component is going to be a wire carrying an integer value. Hence it infers that the type of the other branch and the output wires have to be restricted to integers only. As can be seen by these two examples, despite the fact that the underlying wires are still essentially untyped (the `Wire` data type is simply encapsulated by the `W` data type), when using phantom types, Haskell is still able to infer the correct types the wires have to be in order to create correctly-typed circuits.

Certain disadvantages exist with phantom types. Cheney and Hinze [7] state that while phantom types allow only well-typed objects to be created, they do not allow their deconstruction without the use of run-time checks and extra tagging. For example, if we attempt to write an interpretative function such as `simulate` using just phantom types we get stuck simply upon attempting to define the interpretations of the simplest data constructors `ConstB` and `ConstI`. Haskell cannot infer the type of the interpretation function since these two have different types. One might think that a possible solution would be to enforce the type

`W a -> a` by including it as a type definition for the `simulate` function. Unfortunately, this type definition is too generic and the compiler cannot use it to infer the value to an integer or Booleans type as according to the case. Thus, in order to work with phantom types `simulate` has to be rewritten as follows:

```
simulate :: W a -> Either Int Bool
simulate (W (ConstB b)) = Right b
simulate (W (ConstI i)) = Left i
simulate (W (Not wire)) =
  case (simulate (W wire)) of
    Right i -> Right (not i)
simulate (W (And wire1 wire2)) = ...
simulate (W (Add wire1 wire2)) =
  case (simulate (W wire1), simulate (W wire2)) of
    (Left i, Left j) -> Left (i + j)
simulate (W (Equ wire1 wire2)) =
  case (simulate (W wire1), simulate (W wire2)) of
    (Right i, Right j) -> Right (i == j)
    (Left i , Left j)  -> Right (i == j)
simulate (W (Ifte wire1 wire2 wire3)) = ...
```

If one compares the two `simulate` functions, one can immediately notice their close similarity. Phantom types still require us to make use of run-time checks in the form of case statements and the use of tags in pattern matching in order to be able to interpret the data structure correctly. So while phantom types are able to introduce stronger, static type-checking, we still need to resort to run-time checks in order to extract the required information from the data structure itself. The reason for this is that the interpretative function has no way of finding information about the restrictions upon the relevant type and therefore we have to resort to the use of manual methods. A possible way to allow functions which make use of such data types access to information about the restrictions the latter have is to encode such information within the data constructors themselves. Phantom types do not allow us to supply such information and so are unsuitable. Unfortunately, at its current version, Haskell has no way to achieve this on its own. Apart from this disadvantage, Cheney and Hinze [7] also give an example where phantom types are not enough to infer the correct types, mainly when inferring types where computations are involved. In order to solve these disadvantages, they thus propose another technique called first-class phantom types which mitigates these problems. They call such types first-class because they view classic phantom types we have described here as second-class due to problems mentioned above. First-class phantom types have been implemented as an extension of Haskell under the name generalised algebraic data types which we shall discuss shortly.

Generalised Algebraic Data Types Generalised algebraic data types (GADTs) [7, 28, 55], also known as recursive data types, first-class phantom types and equality-qualified types are generalised recursive data types which by the use of type annotations, similar to the way types are supplied to phantom types, allow greater control on the types of the objects constructed. GADTs achieve an equivalent result to phantom types by an orthogonal method which restricts the data constructors themselves to being less polymorphic depending on the type supplied to each of the data constructors in the first place. Instead of all of the data constructors returning a single generic data type, each of them may return a more specialized and restricted version of the same type according to their needs. They thus provide us with a way to impose a stricter type checking mechanism on our programs which allows them to be used in a similar way to phantom types in order to embed a type system for an embedded language in Haskell. As we shall see later on they also give us the extra advantage of allowing us to write interpretative functions which do not need the use of run-time checking and extra tagging required by phantom types.

In order to make use of GADTs we update our version of the Wire data type as follows:

```
data Wire a where
  ConstB :: Bool -> Wire Bool
  ConstI :: Int -> Wire Int
  Not    :: Wire Bool -> Wire Bool
  And    :: Wire a -> Wire a -> Wire a
  Add    :: Wire Int -> Wire Int -> Wire Int
  Equ    :: Eq a => Wire a -> Wire a -> Wire Bool
  Ifte   :: Wire Bool -> Wire a -> Wire a -> Wire a
```

In the new Wire data type definition each of the data constructors is specialised and we return the required typed wire depending on the particular circuit component's restrictions. It is important to notice that the first use of the variable a in the above definition (data Wire a where ...) is not the same as in the other instances. In that particular case, the a acts as a kind variable in the sense that the data type requires another type as input in order for it to be used. We now define wrapper functions around the above data constructors to act as our language constructs as follows:

```
constB b                     = ConstB b
constI i                     = ConstI i
notC (wire)                  = Not wire
andC (wire1, wire2)          = And wire1 wire2
addC (wire1, wire2)          = Add wire1 wire2
equC (wire1, wire2)          = Equ wire1 wire2
ifteC (wire1, wire2, wire3) = Ifte wire1 wire2 wire3
```

If we make use of the same example programs wrong1 and wrong2 used for phantom types, the error messages we receive are similar for GADTs. They thus allow us to type check our programs at compile-time in a similar fashion to phantom types.

Yet another aspect GADTs share with phantom types is that the host language's type inference engine is not inhibited in any way. This means that despite introducing another abstract layer of types in the objects created themselves, the type inference engine is still capable of correctly detecting the types of the programs created. Moreover, it does so without requiring the users to enter any type information for their programs manually in order to guide the inference engine. If we define the former phantom type examples doubleEqual and ifThenZeroElse using our version of the language with GADTs and again ask Haskell to infer their types the compiler detects correctly that for doubleEqual, the right type definition is to restrict the first two input to a certain type and the second two inputs to another type. It also detects correctly that whatever the types of the inputs the end results is a wire carrying a Boolean value.

So far we have seen how phantom types and GADTs are comparable to one another in regards to their use in rendering embedded languages more strongly-typed. It is when defining interpretative function upon the data structures representing our programs that GADTs have the upper hand. With phantom types type-safe decomposition required us to make use of a workaround in the form of the auxiliary data type, Either which means that in regards to interpretation these types provide no advantage over standard data types. Fortunately, GADTs do not suffer from the same problems and they allow us to create the simulate function we require for interpretation in a very straightforward manner:

```
simulate :: Wire a -> a
simulate (ConstB b) = b
simulate (ConstI i) = i
simulate (Not wire) = not (simulate  wire)
simulate (And wire1 wire2) = (simulate wire1) && (simulate wire2)
simulate (Add wire1 wire2) = (simulate wire1) + (simulate wire2)
simulate (Equ wire1 wire2) = (simulate wire1) == (simulate wire2)
simulate (Ifte wire1 wire2 wire3) = if (simulate wire1)
                                      then simulate wire2
                                      else simulate wire3
```

The guiding process is performed upon the data constructors to allow the definition of such a function. This is achieved simply via their type definition within the `Wire` generalised algebraic data type. As mentioned by Peyton Jones et al. [55], by means of giving each of the constructors an explicit type definition, GADTs allow what is known as type refinement to occur. In our case, they refine the type `a` above into the required types (`Int` and `Bool`) depending on the constructor being pattern matched.

As we saw here, GADTs as a technique are highly versatile for DSEL development as they allow stronger typing and at the same time allow us to write interpretative functions with ease. Various DSELs nowadays make use of GADTs. Hinze, Jeuring and Löh [29], for example, make use of GADTs to implement a contract DSEL which allows them to create contracts and assert properties upon them. A contract in their language is modelled using a generalised algebraic data type in a similar fashion to our `Wire` data type. Asserting properties is then done using a function which interprets the contract similarly to how our `simulate` function works. Another language which makes use of GADTs is Yampa [48].

In this section we have seen how stronger typing can be introduced into a deeply embedded DSEL since it is possible to define languages which are essentially untyped. This allows certain incorrect programs to pass the compilation stage when domain-related restrictions cannot be enforced. It is still possible to capture such type errors at run-time by encoding the required typing rules within the relevant interpretative functions but it is usually preferable to capture such errors at compile-time. The technique of phantom types allows us enrich the DSEL data type by providing it with extra type information such that Haskell own type-checking mechanisms can adequately compile the embedded language programs. This allows us to capture DSEL type errors at compile-time thus providing an advantage over normal deep embedding. There is however one problem of deep embedding which phantom types do not address: interpretative functions are still required to use run-time compilation by means of case statements and tags so as to extract the required information from the data structure since phantom types cannot do this on their own without extra guidance. GADTs solve this later problem by introducing what is known as type refinement which allows the type-inference engine to refine a generic type into a more specialised form of it according to the current particular case. To achieve this GADT type constructors encode more type information within them thus allowing the construction of more specialised cases of the underlying type. Due to this, GADTs allow us to type-check DSEL programs at compile-time similary to phantom types and also allow us to interpret our DSEL programs in a much simpler fashion.

3.8.5 Parameterised Objects

Parameterised objects, also known as object generators, are functions which are supplied with one or more parameters. These parameters guide the function to generate a number of domain-related objects while following a specific pattern encoded within the function itself. In this way, a single concise description can be used to create a whole family of similar structures. Parameterised objects thus act at a level above the domain itself. By acting as generators of the domain objects they provide the users of the embedded language with a simpler way to create the objects such that they do not have to worry about how to implement their creation but rather how to use them once they are created. They thus greatly simplify our descriptions and render them more understandable.

As an example of the use of parameterised objects we shall create an n-bit binary adder generator function. Before creating the parameterised object itself we must first define a full-adder circuit. A simple way to do this is to define it in terms of a half-adder circuit and a XOR gate component. We thus introduced a `Xor` type constructor to our `Wire` data type and created the matching `xorC` constructor function which acts as our construct. The half-adder and the full-adder circuits are then defined as follows:

```
halfAdder (wire1, wire2) = (sum, carry)
  where
    sum = xorC (wire1, wire2)
    carry = andC (wire1, wire2)


fullAdder (carryIn, (a, b)) = (sum, carryOut)
  where
    (sum1, carry1) = halfAdder (a, b)
    (sum , carry2) = halfAdder (carryIn, sum1)
    carryOut = xorC (carry1, carry2)
```

Using the `fullAdder` circuit we can now define an n-bit binary adder generator:

```
fixedSizeNBitBAdder 0 (carryIn, _ )        = (carryIn, [])
fixedSizeNBitBAdder n (_      , [])        = error (msg)
fixedSizeNBitBAdder n (carryIn, (a,b):abs) = (carryOut, sum:sums)
  where
    (carry, sum)     = fullAdder (carryIn, (a, b))
    (carryOut, sums) = fixedSizeNBitBAdder (n-1) (carry, abs)
    msg              = "Circuit generated requires " ++ show n ++ " more input(s)."
```

The function is parameterised over its first input, a compile-time or static integer value which dictates the final size of the circuit being generated. The rest of the parameters required by the function, the carry-in and the list of input wires, are its run-time or dynamic inputs which are used once the circuit itself has been generated. Despite the fact that the two types of inputs are presented similarly in Haskell due to the latter's type system they are actually distinct. The first parameter is the true parameter of the generator function which is used to actualize the generator into an actual circuit. Once the circuit required is obtained the other parameters are used as the actual circuit inputs. Creating an n-bit binary adder now becomes trivial and can be achieved as follows for 4-bit or 6-bit examples:

```
fixedSize4BitBAdder = fixedSizeNBitBAdder 4
fixedSize6BitBAdder = fixedSizeNBitBAdder 6
```

Both definitions act as their respective generated circuit and expect only the run-time inputs consisting of the carry-in and a list of inputs and return a carry-out and list of outputs. As can be noticed by this example, parameterised objects allow us to generate a member of a family of similar circuits quickly and greatly facilitate the circuit generation process of often-used circuits by allowing the users to abstract away from their implementation details. Finally, they allow us to integrate a form of iteration in our DSEL without introducing explicit looping constructs. Thus parameterised objects are an indispensable technique for the users of DSELs.

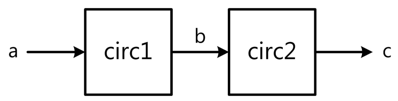


Figure 3.13: Two circuits composed together in series. Reproduced from [8, 12].

3.8.6 Connection Patterns

Domain-specific embedded languages are often known by another name, that of combinator libraries [35]. This is because DSELs, similarly to combinator libraries, often make use of a number of combinator functions or combinators [2, 4, 8, 11, 50]. Combinators are higher-order functions which are used to combine functions together in a certain way in order to create more complex functions. In the realm of DSELs, combinator functions are referred to as connection patterns since they are used to connect objects together according to linking-patterns between them. They are often used when combining constructor functions together to construct larger, composite objects. They are thus a special case of parameterised objects where the parameters are other functions. Using them allows us to introduce new functionality into our DSEL simply by the way they are used to combine functions together.

A common example of a connection pattern is sequential composition. This combinator simply combines two objects together to form a composite of the two where one comes right after the other in series. It would be interesting to add this connection pattern to our example DSEL such that we can combine circuit components one after the other sequentially as shown in Figure 3.13. We can define this connection pattern by means of the following function:

```
(-->) :: (a -> b) -> (b -> c) -> a -> c
(-->) circ1 circ2 a = c
  where
    b = circ1 a
    c = circ2 b
```

As can be noticed, this function `(-->)` acts as a higher-order function which requires as input two other functions, `circ1` and `circ2`, and a normal input `a`. We can use this connection pattern to connect any two circuits as long as the first circuit's output type and the second circuit's input type are compatible to one another. As a trivial example of the use of this connection pattern we shall compose a NAND gate using a description in our DSEL which does not make use of connection patterns and then rewrite the same description using the sequential composition operator we just defined:

```
nandC (a, b) = d
  where
    c = andC (a, b)
    d = notC c

nandC' = andC --> notC
```

We immediately notice that connection patterns give us an advantage in the understandability of the code. It now becomes intuitive how the component is composed out of the other two gates one after the other in sequence. This leads us to yet another advantage: connection patterns allow us to abstract away common implementation details such that the user can focus on what needs to be implemented rather than how. In this case, the connection pattern abstracted away how the output of the AND gate connects with the input of the NOT gate by the passing-around of variables. Instead it does it internally for us such that we do not need to reference the inputs or outputs in any way but simply focus on how the two components are positioned

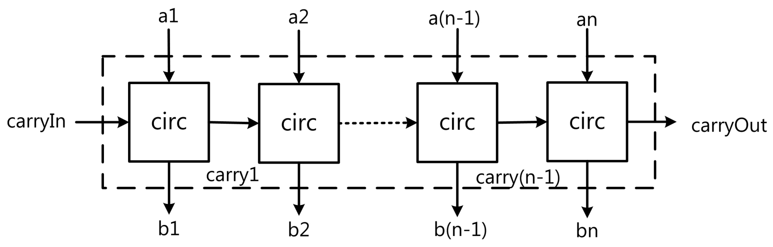


Figure 3.14: The `row` connection pattern parameterised over a generic circuit.

sequentially. As a final advantage, by the use of Haskell's polymorphic types we can overload this operator remarkably easily. In fact, if we want to compose sequentially circuits which have larger or differently structured inputs and outputs we can do so easily as long as they match together in an interlocking-style.

An important use of connection patterns is to describe objects within the embedded language which are structured following a regular pattern such as a row, a column or a grid [8]. By defining an appropriate connection pattern for such structures the advantages described earlier become much more pronounced. As an example of this we shall describe how the row connection pattern can be used within our example DSEL to describe an n-bit adder by composing a number of full-adders in series. As with the example concerning parameterised objects which we discussed earlier we must first add a XOR gate component to our language and define a half-adder and full-adder circuit. If we examine how an n-bit binary adder is constructed we realize that if the carry-out of a full-adder is passed on to another full-adder we can create a two-bit binary adder. Passing the latter's carry-out once more leads to a three-bit binary adder, and so on. This sequential pattern can be described for n-bits as follows:

```
nBitBAdder (carryIn, []) = (carryIn, [])
nBitBAdder (carryIn, ((a,b):abs)) = (carryOut, sum:sums)
  where
    (carry, sum)     = fullAdder (carryIn, (a, b))
    (carryOut, sums) = nBitBAdder (carry, abs)
```

This circuit description generates an n-bit binary adder depending on the size of the input list required. The pattern present in the description, with a base case and a recursive definition, is a common one and often appears exactly the same for different circuits with differently structured inputs. Thus if we abstract away the full-adder circuit from it we can we obtain the required row connection pattern which may be parameterised over any circuit as shown in Figure 3.14. We can describe this connection pattern as follows:

```
row circ (carryIn, []) = (carryIn, [])
row circ (carryIn, a:as) = (carryOut, b:bs)
  where
    (carry, b)     = circ (carryIn, a)
    (carryOut, bs) = row circ (carry, as)
```

If we supply this connection pattern with a full-adder component as an input parameter we can now define an n-bit binary adder trivially:

```
nBitBAdder = row fullAdder
```

It is immediately apparent that making use of the new connection pattern greatly simplifies things for the user. Not only is the n-bit binary adder, which is by no means a trivial circuit to describe in our DSEL, written in one line but it is also much more intuitive to understand that it simply consists of a row of full-adders. Compared to this new description, the former one requires more time to write and comprehend.

By abstracting away from implementation details which clutter our description, this connection pattern allows us to focus on what we want to describe. Regular patterns captured using connection patterns are thus very useful for DSELs as they allow us to greatly simplify the use of the embedded language. Finally, like parameterised objects, they also introduce a way to make use of iteration within our language without explicitly introducing a looping construct. Instead, we make use of connection patterns to create iterative constructs which are related to the domain at hand only such that the user can immediately understand them and visualise them as part of the domain.

3.8.7 Clever Functions

When making use of normal parameterised objects, a static variable such as a Haskell integer is used to control the generation of a structure whilst following a pre-imposed, regular pattern. Clever functions by Sheeran [60, 61, 62] are a form of parameterised object which takes this approach a step forward. Rather than just using the static variable in order to follow a pre-imposed pattern, these functions take certain intelligent decisions based on the values of a set of such static variables. These static variables which guide the generation process are called shadow values and are strongly tied to the structure being generated [61]. In fact, by examining the current shadow values related to a structure, one can obtain a general, overall view of the structure being generated itself. This fact has been used by Sheeran [60] to make use of a "try-and-see" approach where shadow values are paired with dynamic inputs (such as actual input wires) and are used to attempt to generate a tentative part of the structure [62]. This generation process results in a new set of shadow values which can be used once more to infer the new status of the structure being generated. Depending on these resultant shadow values and the way we define the clever function, we might decide to accept the tentative structure as the actual one or for example attempt another possible generation which results in a different set of shadow values equating to a different structure. Ultimately, a set of shadow values, tied to one of many possible structures is selected by the clever function as part of the structure being generated. Using this method, these functions provide their users a means of automatically generating structures which attempt to follow certain biasing criteria without necessarily falling within a regular pattern. In fact, Sheeran [60, 61, 62] has mostly made use of clever functions to describe circuit structures which start of as regular, but due to certain factors which can be varied within such circuits according to the designer's needs, the circuits' regular shapes might change into more irregular ones. To achieve this she has attached shadow values to these factors in various cases and used clever functions to generate the most optimal solution based on a supplied predicate function which detects the fitness of the shadow values. The clever function, basing itself on the fitness of the shadow values supplied by the predicate function which acts as a weighting mechanism then selects by a simple case analysis what it detects as the best way to proceed with circuit generation at that particular point. Using this procedure, the clever function automatically decides for the user what the best circuit structure is. Sheeran has used clever functions for a variety of circuits such as generating median circuits [60], generating multiplier reduction trees [61] and parallel prefix circuits [62]. In these cases the shadow values were attached to the degree of sortedness of the values on the circuit wires, the choice of wiring and non-functional properties, respectively. This technique, however, remains generic enough to be applicable to any embedded language for any domain required.

As an example of the use of clever functions we shall define a component of a multiplexer circuit of variable size. A basic multiplexer can be seen as consisting of three sub-components: a decoder circuit, which selects one of two or more input wires, an array of parallel AND gates which group the multiplexer's input wires with their respective input-selection wires provided by the decoder circuit and a number of OR gates which group all the outputs of the AND gates together, as shown in the example four-input multiplexer in Figure 3.15. Here the actual inputs of the multiplexer are the wires labelled `i1`, `i2`, `i3` and `i4`. Using the two-to-four decoder which receives two inputs `s1` and `s2` and outputs four input-selection wires `si1`, `si2`, `si3` and `si4` we can select the appropriate input from the four possible inputs. This is achieved by combining

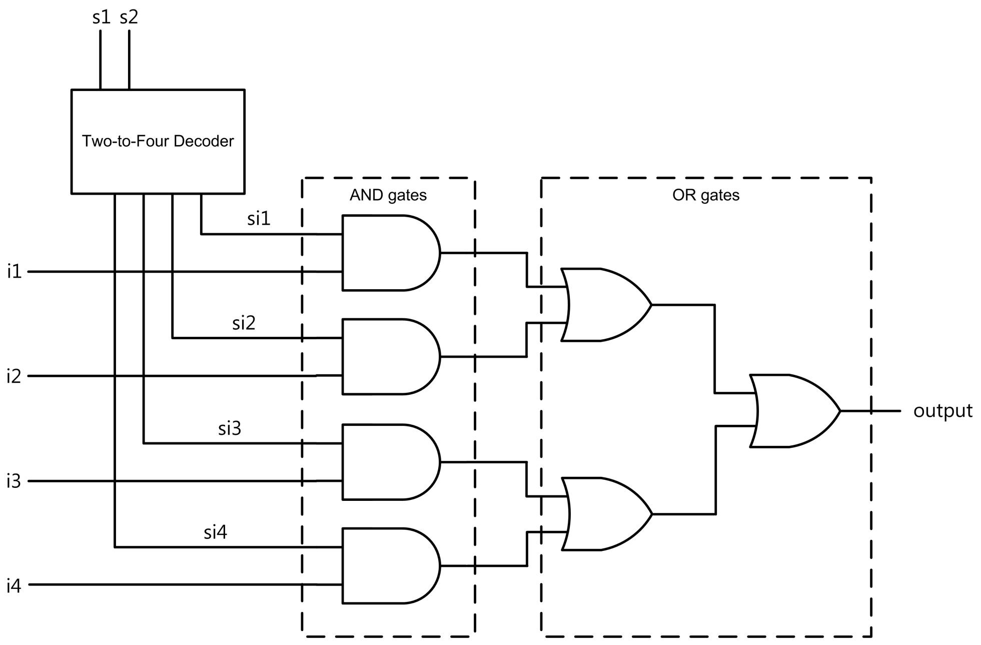


Figure 3.15: A four-input multiplexer with three sub-components: a two-to-four decoder, a number of parallel AND gates connected to the multiplexer's inputs and the decoder's input-selection wires, and a number of OR gates which connect the outputs of the AND gates together.

each of the decoder's input-selection wires with their respective multiplexer input wire by means of parallel AND gates. The multiplexer's resultant `output`, is a disjunction of the AND gates' outputs and can be achieved by means of the OR gates. The component which we are interested in is the one with the parallel AND gates. We shall use a clever function to generate the required number of AND gates automatically based on the number of inputs. So far this can be achieved by a simple connection pattern similar to the `row` pattern mentioned earlier but altered to combine the gates in parallel rather than in series. We wish however, due to external factors such as manufacturing costs or performance issues, that instead of the repeated use of AND gates, we replace some or all of the latter by OR gates and NOT gates via De Morgan's Law. In short, we need the number of AND gates or the number of OR and NOT gates to be alterable according to our needs. We may achieve this by means of a biasing value which we may supply to the generating function, in our case the clever function we are going to use. This biasing value is in turn passed to a predicate function which also receives two shadow values. The latter shadow values are tied to the usage of true AND gates or replacement AND components (OR/NOT gates). Using the biasing value and these shadow values, the predicate function aids the clever function to decide whether to create a true AND gate or its equivalent by means of the replacement AND components. In this way, the clever function can balance the number of AND gates or their equivalent components used according to our needs on its own.

In order to implement the clever function mentioned above we must first alter our basic language to include an explicit OR gate component since we previously limited ourselves to AND and NOT gates. This is achieved as follows:

```
data Wire = Low
          | High
          | Not Wire
          | And Wire Wire
          | Or Wire Wire
```

Hence, we define the appropriate constructor function `orC` as follows:

```
orC :: (Wire, Wire) -> Wire
orC (wire1, wire2) = Or wire1 wire2
```

For convenience we shall also introduce the replacement AND gate made up of three NOT gates and one OR gate by means of the following `andC2` function:

```
andC2 :: (Wire, Wire) -> Wire
andC2 (wire1, wire2) = notC (orC (notC wire1, notC wire2))
```

The clever function itself is defined using the following Haskell code:

```
multiplexerAndComp :: Int -> Int -> [(Wire, Wire)] -> [Wire]
multiplexerAndComp _ _ [] = []
multiplexerAndComp bias n ((i, si):xs) =
  if (balancingFunction bias n1 n2)
    then w2: multiplexerAndComp bias n2 xs
    else w1: multiplexerAndComp bias n1 xs
  where (n1, w1) = (n-1, andC  (i, si))
        (n2, w2) = (n+1, andC2 (i, si))
```

This function receives as input three parameters. The first of these is the biasing value `bias`. As mentioned earlier this can be used by the user to bias the function towards creating more actual AND gates or more replacement AND gates. If the biasing value is zero the clever function attempts to balance the number of both AND gate types. The higher positively the biasing value, the more the number of replacement AND gates. Conversely, the higher negatively the biasing value, the more the number of true AND gates. The user of this function which may be presented as a construct can thus tweak the number of gates of each type of AND gate as required using this input. The clever function then does the actual generation based on this value accordingly and on its own. The second input `n` is a running value which is used by the clever function whilst building the different AND gates to check how far it is from the biasing value. The clever function attempts to hold this value as close as possible to the biasing value whilst at the same time creating the appropriate AND gate type. The final input is a list of tuples of wires. Each of the tuples consist of a multiplexer input wire and an input-selection wire from the decoder which are supplied to an AND gate as required by the multiplexer circuit. The function itself works recursively upon the list of tuples to create the required AND gates. The first case is the base case which given an empty list of tuples simply returns an empty list. The recursive definition, first creates both AND gate types using the head tuple of the input list, and also two respective shadow values `n1` and `n2`. These two shadow values are then used as described above by the predicate function `balancingFunction` to examine, by comparison with `bias`, which of the two created AND gates to actually use. The predicate function is defined simply as follows:

```
balancingFunction :: Int -> Int -> Int -> Bool
balancingFunction bias and1Val and2Val = (diff1 >= diff2)
  where
    diff1 = abs (bias - and1Val)
    diff2 = abs (bias - and2Val)
```

It simply checks the difference both of the shadow values have when compared to the bias value which is used as a reference and then returns a Boolean-typed value which signifies which of the shadow values differs the most from the bias. If the difference value for the true AND gates is greater than or equal to that for the replacement AND gates, the clever function makes use of the replacement AND gate created before while the other AND gate is discarded, otherwise the true AND gate is used instead. The selected gate is returned as part of the output list of created AND gate circuits and a recursive call of the clever function is performed on the rest of the input list whilst supplying the used AND gate type's shadow value as the subsequent running value. In this way the clever function generates the required list of AND gate types automatically for us. The list can then be used as a component to create the final, actual multiplexer circuit itself.

In order to show the use of the clever function we shall emulate a number of calls to this function and show the list of outputs received. Some examples are:

```
> multiplexerAndComp 0 0 [(high, low), (low, low), (low, high), (low,low)]
[Not (Or (Not High) (Not Low)),And Low Low,Not (Or (Not Low) (Not High)),And Low Low]
> multiplexerAndComp (-2) 0 [(high, low), (low, low), (low, high), (low,low)]
[And High Low,And Low Low,Not (Or (Not Low) (Not High)),And Low Low]
> multiplexerAndComp 2 0 [(high, low), (low, low), (low, high), (low,low)]
[Not (Or (Not High) (Not Low)),Not (Or (Not Low) (Not Low)),
 Not (Or (Not Low) (Not High)),And Low Low]
```

In all three calls the input list of tuples is the same. Since we are dealing with a decoder's output, only the third tuple is marked with a high-valued input which signifies that this input is going to be selected by the multiplexer. We also start from a running value of zero. In the first call we supply a bias value of zero. As can be noticed we obtain an alternating list of replacement AND gates and true AND gates, that is, the clever function is unbiased and so returns the same amount of both types indiscriminately. In the second call we bias the function towards true AND gates by means of the value `-2`. Hence, we receive three true AND gates and one replacement AND gate. In the third call we see the opposite situation by supplying a bias of `2`. Changes to the generated sequence of AND gates can also be made by altering the starting running total supplied. As we have seen by means of these example calls to the clever function, the latter automatically generates the gates we need whilst following possible requirements which are supplied to it as static inputs. It is also possible to package and expose different variations of this function to the user by supplying a number of predefined bias values and starting running values thus perhaps simplifying its use, such as for example, extremely biased versions supplied with high positive or high negative bias input values.

As we have seen by means of our multiplexer component example, clever functions, by the use of shadow values and predicates allow us to write parameterised objects which generate structures (and hence programs) in our embedded language where the latter, rather than simply following predefined patterns, are more adaptive to our possible needs. This allows our generators to become more context-aware and subsequently increase productivity by creating embedded language programs, even irregular ones, for us by means of a simple function call.

3.9 Interpreting and Compiling Embedded Languages

An interpretative or compilation function can be considered a form of parameterised object where the parameter supplied is a language program which the function translates into another value of the same language or into a lower-level language program. For example, simulating a circuit structure using a simulation function resulted in one or more wires carrying either a high value or a low value. These values where themselves of the data type which was being used to represent the embedded language programs within the host language. As another example, counting the gates of the circuit resulted into a simple integer value at the Haskell level, that is at the level of the host language. We thus maintained the existence of two basic levels of abstraction and stepped-up or stepped-down as was required. Allowing for two levels gives us various advantages. Of these, the most important, as stated by Claessen and Pace [8, 10], is the fact that for the host language, the embedded language programs are first-class objects. This allows the host language to manipulate them at will, thus giving the programmers the power of a complete programming language when this is required.

Claessen and Pace [8, 10], however proposed another step forward. Rather than stopping at a two-tiered methodology with just an embedded language within its host language, they proposed a multi-tiered framework of embedded languages where it is possible to embedded a language within another language, the latter of which is also itself an embedded language. They have achieved this by embedded a synthesisable behavioural description language called Flash into their Haskell-embedded hardware description language Lava which we discussed earlier [8, 10]. The higher-level language's syntax is implemented as a normal data type in Haskell, but the structures of the latter type are not interpreted or compiled directly to Haskell which is an indirect host, but rather to the direct host language. It is of course possible, if required, to translate into Haskell's level of abstraction, however this can be achieved in a two-stepped translation: first from the higher-level embedded language into the lower-level embedded language and then from the latter into the host language. Using this approach saves a lot of work since the lower-level embedded language acts as an appropriate intermediate language. The intermediate language approach had been suggested by Bentley [3], but by virtue of embedding it becomes much more easier to achieve as no embedded language, neither the higher-level one, nor the lower-level one must be implemented from scratch. All one needs to do is provide the embedded language's syntax as a type in the main host language and provide a translation mechanism, usually a simple compilation function, into a less abstract level.

The approach suggested by Claessen and Pace [8, 10] introduces various advantages. The first of these is the number of embeddings possible. Using multi-tiered embedding leads to potentially an infinite amount of embedded languages since there are in fact various levels of abstraction one might envisage as part of their framework of languages. Allowing for different levels of abstraction allows for faster programming in a specialized, abstract domain which can then be concretised by means of a simple function which takes the abstract program to a more concrete, realisable level. Another advantage is that of combining one or more embedded languages of the same level into a lower level host language. Utilising one of these languages in a lower level language program allows the user to select the best language to facilitate the writing of a solution to solve a particular problem. Combining them together in one host language takes this a step further by allowing two languages of two different specialisations to be utilized in tandem to achieve a solution for their host language. A framework of embedded languages also allows nesting of languages to be achieved where one or more languages of the same level of abstraction are imported in another language also of the same level. This is usually achieved by adding an import primitive used to explicitly import the other language within the nesting language. It is possible to have various such import statements which nest different languages within the same language if required. Nesting allows us to combine languages together without requiring the use of the hosting language to act as common grounds for the languages being combined or without having to create a unifying language for every two or more languages being combined together. Language embedding allows for a further advantage: error handling. If a language at a higher level needs to impose a restriction on its lower-level host language due to its semantics this can be achieved by using an appropriate error marker at the level of the embedded language. This error marker is compiled away if we require compilation to a lower level and can be used with a verification interpretation at the host-language level to check the correctness of the program in the embedded language.

Thus, by the use of an embedded language framework we obtain a variety of benefits similarly to the advantages received when a DSEL is embedded in a host language. Compilers to the actual host language for the new embedded language are made available by means of a simple translation function to the hosting embedded language and the tools available to the hosting embedded language are also available for the more abstract language. Moreover, since programs at all embedded language levels are first-class data structures for Haskell, they can be manipulated and transformed as required by simple host language functions.

3.10 Conclusion

In this chapter, we have introduced domain-specific embedded languages and outlined how they are derived from domain-specific languages. Following this, we considered their aims and positive aspects along with their drawbacks. Next, we discussed how the notion of embedding provides a number of advantages over the traditional approach of implementing a language from scratch often made use of when creating DSLs. Selecting the correct host language for embedding is a fundamental aspect we have also discussed in detail. We then saw how this technique was used for a number of domains including images, animation, contracts, querying databases, music composition, robotics, testing, hardware description and business processes amongst others. We also discussed a number of techniques often utilised when employing DSELs such as shallow and deep embedding, the detection of sharing and loops, ways to embed a type system for a DSEL and functions for the creation of simple and clever program generators. Finally, we showed how the technique of embedding is not limited to one level of abstraction only but can be extended to a create multi-tiered framework of embedded languages, each at a higher level of abstraction and specialisation than the host language it is embedded in. As we shall see in the following chapter, these techniques can be put to use in order to quickly and efficiently create a scripting language for games which may be used for normal simple scripts and even complex AI scripts as was suggested in Chapter 2.

Chapter 4

Case Study: 4Blocks

4.1 Introduction

As we discussed earlier, making use of a scripting language such as a general-purpose language or a proprietary language brings about various advantages over game engine integration. However, off-the-shelf general-purpose languages are often unnecessarily bloated in order to attempt to cater for most scripting needs and do not offer domain-specific constructs which could allow a story-writer to define their own scripts for the game as domain-specific proprietary languages do. As we hinted in Chapter 2, a possible solution would be to develop a domain-specific language of our own which is good for story-writers but which would also be expressive enough to enable AI-scripting. However, this requires us to introduce an extra, non-trivial task into the game-production life-cycle, that of creating a scripting language from the ground up. We believe that the technique of language embedding provides us with the essential tools required in order solve this problem by allowing us to to quickly, efficiently and cost-effectively create such a scripting language. The main aim behind this chapter is to show how this can be achieved by making use of a simple puzzle game called 4Blocks. We shall first show how quick and straightforward it is to develop and integrate a domain-specific scripting language within the game in order to control it. Such a language is simple and abstract enough such that it can be easily understood by story-writers. Next we shall expand upon this language by adding general-purpose language constructs such as conditional and looping statements to it and show how it may be used to write simple, fixed AI scripts which are carried out automatically for us by an AI module. Finally, since embedding is done in a general-purpose language such as Haskell the latter's features are available for use to create powerful manipulation functions which enable us to improve upon fixed AI scripts by allowing us to create adaptive AI scripts which are generated on-the-fly based on the current game state.

4.2 4Blocks

4Blocks is a game written in Haskell in the spirit of the popular game Tetris created by Alexey Pajitnov in 1984[1]. During a session of this game, shown in Figure 4.1, a continuous number of randomly generated bricks appear on the upper side of the screen and fall downwards at a constant rate which increases as the game progresses. Each of the bricks consists of four blocks (hence the name of our game) which are arranged in a number of different shapes. The player can perform a series of moves upon each of these bricks such that they are positioned on the bottom of the game area, also known as "well". The possible moves are: shifting to the left or right, rotating to the left or right, performing a hard (sudden) or soft (slow) drop and

[1]http://www.tetris.com/history/index.aspx (last visited August 2010)

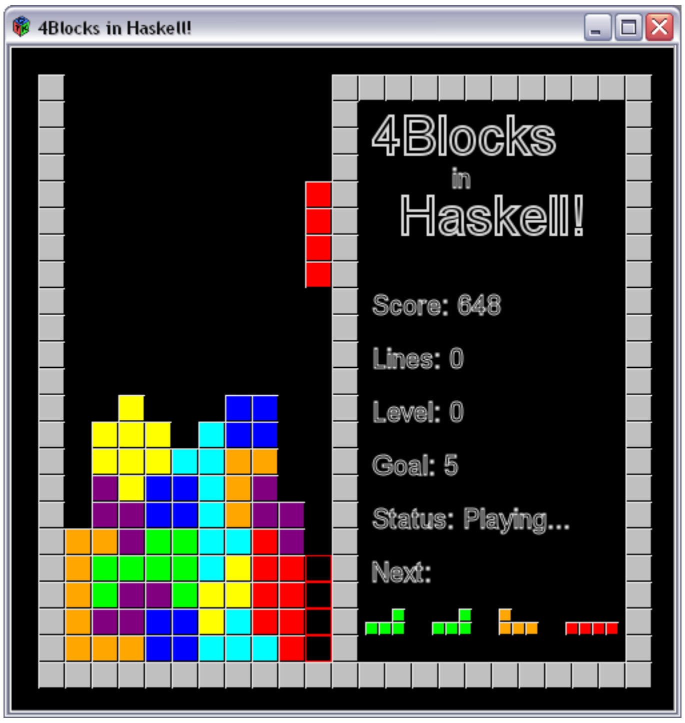


Figure 4.1: A session of 4Blocks

performing no action at all. The objective of the game is to create 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. Figure 4.1 shows an example of an I-shaped brick which is being used to complete four lines. Other bonus points are given when the user forces the brick downwards themselves using a soft drop or a hard drop. 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. The game is lost if the well is filled up with unfinished lines and the point where bricks appear on the screen is blocked.

4.3 Creating a Scripting Language

As an initial introduction to our approach we shall create a simple domain-specific embedded scripting language for 4Blocks which is easy to comprehend and use, even by story-writers. Creating such a language for a puzzle game like 4Blocks simply for use by story-writers is somewhat excessive but our aim here is to show how quick and easy it is to create a scripting language by means of the embedded approach. Moreover, we shall build upon such a language in a later section in order to enable it to express actual AI scripts so this will serve as a gentle introduction.

If a story-writer had to script a game of 4Blocks, such as in an automated demonstration of the game, the language they would require is one which allows them to describe how a brick can move from an initial location into its final one. Since this can be done in one or more steps our language should be able to express either how to perform a single action or how to perform a number of actions one after the other, sequentially. Thus our language's syntax needs two constructs: one to encapsulate an instruction to be performed and one to combine a possibly infinite number of the former. Using the popular Backus-Naur Form (BNF) notation this two constructs can be defined as follows:

program ::= **perform** *instruction*
  | *program* **;** *program*

Thus semantically, a program consists of either simply performing an instruction or two programs chained together. Using the second case we can chain as many programs as we require in order to allow a brick to reach its final location. To complete the definition of our language's syntax we need to define the syntax of what an instruction is. Since seven axiomatic actions are possible within the game, these can be defined simply using the following BNF:

instruction ::= **shiftLeft** | **shiftRight** | **rotateLeft** | **rotateRight**
  | **softDrop** | **hardDrop** | **noAction**

Each of these actions are instructions which the game understands when passed as input to it. Hence their semantics are tied to how the game interprets them by performing their respective transformation upon the current brick. This completes our specification of the language and the design of both its syntax and its semantics.

The steps undertaken so far are in line to traditional language-creation steps. It is in the next step where the embedded language approach differs, that is in the implementation step. Usually this step requires a considerable amount of work in order to create a compiler or interpreter for the language since we have to develop a number of components such as lexical, syntax and semantic source-code analysers, code optimisers and target-machine code generators [1]. Instead of following this path we shall create our scripting language for 4Blocks by embedding our language in the game's implementation language, Haskell. All that is required in regards of syntax-creation is translating the above two BNFs into Haskell data structures. For *program* this is achieved using the `Program` data type as follows:

```
data Program a
  = Perform a                -- perform an instruction
  | Program a :> Program a   -- sequential composition
```

where each of the data type's constructor maps to a case in the BNF. The first of `Program`'s constructors simply encapsulates a single action and allows us to lift an instruction in order to create a basic program consisting of just one instruction. It is the second constructor's job to actually link two programs together so as to create a possible chain of instructions. Similarly, the instructions' enumeration is defined by means of the `Instruction` data type:

```
data Instruction
  = ShiftLeft      -- shift brick to the left
  | ShiftRight     -- shift brick to the right
  | RotateLeft     -- rotate brick to the left
  | RotateRight    -- rotate brick to the right
  | HardDrop       -- perform a sudden drop
  | SoftDrop       -- perform a slow drop
  | NoAction       -- do nothing
```

where each of the actions is mapped to the corresponding `Instruction` constructor.

These two data types allow us to create a deeply-embedded DSEL for 4Blocks where the language's syntax is defined using the data type `Program Instruction`. Structures of this data type form our game scripts. As an example of a script consider the program:

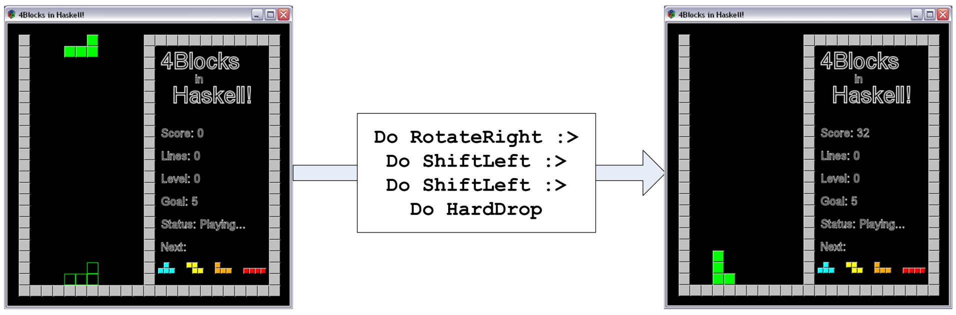


Figure 4.2: Carrying out a simple example script.

```
Perform RotateRight :>
Perform ShiftLeft :>
Perform ShiftLeft :>
Perform HardDrop
```

This program is very simple to understand, even by a non-programmer. It first rotates a brick to the right, shifts it twice to the left and performs a hard drop so as to place it at the bottom of the well. Conceptually, if we are given a game state consisting of an empty well and an L-shaped brick the result of the above script is that shown in Figure 4.2.

In order to allow the game to actually perform this, a scripting engine must be introduced as a module within our game. This module, consisting of a function which we named `step`, takes care of attaching the required semantics to the structures representing our scripts, thus acting as our language's interpreter. Each call to `step` performs an interpretation step which extracts a single instruction from the program and returns an updated program which is then used for the next interpretation step and so on. The extracted instruction is then used, via the game's API, to call the relevant command which the game engine uses to update the game state. This process is visually summarised in Figure 4.3. Since a program might consist of a single instruction only it is possible that `step` returns no program at all. Due to this we make use of Haskell's `Maybe` data type. Thus the function `step` is defined as follows:

```
step :: Program Instruction -> (Maybe (Program Instruction), Instruction)
step (Perform inst) = (Nothing, inst)
step (prog1 :> prog2) =
  case (step prog1) of
    (Nothing    ,inst) -> (Just prog2, inst)
    (Just prog1b,inst) -> (Just (prog1b :> prog2), inst)
```

If a program consists of a single `Perform` we simply return an empty updated program and the instruction within the `Perform`. This serves as our base case. In case of sequential composition we recursively step the first of the two sequentially composed programs. If this program terminates after returning one instruction we simply return this and the second program. However, if the first program does not terminate after a recursive call to `step` we simply return the instruction and the updated first program sequentially composed with the second program. In both cases the instruction returned can then be used to update the game state in a similar manner to how this is achieved when the game is being played by a human player. When the latter plays the game a key-press is mapped to a particular game-updating function via the game's API. This time we do not map a key-press but rather the instruction itself. This is carried out until the program terminates allowing the brick to reach its final location within the well.

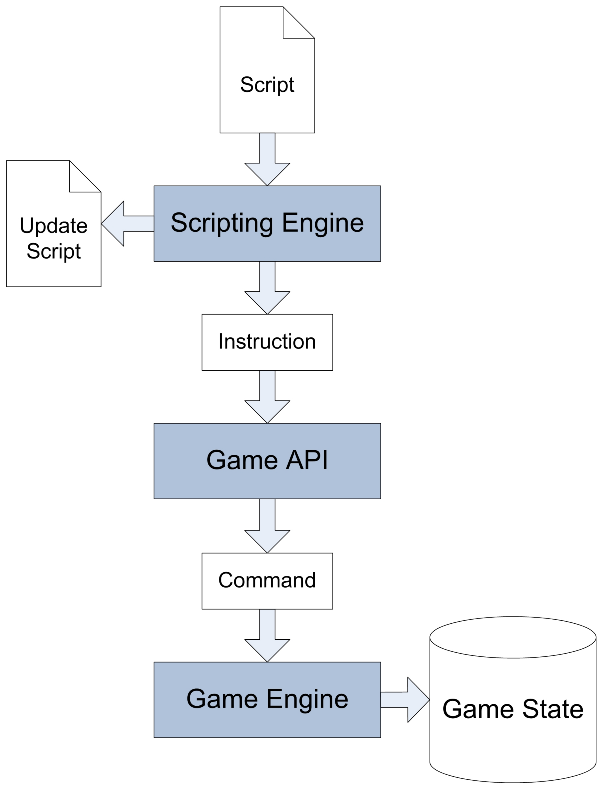


Figure 4.3: How a script is interpreted.

4.4 A Scripting Language for AI

The language we have developed so far is not expressive enough to allow a programmer to write any scripts which encode any form of AI within them. To allow us to write AI scripts, our language needs to be able to react to the game state, that is, it must be able to query the game state and take decisions based upon it. A possible way to achieve this is to extend the language with general-purpose language constructs which allow for branching, looping and exception-handling. In this section we will show how easy it is, by means of the embedded language approach, to update our previously-defined language in order to include these non-trivial notions in the form of language constructs.

4.4.1 Language Syntax

The first construct we require is one which branches program flow based on whether a predicate upon the game state has been satisfied or not. In general-purpose languages this functionality is often performed by an *if-then-else* statement. We can thus add this construct such that it allows us to control which branch of a script is run based on whether the predicate returns a Boolean value of true or false. As an example, consider that we wish to write a program which rotates a brick to the right if it is a certain shape (say, I-shaped), otherwise it rotates the brick to the left. Conceptually, we need something of the form:

If (`brickIsIshaped`)
    Then (`Perform ShiftRight`)
    Else (`Perform ShiftLeft`)

where `brickIsIshaped` is the predicate which performs the required check for us using an appropriate API call. Furthermore, since sometimes we do not wish to perform any program when a predicate is not satisfied (that is, in the else branch), we require a place-holder construct or null-program construct which is simply skipped over when met. This construct also facilitates our language's algebraic semantics [30]. We can name this construct *skip*. It allows us to write programs such as:

```
If (brickIsIshaped)
   Then (Perform ShiftRight)
   Else Skip
```

which cause the else branch to be simply ignored if the brick is not I-shaped.

The third construct we need is one which allows our programs to loop until a predicate is no longer satisfied, a functionality which is often supplied by a *while* statement. We require this statement because sometimes we simply wish to repeat a certain program upon a brick a number of times until its state in the well changes in a certain manner. For example, consider that we wish to shift a brick to the right a number of times until it hits the well's right wall. This can be written by allowing our construct to take the following format:

```
While (brickNotAtRightWall)
   (Perform ShiftRight)
```

where `brickNotAtRightWall` is the predicate that checks the game state for us. As a measure of convenience we will also create a variation of the *while* statement called *forever*. The latter is simply a *while* loop which always evaluates to true and thus loops indefinitely. As an example consider the following script:

```
Forever
   (Perform ShiftRight)
```

This script will cause any brick which appears in the well to shift to the right continuously.

Finally, for a game such as 4Blocks another construct which would be very useful when writing our AI scripts is one which allows us to detect when a new brick enters the well. Such a construct, which we named by the domain-specific term *for-this-brick*, would allow us to ensure that a program which is intended for a brick that has become locked in its place does not overflow its scope and start affecting a new brick. In a way this is a weak form of exception-handling since if there is any outstanding program from the previous brick this must be immediately discarded when the new brick is introduced. As an example, let us consider that we wish that a program which shifts a brick to the right once is only carried out if the brick is still in scope. This could be simply written as follows:

```
ForThisPiece
   (Perform ShiftRight)
```

This construct is more useful with long programs or with looping programs which might not terminate in time and so scope over to the next brick such as for example with the *forever* construct as follows:

```
ForThisPiece
   (Forever (Perform ShiftRight))
```

Using this program only the current brick will shift to the right continuously since the *for-this-piece* construct will cause an exception which exits the loop when a new brick appears within the well.

Thus our final language syntax can be defined by updating our previous language's BNF by adding the new constructs, as follows:

program ::= . . .
  | **ifThenElse** *predicate program program*
  | **while** *predicate program*
  | **forever** *program*
  | **forThisPiece** *program*
  | **skip**

We will also add another rule for predicates:

predicate ::= *. . . various API calls upon the game state . . .*

Using this syntax we can write programs such as:

forThisPiece (
  while (**brickHeightGreaterThan5**)
    (**perform softDrop**) **:>**
  perform shiftLeft)

This program may be used to perform what is know as a side-fit where the brick is lowered to a certain height and then moved laterally into position. Using the **ForThisPiece** construct we restrict the encapsulated code to the current brick only such that any following brick is not affected by this code. We then make use of a **While** construct in order to perform the lowering part itself. The function **brickHeightGreaterThan5** is an API function which we use in order to check whether the brick's height is more than 5. If this is true a soft drop is performed. This goes on until the predicate is false at which point we shift the brick to the left using the last **Perform** statement. As can be seen by this example fairly sophisticated scripts may now be written using our modified language.

Implementing the above language's syntax as an embedded language in Haskell by updating our previous definition is again straightforward. For every new case in the *program* BNF we shall add a new construct to our `Program` data type as follows:

```
data Program a
  = ...
  | IfThenElse Pred (Program a) (Program a)    -- if-then-else statement
  | While Pred (Program a)                     -- while statement
  | Forever (Program a)                        -- repeat a program forever
  | ForThisPiece (Program a)                   -- perform program for this piece
  | Skip                                       -- null statement
```

As can be noticed there is a one-to-one mapping between each of the BNF cases and `Program`'s constructors. The first construct we are adding is the *if-then-else* statement. It requires a predicate type `Pred` and two programs. The predicate is essentially a function over the game state, of the form `Game -> Bool`, where `Game` represents our game state. We use such a function in order to query the current game state to check whether a predicate is satisfied. Several API functions are available fitting this type which allow us to achieve this. The second construct is the *while* statement which takes a predicate and a looping program as input as was described earlier. Similar to the *while* statement in structure are the *forever* statement and the *for-this-piece* statement which, however, do not require a predicate as input. Finally, we have the *skip* statement which requires no input. Our example above can be written in our updated language as follows:

```
ForThisPiece (
  While (brickHeightGreaterThan5)
    (Perform SoftDrop) :>
  Perform ShiftLeft )
```

Instead of exposing the `Program` data type and its type constructors we shall create a number of constructor functions which will act as our language constructs. This allows us to expose a safer interface to our language for the AI programmers to use and also allows us to define a number of language constructs based on our few available type constructors. This latter fact will allow us to keeping our language's semantics as simple and concise as possible as we shall see when we discuss them later on. The instructions themselves will also be created using constructor functions in order to create a more polished look to our language. These are defined as follows:

```
shiftLeft :: Instruction
shiftLeft = ShiftLeft

shiftRight :: Instruction
shiftRight = ShiftRight

...
```

For the `Program` data type we may provide the following constructs as part of our language:

```
perform :: Instruction -> Program Instruction
perform inst = Perform inst

skip :: Program Instruction
skip = Skip

forThisPiece :: Program Instruction -> Program Instruction
forThisPiece prog = ForThisPiece prog

ifThenElse :: Pred -> Program Instruction -> Program Instruction -> Program Instruction
ifThenElse pred prog1 prog2 = IfThenElse pred prog1 prog2
```

The definition of `ifThen` is rendered possible by *skip* as was suggested earlier.

```
ifThen :: Pred -> Program Instruction -> Program Instruction
ifThen pred prog = ifThenElse pred prog skip

while :: Pred -> Program Instruction -> Program Instruction
while pred prog = While pred prog
```

The construct `forever` is defined in terms of `While` itself so in reality we do not need a `Forever` constructor within our `Program` data type. Hence this constructor can be safely removed and the `forever` construct simply defined as a while loop where the predicate is a function which returns always true.

```
forever :: Program Instruction -> Program Instruction
forever prog = while (\_ -> True) prog
```

4.4.2 Language Semantics

Part of our language's design process consists of specifying its semantics. Since our language syntax is based on a small number of constructs: *skip*, *perform*, *sequential composition*, *if-then-else*, *while* and *for-this-piece*, we must define their operational semantics.

Before doing so we must define some preamble. The set of basic instructions defined earlier is denoted by I and the null instruction is denoted by $\bullet$, such that:

$$I^{+} \triangleq I \cup \{\bullet\}$$

The set of all possible programs is denoted by Prg while the terminated program is denoted by $\bullet$, such that:

$$Prg^{+} \triangleq Prg \cup \{\bullet\}$$

The set of game states is represented by Σ where each game state σ contains the information regarding the current brick, the state of the well, the bricks following the active one, the number of accumulated points, the number of lines completed, the number of lines left to reach the level's line-goal and the current level number. Given the aforementioned state σ we will assume a number of predicates $Pred$ over the state such that $c \in Pred$ can be resolved by $\sigma(c)$. We will also assume a predicate $new\_brick \in Pred$ which queries the game state in order to check whether the brick in scope has just been generated.

A script transitions into another script by means of a transition relation defined as:

$$\rightarrow \subseteq Prg^{+} \times \Sigma \times I \times Prg^{+}$$

such that our semantics take the following format:

$$P \xrightarrow[\sigma]{i} P'$$

where $P \in Prg^{+}$ is the script before transition, $P' \in Prg^{+}$ is the updated script, post transition, $i \in I^{+}$ is the current instruction and $\sigma \in \Sigma$ is the current game state. For example, the transition of a program which consists of a left shift followed by a right shift may be written as follows:

$$\textbf{Perform (ShiftLeft)} \;\; \textbf{:>} \;\; \textbf{Perform (ShiftRight)} \xrightarrow[\sigma]{\textbf{ShiftLeft}} \textbf{Perform (ShiftRight)}$$

Following the transition the first instruction within the program (a left shift) is consumed and returned as the current instruction. The updated program becomes one which performs a shift to the right only.

Finally, we also assume a function, *update*. This function updates the game state by applying the instruction extracted from the program transition. It has the following type definition:

$$update : (\Sigma \times I) \;\rightarrow\; \Sigma$$

This allows us to define our operational semantics as follows:

Skip

$$\frac{}{\textbf{Skip} \xrightarrow[\sigma]{\bullet} \bullet}$$

A program consisting of a `Skip` is translated into a terminated program immediately (no step is performed).

Perform

$$\frac{}{\textbf{Perform}\ \alpha \xrightarrow[\sigma]{\alpha} \bullet}$$

A program consisting of a `Perform` is translated into a terminated program in one step and extracts one instruction.

Sequential Composition

$$\frac{P_1 \overset{\gamma}{\underset{\sigma}{\rightarrow}} \bullet}{P_1 :> P_2 \overset{\gamma}{\underset{\sigma}{\rightarrow}} P_2} \qquad \frac{P_1 \overset{\gamma}{\underset{\sigma}{\rightarrow}} P_1'}{P_1 :> P_2 \overset{\gamma}{\underset{\sigma}{\rightarrow}} P_1' :> P_2}\, P_1' \neq \bullet \qquad \frac{P_1 \overset{\bullet}{\underset{\sigma}{\rightarrow}} \bullet \quad P_2 \overset{\gamma}{\underset{\sigma}{\rightarrow}} P_2'}{P_1 :> P_2 \overset{\gamma}{\underset{\sigma}{\rightarrow}} P_2'}$$

The first rule states that if the first program in sequential composition translates into a terminated program, whilst extracting zero or one instructions, the second program is returned as a result of the step. If the first program is still unfinished after the first step, the second rule dictates that the remaining, updated first program is sequentially composed with the second program. The third rule states that if the first program translates into a terminated program and returns no instruction we simply return the updated second program and the instruction which results from this latter's translation.

If-Then-Else

$$\frac{P_1 \overset{\gamma}{\underset{\sigma}{\rightarrow}} P_1'}{\textbf{IfThenElse}\ c\ P_1\ P_2 \overset{\gamma}{\underset{\sigma}{\rightarrow}} P_1'}\, \sigma(c) \qquad \frac{P_2 \overset{\gamma}{\underset{\sigma}{\rightarrow}} P_2'}{\textbf{IfThenElse}\ c\ P_1\ P_2 \overset{\gamma}{\underset{\sigma}{\rightarrow}} P_2'}\, \neg\sigma(c)$$

In the first rule, if the predicate upon the game state is true and the first program translates into another updated program, whilst returning zero or one instructions, this program is returned. Similarly for the second program when the predicate is false.

While

$$\frac{P_1 \overset{\gamma}{\underset{\sigma}{\rightarrow}} P_1'}{\textbf{While}\ c\ P_1 \overset{\gamma}{\underset{\sigma}{\rightarrow}} P_1' :> \textbf{While}\ c\ P_1}\, \sigma(c), P_1' \neq \bullet \qquad \frac{P_1 \overset{\gamma}{\underset{\sigma}{\rightarrow}} \bullet}{\textbf{While}\ c\ P_1 \overset{\gamma}{\underset{\sigma}{\rightarrow}} \textbf{While}\ c\ P_1}\, \sigma(c)$$

$$\frac{}{\textbf{While}\ c\ P_1 \overset{\bullet}{\underset{\sigma}{\rightarrow}} \bullet}\, \neg\sigma(c)$$

The first rule states that if the `While` construct's predicate is true and the looping program translates into an updated program which is not the terminated program, the remaining, updated program is sequentially composed with a fresh copy of the while loop with the original looping program. On the other hand the second rule states that if the looping program terminates a fresh copy of the while loop is returned. Finally, if the predicate is false the while loop translates to the terminated program and returns no instruction.

For-This-Piece

$$\frac{P_1 \overset{\gamma}{\underset{\sigma}{\rightarrow}} P_1'}{\textbf{ForThisPiece}\ P_1 \overset{\gamma}{\underset{\sigma}{\rightarrow}} \textbf{ForThisPiece}\ P_1'}\, \neg\sigma(new\_brick), P_1' \neq \bullet$$

$$\frac{P_1 \overset{\gamma}{\underset{\sigma}{\rightarrow}} \bullet}{\textbf{ForThisPiece}\ P_1 \overset{\gamma}{\underset{\sigma}{\rightarrow}} \textbf{ForThisPiece (Perform NoAction)}}\, \neg\sigma(new\_brick)$$

$$\frac{}{\textbf{ForThisPiece}\ P_1 \overset{\bullet}{\underset{\sigma}{\rightarrow}} \bullet}\, \sigma(new\_brick)$$

The for-this-piece construct is a bit trickier to define. The first rule states that if the encapsulated program translates into an updated program which is not the terminated program and the current brick is not a new one, the result of translation is the updated program encapsulated within a `ForThisPiece`. If, on the other hand, the updated program is the terminated program, the second rule dictates that we return a program consisting of `Perform NoAction` within `ForThisPiece`. This implies that when the encapsulated program terminates we simply perform no action. This goes on until the third rule comes into force, when a new brick is generated. When this occurs the entire program is skipped such that it translates into the terminated program whilst returning no instruction.

Finally, a we define the transition relation for a turn since it ties how a program is interpreted to how it actually affects the game state:

$$\rightarrow_T \subseteq Prg^+ \times \Sigma \times Prg^+ \times \Sigma$$

Here we map a program and a game state to updated, post-turn versions of these. A turn thus has the following semantics:

$$\frac{\begin{array}{ll} P \overset{\gamma}{\underset{\sigma}{\rightarrow}} P' & \textit{(line 1)} \\ \sigma' = update(\sigma, \gamma) & \textit{(line 2)} \end{array}}{(P, \sigma) \quad \rightarrow_T \quad (P', \sigma')}$$

Thus, in order for a turn to occur, we must update the program P into P' and the game state σ into σ'. This is possible if two premises are satisfied. The first of these (line 1) states that by means of the transition relation $\rightarrow$ over the game state σ and an initial program P we obtain an instruction γ and the updated program P'. The second premise (line 2) updates the game state σ into σ' by means of the game-state updating function *update* mentioned earlier.

In order to actually implement the language's semantics these rules must be encoded within our scripting engine function `step` which we have introduced earlier in Section 4.3. This is relatively easy to achieve as we must simply update `step` by adding code which handles the remaining constructs according to the operational semantics just defined. The function `step` is now defined as highlighted here:

```
step :: (Game, Program Instruction) -> (Maybe (Program Instruction), Maybe Instruction)
step (_, Skip) = (Nothing, Nothing)
step (_ ,Perform inst) = ...
step (game,prog1 :> prog2) = ...
step (game,IfThenElse pred prog1 prog2) =
  if (pred game)
    then step (game,prog1)
    else step (game,prog2)
step (game,While pred prog) = ...
step (game,ForThisPiece prog) = ...
```

The function's type signature had to be altered in order to allow `IfThenElse`, `While` and `ForThisPiece` to be evaluated. The game state `Game` was introduced such that the predicates made use of in the latter three constructs can be evaluated upon it. Also, since certain rules may transition without returning an instruction we now return the latter encapsulated inside a `Maybe` structure. As we saw here, creating an interpreter for our DSEL is as simple as defining its operational semantics and translating these into the required function over the data type representing our programs.

4.4.3 Fixed AI Scripting

When a human player plays a game, he or she sees the visual representation of the game state and based on this generates a strategy. This strategy translates into a number of steps which are then carried out one

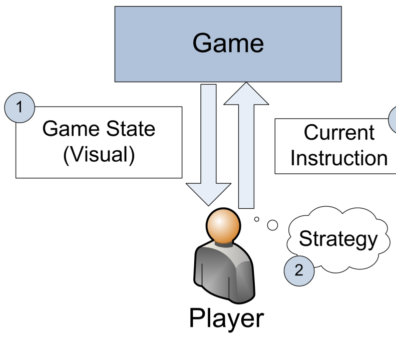


Figure 4.4: A human player playing a game.

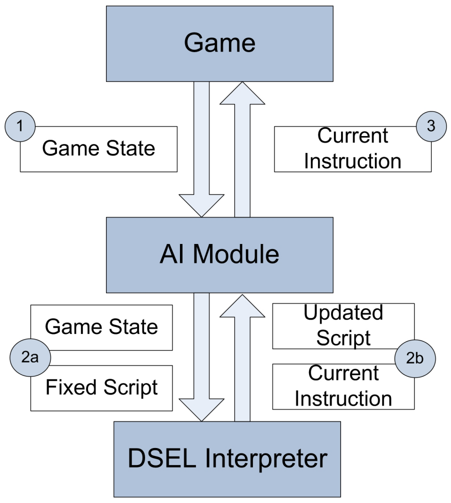


Figure 4.5: An AI player playing making use of fixed scripts.

after the other, automatically. This process is depicted in Figure 4.4. When an AI is involved instead of the player, the first part, that of visual representation, is not required as by means of the API we can extract the required information from the game state. The latter part, that of carrying out the script is carried out by our `step` function. What is still required, is introducing an AI module within our game such that the latter generates the scripts for us. This module will mimic the human player by performing the act of “thinking” by receiving the current game state and returning a fixed AI script. This script and the game state are then passed on to our `step` function which uses them to extract the first instruction to be supplied to the game engine. An updated script is also returned which is retained for the next iteration. In the next iteration, since a script is already present it is processed again to extract the next instruction. This process is repeated until the script finishes at which point the AI must generate a new script for us again. Figure 4.5 shows the initial iteration of this process. Notice how the AI module and the DSEL-interpreter replace the human player. In our implementation, the AI module will be implemented by introducing a simple function which we call `stepAI` consisting of the following code:

```
stepAI :: (Game, Maybe (Program Instruction)) ->
          (Instruction, Maybe (Program Instruction))
stepAI (game, Nothing) = thinkAI game
stepAI (game, Just prog)
  = if (not (isGameOver game))
      then case (step (game, prog)) of
             (maybeProg, Nothing)   -> stepAI (game, maybeProg)
```

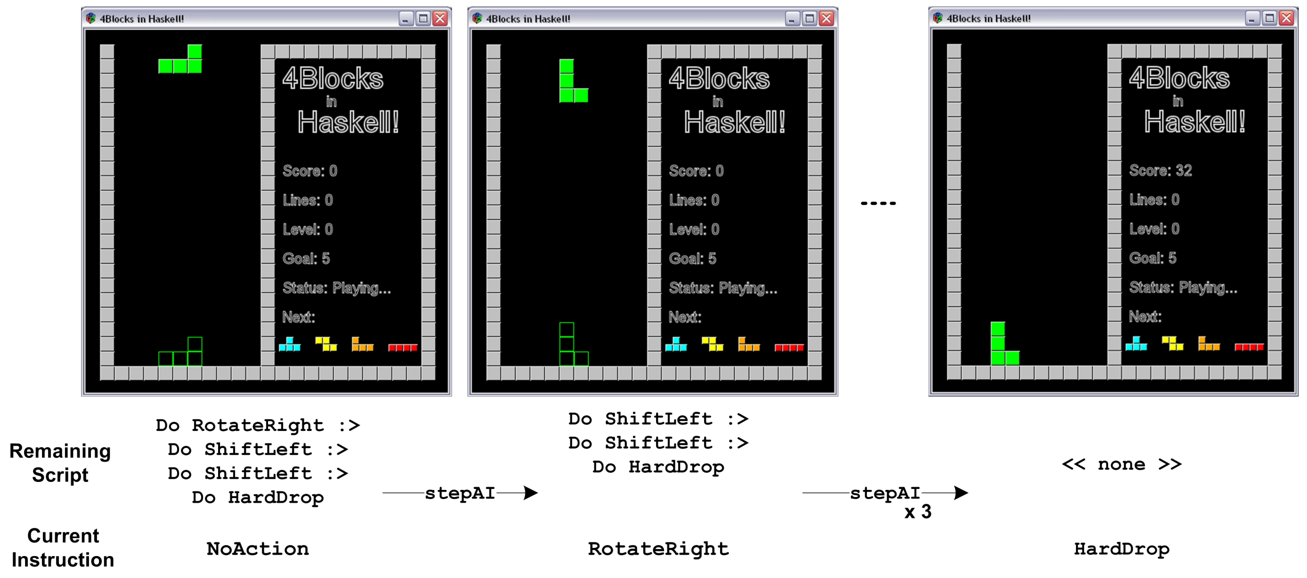


Figure 4.6: Carrying out a simple example script using multiple calls to `stepAI`.

```
            (maybeProg, Just inst) -> (inst, maybeProg)
    else (noAction, Just skip)
```

This function will receive as input the game state and, if it exists, also an outstanding script. It will output an instruction and any remaining script after the former has been extracted. The first instance of `stepAI` triggers when there is no outstanding program and so a new program must be generated. The program generation is performed by an auxiliary function `thinkAI` whose sole purpose is to receive the game state and return the game's predefined AI script and also return a `NoAction` instruction. We are arbitrarily assuming that no action is taken while the AI is "thinking" what it should do. This is, in part, to better copy how a person behaves. The second instance is used when a program already exists. If the game has not ended a call to `step` is performed to extract an instruction from the program. Otherwise, we perform no action and return a program consisting of a skip statement. If `step` returns no instructions we call recursively `stepAI` until finally we receive an instruction. When an instruction is obtained we return it along with the remaining script. As an example we will revisit the example script introduced in Section 4.3, reproduced here:

```
Perform RotateRight :>
Perform ShiftLeft :>
Perform ShiftLeft :>
Perform HardDrop
```

This is the script which our `thinkAI` function will now return. The step-wise result after four calls to `stepAI` is depicted in Figure 4.6 which leads our brick in its final position.

4.4.4 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. Moreover, a human player does not stick to one strategy once the latter is selected. If the game state suddenly changes during the execution of a strategy, which might mean that the current strategy is no longer favourable, the player immediately changes to a strategy better suited for the new situation.

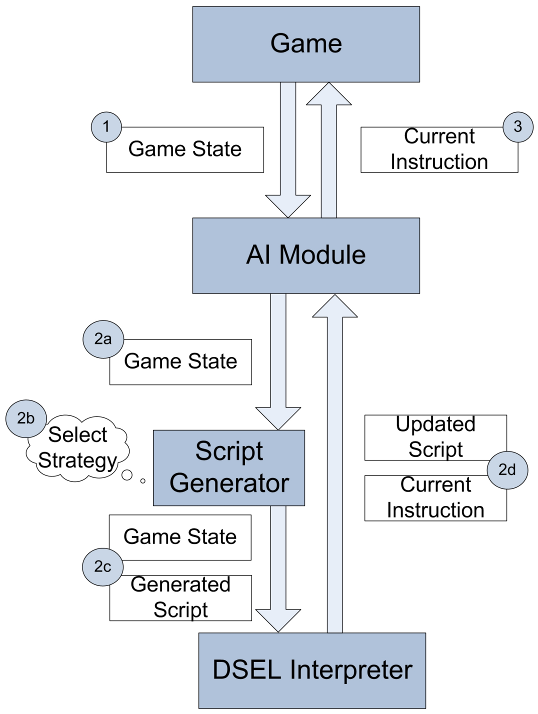


Figure 4.7: An AI player making use of adaptive scripts which are generated by selecting an appropriate strategy based upon the current game state.

On-the-fly strategy-based script generation

The first concept, where different strategies are selected when the need arises, can be mapped to our AI allowing it to generate adaptive scripts based on the current game state. To achieve this we will leverage the fact that our DSEL is embedded within Haskell. Haskell will act as our meta-language such that code written in the latter examines the game state and, based on a number of encoded strategies, generates the required DSEL scripts on-the-fly. This process will take place instead of simply retrieving the same script each time as was done in the previous section.

The design of on-the-fly script generation based on strategies requires us to introduce a new module within our AI which acts as a script generator as shown in Figure 4.7. Now, when the AI module receives the game state the latter is passed onwards to the script generator. The script generator examines the game state and selects a strategy. Based on this strategy it also generates the appropriate script which it then passes, along with the game state to the DSEL interpreter. The DSEL interpreter then extracts the first instruction which is supplied to the game to update it and saves the script as with fixed scripting.

So far our implementation for fixed AI scripts handles naïve script generation using the `thinkAI` function. The only change we need to perform is upon this same function such that it no longer simply fetches a predefined script but rather it selects a strategy from a number of different strategies. Then, based on the selected strategy, it generates automatically the required script. We have thus defined `thinkAI` as follows:

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

think :: Game -> Program Instruction
think game
  | detectCompleteFourLines game
```

```
      = completeFourLinesStrategy game
  | detectCompleteAnyLinesToMinimiseHeight game
      = completeAnyLinesToMinimiseHeightStrategy game
  | ...
  | detectSideFit game
      = sideFitStrategy game
  | otherwise
      = minHeightLowestBestFitStrategy 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 (although this is not necessarily the case). It is however possible to come up with worse or better AI based on the selected strategies and the relative ordering of their corresponding detection predicate. Two example strategies which we employ are to complete four lines whenever the chance prevents itself and to complete lines in order to reduce the well's maximum height as much as possible. The first one is listed amongst the first strategies since it is a strategy which removes the most lines from the well and also gives us the most points. The next strategy attempts to minimise the 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. We proceed in this manner until we reach the default strategy which simply attempts to find the lowest best fit for the current brick. Other strategies are possible of course such as for example catering for the next one or two bricks apart from the current one, equalising the columns' heights within the well to prevent one column to grow to fast, and so on.

The writing of the strategies' detection predicates is made possible by means of the game's API which supplies a number of functions which can be used to query the game state. As an example we will reproduce `detectCompleteFourLines` used to detect the four lines strategy outlined above:

```
detectCompleteFourLines :: Game -> Bool
detectCompleteFourLines game = fourLinesDetected
  where (_, _, _, fourLines) = getLinePossibilities game
        fourLinesDetected = (length fourLines > 0)
```

The function `getLinePossibilities` queries the game state to check all the possible lines a brick might be able to complete at any orientation and position in the well. It returns four lists, one for the fits which complete one line, one for the fits which complete two lines and so on, up to four lines. Since we are interested in completing four lines at once here we are only interested in the last parameter of this function. If this list is of length one or more it means that four lines can be completed and thus the function returns true allowing for this strategy to be selected.

Once a strategy is selected its corresponding script-generating function is used to generate the relevant script. Thus 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 usually followed by these strategy functions is that of first querying the game state for the possible number of brick fit locations. This is once more carried out by means of a number of API functions. 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 specialised functions called parameterised objects (refer to Section 3.8.5) and connection patterns (refer to Section 3.8.6). The first of these generate a script given a number of parameters as input while the latter combine scripts together using a regular patterns. As an example of this approach we provide the script generator for our running example strategy which attempts to complete four lines. It is defined as follows:

```
completeFourLinesStrategy :: Game -> Program Instruction
completeFourLinesStrategy game = generateNormalFitProgram game fLocX orient
  where (_, _, _, fourLines) = getLinePossibilities game
        ((fLocX,_),_,orient) = head $ sortBy (compare ‘on‘ (snd.trip1)) 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 (`fLocX`) and the brick's orientation (`orient`) at that position are saved and supplied to a parameterised object called `generateNormalFitProgram` 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. It is defined as follows:

```
generateNormalFitProgram :: Game -> Double -> Double -> Program Instruction
generateNormalFitProgram game destX orient = finalProgram
  where initX = getBrickXOffset game
        rotatePart = genRotateProgram orient
        shiftPart = genShiftProgram initX destX
        hardDropPart = perform hardDrop
        finalProgram = composeProgram [rotatePart,movePart,hardDropPart]
```

This function makes use of two other parameterised objects. The function `genRotateProgram` generates a program to rotate a brick to the required orientation `orient` while the function `genShiftProgram` generates a program to shift a brick from its initial position (queried from the game state using `getBrickXOffset`) to its final destination `destX`. The latter program simply calculates the offset and generates the number of shifts, left or right, which must be perform to reach the destination x-position. Finally, we make use of a connection pattern named `composeProgram` which takes a list of programs and simply concatenates them one after the other. We use this pattern to create one program consisting of the rotation sub-program, the shifting sub-program and the final hard drop sub-program combined together as our generated script.

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 for us. This was only rendered possible by means of the embedding process which allowed our game scripts to be first-class objects manipulated by the host language Haskell acting as a meta-language.

Changing strategies based on in-game events

In certain games, such as those played with opponents, the game state changes, sometimes drastically, without any intervention by the player themselves. In single-player games this could be caused by a game engine intended event which is designed to make the game more challenging. In multi-player games one never knows how an enemy player reacts to our last move. As suggested earlier, in order to have truly adaptive AI, we need to change it such that when certain events occur we react to them accordingly, sometimes by changing our current strategy and instating a new one.

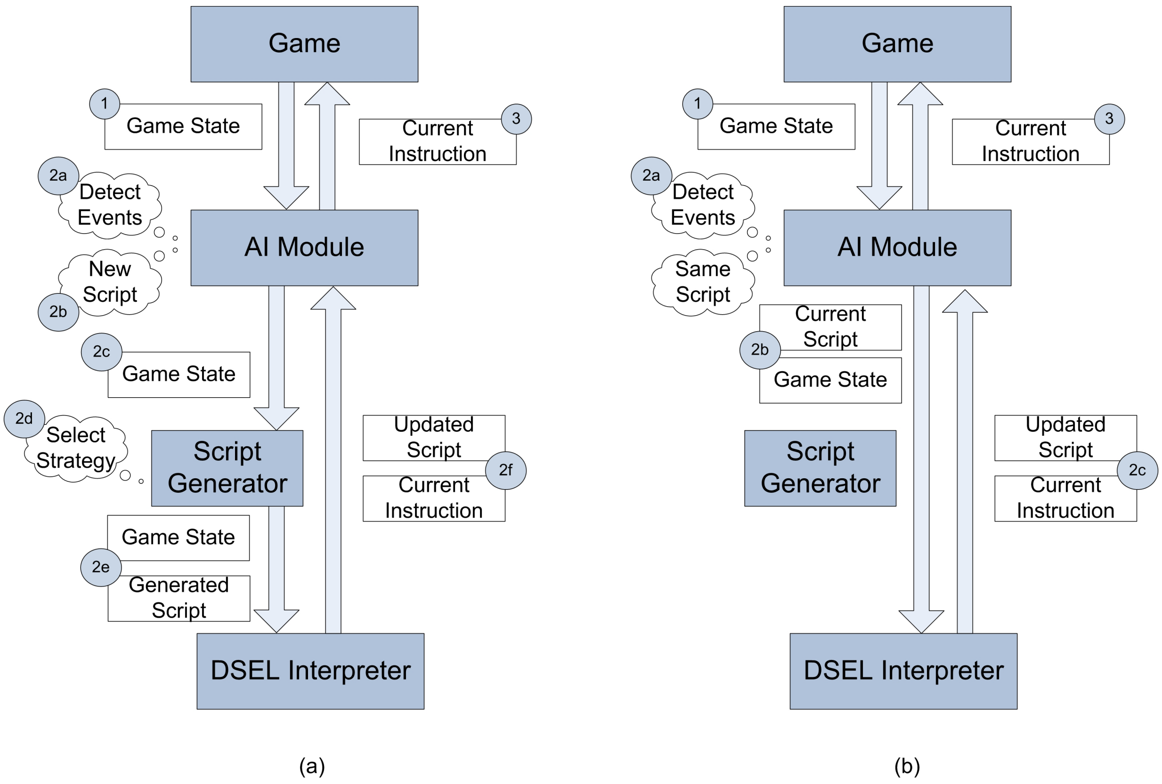


Figure 4.8: An AI player making use of adaptive scripts which are generated by selecting an appropriate strategy based upon the current game state and by reacting to in-game events. (a) Shows what happens when the AI module detects an event: a new script is generated accordingly and run. (b) Shows what happens when the AI module detects that the current script is still valid: script generation is by-passed and the current script is run.

In order to introduce this feature within our AI the AI module must be altered such that it now decides whether a new strategy is needed or not by checking various events. Basically, our `stepAI` function must no longer simply call the `thinkAI` to generate a new script or `step` in order to extract a new instruction, but rather it must also query the game state for any possible events. When an event is detected which requires a change of strategy we call the script generator such that it generates a new script for us which is then interpreted by the DSEL interpreter and used to update the game state as normal. This process is summarised in Figure 4.8 (a). When the AI module detects no event which requires a change to our script we simply by-pass the script generator and interpret the current script, as shown in Figure 4.8 (b).

Coincidentally, we are already detecting an event within `stepAI` which is affecting our script generation. When the game is over we are defaulting to performing no action. This event is being detected prior a call to script generation or interpretation by means of the predicate `isGameOver` upon the game state. We can extend this approach for other events by including other predicates here with relative ease. Unfortunately for a single player game such as 4Blocks, changing strategies due to events is only viable by a forced example such as by introducing random 1-block bricks in the well which appear and disappear or by changing the current well landscape from time to time, thus forcing re-evaluation of our strategies. Due to this we have not deeply tested this approach with this game but rather reserved it for our second case study where our game allows two or more players to affect the same game state simultaneously.

4.5 Results and Discussion

Our results show that using embedded languages as a means of scripting AI is a very viable approach. First of all, by means of the use of a scripting language, it shares the advantages which the proprietary scripting language method and the general-purpose scripting language method have over game-engine integration. As stated earlier, having a language means that we allow a separation of concern between the actual game engine and the game-content. This also enables us to provide a plug-in mechanism where both story-writers and AI-programmers may write their scripts in stand-alone modules which are then swapped in and out accordingly. For example, for 4Blocks we can supply the knowledge of the scripting language constructs and a number of API functions to different AI-programmers who know Haskell and the latter can write their own AI for the game.

The advantage our method has over general-purpose scripting languages and proprietary scripting languages, however, is how quick and easy it is to implement the required scripting language when compared to the processes which the former require to achieve the same result. As we saw once the design phase is complete, the language may be implemented in a very straightforward manner within a host language instead of from scratch. Moreover, as we saw here, updating the language is also simple once the BNF of the language and its operational semantics are finalised. The data structure representing the language is updated in order to reflect the new syntax while the interpretation function (in our case `step`) is updated to include the new semantics. In traditional language-creation techniques this requires quite a considerable amount of work in comparison.

Another advantage our approach has over the other two methods is that we can decide which part of the language to expose. In the case of story-writers we can expose our initial language which consists of domain-related *performs* following one another. Haskell's module system allows us to achieve this with relative ease. For AI-programmers we can expose the entire language since they understand more advance programming constructs. Thus our language can be tailored towards its particular target audience accordingly and does not have to necessarily be tied to programmers or non-programmers exclusively, as was the case with general-purpose scripting languages (which are for programmers only) and proprietary-scripting languages (which are for programmers or non-programmers depending on the language itself).

In regards to AI programming, language embedding allows us to make use of various abstraction levels as required. This is another advantage which it has when compared to other scripting solutions. If needed we can program directly in the embedded language as we did with fixed scripting. However, the true power of embedding shows itself when we have to abstraction levels by combining the scripting language with the host language. Not only does this allow us to make use of a full, general-purpose language to encode our AI but it also allows the host language to act as a meta-language which manipulates and generates the AI scripts for us at run-time as we saw when we used parameterised objects and connection patterns in adaptive scripting.

We have tested the approach by writing the strategies shown earlier and performing a number of test runs. Over a number of these runs the AI has achieved both negative and positive results with the former having an upper bound of reaching at least Level 4 (75+ lines completed) and the latter being able to reach on some occasions more than 600 lines. We believe that the results obtained are comparable to that of an intermediate-level human player. (The writer wrote the strategies himself and is in fact an intermediate level Tetris player). We believe that introducing more strategies and improving our current strategies could allow us to reduce the negative results and improve our AI but this goes beyond the scope of this thesis.

4.6 Conclusion

As we saw here, creating a scripting language using the embedded language approach has a number of benefits. First of all, language creation is a very straightforward process when compared to more traditional approaches. Moreover, we can expose only the DSEL portion of the language to the story-writers and the full language to AI writers. Also, using Haskell's module system we can enforce a plug-in system. Finally, we have the added advantage of meta-programming which allows for on-the-fly adaptive AI scripting, something which the other two scripting methods lack. In the following chapter we shall discuss another case study which consists of a much more complex turn-based strategy game. As we shall see the techniques used in this chapter may be adapted for such a game with relative ease. Moreover, we will show how we may embed two DSELs within Haskell which communicate together using the latter language as a form of common ground. We achieve this either by means of language frameworks, which allow us to import a language within another, or by introducing a number of constructs relating to the domain of message-passing within our scripting languages.

Chapter 5

Case Study: Space Generals

5.1 Introduction

Our previous game, 4Blocks, is very limited in its domain since it is quite a small game. In order to truly test our approach and to see whether it scales up to more sophisticated features we decided to implement another, more complex game, called Space Generals.

Space Generals is a turn-based game which will require us to create two scripting languages in order to write the AI of two different types of agents present within the game. These agents need different languages because their roles are completely different and are at different levels of abstraction within the game itself. Here, we will show how these languages may be easily implemented and appropriately typed by means of the embedded language techniques utilised for 4Blocks and stronger typing techniques mentioned in earlier chapters. Also, using the techniques of connection patterns and parameterised objects we shall show how Haskell can be made use of in order to generate the required AI scripts for us at run-time based on a number of encoded strategies. Space General's adaptive AI will be created this way and also improved upon by adding the notion of event-detection. The latter allows us to write functions in Haskell upon the game state which if triggered may be used to change our scripts half-way through execution. Creating these languages and their adaptive AI will allow us to re-verify the applicability of our approach in a game which is more involved and which requires more complex strategies.

Another reason why we believe Space Generals is a good case study is that it allows us to test whether we can include additional language features within our scripting languages, and if so, whether this can be achieved with ease such that it is amenable to do so within the game's creation life-cycle. Two features which we are considering are parallel composition and concurrency. The first is required since the agents within Space Generals may perform more than a task at a time. This means that in order to appropriately script the agents we are required to include this functionality as part of the scripting languages' features. Concurrency, on the other hand, is required since two agents may work together and with a common goal in this game. Thus a way by means of which they coordinate and synchronise their efforts must be introduced within the scripting language employed. To achieve this we have decided to include concepts from the domain of concurrency related to message-passing within our scripting language. Also, since one of the agent types has jurisdiction over the other we have included a way by means of which direct orders are given. Adding direct orders requires the language of one of our agent types to directly alter the script of the other agent type. We will show how language frameworks discussed in Chapter 3 allow us to achieve this by importing a scripting language within another.

5.2 Space Generals

Space Generals is a turn-based game in the style of Risk which pits two to five generals against each other with the final objective of controlling all the known galaxies of the universe. Each player takes the role of a general who leads a number of captains. A general has a high-level view of the game and concerns themselves with managing galaxies and their respective planets. Each captain, on the other hand controls one planet at a time and manages the planet's various continents and their respective countries. The generals (and their respective captains) take their turns simultaneously and each time the game-board is updated at one go to reflect the next turn.

In a turn a general can reassign their captains to a particular planet they feel needs attention such as for defending it or for using it to stage an attack. They can also support the captains and supply them with resources to aid in their effort. The resources available are credits (money), population and armies. Each turn the general receives bonus income from the galaxies and planets under their control in order to have supplies to help them in their war-effort. Captains also receive bonus income in their own way from the planet which they are currently controlling. The amount of such income which they receive depends on the number of continents and countries within the planet which they control. They can also directly withdraw credits and population from the countries under their control. The less is withdrawn from a country the more its credits and population grow so withdrawing should be done wisely and at the most opportune moments such as when one is sure that the country is going to be lost to the enemy. Countries can be used to station armies and are linked to each other. Some links are between countries on the same planet while others link countries in different planets and even different galaxies. Linked neighbouring enemy countries can be attacked by means of the armies which a captain deploys in a friendly country. Obviously countries which link to other planets and galaxies are harder to defend but provide very strategic locations to control other territories.

When the generals and captains perform their turns the game is updated by resolving all the attacks. Attacks are resolved using dice throws but can be affected by the number of armies involved. Say an army of 20 units has a chance advantage over an army of 5 but if the latter is really lucky it can win. This is done in order to better simulate battles. This carries on until one of the generals becomes the ruler of the known universe by successfully defeating all of his or her opponents. Figure 5.1 shows two screenshots from a typical game. Space Generals is implemented using Google Web Toolkit for its front-end and Haskell for its back-end. The game can be found online[1]. More information about the game can be found in Appendix A.

5.3 Language Syntax

5.3.1 Requirements

For Space Generals, two languages are required due to the fact that we have two different types of agents within the game: one language for generals and another language for captains. The two languages should contain general-purpose language constructs such as conditional statements and looping statements very similar to the ones found in the scripting language for 4Blocks in order to be expressive enough for AI scripting. A feature which they must also include is parallel composition. This type of composition was not included for 4Blocks because only one action could be performed at a time. In Space Generals, this is not the case. A captain for example, can withdraw resources from a country and use them to buy armies in the same turn.

[1]http://spacegenerals.servegame.com

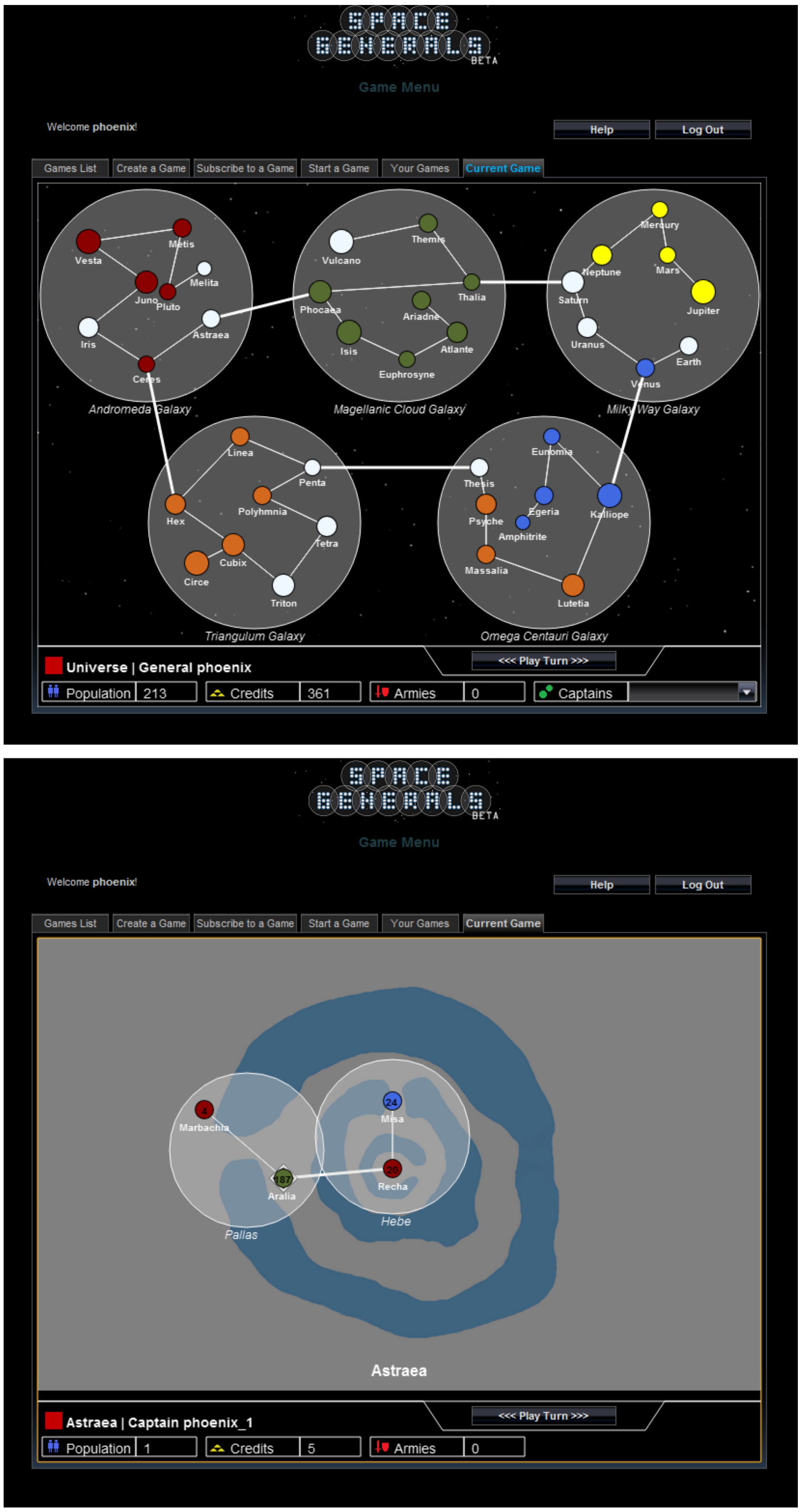


Figure 5.1: Screenshots showing a typical game of SpaceGenerals. Above: Star-map, shows all the planets and galaxies, Below: A Planet-map for planet Astraea, showing its continents and countries.

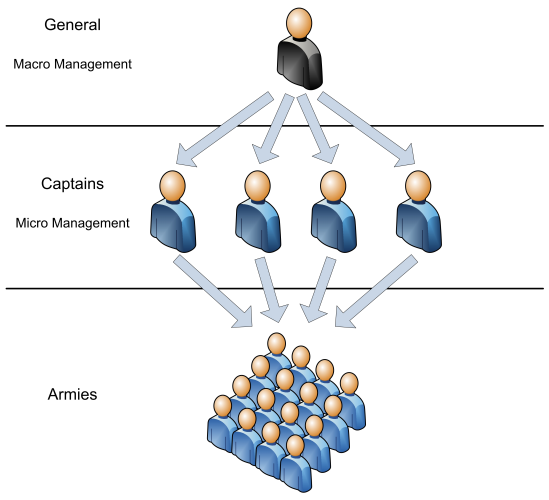


Figure 5.2: Military hierarchy of power employed in Space Generals.

5.3.2 Design

In order to design a language for Space Generals, we first identified the basic actions which the generals and captains are allowed to perform, as was done for 4Blocks. These actions constitute the domain-specific aspect of the game which must be added as language constructs by default. They could be used by story-writers to script the game but are however not enough to encode AI. There are different categories of actions since the generals and captains have a different viewpoint of the game which is based on a simple military hierarchy of power as shown in Figure 5.2. While the general is concerned with macro (high-level) management involving the captains themselves and planets and galaxies, the captains are concerned with micro (low-level) management involving the armies which fight for their general and the countries and continents of the planet where they are stationed. Finally, both generals and captains are allowed to manage their respective personal resources. This leads us to the following three categories of actions: general actions, captain actions and common actions.

A general's actions mainly concern the management of captains and the universe's planets and galaxies:

- **Commission Captain** – Train a new captain, each time a captain is commissioned the cost of the next captain grows exponentially.
- **Reassign Captain** – Move a captain from one planet to another.
- **Retire Captain** – Retire a captain gaining half of his or her cost.
- **Support Captain** – Support a captain by supplying him or her with credits, population and armies.
- **Upgrade Planet** – Upgrade a planet to increase the bonus received from it if controlled.
- **Upgrade Galaxy** – Upgrade a galaxy to increase the bonus received from it if controlled.

The actions which a captain can perform concern his or her armies and the countries and continents of the planet where he or she is stationed. A captain can also support his or her general by supplying the latter with resources:

- **Attack with Armies** – Use armies to attack a neighbouring enemy country from a friendly country.
- **Deploy Armies** – Deploy armies into a friendly country.
- **Move Armies** – Move armies from a friendly country into a neighbouring friendly country.

- **Pay Tribute** – Send resources back to general.
- **Upgrade Country** – Upgrade a country to increase the bonus received from it if controlled.
- **Upgrade Continent** – Upgrade a continent to increase the bonus received from it if controlled.
- **Withdraw From Country** – Withdraw the country's generated resources (credits and population).

The actions which are common to both generals and captains are related to resource management and deal with buying armies, selling them (credits refund) or disband them (population refund):

- **Buy Armies** – Convert 10 credit units and 10 population units into 5 army units, (10 Credits + 10 Population → 5 Armies)
- **Disband Armies** – Convert 5 army units back into 10 population units, (5 Armies → 10 Population)
- **Sell Armies** – Convert 5 army units back into 10 credit units, (5 Armies → 10 Credits)

Therefore both levels of command can change the army size but the general has no other means of controlling their armies except via captains.

Having thus discussed the domain-specific aspect of the language, we now turn to the general-purpose constructs which will allow us to encode AI strategies. The first construct which we shall create within our languages is the one which actually allows a general or a captain to perform the actions described above. This construct which we named *perform* takes the following form:

Perform (`DeployArmies ...`)

where `DeployArmies` represents one of the above actions. The purpose of this construct is thus to encapsulate domain-related actions and lift them up to be used within the AI languages.

The next construct we shall introduce is that of sequential composition. Since the game consists of turns we require a way to indicate that an action will occur in this turn as opposed to the next turn or any other future turns. Thus actions separated by sequential composition are carried out in a by-turn basis. This construct should thus takes the following form:

```
Perform (DeployArmies ...) ;
Perform (MoveArmies ...)
```

where `MoveArmies` represents another one of the above actions. Thus we wish to signify, by means of the above prototype script, that in the current turn a captain will deploy his armies inside a country and in the next turn they will move some armies from one country into another. Adding more *performs* using sequential composition should thus script a captain for any number of subsequent turns.

The constructs designed so far were virtually identical to the ones for 4Blocks. The next construct however is new. Parallel composition is required within the scripting languages for Space Generals since, as stated earlier, a turn may constitute more than one action at a time. We thus need a construct which allows us to signify that a number of actions are composed in such a way as to be carried out in the same turn. Moreover, there should be no limit to how many actions can be done in parallel. Therefore, parallel composition should take a format similar to sequential composition in structure but totally different in meaning, as follows:

```
Perform (DeployArmies ...) /
Perform (MoveArmies ...)
```

Thus in the current turn a captain would deploy his armies inside a country and also move some armies from one country into another. Combining both compositions together will allow us to write more interesting scripts, such as for example:

```
(Perform (DeployArmies ...) / Perform (MoveArmies ...)) ;
(Perform (DeployArmies ...) / Perform (MoveArmies ...))
```

Here our captain will deploy his armies and move them in this turn and also do this again in the following turn.

The next construct we will design is the one which will allow us to branch our scripts based on decisions. The *if-then-else* statement introduced for 4Blocks will also be introduced for both our generals and captains in order to allow us to react to queries upon the game state. For example, we can script our captains to withdraw ten credits from a country if the latter has more than a hundred credits. Otherwise we withdraw only five credits. This could be written with a construct taking the following form:

```
If (countryMoreThanHundredCredits)
   Then (Perform (WithdrawFromCountry 10 ...))
   Else (Perform (WithdrawFromCountry 5 ...))
```

where `WithdrawFromCountry` is a captain construct while `countryMoreThanHundredCredits` is a predicate which makes use of an API function to obtain the required information from the game state. In order to allow for easier use of this construct instead of using just predicates it would be ideal to introduce an expression sub-language with Boolean primitives and the basic operators of conjunction, disjunction and negation. This language should also include a construct, *check*, with the purpose of actually doing the predicate check upon the game state. Thus, as an example consider:

```
If ((Check countryMoreThanHundredCredits) && (Check countryLessThanThousandCredits))
   Then (Perform (WithdrawFromCountry 10 ...))
   Else (Perform (WithdrawFromCountry 5 ...))
```

where `countryLessThanThousandCredits` is a predicate which checks whether a country has less than a thousand credits and && represents our conjunction operator. Here we want to signify that we wish to withdraw ten credits if the country has between a hundred and a thousand credits. If included, the expression sub-language will allow our *if-then-else* statement to become more expressive.

In order to repeat certain scripts we will again make use of a *while* statement. Like in 4Blocks we require the construct to take a program and a condition as input but only run the script if the condition is met. The condition may be defined using the expression sub-language suggested earlier, allowing this construct to take a form such as this:

```
While (countryMoreThanHundredCredits)
   (Perform (WithdrawFromCountry 5 ...))
```

Here, until the condition is no longer met, the captain will withdraw five credits from a country each turn. This construct will thus allow us to write scripts which span more than one turn if required and is very useful in game such as Space Generals where certain actions are required to take place each turn, such as harvesting resources and buying armies.

The final language construct we require is *skip*. Again it will act as a place-holder program which is simply skipped over but which is very useful for cases where one of the branches of the `if-then-else` statement needs to be ignored. It is common for example in the game to withdraw resources from a country at a certain threshold and otherwise do nothing at all. Using *skip* we could write this as follows:

```
If (countryMoreThanHundredCredits)
   Then (Perform (WithdrawFromCountry 10 ...))
   Else Skip
```

Our final language syntax for our game's scripting languages can be defined using the following BNF:

program$_{<C,G>}$ ::= **skip**
  | **perform** *instruction*
  | *program* **;** *program*
  | *program* **/** *program*
  | **if** *expression* **then** *program* **else** *program*
  | **while** *expression* *program*

Where *program*$_{<C,G>}$ refers to the scripting language for captains and generals, respectively.

instruction ::= **attackArmies** *string string integer string string*
  | **buyArmiesCaptain** *string string integer*
  | **buyArmiesGeneral** *string integer*
  | **commissionCaptain** ... | **deployArmies** ... | **disbandArmiesCaptain** ...
  | **disbandArmiesGeneral** ... | **moveArmies** ... | **payTribute** ...
  | **reassignCaptain** ... | **retireCaptain** ... | **sellArmiesCaptain** ...
  | **sellArmiesGeneral** ... | **supportCaptain** ... | **upgradeCountry** ...
  | **upgradeContinent** ... | **upgradePlanet** ... | **upgradeGalaxy** ...
  | **withdrawFromCountry** ...

expression ::= **true** | **false** | **not** *expression*
  | *expression* **<&&>** *expression*
  | *expression* **<||>** *expression*
  | **check** *predicate*

predicate ::= *... various API calls upon the game state ...*

where *string* and *integer* refer to Haskell's own strings and integers which are available since we are embedding our scripting language in the latter. An example of the use of this syntax is the following program which scripts a captain:

while (**not** (**check** (**underAttack** Earth))) (
  perform (**withdrawFromCountry** General1 Capt2 Malta 10 10) **-|-**
  perform (**buyArmiesCaptain** General1 Capt2 Malta 5)
)
->- **perform** (**attackArmies** General Capt2 100 Malta Xenon)

For this program, let us assume that the captain's name is Capt2 and that he is assigned to planet Earth. A country on Earth, Malta is linked to another country Xenon on Mars. Using the program's while loop and the predicate over the game state **underAttack** we can first check whether Earth is under attack. If it is not, each turn, until the loop is false, we collect ten credits and ten population from it and use these to buy five armies in the same turn (since we make use of parallel composition to link the two instructions). Once the predicate is false the loop is no longer run and for one turn the captain performs no action. In the next turn however, since we are using sequential composition, they will perform a counter-attack on Xenon.

On the other hand, an example general script can be written as follows:

if (**check** (**surplusResources** Earth))
  then (**perform** (**reassignCaptain** General1 Captain1 Earth))
  else skip

The script scopes over one turn only and is then consumed. It attempts to check whether Earth has surplus resources using the **surplusResources** predicate and if it does assigns a captain there to manage its resources, otherwise no action is perform in this turn (using **skip**).

5.3.3 Implementation via embedding

Implementing the required syntax into Haskell via embedding is once more quite straightforward and simply requires the translation of the BNF into the equivalent data types. The generals' scripting language will be embodied in the `GeneralProgram` data type defined as follows:

```
data GeneralProgram
  = GPerform Instruction
  | GeneralProgram :>>: GeneralProgram
  | GeneralProgram :||: GeneralProgram
  | GIfThenElse Expression GeneralProgram GeneralProgram
  | GWhile Expression GeneralProgram
  | GSkip
```

Similarly for the captains' language `CaptainProgram`:

```
data CaptainProgram
  = CPerform Instruction
  | CaptainProgram :>: CaptainProgram
  | ...
```

Each of the game-related actions were encoded in the data type `Instruction` as follows:

```
data Instruction
  = AttackArmies {generalName :: Name, captainName :: Name, armies :: Armies,
                  fromCountryName :: Name, toCountryName :: Name}
  | BuyArmiesCaptain {generalName :: Name, captainName :: Name, armies :: Armies}
  | BuyArmiesGeneral {generalName :: Name, armies :: Armies}
  | CommissionCaptain {generalName :: Name, captainName :: Name, planetName :: Name}
  | ...
```

Finally, the expression sub-language was added by the `Expression` data type as follows:

```
data Expression
  = B Bool
  | Not Expression
  | Or Expression Expression
  | And Expression Expression
  | Check Predicate
```

Basically this is all the code we require in order to implement the scripting languages for Space Generals. There are a number of issues however which we have encountered and which we will now discuss.

Stronger-typing The GPerform and CPerform constructs encapsulate an instruction to be performed by a general and a captain, respectively. Due to this GPerform needs to accept only instructions available to the general while CPerform must only accept captain-specific instructions. Moreover, common instructions should be available to both. Since Instruction is itself untyped in this regard we cannot make use of it alone. To allow the required behaviour we shall make use of phantom types as we saw earlier in Section 3.8.4. To do this we have created an auxiliary data type which we call Action defined as follows:

```
newtype Action a = Action Instruction
```

An Action simply encapsulates an Instruction and requires a type to be supplied. This latter type will allow us to categorize our instructions and subsequently embed typing within the two languages. To make use of Action we will provide a number of constructor functions which will now replace the Instruction type constructors as our constructs. Each of these will return an object of type Action Captain or Action General where Captain and General can be dummy data types or even the game's own representations of a captain or general:

```
-- Captain and General as dummy data types
data Captain = Captain
data General = General

-- Captain-specific Instruction
attackArmies' :: String -> String -> Integer -> String -> String -> Action Captain
attackArmies' generalName captainName armies fromCountryName toCountryName
  = Action (AttackArmies generalName captainName armies fromCountryName toCountryName)

-- General-specific Instruction
commissionCaptain' :: String -> String -> String -> Action General
commissionCaptain' generalName captainName planetName
  = Action (CommissionCaptain generalName captainName planetName)

-- Common Instruction "Buy Armies"
buyArmiesCaptain' :: String -> String -> Integer -> Action Captain
buyArmiesCaptain' generalName captainName armies
  = Action (BuyArmiesCaptain generalName captainName armies)

buyArmiesGeneral' :: String -> Integer -> Action General
buyArmiesGeneral' generalName armies
  = Action (BuyArmiesGeneral generalName armies)

...
```

The captain-specific instruction AttackArmies shown here is now constructed using the constructor function attackArmies. The latter constructs an untyped AttackArmies instruction and returns it within an Action.

The actual type-enforcing is done via the function's type definition which now returns `Action Captain` instead of an `Instruction`. This is similar for general-specific instructions such as `CommissionCaptain` shown above which now is defined as being of type `Action General`. For common instructions we simply provide two functions, one for each, which return the captain-specific or general-specific actions as shown in the last two actions above. Our `GPerform` and `CPerform` constructs now obtain the following definition:

```
data GeneralProgram
  = GPerform (Action General)
  | ...

data CaptainProgram
  = CPerform (Action Captain)
  | ...
```

in this way allowing our languages to be more appropriately typed. In fact, since both languages have the same constructs and are now typed in this manner we can simplify our implementation by unifying `GeneralProgram` and `CaptainProgram` into one data type as follows:

```
data Program a
  = Perform a
  | Program a :>: Program a
  | ...
```

where general scripts make use of the type `Program (Action General)` and captain scripts make use of the type `Program (Action Captain)`.

Language interface As was done previously with 4Blocks, we have provided a number of constructor functions in such a way as to provide a unified and polished interface to the language. Our expression sub-language will be accessible via the constructor functions `true'`, `false'`, `not'` (negation), `<&&>` (conjunction), `<||>` (disjunction) and `check'`. For the language itself we shall use a similar method since it allows us to introduce new constructs without inflating our basic syntax. Moreover, this will allow us to create a *forever* construct defined in terms of *while* and also will enable us to add additional keywords such as *then*, *else* and *do* to the *if-then-else* and the *while* statements in order to make them more user-friendly. Thus:

```
perform' :: Action a -> Program (Action a)
perform' action = Perform action

skip' :: Program (Action a)
skip' = Skip

while' :: Expression -> Do -> Program (Action a) -> Program (Action a)
while' exp Do prog = While exp prog

forever' :: Program (Action a) -> Program (Action a)
forever' prog = While true' prog

(->-) :: Program (Action a) -> Program (Action a) -> Program (Action a)
(->-) = (:>:)
```

```
(-|-) :: Program (Action a) -> Program (Action a) -> Program (Action a)
(-|-) = (:|:)

if' :: Expression -> Then -> Program (Action a) -> Else -> Program (Action a)
                                                        -> Program (Action a)
if' exp Then prog1 Else prog2 = IfThenElse exp prog1 prog2
```

where `Do`, `Then` and `Else` are dummy data types used to make the language constructs easier to understand and use. We shall use the constructor functions `do'`, `then'` and `else'` to construct these.

Polished language constructs One of the minor issues we encountered was related to the keywords used within our languages as they conflicted with Haskell's own keywords. This was mostly noticed when we attempted to create the *if-then-else* keywords such as `if`, `then` and `else` and also most of the constructor functions used as constructs instead of making use of the `Expression` data type directly such as `true`, `false`, `not` (negation), `&&` (conjunction), `||` (disjunction) and so on. To alleviate this problem we took the convention of writing all language keywords using their respective name and adding an apostrophe as suffix. For operators we encase them in opening and closing angle brackets such as for example `<&&>` and `<||>`. Despite being a minor detail we feel it detracts from giving the embedding language the feel of an actual language. Other ways to fix this do exist as we discuss later on in our future work.

We can thus use the above language to re-write the captain scripts and general scripts given previously as examples in BNF form as follows:

```
capt2Prog =
  while' (not' (check' (underAttack "Earth"))) do' (
    perform' (withdrawFromCountry' "General1" "Capt2" "Malta" 10 10) -|-
    perform' (buyArmiesCaptain' "General1" "Capt2" 5)
  )
  ->- perform' (attackArmies' "General1" "Capt2" 100 "Malta" "Xenon")

general1Prog =
  if' (check' (surplusResources "Earth"))
    then' (perform' (reassignCaptain' "General1" "Captain1" "Earth"))
    else' skip'
```

5.4 Language Semantics

5.4.1 Requirements

The semantics of our scripting language can be defined in terms of the constructs used to define our general and captain programs. We thus need to provide the operational semantics for both of these languages' *perform* constructs, *sequential composition* constructs, *parallel composition* constructs, *if-then-else* constructs, *while* constructs and *skip* constructs.

5.4.2 Design

In order to define the language semantics we must first define some preamble. The set of basic actions defined earlier which a general and a captain can perform are denoted by A_G and A_C, respectively. The sets of all possible general programs and all possible captain programs are denoted by Prg_G and Prg_C, respectively. The empty program is denoted by $\bullet$, such that:

$$Prg_G^+ \triangleq Prg_G \cup \{\bullet\}$$

$$Prg_C^+ \triangleq Prg_C \cup \{\bullet\}$$

The set of game states is represented by Σ where each game state σ contains the information regarding a game's generals, captains, countries, continents, planets and galaxies. Given σ we will assume a predicate c over the state which can be resolved by $\sigma(c)$. A general's program transitions into another by means of a transition relation defined as:

$$\rightarrow_G \subseteq Prg_G^+ \times \Sigma \times \mathcal{P}^{A_G} \times Prg_G^+$$

As can be noted from this definition, a general's program requires a game state and a number of basic actions in order to evolve into another program using such a transition relation. Thus an instance of the transitional relation takes the form:

$$P \; \xrightarrow[\sigma]{I}_G \; P'$$

where $P \in Prg_G^+$ is the script before transition, $P' \in Prg_G^+$ is the updated script, post transition, $I \in \mathcal{P}^{A_G}$ is the set of current instructions and $\sigma \in \Sigma$ is the current game state. An example of the use of this relation is as follows:

$$\textbf{(Perform (BuyArmies} \ldots\textbf{)} :|: \textbf{Perform (SellArmies} \ldots\textbf{))} :>: \textbf{(Perform (BuyArmies} \ldots\textbf{))}$$

$$\xrightarrow[\sigma]{\textbf{\{BuyArmies,SellArmies\}}}_G$$

$$\textbf{(Perform (BuyArmies} \ldots\textbf{))}$$

The transition extracts two instructions by consuming the part of the program intended for the current turn (the sub-program prior to the sequential operator). The sub-program after the sequential operator is returned as the updated program. A similar approach is taken for captains.

We are also going to assume two functions which transition a game state into an updated game state. One of these functions, $update_G$ performs an update on the game state with the actions performed by a general. It has the following type definition:

$$update_G : (\Sigma \times \mathcal{P}^{A_G}) \rightarrow \Sigma$$

It takes two inputs: the initial game state and the set of basic actions which the general is going to perform in order to transition the initial game state into an updated one. It outputs the updated game state. The second function is $update_C$ with the following type definition:

$$update_C : (\Sigma \times (C \rightarrow \mathcal{P}^{A_C})) \rightarrow \Sigma$$

This function performs an update on the game state with the actions performed by a captain C. It takes two inputs, the initial game state and a function which maps a captain to their set of basic actions and returns as output the updated game state.

The language semantics for the skipping construct, the perform construct, the sequential composition construct, the if-then-else construct and the while construct are virtually identical to the ones shown for 4Block's scripting language in Section 4.4.2 and hence require no further explanation:

Skip

$$\frac{}{\textbf{Skip} \; \xrightarrow[\sigma]{\varnothing}_G \; \bullet}$$

Perform

$$\frac{}{\textbf{Perform}\ \alpha \xrightarrow[\sigma]{\{\alpha\}}_G \bullet}$$

Sequential Composition

$$\frac{P_1 \xrightarrow[\sigma]{\gamma}_G \bullet}{P_1 \textbf{ :>: } P_2 \xrightarrow[\sigma]{\gamma}_G P_2} \qquad \frac{P_1 \xrightarrow[\sigma]{\gamma}_G P_1'}{P_1 \textbf{ :>: } P_2 \xrightarrow[\sigma]{\gamma}_G P_1' \textbf{ :>: } P_2}\ P_1' \neq \bullet \qquad \frac{P_1 \xrightarrow[\sigma]{\varnothing} \bullet \qquad P_2 \xrightarrow[\sigma]{\gamma} P_2'}{P_1 \textbf{ :>: } P_2 \xrightarrow[\sigma]{\gamma} P_2'}$$

If-Then-Else

$$\frac{P_1 \xrightarrow[\sigma]{\gamma}_G P_1'}{\textbf{IfThenElse}\ c\ P_1\ P_2 \xrightarrow[\sigma]{\gamma}_G P_1'}\ \sigma(c) \qquad \frac{P_2 \xrightarrow[\sigma]{\gamma}_G P_2'}{\textbf{IfThenElse}\ c\ P_1\ P_2 \xrightarrow[\sigma]{\gamma}_G P_2'}\ \neg\sigma(c)$$

While

$$\frac{P_1 \xrightarrow[\sigma]{\gamma}_G P_1'}{\textbf{While}\ c\ P_1 \xrightarrow[\sigma]{\gamma}_G P_1' \textbf{ :>: } \textbf{While}\ c\ P_1}\ \sigma(c), P_1' \neq \bullet \qquad \frac{P_1 \xrightarrow[\sigma]{\gamma}_G \bullet}{\textbf{While}\ c\ P_1 \xrightarrow[\sigma]{\gamma}_G \textbf{While}\ c\ P_1}\ \sigma(c)$$

$$\frac{}{\textbf{While}\ c\ P_1 \xrightarrow[\sigma]{\varnothing}_G \bullet}\ \neg\sigma(c)$$

Parallel composition is defined as follows:

Parallel Composition

$$\frac{P_1 \xrightarrow[\sigma]{\gamma_1}_G P_1', \quad P_2 \xrightarrow[\sigma]{\gamma_2}_G P_2'}{P_1 \textbf{ :|: } P_2 \xrightarrow[\sigma]{\gamma_1 \cup \gamma_2}_G P_1' \textbf{ :|: } P_2'}$$

This rule states that in order to evolve a program which consists of two programs composed in parallel to one another, we must first know how they evolve separately. The product of these two separate steps is two sets of actions. The resultant evolution of the entire program consists of the two updated programs composed together in parallel and also a set consisting of the union of the two sets of actions produced separately.

The operational semantics of the captains' scripting language are very similar to those shown here. The only changes required are in using the corresponding keywords and the transition relation $\rightarrow_C$.

Turn

We will also provide the operational semantics of how a turn evolves since it combines the general's and captain's transition relations just described. A turn consists of four steps:

1. The general's program is interpreted to extract a number of instructions and obtain an updated program.
2. Using general's instructions the game's state is updated into an intermediate state.
3. The captains' programs are then interpreted in parallel to extract a number of instructions and obtain the respective updated programs.
4. Finally, the captains' instructions are pushed together upon the intermediate game state in order to obtain the final game state.

We shall represent a turn by the following transition relation:

$$\rightarrow_T \subseteq Prg_G^+ \times (C \rightarrow Prg_C^+) \times \Sigma \times Prg_G^+ \times (C \rightarrow Prg_C^+) \times \Sigma$$

which maps a general program, a function which maps captains to their respective programs and a game state to updated post-turn versions of these.

A turn thus has the following semantics:

$$\frac{\begin{array}{ll} P_g \ \underset{\sigma}{\overset{\Gamma}{\rightarrow}}_G \ P'_g & \textit{(line 1)} \\ \sigma'' = update_G(\sigma, \Gamma) & \textit{(line 2)} \\ \gamma_{cs} = \bigcup\{(c,\gamma) \mid \exists P_c . FP_c\ c \ \underset{\sigma''}{\overset{\gamma}{\rightarrow}}_C \ P_c\} & \textit{(line 3)} \\ \forall c . \exists \gamma . FP_c\ c \ \underset{\sigma''}{\overset{\gamma}{\rightarrow}}_C \ FP'_c\ c & \textit{(line 4)} \\ \sigma' = update_C(\sigma'', \gamma_{cs}) & \textit{(line 5)} \end{array}}{((P_g, FP_c), \sigma) \ \rightarrow_T \ ((P'_g, FP'_c), \sigma')}$$

Informally, the above rule states that in order for a turn to occur we must update the general's program P_g into P'_g, the function which maps captains to their programs FP_c into FP'_c and finally the game state σ into σ'. This is only possible if a number of premises are true. The first premise (line 1) states that using the transition relation $\rightarrow_G$ over game state σ and the initial general program P_g we should obtain the set of general actions Γ and the updated general program P'_g. The next premise (line 2) updates σ into an intermediate state σ'' using the game-state updating function for generals $update_G$. This requires the general's actions Γ obtained from the first premise. The third premise (line 3) defines γ_{cs} which is a set of tuples where an element consists of a captain and their corresponding set of actions. It can thus be seen as a function which maps captains to their actions. In order to obtain γ_{cs} we perform a union on a number of sets consisting of individual tuples, one for each captain c. Each tuple is derived by the existential occurrence of an updated program P_c which is obtained by using the transition relation $\rightarrow_C$ over the intermediate state σ'' and an initial captain program which is obtained by querying FP_c with the current captain c. The result of the transition is the current captain's set of actions γ which are paired with the current captain and returned as a tuple within γ_{cs}. The fourth premise (line 4) allows us to derive the updated captain-to-program mapping function FP'_c. To achieve this we must assume that for all captains there exists a set of actions γ such that the captain's program is updated by means of the captain transition relation $\rightarrow_C$. The updated program resulting from this transition defines a part of the function FP'_c. By quantifying for all captains we can derive the actual function FP'_c. The fifth and final premise (line 5) derives the updated game state σ'. We again employ the use of an update function, this time the one for captains $update_C$. This function when supplied with the intermediate state σ'' and the set $\gamma_c s$ outputs the required resultant game state σ' which as been modified to reflect that captains' actions. We thus obtain P'_g,FP'_c and σ' which allow us to define our turn transition relation $\rightarrow_T$.

5.4.3 Implementation via embedding

We have embodied the semantics for both $\rightarrow_C$ and $\rightarrow_G$ within two script-interpreting functions `stepCaptain` and `stepGeneral` (in a similar fashion to 4Blocks's `step`) allowing them to interpret captain scripts and general scripts, respectively. Since the language constructs and semantics are very similar in nature the new step functions are virtually identical to `step`. We require only three minor changes: removing for-this-piece handling, adding handling for parallel composition and, also due to the latter, allowing the function to return more than one instruction as a list of instructions. Using these two functions we have embedded general and captain scripting languages for Space Generals which may be used to encode our AI.

5.5 Adaptive AI Scripting

As we discussed in Section 4.4.4, one of the main advantages our approach has over the other scripting approaches is that by means of embedding we can work on two levels of abstraction where the host language acts as a meta language over the embedded language. This gives the host language the ability of viewing the embedded language scripts as first-class objects which can be generated and manipulated using the former's functions. We have found a very useful application of this when making use of our scripting languages to implement the AI of our games. In fact we can implement an AI module in Haskell which performs two very useful things: on-the-fly strategy-based script generation and changing strategies based on in-game events.

On-the-fly strategy-based script generation

We can make use of Haskell and a number of embedded language techniques such as paramterised objects and connection patterns in order to write powerful script generators. Haskell can be used to write a number of strategies which, by means of Space Generals' API functions, query and analyse the game state and extract certain information from it. The latter can then be supplied to the script generators in order to generate the actual embedded language scripts automatically at run-time, if required. The benefit of this approach is that the strategies themselves are generic but are then concretised at run-time into the required AI scripts.

When we discussed how to introduce the generation of scripts based on a number of strategies earlier we made use of two basic functions in addition to `step`. These functions where `stepAI` and `thinkAI` which represented different portions of our AI. The function `stepAI` actually acted as the AI's main module itself. It received the game state and if there was no outstanding program it asked the script generator `thinkAI` to generate one for it. The latter function made use of `think` in order to select from a number of strategies and generate the appropriate program for the current game state. If an outstanding program existed `stepAI` called `step` upon it in order to extract an instruction and update the program. We can use the same procedure for Space Generals's AI with some minor modifications.

Since a typical Space Generals game may consist of up to five generals playing simultaneously our `stepAI` function must now be run five times, once for each general. We thus created another function `stepMain` which now acts as the AI's main module:

```
stepMain :: Game -> Programs ->
            (Programs
            ,[Instruction],[Instruction],[Instruction],[Instruction],[Instruction])
stepMain game programs = (updatedPrograms,inst1,inst2,inst3,inst4,inst5)
```

```
  where (prog1Gen,prog1Capts) = getPrograms1 programs
        (prog2Gen,prog2Capts) = getPrograms2 programs
        ...
        (inst1,updatedProg1Gen,updatedProg1Capts)
          = stepAI game generalAI1 captainsAI1 prog1Gen prog1Capts
        (inst2,updatedProg2Gen,updatedProg2Capts)
          = stepAI game generalAI2 captainsAI2 prog2Gen prog2Capts
        ...
        updatedPrograms
          = setPrograms updatedProg1Gen updatedProg1Capts updatedProg2Gen ...
```

We shall make use of an auxiliary type `Programs` here which contains the current programs of all the generals and captains within the game. The function `stepMain` receives a game state and an instance of `Programs` and returns an updated version of the latter and the instructions of every general and their respective captains, grouped by general, to be used to update the game state. In `stepMain` we first extract each general's current program and their captains' current programs. These are then used to call `stepAI` along with the game state and two "thinking" functions (instead of `think`, making `stepAI` a higher-order function), one for generals and one for captains. Since each player might write their AI differently five different versions of the latter functions exist: `generalAI1` to `generalAI5` and `captainAI1` to `captainAI5`. The calls to `stepAI` return the required instructions and updated general and captain programs. The latter are saved inside an updated `Programs` instance and returned along with the instructions themselves.

The function `stepAI` still runs as before. If a script exists for either a general or the latter's captains they are stepped using `stepGeneral` and `stepCaptain`. Otherwise a new script is generated using the supplied AI "thinking" functions. To achieve this `stepAI` delegates the job to two auxiliary functions, one for generals `stepAIGeneral` and one for captains `stepAICaptains`. Here we reproduce one of the two functions `stepAIGeneral`:

```
stepAIGeneral :: (Game -> Program (Action General)) -> Game ->
                 Maybe (Program (Action General)) ->
                 (Maybe (Program (Action General)),[Action General])
stepAIGeneral generalAI game Nothing     = stepGeneral game (generalAI game)
stepAIGeneral generalAI game (Just prog) = stepGeneral game prog
```

Similarly for captains. As can be noticed either the program has ended (first case) whereby we generate a new one using the supplied thinking function `generalAI` and step it, or a program exists and so we simply step it using `stepGeneral` defined earlier.

It is the AI programmer's task to program the "thinking" functions in similar manner to the setup used for 4Blocks's `think` function. The functions must select strategies by querying the game state using a number of API function and then generate the required scripts accordingly using the appropriate program generators, connection patterns and clever functions. Thus a typical module supplied by a programmer handling the AI for the first general and his captains the would consist of the following:

```
generalAI1 :: Game -> Program (Action General)
generalAI1 game
  | detectExpandTerritories game = expandTerritories game
  | detectHarvestResources game = harvestResources game
  | detectDefendOutlyingTerritories game = defendOutlyingTerritories game
  | ...
```

```
captainAI1 :: Game -> String -> Program (Action Captain)
captainAI1 game "capt1"
  | detectDefendBorderLands game = defendBorderLands game
  | detectFortifyLinkedLands game = fortifyLinkedLands game
  | ...
captainAI1 game "capt2"
  | ...
```

The first function returns the general's program while the second returns the captains' programs. This allows us to generate scripts based on strategies for both generals and captains in order to create an AI which is reactive to the game state.

Changing strategies based on in-game events

Since Space Generals is a multi-player game, more than one player acts on the same game board. Since we can only make an educated guess of future enemy player actions our AI script might not be valid within a few turns. This is especially true since the game has a small element of chance in it and what should be a potential victory could quickly turn into an unlucky loss. Due to this, for Space Generals, having dynamically generated scripts is not enough anymore. What happens if during an expansion script we realize that we are stretching ourselves too thin to defend for any possible enemy attack? Ideally we detect such occurrences and re-evaluate our strategy if required, either by rethinking everything or by arbitrarily selecting a particular strategy. So far we had considered this possibility when we discussed 4Blocks's adaptive AI, however we did not implement it since the game was trivial. For Space Generals, however, if a script already exists our AI module should no longer simply step it as was done so far but rather it should take a decision on whether a new script is needed, as we saw in step 2a of Figure 4.8.

For Space Generals we have decided that when an event occurs we simply re-think our strategy from scratch by calling the appropriate "thinking" functions. To achieve this, we need to redefine the stepping functions `stepAIGeneral` and `stepAICaptain` defined in the previous section such that now if a program does exist we first check whether we can detect any event. Thus we need another item to be supplied by the AI programmer which acts as an expression upon the game state. When this evaluates to true we re-think our strategy. For `stepAIGeneral` this is achieved as follows:

```
stepAIGeneral :: (Game -> Program (Action General)) ->
                 Expression -> Game -> Maybe (Program (Action General)) ->
                 (Maybe (Program (Action General)),[Action General])
stepAIGeneral generalAI _   game Nothing     = stepGeneral game (generalAI game)
stepAIGeneral generalAI exp game (Just prog) =
  if (evalExp exp game)
    then stepGeneral game (generalAI game)
    else stepGeneral game prog
```

Similarly for `stepAICaptain`. As can be noticed we have added another case which is triggered only when an expression upon the game state `exp` evaluates to true. In this case we act as if our program has terminated. We thus generate a new program and step the latter instead. Otherwise we proceed normally by stepping the existing program. In this way we can react to changes in the game state and tweak our scripts accordingly.

It is now the AI programmer's job to supply us with the expression itself. An example of such an expression is the following one:

```
generalExp :: Expression
generalExp = (check’ galaxyAttack <&&> check’ largeArmies) <||> (check’ safeGalaxy)
```

This expression checks whether an event has occurred where we have a galaxy under attack, detected perhaps by a loss of outlying planet, in conjunction to the attacking armies being large. If this occurs we might want to react by providing the necessary support to the next possible targets instead of staging an attack ourselves. If this event is triggered, one of the strategies within the thinking function now takes priority allowing for the appropriate script to be generated. Otherwise using disjunction we identify another event which occurs when a planet has been held for more than a number of turns at which point the general can react by upgrading it accordingly. Thus, in order to create a module for an AI player a programmer needs to provide SpaceGenerals only with four items: `generalAI`, `captainAI`, `generalExp` and `captainExp`. By providing these we are able to make use of adaptive AI scripting as explained here.

5.6 Concurrency

So far the scripting languages we have created for generals and captains allow us to script the latter separately from one another as mutually exclusive entities. This by itself is already advantageous as it gives us the power to script them and encode a form of autonomous intelligence within them. However, an important reason why Space Generals was developed as a case study was to examine the possibility of cooperative and concurrent behaviour between agents in order to reach a common goal. It makes no sense for the captains to plough onwards without being given meaningful directions by the general. Also a general can never achieve his or her high-order aim unless he or she knows what his or her captains are doing or how they are fairing. For example, it is probable that during a game session a general might want to change his or her strategy from an aggressive, expansionist one into a more defensive, guarded one. However, if that were to occur the general has no way of instructing their captains to follow suit, that is, there is no way for a general to order his or her captains into following tactics which concur with their over-all strategy. To allow for this kind of behaviour we need to introduce concurrent communication between a general and the latter’s captains. In the rest of this section we will show how easy it is to introduce the message-passing constructs and to import a language within another using the language embedded approach.

In order to introduce concurrency we first identified the three forms of communication which we would like to add:

- **Order** (General → Captain) – Due to their superior position over their captains, the generals should be able, if the need arises, to give the captains direct orders. This can be achieved by a construct, which we named *order*, that allows the general to override a captain’s behaviour by directly giving the latter the script to execute. The construct should take the following form:

  Order `"Captain1"` (`Perform DeployArmies ...`)

  Here we see a general ordering his or her captain to deploy a number of armies. As can be noticed the action to deploy armies is not something which a general is able to do but this new construct expands the generals’ scripting language with the capability to receive such syntax. Thus here we require the general’s scripting language to actually import the captains’ language syntax, something which is not trivial to do using traditional language creation techniques.

- **Tell** (General → Captain) – A general should be able to pass a message to a captain in order to indicate and guide the latter towards what they should do without actually giving them direct orders. In this way the general passes a cue which the captains are able to pick-up on, in order to change what they are doing whilst still retaining a measure of control. Thus, the construct *tell* should take the following form:

Tell `"Captain1"` *Attack*

Here the general is instructing the captain to become aggressive by attacking as they see fit. As can be noticed here apart from the actual keyword we require the introduction of a number of messages such as *Attack*, *Defend*, *Help* and so on. The captain knows how to react based on the scripts tied to these messages.

- **Report** (Captain → General) – A captain should be able to send messages to their general in order to report to them. This is the direct reverse of what the general can do and would allow the general a means of gaining perspective about their captains' possible successes or failures in carrying out what they are required to do. The captain's ability to report requires the introduction of a construct taking the following form which we shall call *report*:

  Report UnderAttack

  Using this script the captain would be reporting to their general that they are under attack. Once more we require also a number of messages such as *UnderAttack*, *Ready* and *Waiting* to be used in conjunction with the construct. Again, the general can know how to react based on these messages, where each message might lead to a different scripted reaction.

The syntax requirements of ordering, telling and reporting constructs can be safely introduced by updating the *instruction* BNF which holds generals' instruction and the captains' instructions, and the introduction of the *message* BNF, as follows:

$$\begin{aligned}
\textit{instruction} ::= &\ \ldots \\
\mid &\ \textbf{order}\ \textit{string}\ \textit{program}_{<C>} \\
\mid &\ \textbf{tell}\ \textit{string}\ \textit{message} \\
\mid &\ \textbf{report}\ \textit{message}
\end{aligned}$$

$$\begin{aligned}
\textit{message} ::= &\ \textbf{help} \\
\mid &\ \textbf{attack} \\
\mid &\ \textbf{defend} \\
\mid &\ \textbf{underAttack} \\
\mid &\ \textbf{ready} \\
\mid &\ \ldots
\end{aligned}$$

The *tell* and *report* constructs follow directly from the above discussion. Also, as stated, the *order* construct requires the import of the captain's scripting-language. How is this achieved using the embedding approach? The language frameworks technique discussed earlier in Section 3.9 allow us to achieve this with ease. We simply import the captain's programming language represented by the `(Program (Action Captain))` data type within the generals' *order* instruction. As can be noticed, again, what we have to do is simply translate the above BNFs into the required Haskell code, as follows:

```
data Instruction
  = ...
  | Order String (Program (Action Captain))
  | Tell String Message
  | Report Message
```

```
data Message
  = Help
  | Attack
  | Defend
  | ...
```

We have subsequently added three language constructs to our language `order'`, `tell'` and `report'` which like other actions are typed such that only a general can perform the former two while only the captain can perform the latter. The semantics of these constructs are are encompassed by the operational semantics of the *perform* semantic rule used for both general and captain actions. However, since *order* can now alter a captain's program during a turn, we now require a change to the operational semantics of the latter. The turn relation $\rightarrow_T$ is now defined as follows:

Turn

$$\frac{\begin{array}{c} P_g \underset{\sigma}{\xrightarrow{\Gamma}}_G P'_g \\ (\sigma'', FP''_c) = update_G(\sigma, FP_c, \Gamma) \\ \gamma_{cs} = \bigcup\{(c,\gamma) \mid \exists P_c.FP''_c\ c \underset{\sigma''}{\xrightarrow{\gamma}}_C P_c\} \\ \forall c.\exists\gamma.FP''_c\ c \underset{\sigma''}{\xrightarrow{\gamma}}_C FP'_c\ c \\ \sigma' = update_C(\sigma'', \gamma_{cs}) \end{array}}{((P_g, FP_c), \sigma) \rightarrow_T ((P'_g, FP'_c), \sigma')}$$

As can be noticed by comparing the new definition to the older one we have changed the second premise to one which makes use of a new version of *update*$_G$. Since now the general's part of the turn might change a captain's program our *update*$_G$ must be altered such that it now also receives the current captain-to-program mapping function FP_c and returns an updated one FP''_c, in case one of the general's actions is actually an order. It thus obtains the following type definition:

$$update_G : (\Sigma \times (C \rightarrow Prg^+_C) \times \mathcal{P}^{A_G}) \rightarrow (\Sigma \times (C \rightarrow Prg^+_C))$$

and behaves accordingly.

Telling and Reporting

In order to integrate telling and reporting within the game we have to enriched the game state such that now a channel exists for every captain a general has. This channel is unique in that it is created when a captain is trained and destroyed when a captain is retired. The tell command is used by the general to put a message on the channel relating to a particular captain. Since a turn is divided into two stages, the general can submit a message in his stage and in the second stage this is visible for the captain to see. A captain can on the other hand report back using the same channel and the message can be viewed in the next general stage in a new turn. In order to allow the generals and captains a way with which to check their channel we have added another expression which we named `told'`. This construct allows both a general and a captain to query their channel for information which may then be used in conditionals and loops to influence the former's programs.

This allows us to write scripts of the following form now:

```
capt1Prog =
  while' (not' (told' "Capt1" attack')) do'
```

```
    (perform' (withdrawFromCountry' "General1" "Capt1" "Malta" 10 10))
  -|-
  if' (check' underAttackFromEnemy) then' (perform' (report' underAttack')) else' skip'

general1Prog =
  while' (told' "Capt1" underAttack' <||> told' "Capt2" underAttack') do'
    (perform' (tell' "Capt1" attack')  -|- perform' (tell' "Capt2" attack'))
  -|-
  if' (told' "Capt1" help')
    then' (perform' (supportCaptain' "General1" "Capt1" 10 10 10))
    else' ((if' (told' "Capt2" help')
              then' (perform' (supportCaptain' "General1" "Capt2" 10 10 10))
              else' skip'))
```

The captain making use of program `capt1Prog` uses a while construct to withdraw resources from a country `Malta` each turn until he is ordered by his general to attack. To check whether he was told to attack we make use of the newly added expression construct `told'` and the message construct `attack'`. Also if while gathering he is attacked (detected by making use of `check'` and the predicate `underAttackFromEnemy`) he reports this to his general by using `report' underAttack'`. The general on the other hand waits until either of two of his captains report to him that they are under attack (using `told'` and `underAttack'` at which point he orders them both to retaliate using the `tell'` construct and the message `attack'`. At the same time, if the first captain asks for help he supports him by sending resources. If the first captain does not require help the general checks if the second captain did and supports him instead. In this way the first captain is given priority but either one of them will be helped by the general if they require aid.

Ordering

Adding the ordering construct requires no change to our implementation apart from allowing the `order'` construct to be interpreted as changing the captain's current program. Also as highlighted above, a new version of $update_G$ must be written which caters for the possibility of a captain's program being changed during a turn.

A general can now order a captain to perform a script as follows now:

```
general1Prog =
  perform' (
    order' "Capt1" (
      perform' (withdrawFromCountry' "General1" "Capt1" "Malta" 10 10) -|-
      perform' (buyArmiesCaptain' "General1" "Capt1" 5)
    )
  )
```

Here the general is ordering a captain to withdraw resources from a country and use them to buy armies. In this way a general can perform micro-management themself if they so wish.

5.7 Results and Discussion

Like 4Blocks, Space Generals has served us as a means of showing how embedded languages allow us to create a scripting language for games. This time the game is much more elaborate but its scripting language creation process is still straightforward to achieve which shows the validity of our approach. Moreover, we have ported the idea behind adaptive scripts to this game as well. This has allowed our AI to be written in such a way as to enable it to select from a number of predefined strategies by itself and then, using such strategies, to generate the required scripts to be used to encode intelligence within our generals and captains. Thus the scripts encoded within the AI are no longer fixed but now, via the appropriately written script-generator functions, adaptive to the game state and concretised based upon the latter. Also, we have shown how it is possible to make our AI module more human-like by allow it to change strategies if required in the middle of executing a program generated by another strategy. We have shown how this is possible using simple predicate-based events which re-trigger the thinking process.

We have carried out a small experiment where eight or so other students met up to play Space Generals. After familiarisation with the game a few sessions were attempted, pitting the players against each other. Each player started to come up with their own strategies which they attempted to use to win the game. Ideally, after the experiment the students would be supplied with the scripting language constructs and the API of the game such that they would write their own AI modules for us in both Haskell and the embedded language. Unfortunately, this could not be carried out and so we invited the participants to send us a short description of their strategies written in plain English which we then encoded ourselves.

The results achieved within this case study have shown us that the embedded-language scripting approach can supply us with the following advantages. First of all, we were able to embed two languages within Haskell which have allowed us to script two types of actors who are able to perform different actions. Having two DSELs inside a common language has allowed us make use of the best language for the particular scripting required. Also, combining them together has enabled us to use two different languages of different specialisation to reach our goal of scripting Space Generals. Moreover, having Haskell as a common host has permitted us to implement a form of message-passing as a means of allowing the two embedded languages to communicate and influence one another thus enabling our AI to become even more dynamic and adaptive. This brings us to another contribution of this case study which is related to how easy it is to improve an embedded language in order to include notions of parallel composition and also concurrency. Neither of these are native to Haskell but we can still add them to our embedded language and use them as we require. Moreover, the process of adding them is also quite straightforward once we have defined the syntax and semantics of the required constructs. Finally, this case study has allowed us to show how a feature of the technique of language frameworks (described in Section 3.9), that of language nesting, may be used to import a language within another. We have achieved language nesting by means of our `order'` construct which allows a higher-level officer to import lower-level scripts within their own scripts. In doing so we are giving the two languages another means of communication which does not require the host language to act as an interfacing medium. This has the added advantage of not requiring us to develop a third language which consists of the general's language needlessly bloated with captain-language constructs but has rather allowed us to simply import the latter language directly.

We believe that this case study has re-confirmed what our previous game's results have shown, that is, that scripting games using the embedded approach is a viable approach. The complexity of the game itself was increased here but the process by means of which the scripting language was created and integrated is still straightforward and requires very little effort when compared to the proprietary and general-purpose scripting methods. All the stages for the creation of the scripting language are still required but the last step, that of translating the language design products such as BNF syntax and operational semantics, into an actual, working language requires simply the translation of these into the appropriate host language code. Our case-studies have shown us that there is often a one-to-one mapping between the BNF syntax definition and the host language data types required. Moreover, by means of the appropriate correct interpretation functions, the operational semantics are also very straightforward to implement once defined.

The advantage which the embedded language approach shares with general-purpose language scripting and proprietary language scripting and which the game engine integration method lacks is the separation of concerns. Since the embedded approach makes use of a language we can separate content and logic in the case of both fixed scripts and adaptive scripts. In fact, for Space Generals we have implemented a five-player plug-in system where the players may script their AI using either fixed scripts or, if they know Haskell, adaptive scripts. These scripts can be added to game files without actually changing anying from the latter and used to pit the AIs versus each other.

Another advantage of our approach when compared to the other ones relates to one of the results discussed above. Advanced language features such as parallel composition, message passing and language nesting are very easy to add to our language. If we are able to specify and design the required features successfully, implementing them is as straightforward as implementing the language in the first place. Doing so using the traditional means used to implement general-purpose scripting languages and proprietary scripting languages requires more work in comparison as all the various component's of the relevant language's compiler or interpreter might need to be altered depending on the feature itself. Instead, using the embedded approach we simply need to alter the host language code implementing the language without having to actually alter the former's compiler or interpreter in any way.

One final advantage our approach has over the other methods is that it takes the idea of having a scripting language to the next level. As we stated many times, game engine integration suffers because there is no actual scripting language involved. On the other hand, in our approach we do not have just one language, as is the case with general-purpose scripting and proprietary language scripting but rather two. Having two languages at different levels of abstraction introduces the possibility of meta-programming. For games meta-programming is a very useful feature, as we saw in both 4Blocks and Space Generals. It allows us to write strategies which encode certain generic patterns which once supplied with an actual case, that is the game state, are able to adapt themselves accordingly by creating the appropriate instance at run-time. In our case this instance is the resultant script.

5.8 Conclusion

Having thus concluded our second and final case study, in the next chapter we shall discuss our results and evaluate them by discussing what we believe are the key components of our approach. This will allow us to compare it to the other available scripting approaches discussed earlier. We shall also consider a number of limitations found within our approach and suggest ways by means of which such limitations may be reduced or solved. Finally, we shall also discuss some work related to ours.

Chapter 6

Discussion

6.1 Introduction

The main aim of this thesis was to suggest an alternative approach to the more traditional game-scripting methods of game engine integration, proprietary language scripting and general-purpose language scripting which were introduced in Chapter 2. Our approach makes use of a scripting language like the latter two do and so acquires most, if not the same, advantages as these: it separates logic and content, it allows for scripting by story-writers if the language used is domain-specific and it also allows for AI scripting if general-purpose language features are added to the language. The approach also allows for third-party programming to interact with the game if the latter exposes the right interface. Our approach however adds its own benefits. Via language embedding the creation of the scripting language is rendered much easier since once the latter is fully designed, translation into actual working code is almost trivial. Proprietary language scripting and general-purpose scripting have a problem here as the method usually employed to implement their language is so demanding that it is often one of the reasons why developers resort to using pre-implemented off-the-shelf general-purpose languages. Another benefit of our approach, as we saw in our case studies, is that of tailoring. Altering the language for the target audience, be it story-writers or AI-writers is fairly easy by exposing the part of the language which is domain-specific only to the former and supplying the entire language to the latter. Doing so ensures that story-writers are able to understand the language and use it themselves easily. More advanced features such as looping and conditional statements found in general-purpose languages can be made available to programmers to aid them in their AI-programming however by exposing their respective constructs via the host language's module system. Yet another advantage of our approach is that it allows a language to be updated easily. Adding new features, even advanced ones related to complex domains such as concurrency, to the language once it has been developed is also easy if these are properly specified and formalised within the language. This once more requires substantial work in the other two scripting methods. Finally, our approach adds the extra benefit of meta-programming which is very useful in domains such as games in order to generate scripts on-the-fly. This is made possible directly by virtue of embedding since the host language itself introduces an extra layer of abstraction on top of the scripting language. In this chapter we will shall first summarise our results and then use the latter to evaluate how successful our approach was. We will then compare and contrast our approach with the other three approaches in light of the criteria just discussed. This will allow us to evaluate our system qualitatively. Next, we shall briefly discuss a number of shortcomings which our approach has and introduce some potential solutions. Finally, we will discuss some work whose nature is related to our own.

6.2 Results

Using the techniques of language embedding within our two case-studies has allowed us to achieve the following results:

Creating game scripting languages We have successfully created scripting languages for our two games. Creating the syntax for our languages was as straightforward as figuring out the basic domain-specific actions of the game, implementing them as a host language data type and then encapsulating the latter in another data type which represented typical high-level language idioms such as sequential composition, parallel composition, looping statements and conditional statements. The latter's language semantics were then implemented by a simple interpretation function which received a script in the language (a data structure for the host language) and returned its interpretation which for our games consisted of the next instruction to be run and an updated script.

Fixed AI Scripting The scripting languages created have allowed us to write simple and complex scripts which were used to create a form of static AI for our games. These scripts are simply run over and over again by an AI module written in Haskell which sits between the game itself and the scripting language interpreter.

Adaptive AI Scripting The true power of the embedding approach shows itself when we used both the scripting language and Haskell itself together to write our AI module. Having two languages available allowed us to use Haskell as a meta-language which manipulated our language scripts. This has enabled us to achieve two very useful results:

- **Strategy-based script generation** We have enhanced the AI module used in fixed AI scripting such that instead of using a predefined script each time it now attempted to use a strategy from a number of encoded strategies within it which is selected based on the current game state. Once one such strategy is selected the latter makes use of a number of Haskell functions: parameterised objects, connection patterns and clever functions, in order to generate the required scripts automatically based on the required parameters extracted from the game state.
- **Event-based strategies** – A further enhancement to fixed AI scripting is the use of functions which attempt to detect unplanned game-state changes. These have allowed us to remove the current script and generate a new script when the event caused the game state to change in such a way as to render the current script no longer advantageous.

Creating strongly-typed scripting languages Since Space Generals required the use of stronger typing in order to differentiate between general actions and captain actions we made us of phantom-typing. These allowed us to embed a type system within Haskell for our embedded languages by making use of Haskell's own type-system.

Introducing advanced features In our latter case study we have implemented a rudimentary form of concurrency as a means of allowing general scripts and captain scripts to communicate, synchronise their efforts, influence each other and work in unison. The result of this are:

- **Communicating via messages (indirectly)** Since our generals' scripting language and captains' scripting language are both embedded in Haskell the latter has allowed us to create a number of channels, one between every captain and their respective general, as a means of communication. Pre-defined messages can be sent and received via these channels and may be used to influence how a script's interpretation is run. Thus if written properly our adaptive AI strategies were now able not only to react to the game state but also to the possible messages which both generals and captains can exchange.

- **Communicating via orders (directly)** We have also implemented another variation of message-passing which consists of generals being able to directly control their subordinates by changing their current program directly. This was useful in delicate strategies or sudden changes in the game state which required immediate attention. This part of message-passing was achieved via language nesting, another feature of language frameworks where the captain's scripting language was simply imported within the general's scripting language without requiring the use of the host language (as in the previous type of message-passing) or the creation of a third-party language combining the two.

6.3 Evaluation

A qualitative assessment of the results obtained by our case-studies was carried out in order to evaluate our approach. A quantitative evaluation was not applicable for our approach since we could not extract physical parameters with which to compare and contrast. We are not comparing AI implementations for a game so as to decide which AI is better or worse, but rather we are supplying an orthogonal method by means of which we believe AI may be introduced into a game system. The evaluation criteria used here are derived from those defined in Section 2.6. However, we combine a number of the latter together in an appropriate manner to facilitate discussion.

Ease of use By using the scripting language normal scripted behaviour can be achieved for the elements within the game world which must be animated. Moreover, the complexity of the language can be tailored by altering its syntax such that it can be easily understood and used by non-programmers such as story-writers. If certain features such as programming idioms like loops and conditionals are extra or not understandable they can be simply hidden. Also, like with other DSELs, we can provide non-programmers with the game-related keywords which they can understand as our language syntax. Finally, Haskell's free-form syntax by itself, which is not littered by non-scripting constructs, allows for easier understanding of the scripting language.

Expressivity As we saw by means of our two case-studies it is possible to use our scripting languages on their own or in conjunction with Haskell in order to write fairly complex AI for games. In fixed AI scripting we made use of the scripting language directly which on its own has allowed us to script fairly complex behaviour. However, since we are embedding in a fully-featured host language it is reasonable to make use of the latter's features if required. This is what we did in adaptive AI scripting. The idea used here bases itself upon a model of how we believe a human player plays a typical game. A player thinks out a strategy, works out the actual game instructions according to this strategy and then executes the instructions almost automatically whilst taking care of any events which might required a change in strategy. We made use of Haskell as a meta-language over our scripting languages in order to implement the thinking behaviour which selects a strategy and generates the required embedded language script. The actual carrying-out of the script is then achieved by interpreter of the scripting language which extracts the instructions and supplies them to the game engine mechanically. Using this approach our embedded language in conjunction to the host language Haskell we have successfully encoded AIs for our two games. Of course it is possible to use Haskell in a similar fashion such that instead of strategy-based AI we make use of neural networks and genetic algorithms to generate values for our script generator functions but this goes beyond the scope of this body of work.

Ease of implementation As we saw, implementing a scripting language using embedded languages is a very easy and straightforward process which should take no more than a few moments. Once the required language has been specified and adequately designed it can be implemented in the host language via the latter's data types and functions without requiring the need to develop a number of tools from scratch as is often done when creating a new language using traditional techniques. Syntax is easily implemented by translating the language's BNFs into the equivalent data types. Semantics are also very simple to implement once their semantics have been defined. In the embedding approach this is achieved simply by encoding the rules themselves in an interpretation function which gives meaning to the programs' data type by extracting from them the next action or actions and returning an updated program.

Ease of updating Another bonus of this approach is that the language may be evolved at ease both if we need to add any constructs related to the game itself, that is domain-specific, and also if we need to add another idiom, such as those often found in general-purpose languages. Once the constructs have been adequately specified and designed, all we require is to introduce the matching data constructors and to update the interpretation function accordingly to reflect the new additions. Moreover, since the code of the language is often compact, identifying where a change needs to be affected is easy. Finally, adding more advanced language features which might not necessarily be related to the actual target domain itself is also very easy. We saw this when we discussed our second case study where we introduced concurrency by means of message-passing.

Ease of integration Integrating the scripting language into the game is an easy process since this is achieved by virtue of the embedding process itself, that is by adding the data type representing the language's syntax and by adding the function which gives semantics to structures of this data type. In a way, implementing the language is synonymous to integrating it.

Separation of scripts from the game engine Since our approach makes use of a scripting language and its scripting engine by nature we are separating the scripts from the game engine. As we discussed earlier doing this gives us the advantage that the game engine does not need to be recompiled from scratch every time a script needs to be altered. Our approach works in this manner as well. For example, if we are making use of fixed AI scripting and our pre-defined script needs to be changed we can implement our changes within the Haskell module containing our script and recompile this module. The same applies for adaptive AI scripting. If we need to update any of the AI written in Haskell or our scripting language, we can change and compile these without affecting other game modules directly. Also, the fact that we are using a scripting language enables us, like in proprietary scripting and general-purpose scripting, to allow for the possibility of third-party scripting in the form of addons or mods. All we have to do is provide the right API to query the game's state and expose the constructs of the language which we wish to make availble to script the game.

6.4 Comparison to other approaches

In the following sections we shall further evaluate our approach by comparing it to the other methods of game scripting discussed in Chapter 2.

6.4.1 Comparison to Game Engine Integration

Most of the advantages which general-purpose language scripting and proprietary language scripting have over game engine integration apply also to the embedded language approach. First of all, our scripts are not integrated within the game engine and so we maintain content/logic separation. In practice our AI modules are still Haskell modules which form part of the game but by means of our approach they are decoupled in such a way that, as stated earlier, we do not need to recompile everything from scratch each time a change is made to a script. Another advantage related to this is that our approach allows for the writing of mods and add-ons to our AI with relative ease. We simply need to change the AI module implementing the AI without affecting the game engine. Also, even though for our case-studies we have restricted ourself to one language, both for the game and for the AI, our approach is generic enough such that it does not require us to make use of the same implementation language as that used to program the game (see Section 7.2). Game engine integration on the other hand requires this by default. Finally, the complexity of our language may be tailored. This means that compared to game engine integration scripting may be performed by both programmers and non-technical people. Programmers can create a language which is more complex and also implement AI as was suggested while story-writers can make use of a simpler language without requiring translation to be done by programmers. The only advantage which game engine integration offers over our approach is that due to the lack of interpretation required it is by default faster. As with the other two approaches this if often considered a minor setback.

6.4.2 Comparison to General-Purpose Language Scripting

Both general-purpose language scripting and our embedded language approach make use of the idea of scripts in order to maintain separation between the game engine and game content. Again this means less unnecessary re-compilations and the possibility of mods and add-ons. To achieve scripting both approaches make use of a general-purpose language even though the approach taken differs. In general-purpose language scripting the language is used by itself to write the required scripts and AI. In our approach the language is first used to implement an embedded game-specific scripting language and then together with the host language they are used to write the scripts and required AI modules. Having a general-purpose language in both cases means that the power of a fully-featured language is available at hand, along with its tools. The main difference between our approach and the general-purpose language approach is the work required to implement the language itself. Unless the general-purpose scripting language used is one which was acquired off-the-shelf it has to be implemented from scratch. The effort required is substantial compared to our approach which by comparison is very straightforward. Moreover, our approach integrates the scripting language directly within the game system. Making use of a general-purpose scripting language means that the latter needs to be somehow integrated such that it functions along-side the game's implementation language. We obtain this affect by default as we are in theory still working with the host language even when programming using the embedded language. Also, both approaches require their scripts to be run somehow. In our case we make use of an interpretation function while general-purpose languages often make use of virtual machines which perform a similar function. Finally, the advantage our approach has over general-purpose language scripting is that the latter is exclusively intended for programmers. Our approach on the other hand caters for both programmers and non-programmers since we can tailor the scripting language accordingly.

6.4.3 Comparison to Proprietary Language Scripting

As with general-purpose language scripting and our approach, proprietary language scripting makes use of scripts to maintain separation between game logic and content giving it the same benefits as the former two. The main difference between our approach and proprietary language scripting is that this time we do not necessarily have a general-purpose language available. A proprietary scripting language ranges from a simple DSEL to a full-blown general-purpose language but the language itself is set in stone and cannot be easily changed. This means that we are limited by what the language allows us to do. Such languages are usually intended towards programmers or non-programmers exclusively and changing the language is often a very tedious process which requires a considerable amount of work updating the language definition itself and any relevant tools related to it. Our approach differs in this manner since we can tailor the language as required very easily and we also always have a general-purpose language available if we wish to make use of it. Another difference is once more the work required to create the languages. Even if the proprietary language is a small one, it still requires a considerable amount of effort. This is not so in our case. Finally, both approaches require a way to run scripts and require some work in order to integrate within the game. In the latter case, like with general-purpose language scripting, our approach allows us to achieve this by virtue of the embedding process itself while in the case of proprietary language scripting this might require some work.

6.4.4 Overall Comparison

Like general-purpose language scripting and proprietary language scripting our approach is an improvement over game engine integration. Our approach however also improves upon the limitations of the former two. General-purpose scripting languages are intended for programmers only and may not be used by story-writers. Our approach mitigates this by allowing for both a general-purpose language and a game-specific scripting language. In comparison to proprietary language scripting we always have a general-purpose available for programmers to use and our scripting language can be tailored to meet the demands of both programmers and non-programmers easily. Also implementing, tailoring and updating a language with extra features is easy using our approach compared to the other methods using scripting languages. In fact using our approach we can mitigate the main problem these methods face, that of being rejected because of the work required to create and maintain their scripting language. Finally, our approach introduces the very useful feature of meta-programming which we have shown to be very useful in game scripting by our two case-studies. Meta-programming is not a feature which is usually found within the other methods unless it is explicitly introduced. Our approach on the other hand includes this by default. The qualitative differences between the four approaches, based on the criteria mentioned in Section 2.6, are summarised in Table 6.1. As can be noticed by the overview this table provides, our approach compares favourably with general-purpose scripting languages and proprietary scripting languages, and similar to the latter two, supplies certain improvements upon game engine integration.

6.5 Limitations

During the use of our approach we have met the following limitations which we believe can be improved upon:

- **Adaptive AI's event detection** – Our current implementation of adaptive AI's event detection is very basic since we base ourself on the current game state each time rather than a stream of changing game states. This is a very limited scope since it does not allow us to detect events which might occur over more than a number of turns. For example, in Space Generals we cannot detect that a galaxy is under attack unless we simply assume that this means that in the current state one of its planets is not ours. In reality such a definition would be more accurately attributed to the predicate "contested" rather than "under attack" since the latter requires us to detect a change in state to be true. Thus cases where we have almost conquered a galaxy also fall under the predicate "under attack" even if this is not the case. Ideally, further techniques are introduced which might allow us to better detect and handle such in-game events. A potential candidate to solve this problem is functional reactive programming (FRP) which we mentioned when we discussed example DSELs in Section 3.7. FRP is based on the notions of behaviours and events which we believe might provide us with a possible solution of this limitation in our future work.

Category	Criteria	Game Engine Integration	General-Purpose Scripting Language	Proprietary Scripting Language	Embedded Scripting Language
Scripting Language	Ease of implementation	N/A	No	No	Yes
	Ease of tailoring	N/A	No	No	Yes
	Ease of updating	N/A	No	No	Yes
	Ease of integration	N/A	Depends	Depends	Yes
Target Audience	Programmers	Yes	Yes	Yes	Yes
	Non-programmers	No	No	Depends	Yes
Features	Logic/content separation	No	Yes	Yes	Yes
	Normal scripting	Yes	Yes	Yes	Yes
	AI scripting	Yes	Yes	Depends	Yes
	Add-ons/mods	No	Yes	Yes	Yes
	General-purpose language available	Yes	Yes	No	Yes
	Requires interpretation/compilation	No	Yes	Yes	Yes
	Meta-programming	Depends	Depends	Depends	Yes

Table 6.1: Four scripting language approaches compared qualitatively based on a number of criteria.

- **Creating a language with the required syntax** – Sometimes due to collisions with the host language keywords we could not create the most ideal syntax for our embedded language. While this is a very minor setback it could be determintal if the scripting language is used by third party programmers, especially if these are not programmers by profession. In our implementation we went around this limitation by making use of a unified notation but other possible solutions exist as we shall see in Chapter 7. Another problem related to syntax is the elimination of what appear to be extranous brackets which are required by the host language rather than the embedded language itself. Sometimes these are unavoidable due to the way Haskell's operator precedence works. A final problem due to syntax is caused by Haskell's inability to natively accept functions whose variables may change in number and sometimes in type. An example of this was when we attempted to make the *if-then-else* statement to sometimes take one branch and sometimes take two, depending on whether the else branch is needed. To solve this problem we made use of the *skip* statement in our implementation.

6.6 Related Work

A variety of games have been implemented in Haskell and used either as a means of learning Haskell itself, for entertainment, as teaching tools or as test-beds for other research. For example, Lüth [43] has implemented a clone of the arcade game Asteroids in Haskell and used it as means of teaching functional programming while Courtney et al. [14] have used it as means of showing how functional reactive programming (FRP) may be used to adequately implement a game, in this case a clone of another arcade classic Space Invaders. Other games implemented in Haskell can be found on Haskell's online package repository HackageDB[1] in the Game[2] category.

Dobbe [17] made a similar study to ours where he attempted to make use of DSLs in order to add scripting features to a proprietary game engine by Cannibal Game Studios. Like us, he believes that DSLs provide the best abstraction level in order to script games. Despite also agreeing that creating a DSL from the bottom up is expensive and time consuming, his approach is to create one from scratch as he believes it gives him better control over the final scripting language itself and the integration mechanism it makes use of. We believe that the DSEL approach to DSL-creation provides us with a simpler and more cost-effective implementation process which still gives us considerable control upon our final scripting language depending on the host language selected.

A number of tools exist which allow for easier creation of DSLs in a similar manner to our approach. One of such tools is Stratego/XT [5] which consists of the Stratego language and the XT toolset. Stratego allows implemenation of program transformations using rewriting strategies while XT consists of a set of tools which allow for the development of transformation systems. Combined together these allow for the analysis, manipulation and generation of programs and subsequently may be used to formalize and create languages such as DSLs with relative ease. Another tool is the Algebraic Specification Formalism + Syntax Definition Formalism Meta-Environment (ASF+SDF) [67] allows the generation of languages such as DSLs by generating the tools which these require such as editors, parsers, debuggers, interpreters and compilers. To achieve this it makes use of the ASF+SDF formalism which allows the specification of syntax and semantics of the required DSL. A final tool is Microsoft's DSL Tools [13]. It consists of a suite integrated with Microsoft's Visual Studio which allows the creation of a DSL and its related tools. These are however confined to work within the .NET framework. If we compare these tools to our approach we see that in all three cases we have an environment which acts as meta over the required DSL in order to define it and create its tools. In our case Haskell's abstraction acts as the meta-environment we require to define the DSL's syntax and semantics and any other tools the language might need.

[1]http://hackage.haskell.org/packages/hackage.html (last visited August 2010)
[2]http://hackage.haskell.org/packages/archive/pkg-list.html#cat:game (last visited August 2010)

Other forms of adaptive AI have been discussed apart from ours. One of these is that suggested by Spronck et al. [63] where the AI makes use of an online machine learning technique called dynamic scripting in order to better itself and learn from past steps. Initially, the approach is similar to ours in the sense that a number of strategies are encoded within the AI with a certain priority and these generate the required scripts. Like us the strategies are given priorities based on preconceived domain-specific knowledge of what good game-play is for the particular game. Also, when selecting a strategy each of the strategies' conditions are checked and the first strategy to match a condition is selected. The main difference between our approach and Spronck et al.'s is that they make use of a system of changeable weights in order to affect the priorities of a number of rules which form strategies themselves. Due to this, if two strategies have the same priority, the one with the heavier weight is used. To update the weights a form of fitness is employed which depends on whether or not the rule has lead to a good or bad result. The fitness results are then used to update the weights by means of a weight-update function. This approach was applied successfully to the commercial game Neverwinter Nights.

Other work which relates to ours is ScriptEase, a graphical-user interface tool developed by Cutumisu et al. [16]. ScriptEase is designed to provide non-programmers such as story-writers with the facility to write their own scripts such that they no longer have to work with programmers in order to integrate content within the game. In order to achieve this story-writers make use of a menu-system based on a three step method. The first step involves a story-writer selecting what is known as a design pattern. Design patterns are axiomatic concepts or idioms often found in stories such as character/object interactions and dialogues. The second step involves the story-writer tweaking the design pattern in order to apply it to the story at hand. This involves specialising it by selecting context information related to the story being told. The third and final step is optional as far as story-writer goes and is carried out by pressing a button in order to generate the low-level code. This code can then be tweaked and integrated by a programmer. We liken design patterns to either parameterised objects which are supplied with parameters to generate low-level code or high-level language constructs which are then translated into lower-level scripts as in language hierarchies. Thus by means of an orthogonal method, ScriptEase solves the problem of creating a simple-enough language to be used by non-programmers by providing an appropriate interactive tool.

6.7 Conclusion

Having thus concluded our discussion on what our study has managed to achieve, how it compares to other approaches, its limitations and work related to it, in the next chapter we shall discuss ways by means of which we can expand upon our approach. We will then conclude with the contributions which this thesis has achieved and our final remarks.

Chapter 7

Conclusions

7.1 Introduction

This final chapter starts concluding our thesis by discussing some future work which we believe can strength our approach. Most of this work relates to improving the embedding approach itself such as via allowing the definition of better, more understandable syntax and by allowing us to create embedded languages even quicker and automatically. Other future work can be done to allow our adaptive AI to detect events better. We shall propose two ways to achieve this. One involves the use of FRP discussed in earlier chapters and one involves the use of lazy script generation. Finally, we shall conclude this thesis by discussing our achieveiments and giving our final remarks.

7.2 Future Work

Our work is far from complete and there are a number of ways that we can think of in order to expand our approach even further:

7.2.1 Implementing the required syntax

Haskell is very amenable to embedding, something which other languages are less capabile of sometimes. However, it stil causes problems when defining the language's syntax. One of these problems was mentioned earlier when we discussed Haskell's ability to let us chose an ideal syntax for our embedded language. As we discussed earlier, the process of embedding has some problems when it comes to permitting us to select certain keywords for our scripting language's syntax. A possible solution is to hide the host language's own keywords. Haskell allows this via a compiler directive. In the case of non-programmers this is ideal as the least of the host language is exposed the easier the former find it to work with the embedded language. However, this is problematic for programmers especially if we require the use of both host and embedded language constructs as in adaptive AI. Ideally, the host language should allow you to overload its keywords somehow. Further work is required to investigate a solution to this problem. A possible candidate is perhaps the use of Haskell meta-programming via Template Haskell [59].

Another problem related to syntax is that sometimes Haskell requires extract bracketing in order to be able to compile our embedded language. This sometimes causes confusion since the brackets would not be required by the scripting language. This problem can be somewhat reduced by selecting an adequate operator precedence for the constructs or by including the brackets themselves as part of the BNF of the language. Other languages encountering this problem in fact introduce curly braces to achieve a similar affect.

A final problem we met related to syntax is the need of what are known as polyvariadic functions. These are functions who's inputs could change in number as well as in type. As stated earlier, an example of this is the *if-then-else* statement which sometimes does not require the use of an else statement. Haskell by itself does not allow us to compile two different *if-then-else* constructs as we wish to do. We solved this by making use of a *skip* statement. However, it would be ideal to attempt the use of type-classes to implement this construct as one of these functions.

7.2.2 Quicker prototyping

A possible extension to our work which would make the process of language creation even faster is designing a tool which would allow us to generate the syntax and semantics of an embedded scripting language automatically. The tool would allow the creator of a new language to supply the language's syntax via a commonly used notation in the design stage such as BNF. In regards to semantics, the same approach can be taken by allowing the definition within the tool of the language's operational semantics. Since the translation of these is often very direct in Haskell it would be possible to then make the tool generate the required host language code for the embedded language by creating the required data types for the language's syntax and also by automatically producing the required `step` function for the language's semantics. In this way our approach can be used to quickly prototype languages and use them directly or implement them using traditional approaches once they have been used in embedded form.

7.2.3 Better event-detection

Integrating notions from functional reactive programming in our approach may allow us to solve the latter's second limitation which relates to properly detecting in-game events as was discussed in Section 6.5. A possible way to achieve this, perhaps, is to make use of a similar approach to that taken by a DSEL called Yampa [14] within our embedded scripting language. Yampa makes use of the idea of signals, or values which change over time in order to become more "time-aware" or reactive to time flow. In this way it is able to detect events. Our hope is that by introducing time in a similar manner we might be able to detect events better and hence improve our adaptive AI.

7.2.4 Lazy program generation

As we saw, a game's state might change by the time a script is fully run. Due to these changes our script might become stale and no longer advantageous. In order to mitigate this we attempted to make use of events which actually force a form of exception upon our current script by making our AI module rethink our strategy. Another possible solution to mitigate this problem is to introducing a lazy form of program generation. The basic idea behind lazy program generation is that instead of generating our code at one go we leave stubs which are generated later. These stubs are simply constructs which are supplied with the required code generating function that will generate the required code when it is its turn to be run. As an example of a program with lazy program generation consider the following partial implementation:

```
type GeneratorF = Game -> Program Game

data Program Game = ...
                  | Think GeneratorF

genF1 :: GeneratorF
genF1 gs = Perform a :>: Perform b :>: Think genF2
```

```
genF2 :: GeneratorF
genF2 gs = ...
```

where the type synonym `GeneratorF` represents a generator function which given a game state returns back a new program and the construct `Think` receives a generator function for later use. As can be noticed by the definition of `genF1` the program generated now will run the first perform in the current turn, the second perform in the next turn and in the third turn it will generate a new program based on the game state at that turn using `genF2`. If changes have occurred to the game state the program generated will now reflect these rather than basing itself on the older game state used by `genF1`.

7.2.5 Translating into general-purpose scripting languages

Since we are making use of deep embedding the semantics of our scripting languages do not necessarily require us to confine ourselves to normal language interpretation using `step`. Various other functions can be written which perform different transformations upon our abstract syntax tree. In fact we have provided a pretty printer for our Space Generals game accessed via the function `printOut`. Other, perhaps more interesting, translations could be to general-purpose scripting language code such as Lua or Python which are common ways of incorporating scripting within commercial games. The advantage of this is that it would allow us to create a simpler language for use by non-programmers such as story-writers which is then automatically translated into the general-purpose language code for integration or modification by the game-programmer.

7.3 Summary of Achievements

Our first aim in this thesis, which we have achieved successfully, was to propose an alternative method to more traditional scripting methods which allows for faster and quicker language creation. As we saw, using the embedded approach this is almost trivial which means that programmers no longer have to resort to using the language used to implement the game itself as in the case of game engine integration or to employ the use of a complex general-purpose off-the-shelf language in order to script their games. Instead they can now easily create their own language quickly and efficiently. Also if they still require to create the language themselves, such as when packaging a game engine to be sold for other game-developers, they can use our approach to quickly protoype their language and test it out before actually going through the implementation process itself. Our approach is thus very useful within the game-development lifecycle. Our next aim was to see whether a scripting language created using the embedded approach is fit for game-scripting. Using our two case-studies we have shown how fixed AI scripting makes it possible to use an embedded scripting language to script a game. Moreover, we have improved upon this approach by showing how the host language may be used in conjunction to the embedded language in order to create very powerful adaptive scripts which are generated based upon the game state. Having the power of a general-purpose language as a meta-language over the embedded language is the pivotal aspect of our approach which renders it powerful enough to encode complex AI strategies with ease. Finally, we have evaluated our method qualitatively and have shown, by means of a number of ideal criteria, that it compares well enough to other scripting methods.

7.4 Final Remarks

Game scripting is often overlooked in the game production process. However, scripts, as content, can make or break a game by themselves. There are various methods available for adding scripting, each with its own strengths and weaknesses. However, we believe that our approach is one of the first attempts at bringing the advantages of DSELs to the area of game scripting. Game scripting greatly benefits from the abstraction offered by DSLs and our approach improves upon this by making the process of DSL creation easier, faster and efficient. Having both a domain-specific scripting language and a general purpose meta-language available for use is the key feature of this approach which allows us to not only script games but also to encode the AI we require with relative ease. Our approach attempts to combine all the advantages more common methods have to offer and provides an adequate alternative method to aid game creators in their endeavours.

Appendix A

Space General User Manual

A.1 Introduction

Space Generals is a turn-based strategy game in the style of Risk which pits two to five generals against each other with the final objective of controlling all the known galaxies of the universe. Each player takes the role of a general who leads a number of captains. A general has a high-level view of the game and concerns themselves with managing galaxies and their respective planets. Each captain, on the other hand controls one planet at a time and manages the planet's various continents and their respective countries. The generals (and their respective captains) take their turns simultaneously and each time the game-board is updated at one go to reflect the next turn. This carries on until one of the generals becomes the ruler of the known universe by successfully defeating all of his or her opponents.

Section A.2 provides a guided tutorial on how to create and play a game. Section A.3 explains some of the basic gameplay details such as what actions generals and captains can perform, the attributes of galaxies, planets, continents and countries, how a turn is resolved and how attacks themselves are resolved.

A.2 Playing the Game

1. **Website** — Point your web-browser to the Space Generals website[1]. You will be presented with the website shown in Figure A.1. Currently only two web-browsers are supported: Mozilla Firefox and Google Chrome.

2. **User Registration** — The first step is to register a new user. Click the Register button. A new pop-up appears shown in Figure A.2. Enter your name, surname and email address details in the User Details section. The email address should be an active one since notifications relating to game progression are sent to it. In the Login Details section enter a username and password. Your password must be entered twice to ensure that it has been entered correctly.

3. **Logging In** — Once your details are accepted you can login by entering your new username and password and clicking the Login button.

4. **Game Menu View** — On successful login you will be presented with the Game Menu View shown in Figure A.3. This view consists of a Log Out button which may be used to log out of the game and a number of tabs. There are six tabs in all:

   - **Games List** – All the games created are shown here.

[1]http://spacegenerals.servegame.com/

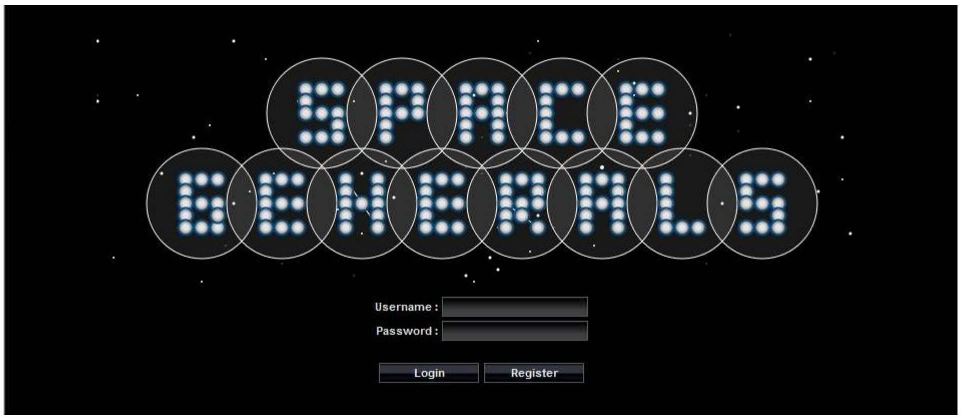
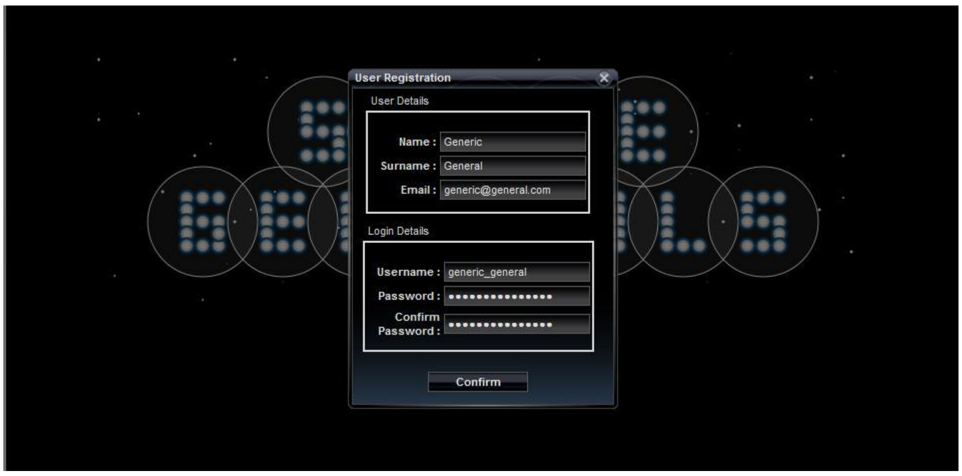


Figure A.1: Space Generals — Login Page



Name : Generic
Surname : General
Email : generic@general.com
Username : generic_general
Password : •••••••••••••••
Confirm Password : •••••••••••••••

Figure A.2: Space Generals — User Registration Popup

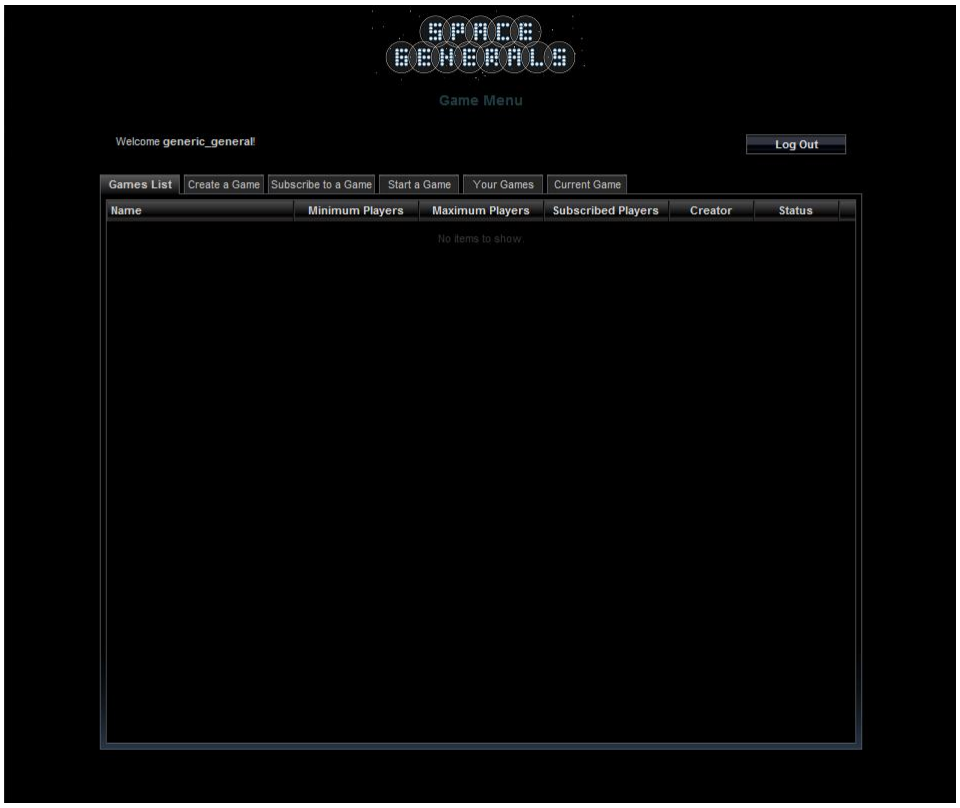
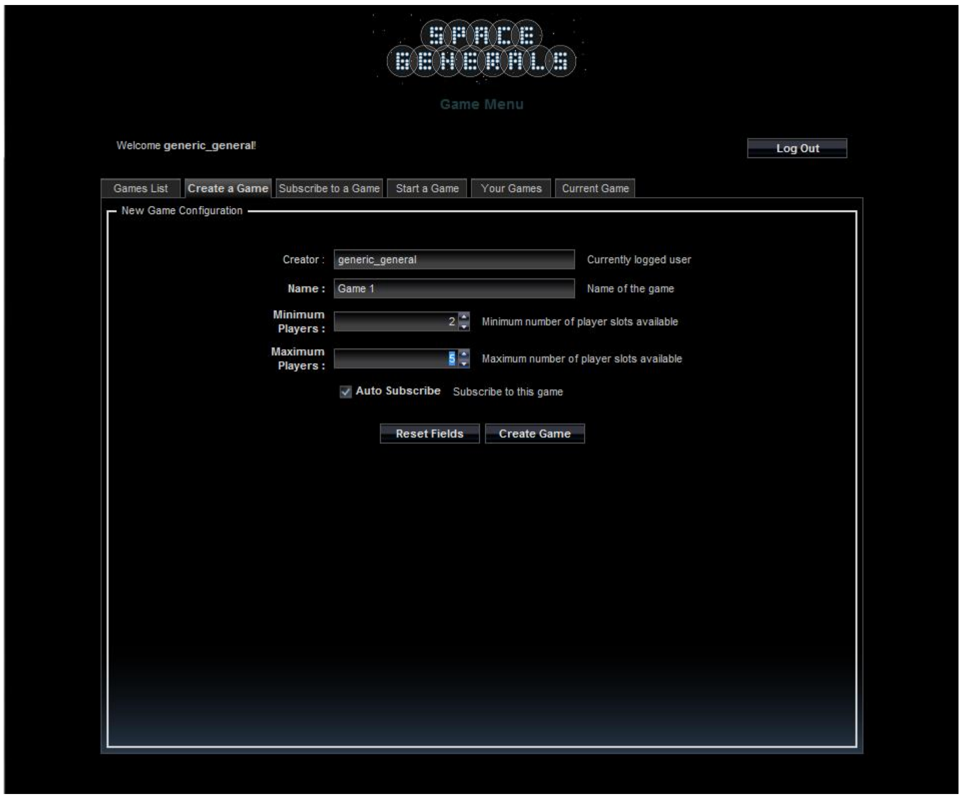


Figure A.3: Space Generals — Game Menu View (Game List Tab)

- **Create a Game** – Allows you to create a new game.
- **Subscribe to a Game** – Allows you to subscribe to a game.
- **Start a Game** – Allows you to start a game which has enough players subscribed.
- **Your Games** – Shows you all the games which you have subscribed to and which have been started.
- **Current Game** – This tab is initially disabled until you select a game from the previous tab. It displays the game you have selected and allows you to play the game.

5. **Creating a Game** — To create a game click on the Create a Game Tab. This brings up the view shown in Figure A.4. Enter a game name which will help you identify your game. Select a minimum number and a maximum number of players. The minimum number of players may vary from two to five players and signifies the minimum number of players required to start the game. The maximum number of players varies from the minimum value selected up to five players and marks the maximum number of players which may subscribe to the game. Finally select whether you want to subscribe yourself to the game automatically. You may do so later if you wish. Once the details are entered you may create a game using the Create Game button.
6. **Subscribing to a Game** — The game created appears in the Subscribe to a Game tab shown in Figure A.5 if you are **not** subscribed to it. Thus if you auto-subscribe to it on creation, you will not be able to see the game in this tab. Any other player will however be able to see the game listed in here. By selecting the game and clicking the Subscribe button in the lower-right corner of the view your opponents are able to subscribe to the game.
7. **Starting a Game** — If enough players have subscribed to your game (equal to or greater than the minimum number of players up to the possible maximum number of players), you receive an email

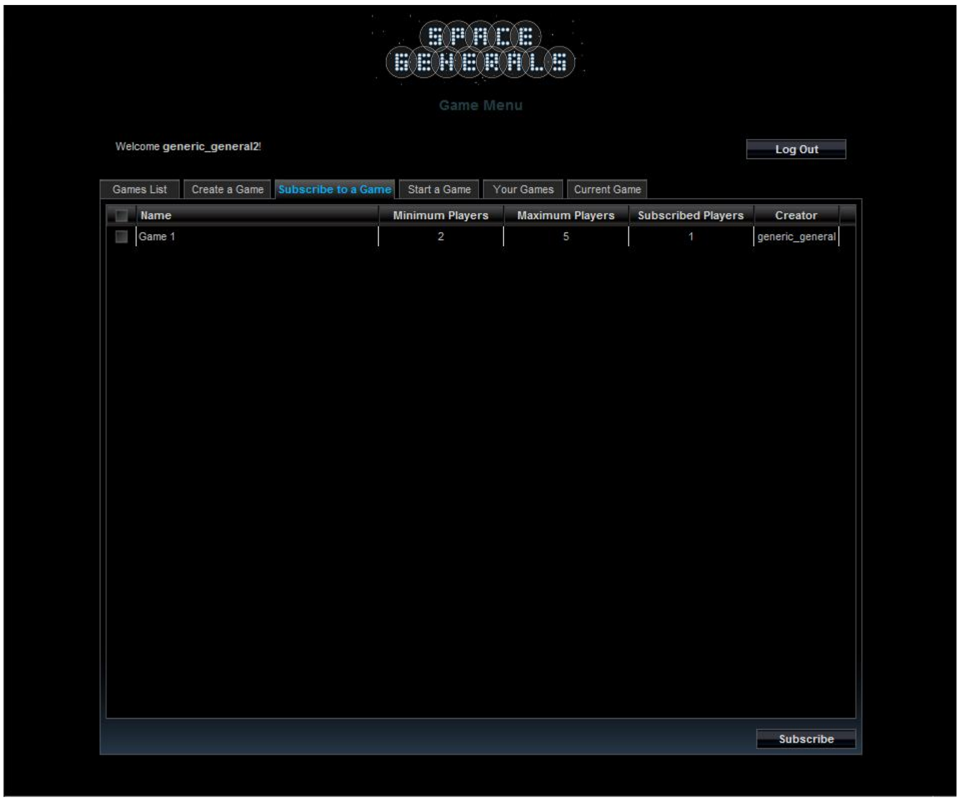


Welcome **generic_general**!

New Game Configuration

Creator :	generic_general	Currently logged user
Name :	Game 1	Name of the game
Minimum Players :	2	Minimum number of player slots available
Maximum Players :	5	Maximum number of player slots available
✓ **Auto Subscribe**		Subscribe to this game

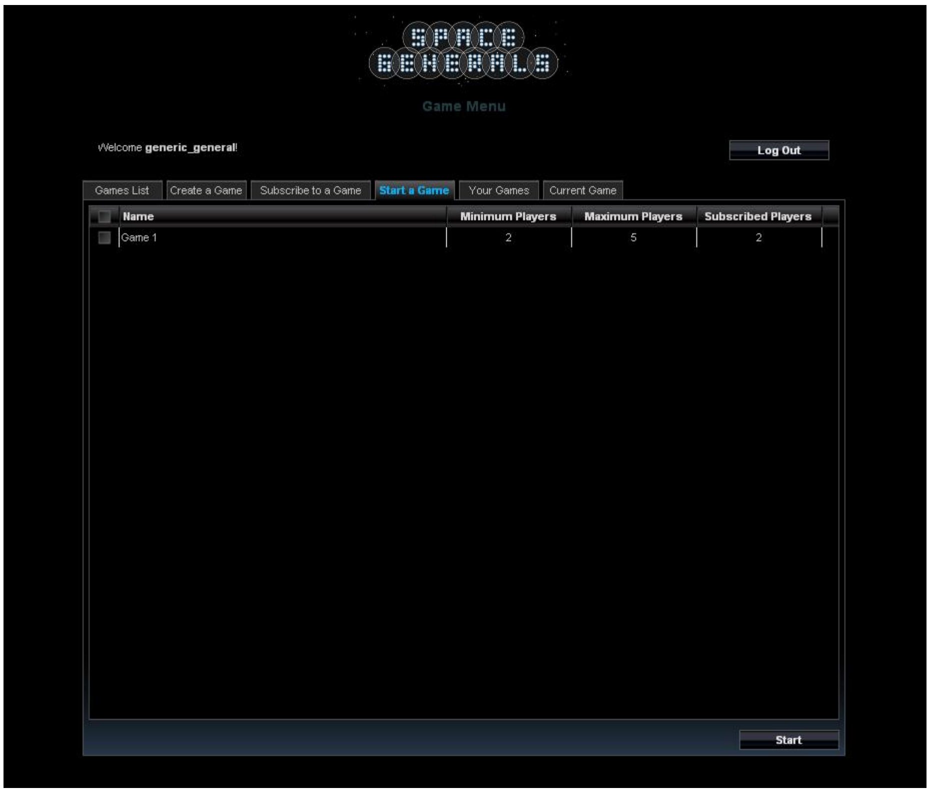
Figure A.4: Space Generals — Game Menu View (Create a Game Tab)



Welcome **generic_general2**!

	Name	Minimum Players	Maximum Players	Subscribed Players	Creator
☐	Game 1	2	5	1	generic_general

Figure A.5: Space Generals — Game Menu View (Subscribe to a Game Tab)



Name	Minimum Players	Maximum Players	Subscribed Players
Game 1	2	5	2

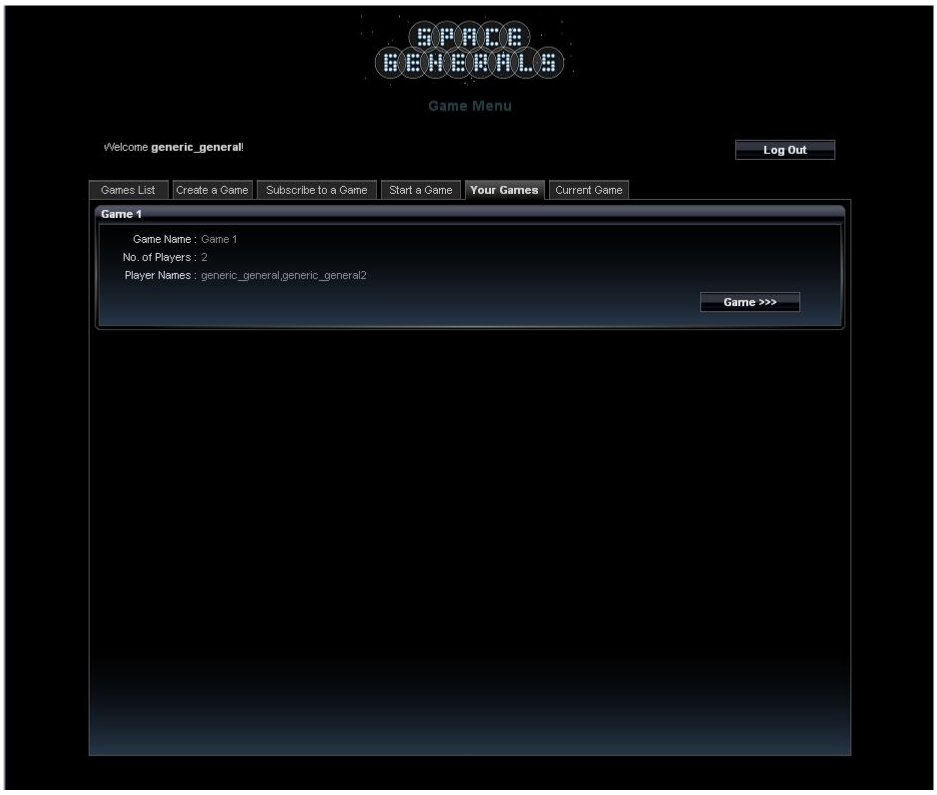
Figure A.6: Space Generals — Game Menu View (Start a Game)

which notifies you that your game is ready to be started. At this point you can head to the Start a Game tab, shown in Figure A.6, where you can select the relevant game and start it using the Start button. Once the game is started, a notification will be sent via email to all the subscribed participants denoting that the game has started and that they can login to perform the first turn.

8. **Accessing the Game** — Once a game has been started it appears listed in the Your Games tab shown in Figure A.7. To view the game and perform a turn you can press its corresponding Game >>> button. Pressing this button forwards you to the final tab which displays the game view itself and allows you to actually play the game.

9. **Viewing the Game Board – General's View** — An example game is shown in Figure A.8. Here you are presented with the general's view which displays all the galaxies and planets of the universe.

    - Planets are denoted by circles with their name underneath while galaxies are denoted by larger enclosing circles around the planets, also with their respective name underneath. The planets also have links between them, with links between planets in the same galaxy showing as thin lines and links between planets in different galaxies showing as thick lines.
    - In the lower part of the screen there is a control panel which denotes your name, colour, available resources and options. A general's four available resources are: population, credits, armies and captains. The first three resource are viewable in this panel and may be consumed to perform a variety of actions which we shall describe soon. Captains differ from other resources in the sense that they can be considered as players themselves since they have their own set of population, credits and armies available to them. We shall see what actions they can perform soon as well. For the time being one should know that they can be assigned to a planet and that by using the dropdown control labelled Captain, you can navigate to where the captain is stationed.

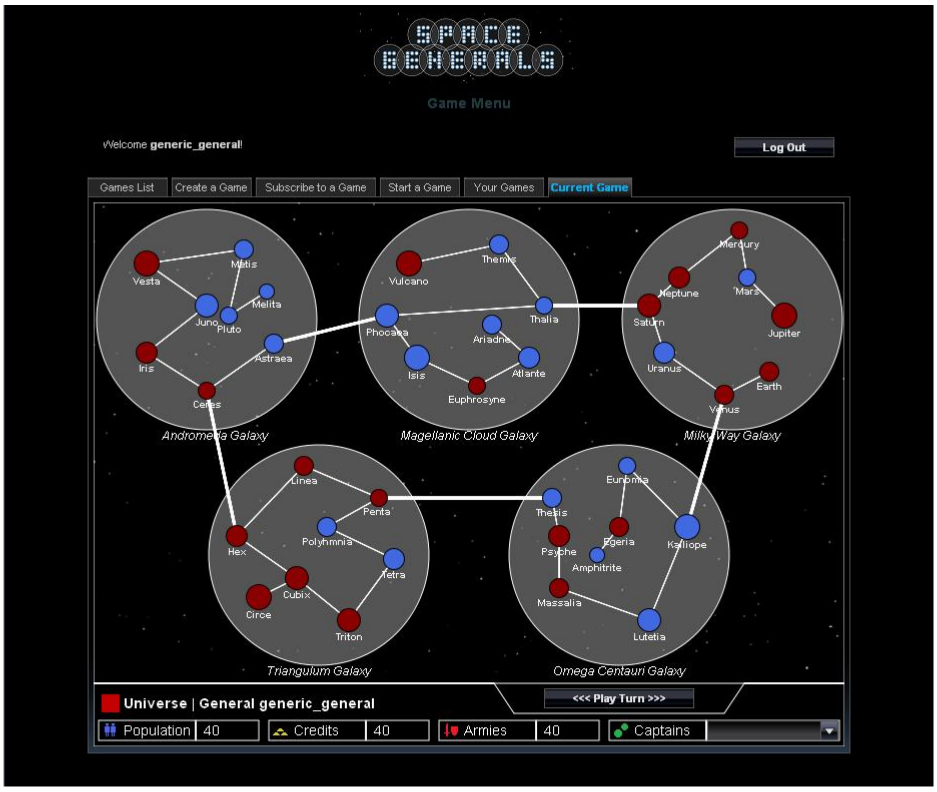


Welcome **generic_general**!

Game 1

Game Name : Game 1
No. of Players : 2
Player Names : generic_general,generic_general2

Figure A.7: Space Generals — Game Menu View (Your Games Tab)

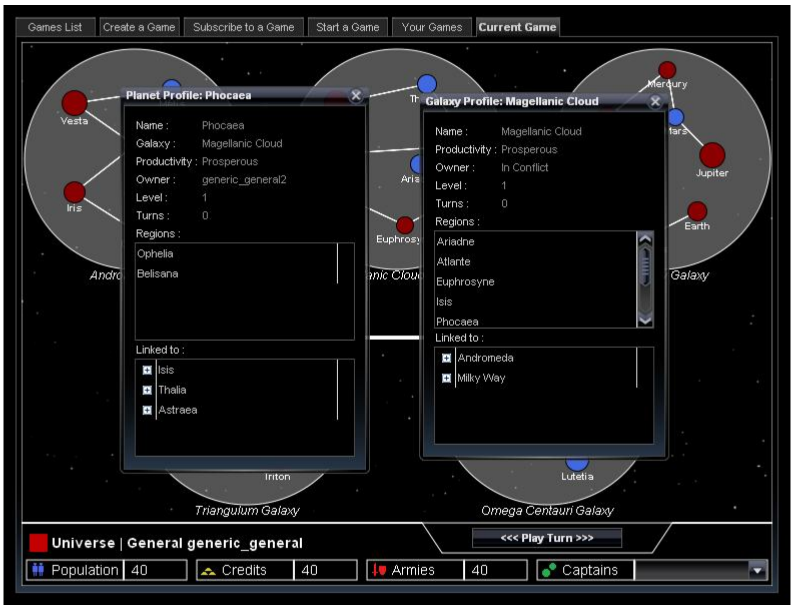


Welcome **generic_general**!

Universe | General generic_general

Population 40 | Credits 40 | Armies 40

Figure A.8: Space Generals — Game Menu View (Current Game Tab)



Planet Profile: Phocaea

Name : Phocaea
Galaxy : Magellanic Cloud
Productivity : Prosperous
Owner : generic_general2
Level : 1
Turns : 0
Regions :
Ophelia
Belisana
Linked to :
Isis
Thalia
Astraea

Galaxy Profile: Magellanic Cloud

Name : Magellanic Cloud
Productivity : Prosperous
Owner : In Conflict
Level : 1
Turns : 0
Regions :
Ariadne
Atlante
Euphrosyne
Isis
Phocaea
Linked to :
Andromeda
Milky Way

Universe | General generic_general

Population 40 Credits 40 Armies 40 Captains

Figure A.9: Space Generals — Game Menu View (Planet & Galaxy Popups)

- Initially, a general is given a home galaxy which is not completely conquered. He or she also has four captains which are assigned to four planets in this galaxy. The planets which the general owns are marked by their colour, which is the same as the general's colour in the lower panel. It is up to the player to decide whether to capture their home galaxy first or expand elsewhere to stop their opponents from obtaining their home galaxy.
- Capturing a galaxy is achieved by capturing all its planets and is recommended as it provides a good bonus to the general. Galaxies which a general owns are highlighted by his/her colour.
- More information regarding a relevant planet or galaxy may be accessed by clicking them with the left mouse button. This will present you with a pop-up which displays more information about the selected land as shown in Figure A.9.
- To view a particular planet right click on it with your mouse. This will take you to the captain's view.

10. **Viewing the Game Board – Captain's View** — An example captain view is shown in Figure A.10 which shows a planet called Saturn.

    - In this view, countries are denoted by the smaller circles with their name underneath, while continents are denoted by the larger circles, also with their name underneath. The countries have links between them, with links between countries in the same continent showing as thin lines and links between countries in different continents showing as thick lines.
    - Certain countries are marked slightly different then others due to them having links to other countries which are in other planets. Such countries are immensely strategic as they provide you with a means of defending a planet from attack or vice-versa, provide you with a staging point to attack a planet. An example of such countries is Li in Figure A.10.
    - As with planets and galaxies, countries and continents which you own have your colour.
    - Capturing a continent by capturing all its countries is recommended as it provides a good bonus to the captain.
    - Countries, differing from larger land types, have credits, population and armies within them. The first two are generated each turn while the armies must be deploy there directly. We shall discuss

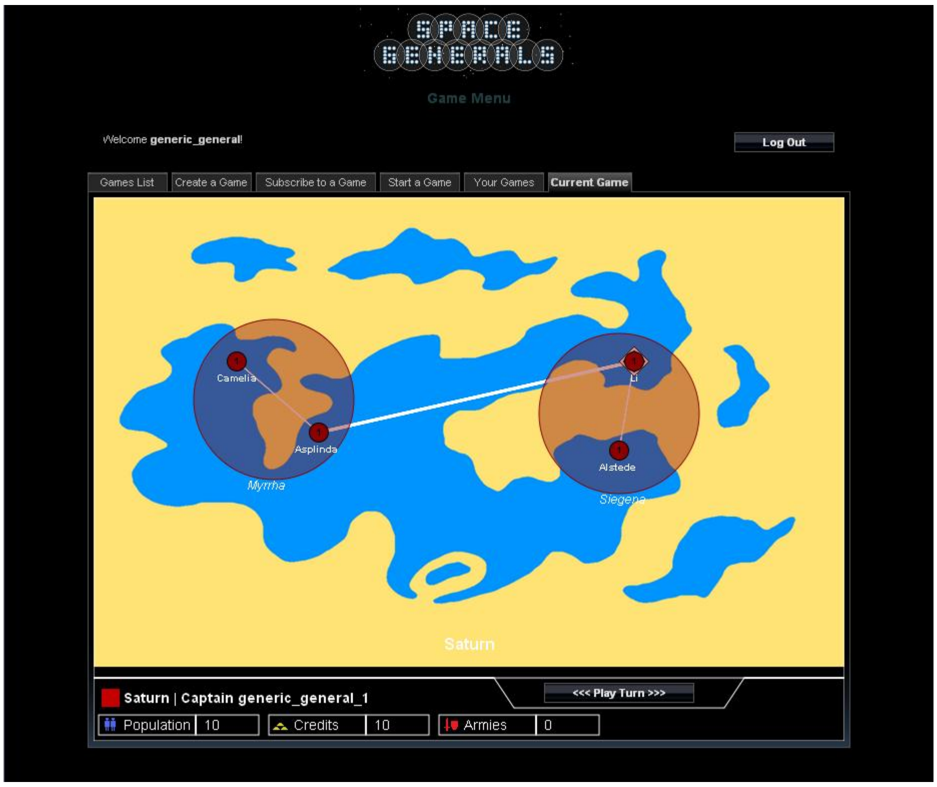


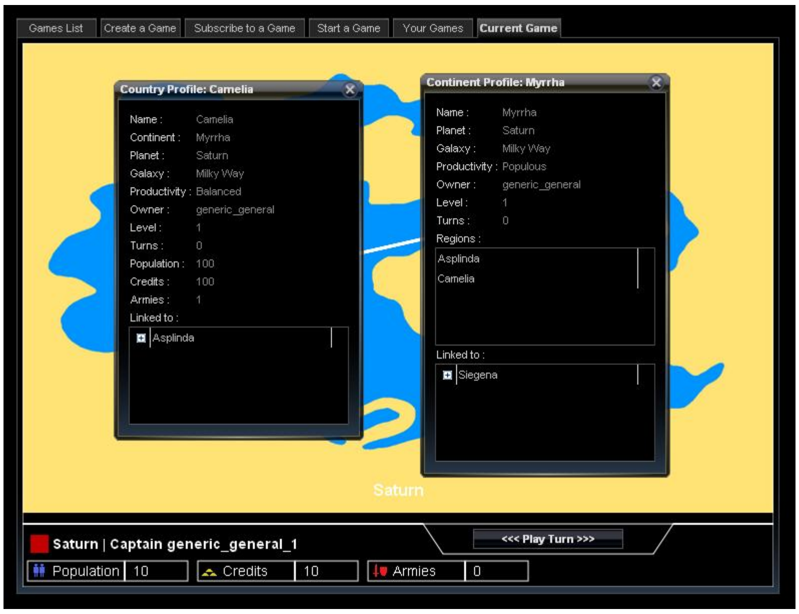
Figure A.10: Space Generals — Game Menu View (Captain's View)

more on this later on. It is however important to know that each country must have one or more armies stationed in it. This is denoted by the number within the circle representing the relevant country. To view the other two resources hover your mouse over a country. This shows a small tooltip which displays how much population, credits and armies, the country has within it.

- Similarly to planets and galaxies, all the information regarding a country or continent may be accessed by clicking them with the left mouse button. This will present you with a pop-up which displays more information about the selected land as shown in Figure A.11.
- If a captain is stationed upon the current planet a panel similar to the one for the general (with credits, population and armies values) is displayed on the lower section of the view. If a captain is not assigned this panel is not shown.
- You can right click anywhere in this view to go back to the general's view.

11. **Playing your turn** — The <<< Play Turn >>> button on the lower right of the general's view or captain's view may be clicked to perform your turn. Clicking this button makes the Turn Menu pop-up show. This popup is divided into three expandable sections as shown in Figure A.12. The three sections are:

    - **Status Update** – This section gives you a brief overview of what has happened in the previous turn such as who gained which land and how attacks were resolved.
    - **Turn** – This section allows you to actually perform your turn both as a general or as a number of captains which as a general you control. The first phase of the turn consists of the general's actions which must be done before proceeding to the next phase consisting of the captains' turns. The latters' turns may be performed in parallel to one another as their actions are independent.
      Phase 1: General's Turn – The upper panel in this section displays the resources available to the general while the lower section is divided into three sub-sections as shown in Figure A.13.



Country Profile: Camelia

Name : Camelia
Continent : Myrrha
Planet : Saturn
Galaxy : Milky Way
Productivity : Balanced
Owner : generic_general
Level : 1
Turns : 0
Population : 100
Credits : 100
Armies : 1
Linked to :
Asplinda

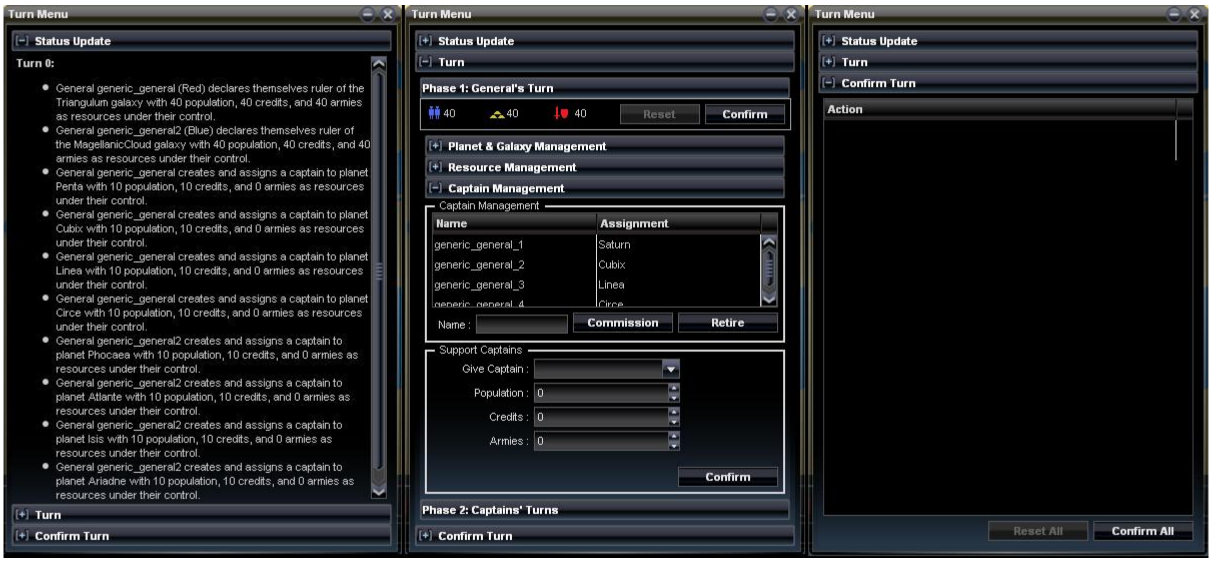
Continent Profile: Myrrha

Name : Myrrha
Planet : Saturn
Galaxy : Milky Way
Productivity : Populous
Owner : generic_general
Level : 1
Turns : 0
Regions :
Asplinda
Camelia
Linked to :
Siegena

Saturn | Captain generic_general_1

Population 10 Credits 10 Armies 0

Figure A.11: Space Generals — Game Menu View (Country & Continent Popups)



Turn 0:

- General generic_general (Red) declares themselves ruler of the Triangulum galaxy with 40 population, 40 credits, and 40 armies as resources under their control.
- General generic_general2 (Blue) declares themselves ruler of the MagellanicCloud galaxy with 40 population, 40 credits, and 40 armies as resources under their control.
- General generic_general creates and assigns a captain to planet Penta with 10 population, 10 credits, and 0 armies as resources under their control.
- General generic_general creates and assigns a captain to planet Cubix with 10 population, 10 credits, and 0 armies as resources under their control.
- General generic_general creates and assigns a captain to planet Linea with 10 population, 10 credits, and 0 armies as resources under their control.
- General generic_general creates and assigns a captain to planet Circe with 10 population, 10 credits, and 0 armies as resources under their control.
- General generic_general2 creates and assigns a captain to planet Phocaea with 10 population, 10 credits, and 0 armies as resources under their control.
- General generic_general2 creates and assigns a captain to planet Atlante with 10 population, 10 credits, and 0 armies as resources under their control.
- General generic_general2 creates and assigns a captain to planet Isis with 10 population, 10 credits, and 0 armies as resources under their control.
- General generic_general2 creates and assigns a captain to planet Ariadne with 10 population, 10 credits, and 0 armies as resources under their control.

40 40 40

Captain Management

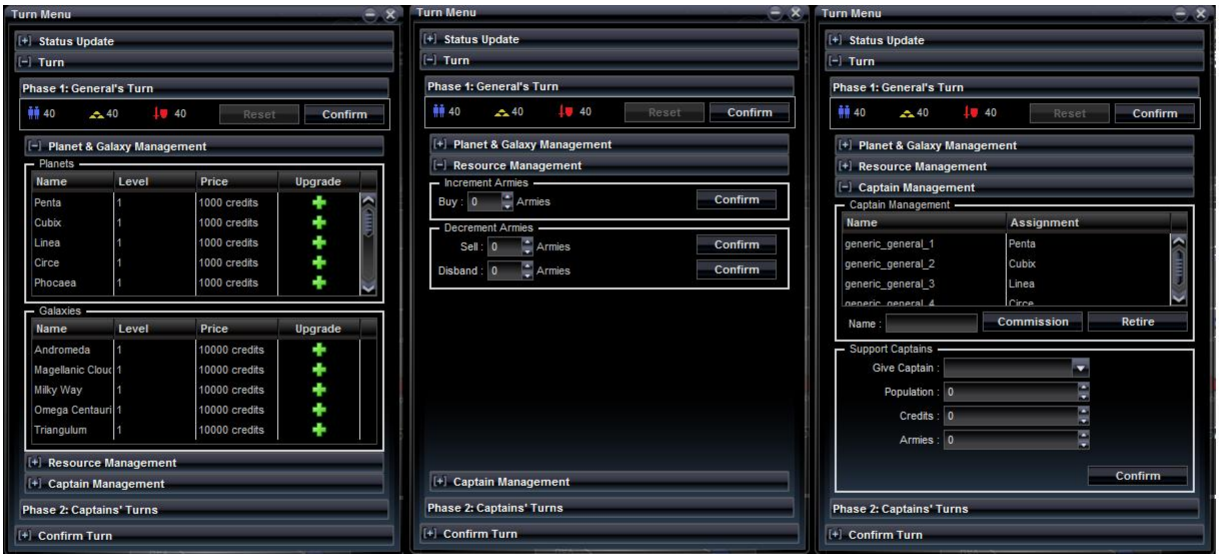
Name	Assignment
generic_general_1	Saturn
generic_general_2	Cubix
generic_general_3	Linea
generic_general_4	Circe

Name :

Support Captains

Give Captain :
Population : 0
Credits : 0
Armies : 0

Figure A.12: Space Generals — Turn Menu



Turn Menu

[+] Status Update

[−] Turn

Phase 1: General's Turn

40 40 40 Reset Confirm

[−] Planet & Galaxy Management

Planets

Name	Level	Price	Upgrade
Penta	1	1000 credits	✚
Cubix	1	1000 credits	✚
Linea	1	1000 credits	✚
Circe	1	1000 credits	✚
Phocaea	1	1000 credits	✚

Galaxies

Name	Level	Price	Upgrade
Andromeda	1	10000 credits	✚
Magellanic Clouc	1	10000 credits	✚
Milky Way	1	10000 credits	✚
Omega Centauri	1	10000 credits	✚
Triangulum	1	10000 credits	✚

[+] Resource Management

[+] Captain Management

Phase 2: Captains' Turns

[+] Confirm Turn

Turn Menu

[+] Status Update

[−] Turn

Phase 1: General's Turn

40 40 40 Reset Confirm

[+] Planet & Galaxy Management

[−] Resource Management

Increment Armies

Buy : 0 Armies Confirm

Decrement Armies

Sell : 0 Armies Confirm

Disband : 0 Armies Confirm

[+] Captain Management

Phase 2: Captains' Turns

[+] Confirm Turn

Turn Menu

[+] Status Update

[−] Turn

Phase 1: General's Turn

40 40 40 Reset Confirm

[+] Planet & Galaxy Management

[+] Resource Management

[−] Captain Management

Captain Management

Name	Assignment
generic_general_1	Penta
generic_general_2	Cubix
generic_general_3	Linea
generic_general_4	Circe

Name : Commission Retire

Support Captains

Give Captain :

Population : 0

Credits : 0

Armies : 0

Confirm

Phase 2: Captains' Turns

[+] Confirm Turn

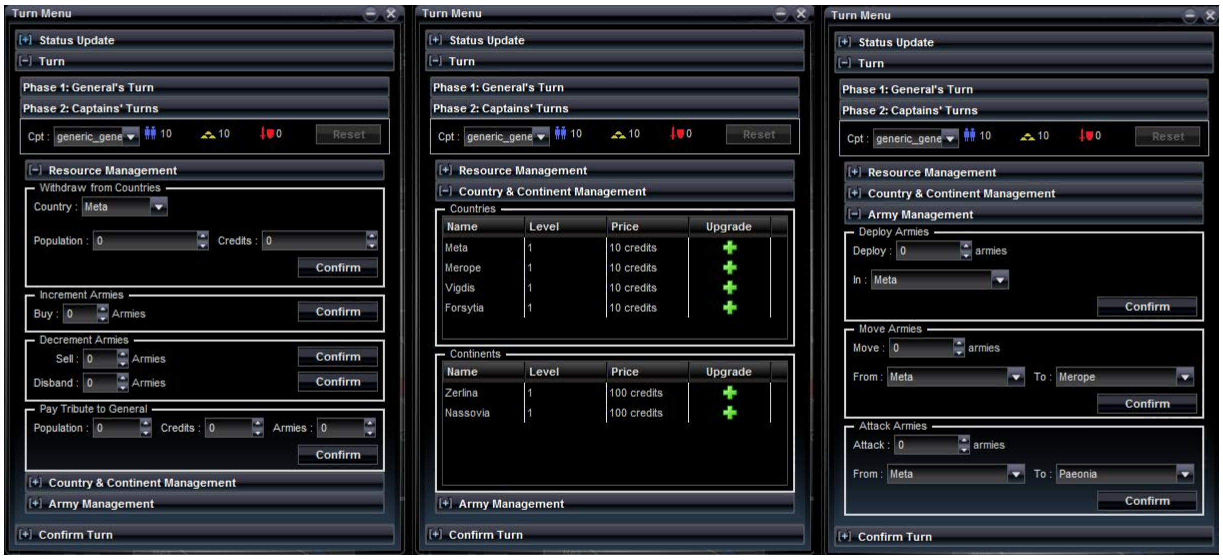
Figure A.13: Space Generals — Turn Menu: General's Turn Sections

These sections categorize the different available actions of the general by the area which they affect, be it lands (planets and galaxies), resources (population, credits and armies) and finally captains:

– **Planet & Galaxy Management** (Figure A.13, left) – This section allows the general to upgrade the level of a planet or galaxy. The level of a planet or galaxy affects the bonus generated and given to the general each turn and the higher the level the larger the bonus obtained. Each land type, including planets and galaxies, always start at level one and the general can decide whether or not to spend credits to increment a land's level. Ideally a land is levelled if the general plans to hold it at all costs (such as the home-galaxy) since if the planet or galaxy is lost the bonus is given to the general who owns it and it is not possible to reduce or refund the level upgrade performed. To perform an upgrade, find the planet or galaxy in the list which you want to upgrade and click on their respective "+" icon.

– **Resource Management** (Figure A.13, middle) – This section deals with incrementing or decrementing of armies by trading them with credits and/or population. Armies are an important aspect of gameplay as they allow you to actually conquer new territories and expand your empire. Here you can decide to spend some credits and sacrifice some population in order to train armies, or sell or disband armies and receive a refund in money and population (always less than what you spent to buy them), respectively. To buy armies, sell armies or disband armies enter a valid value in the respective section and click the confirm button. Buying, selling or disbanding armies can only occur in multiples of 5: 5, 10, 15 and so on.

– **Captain Management** (Figure A.13, right) – This section allows the general to manage their captains and is divided into two. The upper part allows the general to re-assign a captain to another planet or unassign them completely. The reason for reassigning is that planets without a captain may contain countries/continents which are owned by the general, but without a captain present the armies in these countries may not be used for attack or any other command. This means that, for example, if a captain attacks a country in another planet with armies from a country in the current planet and there is no captain in the receiving planet, the armies are leaderless and cannot be used to attack again until a captain is assigned to the receiving planet. Also planets without captains do not receive bonuses due to the number of countries owned or whole continents owned. This means that



Name	Level	Price	Upgrade
Meta	1	10 credits	✚
Merope	1	10 credits	✚
Vigdis	1	10 credits	✚
Forsytia	1	10 credits	✚

Name	Level	Price	Upgrade
Zerlina	1	100 credits	✚
Nassovia	1	100 credits	✚

Figure A.14: Space Generals — Turn Menu: Captain's Turn Sections

since captains are quite expensive to commission one might have to decide which planet to assign a captain to in order to obtain the highest bonus. In order to re-assign or unassign a captain find the captain in the list and select the appropriate planet. Once the planet is selected the captain is automatically re-assigned or unassigned. As mentioned here it is also possible to commission a new captain or retire a captain. In the first case the cost of the captains grows the more captains you have, so one might have to consider one's options carefully before deciding to spend credits in training a new captain. Also retiring a captain never refunds the full value of credits used to train one so this should be done only in dire situations only. To buy commission a captain enter a name in the textbox provided and click the Commission button. If you wish to retire a captain enter his name in the same textbox and click the Retire button. Any resources which the retired captain might have held are given directly to the general as well. Finally, the lower part of this section allows you to support any one of your captains with any population, credits and armies you might decide to supply them with in order to help them in their efforts to win or defend a planet. To do this select a captain from the dropdown control and enter the required population value, credits value and/or army value and then click Confirm.

All of the above actions may be performed any number of times required, depending on your selected strategy. You can review your actions in the following section of this pop-up (see below). When you are satisfied you can select the Confirm button which is situated in the upper part of this section near the resources display. This confirms the general's actions and takes you to phase two to perform the captains' actions.

Phase 2: Captains' Turns – This phase is divided amongst the captains the general commands and can be seen in Figure A.14. In the upper part one can find a dropdown which allows you to select which captain you wish to perform actions with, along with their respective resources. The lower part on the other hand is divided into three section in a similar fashion to the general's. This time however the sections are concerned with resources (population, credits and armies), lands (countries and continents) and finally armies:

– **Resources Management** (Figure A.14, left) – For the captains this section is divided into four parts. The first part is used to withdraw resources directly from a country. It is worthwhile to note that the amount of population and credits left in a country affect its growth (population- and credits-wise) in the next turn. So it is up to the captain to decide when is the right moment to withdraw resources. For example, when the captain needs armies to ensure that a country is captured he might withdraw all the resources completely. Otherwise they might leave the resources in certain countries so these can act as resource-generators and then withdraw them at a more suitable moment such as in the case that they are going to be overrun by an enemy captain so that the latter find them completely empty. To withdraw resources from a country, select the country from the dropdown control, enter a value in the population and/or credits box and click Confirm. Remember that you can view the resources available in a country by hovering over it with the mouse or by clicking on it to show its pop-up. The next two sections are related to buying, selling and disbanding armies and behave in exactly the same way as those for generals. The last section is also similar to the general's captain-supporting section. A captain may decide to act as a supplier for his general and give them any surplus resources which they might have collected. To do this enter a value for population/credits and/or armies and click the Confirm button.

  – **Country & Continent Management** (Figure A.14, middle) – This section in analogous to the Planet & Galaxy Management section for generals and may be used to upgrade countries or continents to increase the bonuses these generate.
  – **Army Management** (Figure A.14, right) – This section is what distinguishes a captain from a general and shows their division of concerns. While a general controls captains, it is the captains' job to control directly how armies are used. This section is dedicated to helping the captains perform this job. The upper part allows a captain to deploy armies into a country. Such armies may be used either to defend the country better or to then use those armies to attack an adjacent enemy country. To deploy armies, select the required country on the planet, enter a value of armies and click Confirm. The middle part is concerned with moving armies from a controlled country into another controlled country. This is useful since once deployed armies cannot be removed from the country but can only be moved from country to country. Note that the armies may move between planets this way. Again if the to-country is in a planet with a captain assigned, these armies may be used by the receiving captain. Otherwise they simply defend the country in which they land. To move armies, select a from-country and the required to-country, enter a number of armies and click Confirm. Finally, in the last section one can attack with armies from an owned country to an adjacent enemy one. This allows you to conquer new territories with the end-purpose of winning a country and possibly a continent, a planet and a galaxy. While attacks are entered the same into the game, their resolution differs depending on the situation at hand. For more information see Section A.3.4. To attack a country, similarly to moving armies, select a from-country and the required to-country, enter a number of armies and click Confirm.

- **Confirm Turn** – Once all the captains actions are performed you can review them (along with the general's actions) in the Confirm Turn section. If you are satisfied you can confirm them using the Confirm button in this section at which point the game may proceed to the next turn once all the players have performed their turn. Once this occurs you receive an email which notifies you that all the players have performed their turn and that a new turn has begun. If you are logged out of the game, simply relogging shows you the new game turn. If you are logged in, you can refresh the page by reselecting it from the Your Games tab or simply refresh the page through your web-browser's refresh button directly.

Note: At the time of writing of this manual the reset buttons found in the Turn Section and Confirm Turn Section are disabled. These are planned to allow you to reset your actions performed either as a general only or as a general and captains together in case you change your mind about a particular action taken. For the time being you can reset your actions simply by refreshing the web-page via your web-browser. This also allows you to re-enter your actions in case you change your mind about your previous, confirmed submission.

12. **End Game** — When a number of turns have been performed and a general manages to conquer all the lands from other generals they become the ruler of universe and have won the game. When a game is won, no further turns may be performed but the game can always be viewed if needed. Also players which lose all their countries lose the game and may no longer perform any turns. However they can still act as spectators and view the game's progress.

A.3 Gameplay

This section provides a brief overview of the gameplay and acts as a quick reference to the game's roles, land types and how turns and attacks are resolved.

A.3.1 Roles

As highlighted earlier there are two roles:

General

A general can perform a number of actions:

- Upgrade a planet using credits
- Upgrade a galaxy using credits
- Buy armies – convert population and credits into armies
- Sell armies – convert armies back into credits
- Disband armies – convert armies back into population
- Commission a new captain using credits
- Retire a captain receiving back a number of credits
- Reassign a captain from one planet to another
- Support a captain by giving him directly population, credits and/or armies.

Captain

A captain can perform the following actions:

- Upgrade a country using credits
- Upgrade a continent using credits
- Withdraw population and/or credits from a country under their control
- Buy armies – convert population and credits into armies
- Sell armies – convert armies back into credits
- Disband armies – convert armies back into population

- Pay tribute to general – give any surplus resources to the respective general
- Deploy armies in a controlled country
- Move armies from a controlled country into a controlled adjacent one
- Attack with armies from a controlled country into an enemy one

A.3.2 Lands

There are four land types:

Country

- the most fundamental land type
- generates population each turn based on the population available
- generates credits each turn based on the credits available and the number of turns held
- holds armies which defend it against attacking armies
- generates a population bonus and a credit bonus per turn. These bonuses is given directly to the captain controlling the planet the country is in if one is assigned there, otherwise the bonuses are lost. The bonuses generated are based on the country's level (upgradeable) and productivity factor (fixed). There are four productivity factors:

  Populous – best for population bonus/worst for credits bonus

  Prosperous – worst for population bonus/best for credits bonus

  Barren – worst for population bonus/worst for credits bonus

  Balanced – mildly good for population bonus/mildly good for credits bonus
- may be linked to other countries via links – these links are used to transfer armies and to attack adjacent enemy countries. A country can be linked to another country in the same continent or in two different continents. In the second case the continents may be either in the same planet or in two different planets and so on up to the galaxy level.

Continent

- regional land – a collection of countries
- if all the countries in a continent are held it generates a population bonus and a credit bonus per turn. These bonuses are given directly to the captain controlling the planet the continent is in if one is assigned there, otherwise the bonuses are lost. The bonuses generated are based on the continent's level (upgradeable) and productivity factor (fixed). The productivity factors are similar to the ones described for countries earlier.

Planet

- regional land – a collection of continents
- if all the continents in a planet are held it generates a population bonus and a credit bonus per turn. These bonuses are given directly to the general who owns the planet. The bonuses generated are based on the planet's level (upgradeable) and productivity factor (fixed). The productivity factors are similar to the ones described for countries earlier.

Galaxy

- regional land – a collection of planets
- if all the planets in a galaxy are held it generates a population bonus and a credit bonus per turn. These bonuses are given directly to the general who owns the galaxy. The bonuses generated are based on the galaxy's level (upgradeable) and productivity factor (fixed). The productivity factors are similar to the ones described for countries earlier.

A.3.3 Resolving a Turn

A turn is resolved in the following manner:

- **Independent Instructions** – instructions which do no affect one another such as the buying or selling of armies per captain or general are performed here en-masse and sequentially in the order entered.
- **Resolve Attacks** – attack actions are grouped together and categorized and filtered such that they are then resolved together simultaneously. See Section A.3.4 for more information on attack resolution.
- **Upgrade Lands** – all the countries, continents, planets and galaxies are upgraded here. The reason they are not upgraded earlier is that it is possible that two or more generals or captains may decide to upgrade the same land at the same time. When this occurs the cost is split evenly between them and the rest is refunded to them.
- **Update all countries population and credits** – each country's population and credits are updated according to their current values.
- **Give bonuses to Captains/Generals** – bonuses are given to each captain per country and continent which they own in the planet they are assigned to. The same applies to generals with regards to planets and galaxies. If a planet is without a captain the bonuses generated by the former are lost.
- **Pay Taxes to Generals** – a percentage of the bonus gained by the captains is taken by the general as tax.

A.3.4 Resolving Attacks

When resolving attacks, the former are resolved simultaneously. To achieve this they are first categorized into four categories and then are carried out one after the other a category at a time. The four categories are:

- **Mass Invasions** – Mass Invasions occur when a country is attacked by more than one neighbour, possibly by more than one captain. When this occurs a battle my result in either the defending country successfully defeating the attackers, or one ore more attackers winning the country from the attacker. If one attacker remains he/she becomes the new owner of the country. If more than one attacker remains the battle becomes a Spoils of War (see later) which is resolved later on.
- **Border Clashes** – Border Clashes occur when two countries decide to attack one another. When this occurs a border clash is first fought. Either of the winners is give the right to attack the country which lost as a Normal Invasion (see later).
- **Normal Invasions** – These are the most basic form of battles where one country is attacking another country where the latter simply defends itself. If the country attacking wins the battle, the armies left after the battle are stationed in the new country after the turn. If the attacker loses the defending army holds the country for the owner minus any loses incurred during the battle.

- **Spoils of War** – Spoils of war battles are fought between attackers who have won the same country and always end up with one attacker winning over another. The remaining armies of the attacker who won the spoils of war battle are stationed in the country which was originally being attacked and the country becomes theirs.

Some notes:

- The game operates on a dice roll mechanism where higher rolls win over lower ones.
- Each battle consists of a number of dice rolls until one of the armies is defeated completely (reduced to zero).
- Normally if an attacker and a defender roll the same it is the defender who loses. This is to encourage attackers.
- The dice used is a 6-sided one but the one used is never unbiased. This fact is used to allow for larger armies to have a slight advantage over smaller ones. There are 6 types of biased dice in all and which one is used depends solely on the army size used to attack or defend. The larger the army the more the diced is biased towards a particular number rather than another with smaller armies having dices which are biased towards 1s and 2s and larger armies having dices which are biased towards 5s and 6s.
- When two armies roll the loser of the roll has a number of armies deducted from their count. This affects which dice is used in the next roll with dwindling armies having a lesser chance to win. However, it is possible that with luck a smaller army manages to win versus a larger one.
- The number of armies deducted on a loss depends on the smallest army. If the smallest army is small only 1 army point is removed per dice roll while if the armies are both large more than one army value is deducted.

Thanks for taking the time to read this manual and we hope that you will enjoy playing the game!

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[2] Emil Axelsson, Koen Claessen, and Mary Sheeran. Wired: Wire-aware circuit design. In *CHARME*, pages 5–19, 2005.

[3] Jon Bentley. Programming pearls: little languages. *Communications of the ACM*, 29(8):711–721, 1986.

[4] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In *In International Conference on Functional Programming*, pages 174–184. ACM Press, 1998.

[5] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.

[6] Gaetano Caruana and Gordon Pace. Embedded languages for origami-based geometry. In *Proceedings of Computer Science Annual Workshop*. Departments of Computer Science and AI, University of Malta, 2007.

[7] James Cheney and Ralf Hinze. First-class phantom types. Technical report, Cornell University, 2003.

[8] Koen Claessen. *Embedded Languages for Describing and Verifying Hardware*. PhD thesis, 2001.

[9] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM.

[10] Koen Claessen and Gordon Pace. An embedded language framework for hardware compilation. In *In Designing Correct Circuits*, 2002.

[11] Koen Claessen and David Sands. Observable sharing for functional circuit description. In *In Asian Computing Science Conference*, pages 62–73. Springer Verlag, 1999.

[12] Koen Claessen and Mary Sheeran. A tutorial on Lava: A hardware description and verification system, 2000.

[13] Steve Cook, Gareth Jones, Stuart Kent, and Alan Wills. *Domain-specific development with visual studio dsl tools*. Addison-Wesley Professional, 2007.

[14] Antony Courtney, Henrik Nilsson, and John Peterson. The yampa arcade. In *Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 7–18, New York, NY, USA, 2003. ACM.

[15] Maria Cutumisu, Curtis Onuczko, Matthew McNaughton, Thomas Roy, Jonathan Schaeffer, Allan Schumacher, Jeff Siegel, Duane Szafron, Kevin Waugh, Mike Carbonaro, Harvey Duff, and Stephanie Gillis. Scriptease: A generative/adaptive programming paradigm for game scripting. *Science of Computer Programming*, 67(1):32–58, 2007.

[16] Maria Cutumisu, Curtis Onuczko, Matthew McNaughton, Thomas Roy, Jonathan Schaeffer, Allan Schumacher, Jeff Siegel, Duane Szafron, Kevin Waugh, Mike Carbonaro, Harvey Duff, and Stephanie Gillis. Scriptease: A generative/adaptive programming paradigm for game scripting. *Science of Computer Programming*, 67(1):32–58, 2007.

[17] Jeroen Dobbe. A domain-specific language for computer games. Master's thesis, 2007.

[18] Conal Elliott. Modeling interactive 3d and multimedia animation with an embedded language. In *DSL'97: Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*, pages 285–296, Berkeley, CA, USA, 1997. USENIX Association.

[19] Conal Elliott. An embedded modeling language approach to interactive 3d and multimedia animation. *IEEE Transactions On Software Engineering*, 25(3):291–308, 1999.

[20] Conal Elliott. Functional image synthesis. In *Proceedings of Bridges*, 2001.

[21] Conal Elliott. Functional images. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, "Cornerstones of Computing" series. Palgrave, March 2003.

[22] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.

[23] Conal Elliott, Sigbjørn Finne, and Oege De Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(3):455–481, 2003.

[24] S. Peyton Jones et al., editor. *Haskell 98 Language and Libraries, the Revised Report.* Cambridge University Press, April 2003.

[25] David Flanagan and Yukihiro Matsumoto. *The ruby programming language.* O'Reilly, 2008.

[26] Maria Grima and Gordon 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.

[27] Tom Gutschmidt. *Game Programming With Python, Lua, and Ruby (The Premier Press Game Development Series).* Premier Press, 2003.

[28] Ralf Hinze. Fun with phantom types. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, "Cornerstones of Computing" series. Palgrave, March 2003.

[29] Ralf Hinze, Johan Jeuring, and Andres Löh. Typed contracts for functional programming. In Philip Wadler and Masami Hagiya, editors, *Proceedings of the Eighth International Symposium on Functional and Logic Programming (FLOPS 2006), 24-26 April 2006, Fuji Susono, Japan*, volume 3945 of *LNCS*, pages 208–225. Springer, April 2006.

[30] C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–686, 1987.

[31] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28, June 1996.

[32] Paul Hudak. Haskore music tutorial. In *In Second International School on Advanced Functional Programming*, pages 38–67. Springer Verlag, 1996.

[33] Paul Hudak. Modular domain specific languages and tools. In *in Proceedings of Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[34] Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia.* Cambridge University Press, New York, NY, 2000.

[35] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

[36] Paul Hudak and Mark P. Jones. Haskell vs. ada vs. C++ vs awk vs ... an experiment in software prototyping productivity. Technical report, 1994.

[37] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua—an extensible extension language. *Software: Practice & Experience*, 26(6):635–652, 1996.

[38] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering (functional pearl). In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 280–292, New York, NY, USA, 2000. ACM.

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

[40] Samuel N. Kamin and David Hyatt. A special-purpose language for picture-drawing. In *In USENIX Conference on Domain-Specific Languages*, pages 297–310, 1997.

[41] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.

[42] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *DSL'99: Proceedings of the 2nd conference on Conference on Domain-Specific Languages*, pages 109–122, Berkeley, CA, USA, 1999. USENIX Association.

[43] Christoph Lüth. Haskell in space: An interactive game as a functional programming exercise. *Journal of Functional Programming*, 13(6):1077–1085, 2003.

[44] Alex Martelli. *Python in a Nutshell.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2003.

[45] Kenneth Lauchlin McMillan. *Symbolic model checking: an approach to the state explosion problem.* PhD thesis, Pittsburgh, PA, USA, 1992.

[46] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[47] Luana Micallef and Gordon 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.

[48] Henrik Nilsson. Dynamic optimization for functional reactive programming using generalized algebraic data types. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 54–65, New York, NY, USA, 2005. ACM.

[49] John T. O'Donnell. Generating netlists from executable circuit specifications in a pure functional language. In *In Functional Programming Glasgow, Springer-Verlag Workshops in Computing*, pages 178–194. Springer-Verlag, 1992.

[50] B. O'Sullivan, D. Stewart, and J. Goerze. *Real World Haskell.* O'Reilly, August 2008.

[51] Gordon 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.

[52] Gordon Pace and Christine Vella. Describing and verifying FFT circuits using SharpHDL. In *Computer Science Annual Workshop 2005 (CSAW'05)*. University of Malta, September 2005.

[53] John Peterson, Gregory D. Hager, and Paul Hudak. A language for declarative robotic programming. In *In International Conference on Robotics and Automation*, pages 1144–1151, 1999.

[54] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with haskell. In *First International Workshop on Practical Aspects of' Declarative Languages (PADL)*, January 1999.

[55] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for gadts. *SIGPLAN Not.*, 41(9):50–61, 2006.

[56] Morten Rhiger. A foundation for embedded languages. *ACM Transactions on Programming Languages and Systems*, 25(3):291–315, 2003.

[57] Erik Sandewall. Programming in an Interactive Environment: the "Lisp" Experience. *ACM Computing Surveys*, 10(1):35–71, 1978.

[58] André Santos. Embedding a firewall programming language into Haskell. In *SBLP'99: III Brazilian Programming Language Symposium*, Porto Alegre, Brazil, May 1999.

[59] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16, New York, NY, USA, 2002. ACM.

[60] Mary Sheeran. Finding regularity: describing and analysing circuits that are not quite regular. In *Proceedings 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 4–18. Springer-Verlag, 2003.

[61] Mary Sheeran. Generating fast multipliers using clever circuits. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design, FMCAD 2004*, LNCS 3312. Springer-Verlag, November 2004.

[62] Mary Sheeran. Hardware design and functional programming: a perfect match. *Journal of Universal Computer Science*, 11(7):1135–1158, July 2005.

[63] Pieter Spronck, Marc Ponsen, Ida Sprinkhuizen-Kuyper, and Eric Postma. Adaptive game ai with dynamic scripting. *Machine Learning*, 63(3):217–248, 2006.

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

[65] Walid Taha, Paul Hudak, and Zhanyong Wan. Directions in functional programming for real(-time) applications. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 185–203. Springer-Verlag, 2001.

[66] Simon Thompson. *Haskell: The Craft of Functional Programming.* Addison Wesley, July 1996.

[67] Mark G. J. van den Brand, Arie van Deursen, Jan Heering, H. A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. The asf+sdf meta-environment: A component-based language development environment. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 365–370, London, UK, 2001. Springer-Verlag.

[68] Alex Varanese. *Game Scripting Mastery (The Premier Press Game Development Series)*. Course Technology Press, Boston, MA, United States, 2002.

[69] Philip Wadler. Monads for functional programming. In *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*. Springer-Verlag, 1993.

[70] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *In ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–252, 2000.