Dissertation

THE WAYFINDING METAPHOR— COMPARING THE SEMANTICS OF WAYFINDING IN THE PHYSICAL WORLD AND THE WWW

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften unter der Leitung von

O.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Frank

E127

Institut für Geoinformation und Landesvermessung

eingereicht an der Technischen Universität Wien

Fakultät für Technische Naturwissenschaften und Informatik

von

Dipl.-Ing. Mag. art. Hartwig Hochmair

Matrikelnummer 9272156

Margaretenstraße 52/12a

1040 Wien

Wien, Juni 2002

Dissertation

THE WAYFINDING METAPHOR— COMPARING THE SEMANTICS OF WAYFINDING IN THE PHYSICAL WORLD AND THE WWW

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Technical Sciences

submitted to the Vienna University of Technology

Faculty of Science and Informatics

by

Dipl.-Ing. Mag. art. Hartwig Hochmair

Margaretenstraße 52/12a

1040 Wien

Advisory Committee:

Univ.Prof. Dipl.-Ing. Dr.techn. Andrew U. Frank

Institute for Geoinformation

Vienna University of Technology

Univ.Prof. Dipl.-Ing. Dr.techn. Werner Kuhn

Institute for Geoinformatics

University of Münster, Germany

Vienna, June 2002

For my father

per aspera ad astra

ABSTRACT

Wayfinding is a common human task. The terms ‘wayfinding’ and ‘navigation’ are traditionally associated with an activity that takes place in the real world. The development of new electronic media induces humans to navigate artificially created environments, e.g., the World Wide Web (WWW), computer games, or virtual environments. Although real environment and artificial environment show different features—e.g., in the definition of a distance between places or in the organization of space—we claim that the *concepts* of wayfinding in the real world can also be found in the WWW.

A goal of the thesis is to determine what the term *wayfinding* means, i.e., to describe the semantics of *wayfinding*. Analyzing several wayfinding definitions in literature we found that there is no unique meaning for the term *wayfinding*, although there seem to be some core properties of the underlying process. Therefore we consider wayfinding to represent a radial category. From the definitions analyzed we get the central meaning of wayfinding, and describe it through a set of axioms. The axioms define constraints on agent and environment. If the axioms are satisfied, the activity performed by the agent describes a wayfinding process.

Another goal of the thesis is to show that within the wayfinding metaphor, the semantics of wayfinding is similar for both the real world and the WWW. We hereby abstract the conceptual wayfinding model through algebraic specifications and give two parallel instantiations. We show that both instantiations satisfy the axioms, and thus the term ‘wayfinding’ can also be used for the Web space—expressing a similar semantics as in the physical world.

The axioms are invariant under the applied strategy and the type of environment. Therefore we can choose any wayfinding strategy that is capable of coping the wayfinding tasks given in the two cases studies (where the environment is unknown to the agent). The chosen wayfinding strategy relies on ‘information in the world’ and applies a semantic decision criterion. A wayfinding simulation shows that the formal algebraic specifications of the agent-based model are executable.

KURZFASSUNG

Wegesuche ist ein Teil des täglichen Lebens. Die Begriffe *Wegesuche* und *Navigation* werden traditionsgemäß mit der realen Welt assoziiert. Durch die Anwendung von neuen Technologien findet Wegesuche auch in künstlich geschaffenen Umgebungen statt (z.B. im World Wide Web, in Computerspielen oder in der virtual reality). Obwohl sich reale Welt und künstlich geschaffene Umgebung in bestimmten Punkten unterscheiden—wie etwa in der Definition von Distanzen oder in der Strukturierung des Raumes—nehmen wir an, dass die wesentlichen Konzepte, die den Wegesuche-Prozess in der realen Welt beschreiben, auch im WWW angewendet werden können.

Ein Ziel der vorliegenden Arbeit ist es, die Bedeutung des Begriffs *Wegesuche* zu klären. Durch die Analyse von verschiedenen Wegesuche-Definitionen haben wir festgestellt, dass Wegesuche kein eindeutig beschreibbarer Prozess ist. Trotzdem scheint es einige zentrale Eigenschaften eines Wegesuche-Prozesses zu geben. Daher sehen wir den Begriff *Wegesuche* als Vertreter einer radialen Kategorie. Aus den verschiedenen Wegesuche-Definitionen extrahieren wir die zentralen Eigenschaften von Wegesuche und beschreiben diese mit Hilfe von Axiomen. Die Axiome stellen bestimmte Anforderungen an den Agenten und seine Umgebung. Wenn diese erfüllt sind, kann man den beschriebenen Prozess als *Wegesuche* bezeichnen.

Ein weiteres Ziel der Arbeit ist es, zu zeigen, dass durch die Wegesuche-Metapher die Bedeutung des Begriffs *Wegesuche* sowohl in der realen Welt als auch im Web-Raum eine ähnliche ist. Dazu formalisieren wir das konzeptuelle Wegesuche-Modell mittels algebraischer Spezifikationen, welche für zwei Typen von Agent und Umgebung instanziiert werden. Wir zeigen, dass beide Instanzen die Wegesuche-Axiome erfüllen und daher der Begriff *Wegesuche* auch im Web-Raum sinnvoll angewendet werden kann—und zwar in einer ähnlichen Bedeutung, wie er sie auch in der realen Welt hat.

Die Axiome sind unabhängig von verwendeter Wegesuche-Strategie und Art der Umgebung. Deshalb können wir dem modellierten Agenten eine beliebige Strategie zur Lösung des Wegesuche-Problems, das ihm in einer Simulation gestellt wird, geben. Dabei bewegt sich der Agent in einer ihm unbekannten Umgebung. In seinem Entscheidungsprozess verwendet der Agent Information aus der Umgebung und trifft die Entscheidungen vorzugsweise aufgrund semantischer Kriterien. Die Simulation zeigt, dass die algebraischen Spezifikationen ausführbar sind.

ACKNOWLEDGEMENTS

Writing a dissertation is a long journey. Finally reaching the destination would not have been possible without the support of a number of people.

First of all I want to thank my advisor Andrew Frank for his encouraging discussions throughout the last years. Some insight in his broad and interdisciplinary knowledge helped me to find ways out of dead ends, and circumnavigate research cliffs and obstacles along the way. The research directions and approaches he suggested were essential for writing this work. I also want to thank my second advisor, Werner Kuhn, who provided me with comments and suggestions for improvement.

I appreciated the discussions with my colleagues of the institute about my topic. Here, I want to thank my former roommate Martin Raubal, who patiently listened to all my questions and shared much of his experience with me. I think that we spent some good years together. Further thanks go to Gerhard Navratil and Andreas Grünbacher for their formal help.

I also want to thank Albert McMahon who helped me to improve the grammar and style on some sections of this treatise.

Last but not least I want to thank my parents for their encouraging words throughout all the past years. They supported me in all my decisions, let me go my own way, and gave me the feeling to do the right thing. I am sure that my father would have also been proud of me in this moment.

TABLE OF CONTENTS

1. Introduction 1
1.1 Motivation 1
1.2 Hypothesis 1
1.3 Approach 2
1.4 Contribution of the Thesis 3
1.5 Audience 4
1.6 Organization of the Thesis 4

2. The Case Studies: Wayfinding in Airports and the WWW 7
2.1 Airport Environment 8
2.1.1 Setting and Task 8
2.1.2 Ontology of the Airport Environment 9
2.1.3 Semantic and Metric Information in the Airport Environment 10
2.1.4 The Abstract Simulated Test Area 12
2.2 The Case Study in the World Wide Web 13
2.2.1 What is the World Wide Web? 13
2.2.2 Searching Tactics in the WWW 14
2.2.3 Setting and Task 16
2.2.4 Ontology of the WWW Environment 17
2.2.5 Semantic and Metric Information in the WWW Environment 18
2.2.6 The Abstract Simulated Test Area 18
2.3 Comparing the Abstract Environments 21
2.4 Summary 22

3. Contributing Disciplines 23
3.1 Metaphors 23
3.1.1 What are Metaphors ? 23
3.1.2 Classification of Metaphors 24
3.1.3 The Mapping of Semantics with Metaphors 25
3.1.4 Formal Approaches of Expressing a Metaphor 27
3.2 Wayfinding 28
3.2.1 What is Wayfinding? 28
3.2.2 Cognitive Models of Space 31
3.2.3 Epistemology—The Use of Affordances for Wayfinding in Airports and the WWW 33
3.2.4 Spatial Decision Making and Wayfinding Strategies 36
3.2.5 Simulating Human Wayfinding 38
3.3 Agent Theory 39
3.3.1 The Term ‘agent’ 39
3.3.2 Abstract Architectures of Agents 41
3.3.3 A Two-tiered Conceptual Model 42
3.3.4 The Sense-Plan-Act Paradigm 43
3.3.5 Basic Operations of an Agent-System 43
3.3.6 Properties of an Environment 45
3.4 Summary 47

4. Formal Tools 49
4.1 Algebraic Specifications 49
4.1.1 Definition 49
4.1.2 Example 50
4.2 Morphisms 51
4.2.1 Signature-Morphism 52
4.2.2 Homomorphism 53
4.2.3 Isomorphism 55
4.2.4 Morphisms in Formalized Metaphors 56
4.3 Morphisms in the Wayfinding Model 59
4.4 Category Theory 60
4.4.1 Definition 60
4.4.2 Functors 61
4.5 Type Systems and Polymorphism 63
4.5.1 Why Do We Need Type Systems? 63

4.5.2 Type Inference and Strong Typing 63
4.5.3 Polymorphism 64
4.5.4 Algebraic Data Types 65
4.5.5 Data Types and Functors 68
4.6 Functional Programming 70
4.6.1 Each Expression Is a Function 70
4.6.2 No Side-Effects 71
4.6.3 Higher-order Functions 71
4.6.4 Why We Use a Functional Programming Language 72
4.7 Haskell 73
4.7.1 Classes 73
4.7.2 Instances 74
4.7.3 Classes with Multiple Parameters 75
4.7.4 Context 75
4.7.5 Pattern Matching 76
4.8 Graph Theory 78
4.8.1 Definitions 78
4.8.2 Shortest-Path Algorithms 79
4.9 Summary 80

5. The Wayfinding Axioms 81
5.1 Informal Description 81
5.1.1 First Axiom: Decision Points 81
5.1.2 Second Axiom: The Agent Has a Goal 82
5.1.3 Third Axiom: Moving Towards a Goal 82
5.1.4 Fourth Axiom: No Impact on Environment 83
5.1.5 Fifth Axiom: Order of Actions 83
5.2 Formal Description 85
5.3 Excluded Features 87
5.4 Summary 88

6. Conceptual Features of The Wayfinding Agent 89
6.1 The Structure of the Wayfinding Agent 89
6.2 The Cognitive Map 90
6.2.1 The Role of a Cognitive Map for Wayfinding 90
6.2.2 The Cognitive Map of the Airport Navigating Agent 91
6.2.3 The Cognitive Map of the WWW Navigating Agent 92
6.2.4 Comparing the Cognitive Maps of Both Agents 98
6.3 The Decision Making Process 99
6.4 Summary 100

7. A Formal Model for Agent-Based Wayfinding in the Real World and the WWW 101
7.1 World 102
7.1.1 Structure of the World 102
7.1.2 Operations in the World 102
7.2 The Environment 103
7.2.1 Nodes 103
7.2.2 Edges 104
7.2.3 Graphs 105
7.3 Agent Structure 105
7.3.1 Fact and Beliefs 105
7.3.2 Mental Position 106
7.3.3 Cognitive Map 107
7.3.4 Perception 107
7.3.5 Decision 107
7.3.6 Preferences 107
7.3.7 Incoming Direction 107
7.4 External Operations 108
7.4.1 Class Definition and Derived Functions 108
7.4.2 The Class Context 108
7.4.3 The Semantics of the External Operations 110
7.5 The Agent's Decision Process: An Internal Operation 111
7.5.1 Sub-Processes of Semantic Decision Making 112
7.5.2 Moving Towards the Goal 112
7.5.3 The Semantic Distance to the Goal 114

7.5.4 Combining the Sub-steps 116
7.6 Summary 117

8. Discussing the Computational Model 118
8.1 Verification of the Wayfinding Axioms 118
8.2 Totality: Mappings and Functions 119
8.2.1 Totality of Mapping 119
8.2.2 Totality of Functions 120
8.3 Comparing Theory and Implementation 121
8.4 Summary 122

9. Simulation of the Wayfinding Process in an Airport Environment and the WWW 123
9.1 The Environments 123
9.1.1 Area of the Simulated Airport Environment 123
9.1.2 Area of the Simulated WWW Environment 125
9.2 The Agents 126
9.2.1 Creating the Agent’s Cognitive Map 126
9.2.2 Creating the Agent 127
9.3 The World 128
9.4 Running the Simulation 129
9.4.1 Wayfinding in the Airport 129
9.4.2 Wayfinding in the WWW 133
9.5 Summary 138

10. Conclusions and Future Work 139
10.1 Summary 139
10.2 Results and Major Findings 141
10.2.1 The Semantics of Wayfinding 142
10.2.2 Formalized Wayfinding Model 142
10.3 Future Work and Open Questions 143

References 146

Appendix 152

Biography of the Author 169

Chapter 1 Introduction

1.1 Motivation

The internet-jargon uses many metaphors that stem from the domain of wayfinding in the physical world. Examples are *move* to the previous web site, to be *lost*, *visit* a web page, *find* a web page, or *navigate* the WWW. *Wayfinding*, like many other everyday metaphors, is almost invisible as we understand it immediately. But what makes common metaphors so easy to be used? We think that the secret can be found in a small number of axioms which define the semantics of the source domain of a metaphor and which are also satisfied in the target domain. We explore if such axioms exist for the process of *wayfinding*, and if this core of the wayfinding metaphor can be represented formally. If this method was possible in general, abstract domains (e.g., a user interface) may be checked for the correct use of a specific metaphor. We will see that, on an abstract level, a metaphor can be described through a functor, i.e., a *homomorphism* between two categories, and that the semantics of a source domain can be defined by the behavior of its operations.

As the WWW has rapidly grown over the last decade, web navigation plays an increasing role in humans' everyday life. Based on their searching habits, "an alarming number of web users are not particularly efficient at reaching their online destinations" (Pastore 2001). This lack of efficiency is partly based on missing wayfinding concepts in the WWW. We assume that consideration of user needs and implementing user concepts of navigating the real world into the WWW may help to make the WWW easier to navigate. For this reason, it is important to determine what wayfinding exactly is. Metaphors have become an essential feature of human-computer interaction (Carroll and Rosson 1994) and represent a necessary ingredient to almost any user interface.

1.2 Hypothesis

Many metaphors are used for activities that take place in the internet. Metaphors map semantics from one domain to the other. An important part of this metaphor concept is therefore the task, how the semantics of the source domain can be defined. We focus on the wayfinding metaphor and explore the semantics of the term *wayfinding*.

The hypothesis of this work is: *The semantics of the term wayfinding can be defined through a set of axioms. The axioms describe minimum requirements on a domain to support a wayfinding process. This set of axioms needs to be satisfied in the source and the target domain so that the term 'wayfinding' is correctly used as a metaphor.*

The axioms will be shown to be invariant under the chosen wayfinding strategy and the type of environment.

1.3 Approach

The approach we use to explain the wayfinding metaphor is twofold. First, we find a set of axioms that define minimum requirements on property and behavior of a domain so that the process it defines can be called *wayfinding*. The axioms hereby refer to wayfinding in the real, physical world. For creating the axioms we use wayfinding definitions found in literature, which do not provide one unique meaning of the term 'wayfinding', although central elements can be found. Therefore the wayfinding process can be seen as a radial category. As there is no unique opinion about what exactly *wayfinding* is, the presented axioms cannot describe *the* correct constraints on a domain to describe a wayfinding process. The axioms rather cover a common and convenient meaning of wayfinding.

Second, we want to show that the axioms are satisfied within the WWW domain, too. For this part of the approach, we choose an algebraic wayfinding model that has two parallel instantiations, one for an abstract real world (an airport domain) and one for an abstract Web environment (a small part of the Yahoo search engine domain). We need to show that the wayfinding axioms—that are based on human wayfinding in the real world—are satisfied for both instances. We hereby try to express conceptual similarities between both instances with polymorph data types and polymorph functions. Using such formal tools, on the one side, demonstrates homomorph mappings between both domains, and on the other side, reduces the effort for the proof that both instantiations satisfy the wayfinding axioms. We demonstrate the conceptual similarity of wayfinding in the real world and the WWW on a formal level with the help of algebraic concepts.

Previous work in human wayfinding, metaphor theory, cognition, psychology, and philosophy, and existing cognitively-based computational models for wayfinding serve as a foundation for the development of the agent-based wayfinding model. We use specific concepts from the fields of artificial intelligence (i.e., agents, semantic networks), ecological psychology (i.e., affordances), computer science (i.e., interface design, WWW), and cognitive science (i.e., information processing) to design the process model. Formal tools, such as category theory, data type theory, and the theory of algebras, are the basis for the discussion of metaphors on an abstract level.

The simulation is developed as an executable specification in Haskell (Thompson 1996), a functional programming language with a syntax close to ordinary algebra. The prototype allows us to simulate the proposed wayfinding strategy of human wayfinding in an airport-environment and the WWW.

1.4 Contribution of the Thesis

The major contribution of this thesis is the ability not only to explain how a given metaphor works—which has in various approaches already been achieved (e.g., by Lakoff and Johnson 1980; Carroll, Mack et al. 1988; Fauconnier 1997; Maglio and Matlock 1998) and formalized (e.g., by MacCormac 1985; Kuhn and Frank 1991; Goguen 1999)—but also why certain metaphors do *not* exist. Let us assume some examples in the German language. The translation into English is printed in brackets:

- “Er findet einen Weg, das Problem zu lösen“ (He finds a way to solve the problem)
- „Er sucht sich mühsam seinen Weg durchs Studium“ (He tries to make his way through his studies)
- “Nach langer Krankheit fand die Sängerin zurück auf die Bühne” (After her illness she found the way back to her daily working routine)

These metaphors are used correctly and understood by (most) German native speakers. The next statements may sound somewhat strange:

- “Sie findet ihren Weg durchs Leben” (Life is searching for a way)
- “Ich finde meinen Weg durch das Turnier” (I’m trying to find my way through this tournament)

What we figure out in this thesis is, why examples like the first two mentioned are good representatives for metaphors, whereas the latter examples cause problems in their use: The latter examples violate some of the wayfinding axioms. A major task of this thesis is to provide:

- A set of axioms that define the meaning of wayfinding
- A formal representation of these axioms
- A demonstration of how abstract models can be checked for satisfying these axioms

- A summary of formal concepts that play a role in the mapping of semantics between two domains
- A computational wayfinding model that allows us to formally demonstrate the metaphorical relation between two domains (i.e., the real world and a Web domain). The formalized model hereby includes parts of Raubal's work (Raubal 2001a)—a model of agent based wayfinding in airport environments.
- The role of homomorphism for the metaphor concept can be demonstrated within the formal model.

1.5 Audience

This exploration of 'wayfinding' involves several scientific fields. Due to the inclusion of a 'physical world' realm, a 'WWW' realm, and the metaphorical connection between these two spaces, the work is interdisciplinary and targeted to researchers in the following areas:

- Software and web designers, especially those who are responsible for designing the navigation interface: They can use the basic method proposed in the thesis (axioms, formalization, proof) for checking any metaphor for its correct use in the user interface.
- Linguists: We show how a metaphor can be formally defined. The method describes the domains of a metaphor independent of a specific natural language and therefore has a high grade of generality.
- Cognitive scientists and psychologists can apply the perceptual wayfinding model in research on human wayfinding behavior in unfamiliar environments. The model can function as a starting point for human subjects testing in this area.
- Researchers in artificial intelligence: They can use the formalized agent-based model for an increasing domain of people's everyday lives, namely navigation in the WWW.

1.6 Organization of the Thesis

In the next chapter we present the two case studies employed in this thesis, i.e., wayfinding at the Vienna International Airport and wayfinding in the directories of the Yahoo-domain. We describe the particulars of wayfinding in an airport and the WWW and the ontology of both domains. We give a description of the settings and an abstraction of the test area. Further, we specify the wayfinding task for both types of agents.

Chapter 3 reviews previous research in related disciplines. We look at the historical understanding of the role of metaphor in human life, and discuss some models that describe the mapping of semantics between domains with the help of metaphors, including formal approaches. We review scientific fields concerning the modelling of wayfinding, covering wayfinding definitions, cognitive models of space, decision making, epistemological concerns, and existing computational models. Further we introduce agent theory including abstract models of agents and their environments.

In chapter 4 we look at those formal tools that we need to build the computational model of the wayfinding agent and that provide a basis for the description of the wayfinding metaphor on an abstract level. We discuss the role of morphism, category theory, and polymorphism in respect to metaphors, and introduce the functional programming language Haskell.

In chapter 5 we define those axioms that express the minimum requirements on agent and environment so that the described activity can be considered as wayfinding process. The axioms are based on analyzed wayfinding definitions. The formalization of the axioms provides a high grade of generality which in turn allows us to map the axioms from the physical world to abstract domains.

In chapter 6 we discuss conceptual features of the wayfinding agent including the agent's structure, cognitive map, and decision strategy. We hereby point out commonalities between the concepts used in the two instantiations of the agent. We look at the requirements concerning content and design of the agent's cognitive map with respect to a given wayfinding task.

In chapter 7 we develop a formalized wayfinding model that is based on the conceptual model for perceptual wayfinding. The features from the domains (i.e., airport and WWW) are separated into two parallel instantiations. In the description of the formal model we focus on those parts that are fundamental for the verification of the wayfinding axioms.

Chapter 8 discusses the formal model on several points. We summarize the verification of the wayfinding axioms for both instantiations. We discuss the totality of operations and the totality of morphisms between both instantiations in the formal model. Further, we look at which of the theoretical aspects of formalizing a metaphor have been in fact realized in the formal model and have been used to express the wayfinding metaphor.

In chapter 9 we simulate wayfinding at the Vienna International Airport and in the Yahoo-domain. We hereby feed the formalized wayfinding models with two different data sets. The simulation checks if the formal model is executable for both instantiations of agent and environment. We compare and analyze decisions made by both types of agent within the simulation.

Chapter 10 presents the conclusions and directions for future work. The appendix shows the complete Haskell code of the agent based wayfinding model.

CHAPTER 2 THE CASE STUDIES: WAYFINDING IN AIRPORTS AND THE WWW

We use two case studies that clarify the concepts and mechanisms underlying a wayfinding process. Wayfinding—the agents' tasks in the two case studies mentioned here—is just an example for one of many metaphors and therefore a contribution to the long quest for a definition of what a metaphor is (Fauconnier and Turner 1998). The case study introduced in this section demonstrates that the meaning of a term can be mapped from the real world to another domain, keeping its conceptual features, and therefore expressing a metaphor.

The first case study describes a wayfinding task at the Vienna International Airport (VIE). Raubal's PhD-thesis (Raubal 2001a) focuses on a conceptual model of understanding signage at airports, the informational needs of the navigator at decision points, and how the perceived information can be used for wayfinding in an unknown airport environment. His work further classifies potential errors of the signage in airports. In contrast to Raubal, this thesis does not focus on details of signage and errors but intends to show that the basic operations of the wayfinding process are also used in the WWW, i.e., that the wayfinding metaphor maps this set of basic operations from the physical world to the WWW. The second case study is situated in the portal of the Yahoo-directories in the WWW. It describes an agent's task to find a Web page of a specific content within the directory structure.

Both, wayfinding in an airport environment and searching the directories of a Web portal are two specific modes of wayfinding. Within the two instantiations in the formalized model, we show that particularities of the two domains and task specific instantiations (i.e., those features that are expressing additional functionality compared to the wayfinding axioms) are not taken into consideration during the proof of satisfying the wayfinding axioms, and therefore do not play a role in defining the wayfinding metaphor.

The ontology of the environment plays an important role for wayfinding, as the percepts from the environment function as input for the decision making process. For each of the two settings we will discuss the ontology and its abstractions for the wayfinding model.

For the conceptualization and simulation of the wayfinding process we presume the following:

- The navigator has never visited the environment before.

- The navigator does not have a map with information about the environment.
- Communication with other passengers is not included, i.e., the navigator cannot ask another person the way to his goal.

2.1 Airport Environment

2.1.1 Setting and Task

The first case study is situated in the departure level of Terminal 1 in the Vienna International Airport (VIE). Figure 1 shows the central part of the departure level indicating the major possibilities for passengers' movements.

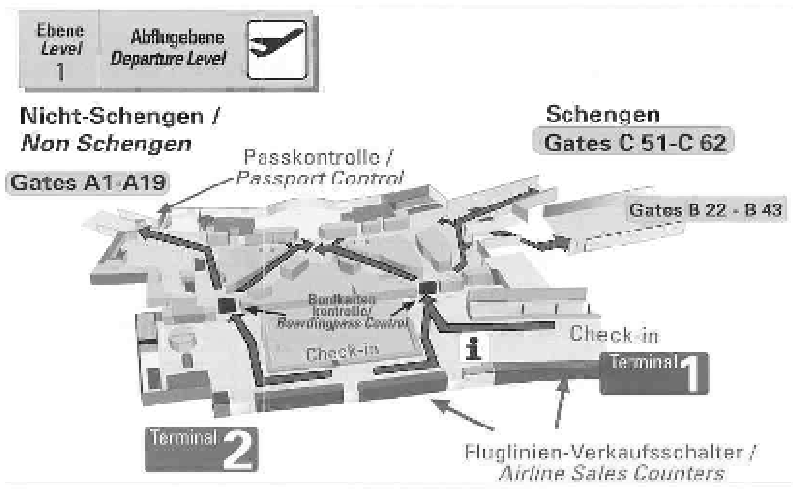


Figure 1: Central part of the departure level at VIE (from Raubal 2001a)

When planning to flight departure from the airport, passengers first have to check in their luggage at one of the check-in counters after which they receive a boarding pass, which tells them their boarding gate and the latest time by which they must arrive at this gate. The gates are labeled with the letters A, B, or C—denoting the three different gate areas at VIE—and a number. The navigator's task in our case study is to find the way from one of the check-in counters to the gate 'C 54' which is located in the Schengen-terminal of the airport (upper-right corner in Figure 1).

As the environment is unknown to the simulated passenger, he relies partly on his general topological knowledge of airport environments ('knowledge in the head'), and navigational cues from the environment ('knowledge in the world') (Norman 1988). The wayfinding strategy that the agent applies will be discussed in section 6.3.

2.1.2 Ontology of the Airport Environment

Before we explain the ontology of airport environments it is necessary to clarify the term *ontology*, as there exist several views about what ontology is. By defining the ontology of a specific domain, one describes what is in this domain in a general way (Gruber 1993). From an Artificial Intelligence (AI) approach, ontologies are content theories which identify specific classes of objects and relations that exist in some domain (Chandrasekaran, Josephson et al. 1999). In its most prevalent use in AI, an ontology refers to engineering artifacts, constituted by a specific vocabulary used to describe a certain reality, plus a set of explicit assumptions regarding the intended meaning of the vocabulary. In a more philosophical sense, ontology represents a subfield of philosophy, which can be defined as the science of what is: the science of various types and categories of objects and relations in all realms of being (Smith 2001a). Ontology in simple terms, attempts to classify entities. It deals with the question which basic kinds of things exist. The basic kinds of things are known as categories. The system of types and categories does not depend on a particular language: Aristotle's ontology is always the same, independent of the language used to describe it. Philosopher-ontologists are concerned with the things themselves (the objects, properties and relations, the states, events and processes) within a given domain. Contrarily, ontological engineers are concerned rather with languages, descriptions, or concepts, and with software representations constructed to a given domain, or with representations in people's heads (Smith 2001b). To solve terminological impasse between these two meanings of ontology, Guarino (1997) introduces the term *conceptualization* for the philosophical reading, using the term ontology for the AI reading.

In his wayfinding model on airports, Raubal (2001a) uses the term 'ontology' in Gruber's sense, i.e., as a description of what is in a specific domain or microworld in a general way. Following Gibson's approach (Gibson 1979), Raubal subdivides the wayfinding environment into *medium*, *substances*, and *surfaces* and constructs the ontology of navigational elements from interviews in which people described their experiences during wayfinding in airports (Raubal, Egenhofer et al. 1997). This method is based on ontology from texts. A taxonomy of substances that is based on "IS-A" relations which allows making transitive inferences is given in (Raubal 2001b). Among the non-cognizing objects of an airport it distinguishes between bona-fide objects (architectural component, information device, counter, gate) and fiat objects (area, navigational elements).

Aristotle characterizes everything that exists into certain categories: substance, quality, quantity, relation, etc. *Substance* (a synonymous for 'individuals') is prior to the other categories since substances exist as separate entities, while the other categories exist only as the *qualities* of substance. For each substance there is one or more qualities which are inseparable from it. This would be its *essence*, its essential quality.

For the proposed simulated wayfinding strategy used in this thesis, gates, information devices (signs), and navigational elements (decision points and paths), are the objects with which the agent interacts during his wayfinding process. All other substances in the discussed hierarchy, such as airport staff, terminal, or recreational areas, are excluded from the abstract environment in the computational model.

We consider the *semantic* information represented on signs as a separate layer of ontology (besides the *substances*), as this layer describes a logical structure of the airport environment in a semantic way. The structure of this layer will be discussed in the next section. Thus, we have following classification within the airport ontology:

- Layer of substances
- Layer of semantic structure (represented through information on airport signs)

2.1.3 Semantic and Metric Information in the Airport Environment

As well as semantic information, the metric of the environment plays a role in the simulated decision making process. Signs are essential information devices, as they represent ‘knowledge in the world’ (Norman 1988) and allow the agent to make wayfinding decisions that lead him closer to the goal. A sign is attached to a corridor which leads towards the gate(s) displayed on the sign. The gate signs can be classified into three categories (Raubal 2001b):

- signs with a single content (e.g., ‘A’, ‘C54’)
- signs with a list of content (e.g., ‘C52, C53’, ‘A,C’)
- signs with a range (e.g., ‘C52-C54’, ‘A-C’)

Concerning the number of attached signs and the usability for the decision process we classify corridors (we abstract them as edges between nodes in the computational model) into three categories (Figure 2). An edge can have a sign on none of its nodes, on one of its nodes, or on both nodes.

(a) No signpost is attached to either of the two nodes. The edge as topological connection between two nodes exists, but there is no information available (at either of the two nodes) that could be used as semantic input for the decision making process. Thus the edge will not be passed in any direction by the agent. We call this a *non-labeled* edge.

(b) A signpost is attached to one of the two nodes: Based on the information on the sign the agent may enter the edge from the node where the signpost is attached (*directed* edge).

(c) A signpost is attached to both of the two nodes: The edge can be entered from both sides (*undirected* edge).

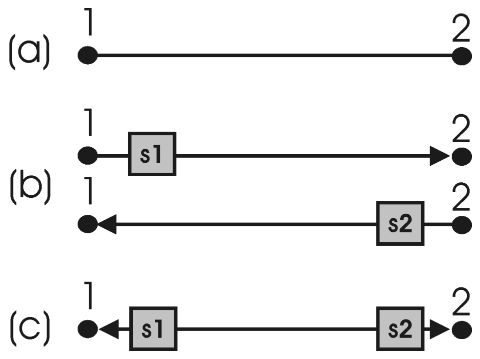


Figure 2: Classification of edges after the number of attached signs

The ‘direction’ of edges in the computational model therefore is defined by the agent’s need for information from signs, and not through topologic or physical constraints (this also holds for the WWW case). Thus, the graph itself is undirected in the model.

Besides the topology of a decision point and the number of perceived signs, metric attributes of outgoing edges (i.e., the configuration) plays a role in the decision process. To model the influence of metric (i.e., directions) in the decision making process, it is sufficient to schematize the directions of edges on decision points for the computational model (Casakin, Barkowsky et al. 2000), e.g., into a schema of 45 degree-angles. Each node can be given a local reference frame with eight directions (Raubal 2001a), where the direction of each of the outgoing edges falls into one of the eight directions. Thus, a wayfinding agent is offered semantic and metric input when reaching a decision point. In the example visualized in Figure 3, three signs (‘A’, ‘A,C’, and ‘B,C’) can be perceived from the position (indicated through the black circle), where the angles between the edges (i.e., the signs) are indicated through curved arrows.

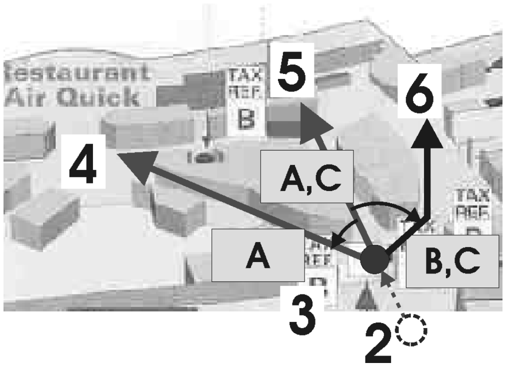


Figure 3: Perceiving semantic and metric input at a decision point

2.1.4 The Abstract Simulated Test Area

In the selected area, signposts are attached either to one or to both sides of a corridor. Thus, the simulated environment—which is abstracted as a *graph* (see section 4.8)—contains directed and undirected edges (Figure 2). The visualization of the graph for Figure 4 omits signposts but displays nodes with their IDs, the geometry of these decision points, the position of the check-in counter (start point), and the agent's goal 'C 54'.

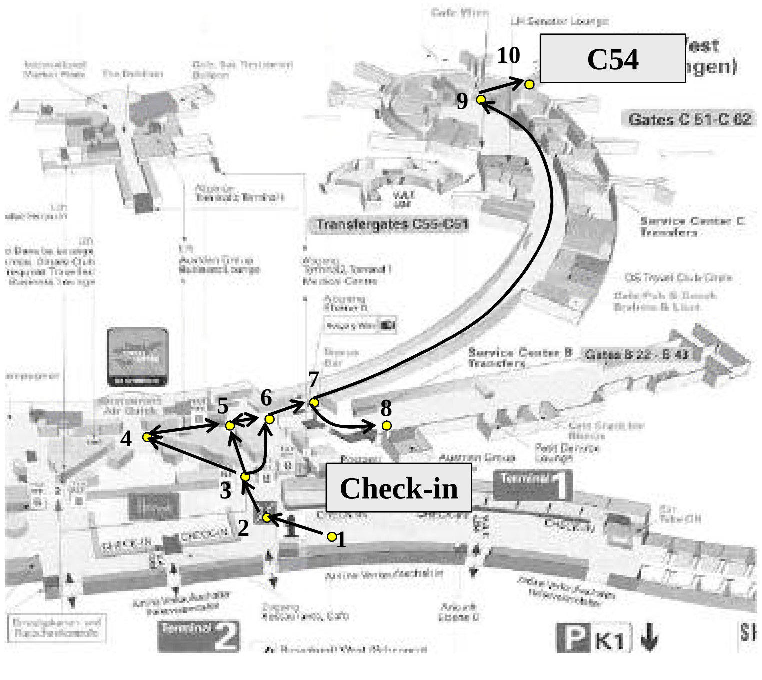


Figure 4: Environmental graph in the airport area

The test graph contains 11 nodes (including a fictive node 0 as reference for the agent's previous position at the start point) with connecting edges. Table 1 summarizes the mentioned properties of all edges of the graph line by line. The value in the first column gives the node ID, the second column gives the direction of the signpost in the local reference frame of the first node (for a detailed description see Figure 46 in the simulation chapter), the third column describes the semantic content of the sign. Columns 4-6 describe the same elements for the second node of the edge. Either a sign offers metric and semantic information, or it is not attached (which is expressed through an empty value in the columns 'Direction of sign' and 'Sign content'). Correspondingly, edges in the test environment are *undirected* or *directed*.

Position	**Direction of Sign**	**Sign content**	**Position**	**Direction of Sign**	**Sign content**
1	1	A – D	2	-	-
2	0	A, B, C, D	3	-	-
3	1	A	4	-	-
	0	A, C	5	-	-
	7	B, C	6	-	-
4	6	C	5	2	A
5	6	B, C	6	2	A
6	6	B, C	7	-	-
7	5	B	8	-	-
	6	C 51 – C 62	9	-	-
9	7	C 54	10	-	-

Table 1: Directed and undirected edges in the airport graph

2.2 The Case Study in the World Wide Web

2.2.1 What is the World Wide Web?

The World Wide Web (WWW) does not have a physical location like physical environments, e.g., airports, have. It is "the universe of network-accessible information, an embodiment of human knowledge", as the World-Wide-Web Consortium, an organization the Web inventor Tim Berners-Lee helped found, defines the WWW. A more technical definition defines the World Wide Web as "all the resources and users on the Internet that are using the Hypertext Transfer Protocol (HTTP)" (whatis.com 2002).

The Word Wide Web is sometimes considered as identical to the internet. Others define the WWW as part of the internet, as the internet—besides the WWW—also comprises electronic mail (e-mail) or electronic telephony (chat). An outstanding feature of the WWW is *hypertext*, a method of instant cross-referencing. Hypertext is the organization of information units into connected associations that a user can choose to make. An instance of such an association is called a link or hypertext link. Most Web browsers underline hypertext links and represent them in a different color. Hypertext was the main concept that led to the invention of the World Wide Web.

The number of web users increases enormously. Industry analysts estimate the number of World Wide Web users to have climbed to over 150 million in the year 2000 (Figure 5) (greatlook.com 2002).

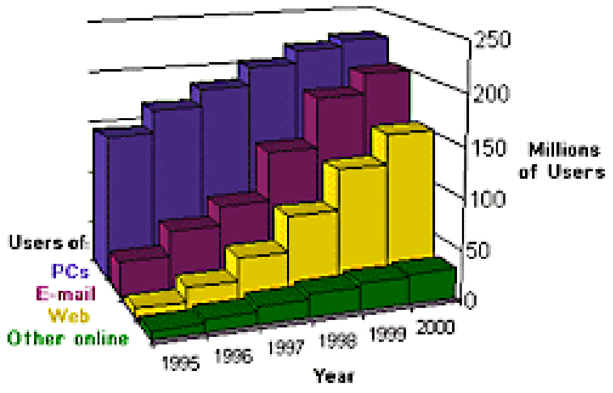


Figure 5: Worldwide users of the PCs, e-mail, WWW, and other online-services between 1995 and 2000 (from greatlook.com 2002)

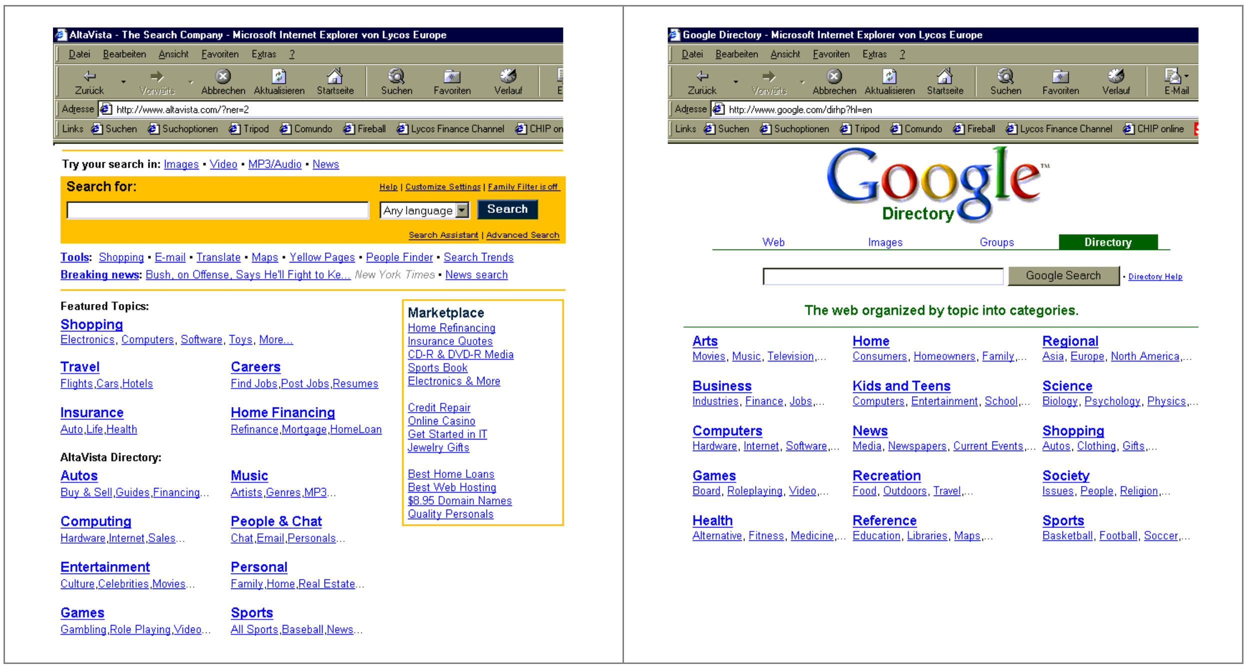
A two-year study by Alexa Research (www.alexaresearch.com) has revealed that—based on their searching habits—a high number of web users is not efficient at reaching their online destinations (Pastore 2001). Matthew Work, vice president of Alexa Research, says that the study reveals that “for many web users there is a conceptual misunderstanding of how to effectively navigate the Web”. Considering this remark from a web-designer’s view we can conclude that essential features being necessary for applying common wayfinding strategies in the real world are missing in numerous web interfaces. Otherwise, Web users would find their target web page faster. Although this thesis does not discuss all elements involved in wayfinding in the real world and the WWW, the contribution of this work is to determine which basic concepts of wayfinding should also be represented in the Web interface.

2.2.2 Searching Tactics in the WWW

Although the term *wayfinding* for the physical world is discussed in section 3.2 we classify searching tactics in the WWW, which is needed to specify the wayfinding task for this case study. Jul and Furnas (1997) distinguish between two tactics for searching and browsing activities in electronic spaces:

1. querying
2. navigation

Both of these tactics can be applied on various search engines, e.g., Yahoo, Google, or Altavista (Figure 6). Whereas the first method describes looking for a web page through entering one or several search terms in an edit field, the second method describes step-wise ‘clicking-through’ the pages on the Web domain until the desired Web page is found.


AltaVista - The Search Company - Microsoft Internet Explorer von Lycos Europe
Datei Bearbeiten Ansicht Favoriten Extras ?
Zurück Vorwärts Abbrechen Aktualisieren Startseite Suchen Favoriten Verlauf
Adresse http://www.altavista.com/?ner=2
Links Suchen Suchoptionen Tripod Comundo Fireball Lycos Finance Channel CHIP on
Google Directory - Microsoft Internet Explorer von Lycos Europe
Adresse http://www.google.com/dirhp?hl=en
Links Suchen Suchoptionen Tripod Comundo Fireball Lycos Finance Channel CHIP online
Search
Google Search
Any language

Try your search in: Images • Video • MP3/Audio • News

Search for: Help | Customize Settings | Family Filter is off

Search Assistant | Advanced Search

Tools: Shopping • E-mail • Translate • Maps • Yellow Pages • People Finder • Search Trends
Breaking news: Bush, on Offense, Says He'll Fight to Ke... New York Times • News search

Featured Topics:
Shopping
Electronics, Computers, Software, Toys, More...
Travel
Flights,Cars,Hotels
Careers
Find Jobs,Post Jobs,Resumes
Insurance
Auto,Life,Health
Home Financing
Refinance,Mortgage,HomeLoan

AltaVista Directory:
Autos
Buy & Sell,Guides,Financing...
Music
Artists,Genres,MP3...
Computing
Hardware,Internet,Sales...
People & Chat
Chat,Email,Personals...
Entertainment
Culture,Celebrities,Movies...
Personal
Family,Home,Real Estate...
Games
Gambling,Role Playing,Video...
Sports
All Sports,Baseball,News...

Marketplace
Home Refinancing
Insurance Quotes
CD-R & DVD-R Media
Sports Book
Electronics & More

Credit Repair
Online Casino
Get Started in IT
Jewelry Gifts

Best Home Loans
Best Web Hosting
$8.95 Domain Names
Quality Personals

Google Directory

Web Images Groups Directory

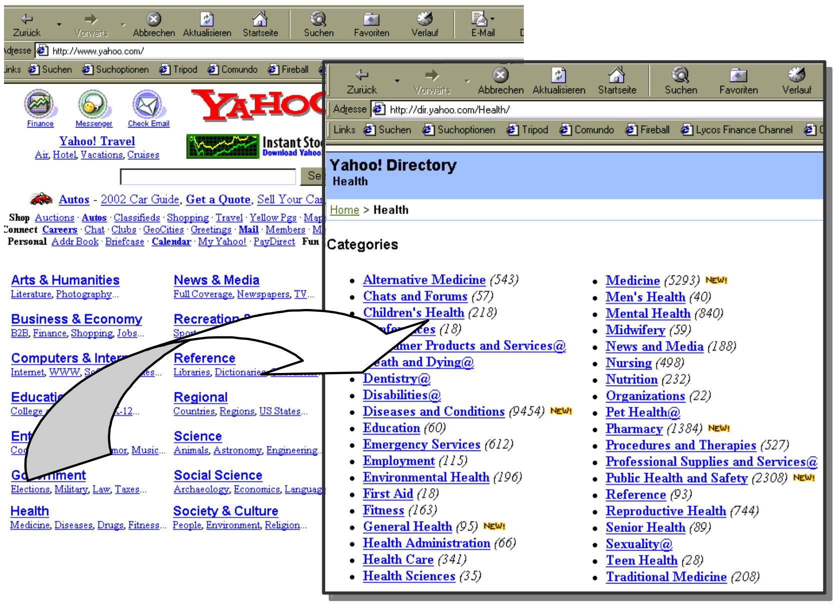
Directory Help

The web organized by topic into categories.

Arts
Movies, Music, Television,...
Home
Consumers, Homeowners, Family,...
Regional
Asia, Europe, North America,...
Business
Industries, Finance, Jobs,...
Kids and Teens
Computers, Entertainment, School,...
Science
Biology, Psychology, Physics,...
Computers
Hardware, Internet, Software,...
News
Media, Newspapers, Current Events,...
Shopping
Autos, Clothing, Gifts,...
Games
Board, Roleplaying, Video,...
Recreation
Food, Outdoors, Travel,...
Society
Issues, People, Religion,...
Health
Alternative, Fitness, Medicine,...
Reference
Education, Libraries, Maps,...
Sports
Basketball, Football, Soccer,...

Figure 6: Search engines (Altavista and Google) containing an edit-field for querying, and categories for navigating

For the second method, the user is offered a number links that represent a categorization of the content of a Web site. The categories are organized as taxonomies or partonomies. By clicking on one of the links, the user is moved to a web page describing this category. Then, a number of sub-categories is offered, and so on. Figure 7 shows how a mouse-click on the category ‘Health’ on the Yahoo-portal opens a new Web page with sub categories of ‘Health’.


Zurück Vorwärts Abbrechen Aktualisieren Startseite Suchen Favoriten Verlauf E-Mail
Adresse http://www.yahoo.com/
Links Suchen Suchoptionen Tripod Comundo Fireball
Finance Messenger Check Email
YAHOO
Instant Sto
Download Yahoo
Se
Adresse http://dir.yahoo.com/Health/
Links Suchen Suchoptionen Tripod Comundo Fireball Lycos Finance Channel

Yahoo! Travel
Air, Hotel, Vacations, Cruises

Autos - 2002 Car Guide, Get a Quote, Sell Your Car

Shop Auctions · Autos · Classifieds · Shopping · Travel · Yellow Pgs · Map
Connect Careers · Chat · Clubs · GeoCities · Greetings · Mail · Members · M
Personal Addr Book · Briefcase · Calendar · My Yahoo! · PayDirect Fun

Arts & Humanities
Literature, Photography...
News & Media
Full Coverage, Newspapers, TV...
Business & Economy
B2B, Finance, Shopping, Jobs...
Recreation
Sport
Computers & Inter
Internet, WWW, S es...
Reference
Libraries, Dictionaries
Educati
College K-12...
Regional
Countries, Regions, US States...
Ent
Co mor, Music...
Science
Animals, Astronomy, Engineering...
Go ment
Elections, Military, Law, Taxes...
Social Science
Archaeology, Economics, Languag
Health
Medicine, Diseases, Drugs, Fitness...
Society & Culture
People, Environment, Religion...

Yahoo! Directory
Health

Home > Health

Categories

- Alternative Medicine (543)
- Chats and Forums (57)
- Children's Health (218)
- nfe ces (18)
- mer Products and Services@
- eath and Dying@
- Dentistry@
- Disabilities@
- Diseases and Conditions (9454) NEW!
- Education (60)
- Emergency Services (612)
- Employment (115)
- Environmental Health (196)
- First Aid (18)
- Fitness (163)
- General Health (95) NEW!
- Health Administration (66)
- Health Care (341)
- Health Sciences (35)
- Medicine (5293) NEW!
- Men's Health (40)
- Mental Health (840)
- Midwifery (59)
- News and Media (188)
- Nursing (498)
- Nutrition (232)
- Organizations (22)
- Pet Health@
- Pharmacy (1384) NEW!
- Procedures and Therapies (527)
- Professional Supplies and Services@
- Public Health and Safety (2308) NEW!
- Reference (93)
- Reproductive Health (744)
- Senior Health (89)
- Sexuality@
- Teen Health (28)
- Traditional Medicine (208)

Figure 7: Clicking on a category to reach its sub categories

Finding information with the technique of querying only is often insufficient, as in only a few searching situations will the result of the query correspond to exactly what the user is looking for. Searching is additionally combined with the second method, i.e., navigation. As the navigation tactic comprises an imagined movement of the agent in addition to other features of wayfinding in the real world, navigation seems closer to the wayfinding concept in the real world than querying. Therefore we mean the *navigation* tactic as used in (Jul and Furnas 1997) when we talk about ‘wayfinding’ in the WWW.

Maglio and Matlock (1998) found that Web users think of the Web as a kind of physical space in which they move, although the Web is not physical and Web users do not locomote. This result can be concluded from an extensive use of spatial metaphors when people talk about the WWW. Such metaphorical thought is motivated by the same basic image schemata that people rely on to structure the physical world and the WWW. Image schemata (Lakoff and Johnson 1980) are recurring mental patterns that help people to structure space and arise from our embodied experience. Image schemata are claimed to shape both metaphorical and non-metaphorical thought (Johnson 1987; Lakoff 1987). A phrase like “in Yahoo” expresses the CONTAINER schema, just like the activity of moving up and down within a hierarchically structured Web domain. In distinction, “at Alta Vista” suggests using the PLATFORM schema. As well as active physical motion towards objects or destinations (concrete or abstract), as abstract motion towards a goal (e.g., going to the ‘Yahoo’ homepage) involve the image schema TRAJECTORY, comprised of a starting point, an end point, and a path between the two.

2.2.3 Setting and Task

The first case study describes wayfinding in the directories of the Yahoo!-domain (*http://www.yahoo.com/*). As the first online navigational guide to the Web, Yahoo! is reaches over 219 million unique users in 24 countries and 12 languages. As in the portal of many Web searching engines, the structure of the directories is hand-built and continually improved. Due to these modifications, differences between the simulated environment and the online directory structure may appear. For an abstraction of the simulated environment see Figure 9.

Corresponding to the airport case study, the agent’s task is to find a goal through stepwise decision making, i.e., using a *navigation* tactics (section 2.2.2). The simulated goal is defined as a web page that should provide the web agent with the possibility of purchasing ‘Nike’ sneakers size 9 1/2. In the simulation the links are restricted to within the hierarchy of the Yahoo-directories, excluding links leading out of the domain. The agent starts at the index page of the domain, makes his decision and chooses a link that may lead him closer to the desired page.

We see commonalities between the two wayfinding tasks: Both tasks contain a goal, the abstract navigators interact with an environment, and they need to make decisions during the navigation process. Using these cases studies, we will show that both of the task related processes can be classified as wayfinding, and that the abstract concepts of the physical world can be mapped to the Web space.

2.2.4 Ontology of the WWW Environment

Corresponding to the airport domain, where the ontology has two layers (section 2.1.2), the Web space has an ontology that consists of objects and information. The topologic layer of ontology describes the arrangement of documents and hyperlinks in the Web space, metric information excluded. Thus, for the representation of this layer, Web domains can be abstracted as graphs, web pages as nodes, and hyperlinks as edges connecting two nodes.

The second layer represents a semantic network that describes the semantic content of a Web site and is visualized through information on the links (corresponding to information on the airport signs). A semantic network is based on the idea of associations, and semantically close information is stored close together. With a semantic network one can express relationships such as synonymous symbols, antonymous symbols, parent categories, child categories, or visual similarity. This type of system is most useful for organizing groups of related symbols. Semantic nets can be visualized as directed graphs, where the nodes represent terms (concepts), and the edges represent relations between the terms. The most important relation is the IS-A relation that sorts terms after their generality. The properties of general terms are inherited to elements of a lower hierarchy.

In summary, we distinguish between two layers in the ontology of the Web space. In our model, the second layer with its semantic information is used for decision making in the proposed strategy of this thesis. The semantic layer—if describing a physical object— can recall action affordances in the Web user’s imagination (and therefore partly corresponds to the second layer of affordances used in the WWW navigation, see section 3.2.3.2).

- the *topologic* structure of documents that are physically available in the web: In this layer, the WWW can be seen as collection of multimedia documents in the form of HTML pages connected through hyperlinks (Li and Shim 1999).
- the *semantic* information that is represented through the information carried by the links: The web represents an ontology of the world from the web designer’s point of view.

Both of these layers are considered static for the duration of the agent’s navigation process, i.e., no *external* impact changes the structure of the environment. Even if the environment would change during the wayfinding process, the navigator would not notice, as he does not visit a node twice with the proposed wayfinding strategy. The web space is a graphical representation of semantically and topologically related information; the physical components are hardly recognized by the user (except the hardware as physical basis for the user-interface).

2.2.5 Semantic and Metric Information in the WWW Environment

Besides semantic and topologic information, a metric property of the links is provided by the Web interface, as each link is given a position in the coordinate system of the screen. In Figure 8, five links are visualized in the user interface. Each of these links contains *semantic* information expressed as keywords, the *metric* attribute is given as distance from the upper border of the screen. In this visualization of a directory structure, the *metric* position of a link does not tell the navigator anything about the content of the page to which it leads (only the semantic content does). Therefore this metric attribute does not help the navigator to reach his target.

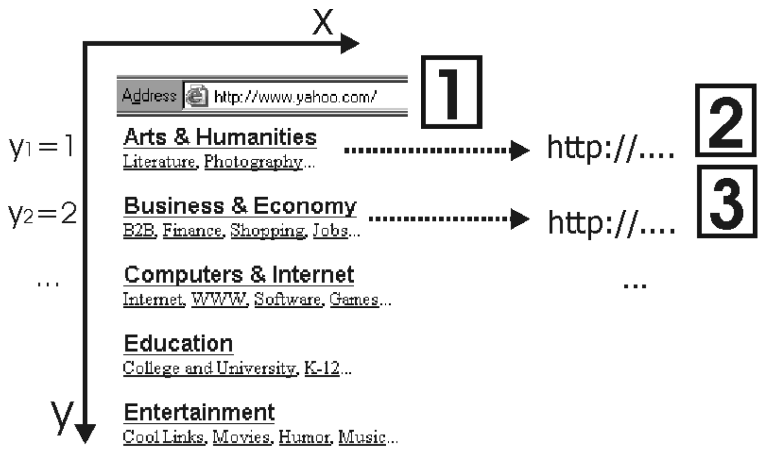

X
Address http://www.yahoo.com/
1
y1=1
Arts & Humanities
Literature, Photography...
http://....
2
y2=2
Business & Economy
B2B, Finance, Shopping, Jobs...
http://....
3
...
Computers & Internet
Internet, WWW, Software, Games...
...
Education
College and University, K-12...
y
Entertainment
Cool Links, Movies, Humor, Music...

Figure 8: Metric and semantic information of hyperlinks displayed in a user interface

2.2.6 The Abstract Simulated Test Area

As with most search-engine portals (Figure 6 and Figure 7 in section 2.2.2), the abstract simulated test area consists of several categories which are hierarchically structured (Figure 9). The test data used for the simulation represent only a small fraction of the complete domain. Terms of up-links are printed in italic font and gray color, those of down links or crosslinks in regular font and black color. Crosslinks between different categories are visualized as dashed arrows. Links which lead to a ‘dead end’ in reference to the predefined goal and therefore would require backtracking, are visualized as thin arrows. The corresponding web pages are labeled with an additional ‘X’ before the id. The characterizing property of cross links is that from a Web page that has been reached through a cross link, one can only go back to the previous page through the ‘back’ button of the browser, but not through a back-link (this does not exist in such case). Thus such an edge is directed. Edge 8-12 in Figure 9 is an example of such a situation.

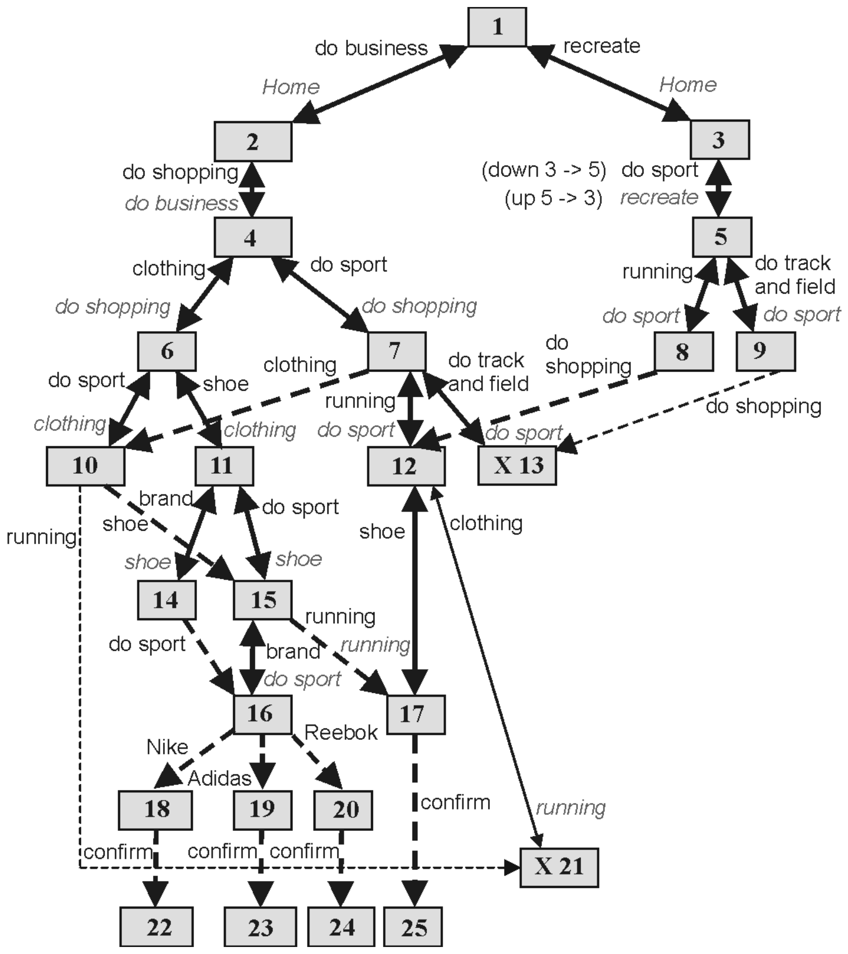
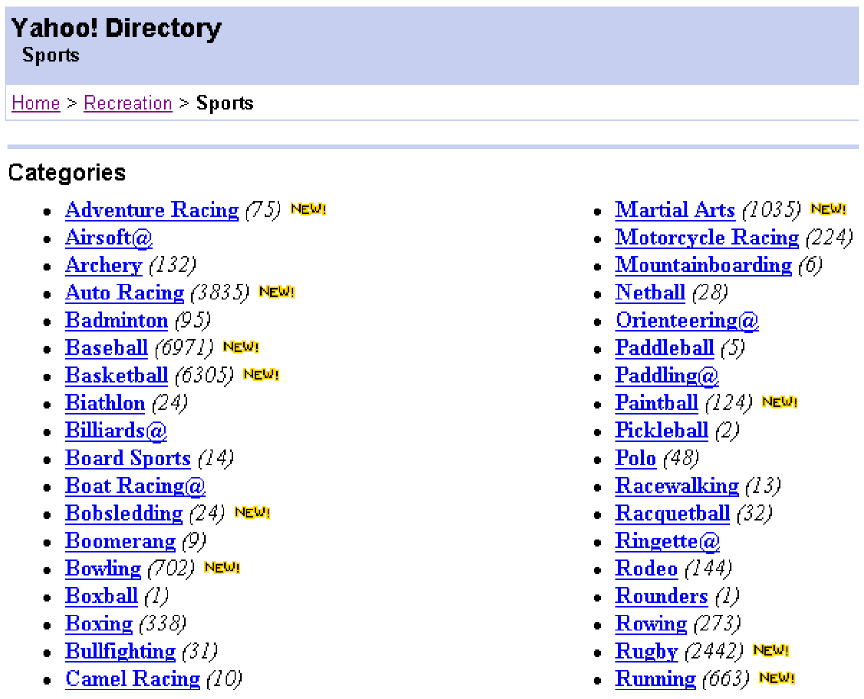


Figure 9: Simplified link structure of an existing web domain

In addition to the visualized links in Figure 9, each web page has links to *each* of its upper category levels. For example, node 5 has up-links to node 3 and node 1. Figure 10 gives a screen shot of the decision situation at node 5 in Figure 9, where the up-links (‘Home’ and ‘Recreate’) can be found below the screen title “Yahoo! Directory Sports”.


Yahoo! Directory
Sports

Home > Recreation > Sports

Categories

- Adventure Racing (75) NEW!
- Airsoft@
- Archery (132)
- Auto Racing (3835) NEW!
- Badminton (95)
- Baseball (6971) NEW!
- Basketball (6305) NEW!
- Biathlon (24)
- Billiards@
- Board Sports (14)
- Boat Racing@
- Bobsledding (24) NEW!
- Boomerang (9)
- Bowling (702) NEW!
- Boxball (1)
- Boxing (338)
- Bullfighting (31)
- Camel Racing (10)
- Martial Arts (1035) NEW!
- Motorcycle Racing (224)
- Mountainboarding (6)
- Netball (28)
- Orienteering@
- Paddleball (5)
- Paddling@
- Paintball (124) NEW!
- Pickleball (2)
- Polo (48)
- Racewalking (13)
- Racquetball (32)
- Ringette@
- Rodeo (144)
- Rounders (1)
- Rowing (273)
- Rugby (2442) NEW!
- Running (663) NEW!

Figure 10: Up- and down-links on a Web page

For reasons of readability, we skip all of these up-links in the model, except the one to the category directly above. Due to the agent's strategy of moving *towards* its goal with each step (i.e., *not* to move up in a hierarchy backwards to the start node), such up-links would not be chosen at a decision point, except if the agent was lost.

The test graph contains 25 nodes with connecting edges. Similarly to Table 1 (airport environment), the edges of the WWW graph can be summarized in Table 2. It lists some of the edges in the simulated WWW environment. The values in the 'direction' column denote the position of the link from the upper border of the user interface. The direction is therefore not oriented within a local reference frame.

Position	Direction of Link	Link content	Position	Direction of Link	Link content
1	2	"do business"	2	1	"Home"
	9	"recreate"	3	1	"Home"
2	2	"do shopping"	4	1	"do business"
3	19	"do sport"	5	1	"recreate"
4	3		6	1	"do shopping"
	61	"do sport"	7	1	" do shopping "
…					
7	1	"clothing"	10	-	-
	59	"running"	12	1	"do sport"
	76	"do track and field"	13	1	"do sport"
8	12	"do shopping"	12	-	-
…				-	-
20	1	"confirm"	24	-	-

Table 2: Directed and undirected edges in the WWW graph

2.3 Comparing the Abstract Environments

The concepts of the environments used in the two case studies show a number of similarities:

- Both environments can be abstracted as graphs using the same classification of edges (non-labeled, directed, undirected).
- Nodes represent decision points.
- The edges contain semantic information:
  - An edge in the airport environment can have a sign attached which contains numbers and letters representing gate names.
  - A link in the WWW interface describes the page to which it leads with a meaningful keyword.
- The edges contain a metric attribute:
  - The outgoing edges from a node in the airport environment enclose a certain angle and are oriented within a local reference frame.
  - The Internet-links are visualized on a certain position of the screen which can be described through coordinates.

The similarity of concepts allows a number of operations to be applied for both abstract domains, for example:

- Find edges with a sign.

- Get the semantic information that can be perceived from a node.
- Determine the degree of a node.

2.4 Summary

This section introduced the settings of the case studies used in this thesis, i.e., wayfinding at the Vienna International Airport and in the Yahoo-domain. The task for both agents is to find a specific place in the environment: The gate 'C 54' for the airport navigating agent, and a WWW page where one can purchase sneakers with certain attributes for the WWW-navigating agent. The conceptual and formal wayfinding model to be developed in this thesis describes a cognitive agent which is able to cope with the given tasks in the case studies.

We described the ontology of airport environments and the Web space, and classified the information provided by these environments into metric and semantic. Despite differences in the physical constellation between both environments, conceptual commonalities of operations performed in the environment can be found on a more abstract level.

Chapter 3 Contributing Disciplines

Discussing the wayfinding metaphor obviously involves two scientific fields, namely metaphor theory and wayfinding theory: First we explain what metaphors are and how they can be classified, followed by a discussion of the wayfinding process in the physical world, i.e., the source domain of wayfinding metaphor. Agent theory allows for elaboration of wayfinding on a more abstract level. We hereby restrict our discussion of human wayfinding to some characteristic features that will be modeled within an agent based model and that formally show the mapping of semantics between the two domains.

3.1 Metaphors

3.1.1 What are Metaphors ?

Johnson (1987) characterizes a metaphor as "…a pervasive mode of understanding by which we project patterns from one domain of experience in order to structure another domain of a different kind." Following Sweetser (1990, p.8), "metaphors allow people to understand one thing in terms of another, without thinking that the two are objectively the same". Research on metaphors presents a number of obvious problems: how to determine its truth value—literally, metaphors are almost always false—and how to recognize an expression as a metaphor (metaphors have no consistent syntactic form) (Scaruffi 2001).

Over the years the understanding of the role of metaphors in human life has changed. Different theories use different approaches to describe the nature of metaphors. Most traditional theories treat a metaphor chiefly as a theoretical or artistic figure of speech whereas contemporary theories extend the scope of metaphor to include its role in scientific reasoning.

The view of *literal-core theories* is that metaphors are cognitively reducible to literal propositions. Treated as literal figures, metaphors were considered to be nothing more than a rhetorically powerful mode of expression without its own cognitive content (Johnson 1987). The literal-core theories hold the *objectivistic* view of metaphors, which says that the objective world has its structure, and concepts and propositions, to be correct, must correspond to that structure. It is only literal concepts and propositions that can do that.

In *metaphorical proposition theories* metaphoric imagination can create new unified wholes within human experience rather than merely supplying novel perspectives on already interpreted experiences. A broad analysis of metaphors was carried out during the 1970's and 1980's by Lakoff and Johnson (Lakoff and Johnson 1980; Johnson 1987; Lakoff 1987). Contrary to the current opinion of this time that a metaphor is a linguistic expression favored by poets, the authors found two fundamental conclusions in their analysis: (1) all language is metaphorical and (2) all metaphors are ultimately based on our bodily experience. They claim that metaphor is not in the words but in the ideas and that metaphor is used for reasoning. Once metaphor is defined as the process of experiencing something in terms of something else, metaphors turn to be pervasive, in action and thought. In their work, Lakoff and Johnson show how metaphors reveal the limitations of objectivism, namely the assumption that the world is made of distinct objects with inherent properties and fixed relations between them.

Although metaphors are literally false there is some sense in which they are not only not false, but can provide very valuable insights (Grey 2000). Thus, metaphors must consist of a deep as well as a surface level. When the literal meaning is deactivated because of the falsehood of the sentence a mental switching happens that activates the secondary meanings. Let us consider the metaphor "Time is money" as an example. As soon as we apprehend that the description is literally false, the expression becomes semantically charged with secondary meanings: Time in our culture is a valuable commodity, it is a limited resource that we use to accomplish our goals. Work is usually associated with the time it takes, and it has become customary to pay people by the amount of time for their work. Corresponding to the fact that we act as if time is limited and a valuable resource such as money, the metaphor is true on the deeper level of cultural experience.

3.1.2 Classification of Metaphors

Lakoff and Johnson (1980) define three types of metaphors:

- *Structural metaphor* where one concept is metaphorically structured in terms of another, e.g., 'The meaning is *right there* in the words.'
- *Orientational metaphors* which organize a whole system of concepts with respect to another, transferring spatial orientation, such as up-down, in-out, front-back. These metaphors use humans' experience with spatial orientation, for example in the phrase 'He *fell* asleep'.
- *Ontological metaphors* which are based on humans' experience of physical objects and substances. Once we identify our experience as entities or substances, we can

refer to them, categorize them, group them, quantify them, and reason about them. Ontological metaphors are ways of viewing events, activities, and ideas as entities and substances.

When using *container metaphors*, as members of ontological metaphors, we use for example the human property to be like a container, with a bounding surface and an in-out orientation. We project our own in-out orientation onto other physical objects that are bounded by surfaces, as for example used in the following phrase 'He's *out* of sight now'.

Grey (2000) classifies metaphors into dead, dormant, and live. A dead metaphor is an ordinary part of our literal vocabulary and commonly not regarded as metaphor at all. The author takes the verb 'run' as example. Running in its basic meaning is considered as a simple activity which involves putting one leg in front of the other in a certain systematic and rhythmic fashion. Through metaphorical extension the expression comes to be applied to objects which lie outside its basic reference class, such as rivers. So it comes about that rivers run, taps run, and fences run—the last example showing another feature of metaphors: By abstracting certain elements of the activity we are able to produce a generalized meaning of the basis sense word. For example, if we speak of fences 'running' around a boundary, there is no suggestion of motion. Instead, the metaphor creates a static sense of running, in this case, the sense of following a path.

A live metaphor is a metaphor which we are conscious of interpreting. As the previously mentioned example ("Time is money"), such a metaphor cannot be taken at its literal face value but has to be decoded. Dormant metaphors represent an intermediate category. They consist of expressions we use without being conscious of their metaphorical character, but if we attend to them they are recognized as metaphors. The border between dormant and dead metaphors is fuzzy. Metaphors that suffer the abuse of overuse, e.g., 'the bottom line', degenerate into cliché. Overuse is a process by which a living metaphor can become dormant or dead.

3.1.3 The Mapping of Semantics with Metaphors

Metaphors map semantics from the source to the target domain. In a metaphorical sentence the terms *tenor* and *vehicle* denote the two parts of a metaphor. The *tenor* is the literal subject whereas the *vehicle* is the figurative connection, i.e., the thing that is compared to the subject or the carrier. For example, in the metaphor "a Sahara of snow" in a poem by Robert Lowell, the tenor is *snow*, while the vehicle is the *Sahara* desert.

The human interpretation of metaphors is discussed in several competing models reported in the literature. In the *similarity* or *comparison view* (e.g., Ortony 1979), preexisting similarities between the constituent terms of a metaphorical sentence are an important source of information for generating figurative meaning. In order to generate an interpretation, a metaphorical sentence first has to be translated into an explicit comparison statement. Then, a feature-matching process is applied to the representations of the noun-concepts involved in the metaphor. If we take the example “Life is a journey”, the features of the vehicle-concept *journey* are compared to features of the tenor-concept *life* in order to identify common features. In the given example, possible features for the interpretation may be ‘surprise’, ‘decision point’, ‘comrade’.

In contrast to the comparison view, the *interaction approach* (e.g., Black 1979) claims that similarity is not antecedent but a product of comprehension. Metaphorical meanings are constructed by means of *emergent features* that appear when the representations of tenor and vehicle as well as their corresponding domains are brought into interaction. As example we use ‘Hercules is a lion’. Here, a feature that is neither characteristic of the tenor nor of the vehicle, but surfaces only in the interpretation, is *mythical feature* (Nückles and Janetzko 1997).

There is no clear evidence for which of the two theoretical approaches should be accepted and which rejected. From empirical tests, Nückles and Janetzko (1997) make the assumption that the two theories support a complementary cognitive process, and that metaphor comprehension proceeds in two stages that the authors call *analysis* stage and *synthesis* stage. First, an analysis of the lexical meanings of tenor and vehicle is attempted. In case of enough similarities to produce a coherent interpretation, the comprehension process will cease. In the other case, a synthesis of the two terms follows that requires the activation of broader world knowledge about the domains involved.

An open question in metaphor theory is, why only parts of the semantics of the source domain are mapped to the target. The metaphor ‘Theories are buildings’, e.g., maps ‘foundation’ and ‘support’ onto the target domain, whereas ‘doors’ and ‘windows’ are not mapped. Recent approaches have attempted to solve this problem by introducing several types of metaphors, including primitive and compound (Grady, Taub et al. 1996).

Kuipers (1982, p.3) defines operations as the relevant parts of a domain that need to be mapped so that one can talk of a metaphor. He claims a metaphor to be correctly used if corresponding operations in both domains, i.e., a graphical map and the map in the head, exist: “The ‘Map in the Head’ metaphor states that the functional behavior is the same in the two contexts”.

3.1.4 Formal Approaches of Expressing a Metaphor

Formal approaches to define the metaphor are rarely found in the literature. A fuzzy-logical approach that uses a four-valued logic has been formalized by MacCormac (1985). Besides truth and falsity, the values also embrace metaphor in two forms, diaphors (metaphors that imply the possibility of something), and epiphors (metaphors that express the existence of something). Employing a system of fuzzy semantic markers, MacCormac defines the fuzzy membership of one category in another as a real number ranging from zero (absolute falsehood) to one (undeniable truth). Within this range exist the delimiters a, b, c, such that $0 < a < b < c < 1$, where the interval 0 to a represents falsehood, a to b represents diaphor, b to c represents epiphor, and c to 1 represents literal truth. Metaphoric set membership is thus indicated by a value in the range a to c. Novel metaphors begin life as diaphors, and migrate along this fuzzy scale into epiphors as they lose their emotive tension through commonplace use, to eventually find rest as dead metaphors in the literal truth interval (see the example of the ‘bottom line’ in section 3.1.2).

Gentner’s *structure mapping theory* (Gentner 1983) describes analogies as mappings between source and target domains, each represented by semantic networks. The mappings themselves are not formalized but rest on a syntactical distinction of different kinds of relations. The author presumes that knowledge is represented as propositional network of nodes and predicates, where the nodes represent concepts treated as wholes, and the predicates applied to the nodes express propositions about the concepts.

An analogy between the base domain A and the target domain T maps the *object nodes* of A onto the object nodes of the target domain T. Further, *predicates* from A are carried across a mapping function to T. Gentner distinguishes between four different kinds of domain comparisons which are determined by the number of *attributes* (sorts) and *relations* (functions) mapped between the two domains. The types of domain comparison are: literal similarity, analogy, abstraction, and anomaly. Table 3 illustrates which features are mapped in which type of structure mapping. The right column gives an example of domain comparison for a solar system. Other formal approaches that use algebraic structures for metaphors are discussed in section 4.2.4.

	Number of attributes mapped to the target	Number of relations mapped to the target	Example
Literal similarity	Many	Many	The K5 solar system is like our solar system
Analogy	Few	Many	The atom is like our solar system
Abstraction	Few	Many	The atom is a central force system
Anomaly	Few	Few	Coffee is like the solar system

Table 3: Number of attributes and relations mapped in different types of domain comparison (Gentner 1983)

Goguen (1999) proposed a mathematically precise theory of semiotics, called *algebraic semiotics*, as a tool to study the ways in which information is mediated in computer systems. A user interface can be considered as a representation of the underlying functionality to which it provides access. Both the interface and the underlying functionality are considered as sign systems. In this setting, representations appear as mappings (morphisms) between sign systems, which should preserve as much structure as possible. A sign system can be formalized as many sorted loose algebra plus some specific semiotic items. A *Semiotic morphism* M: $S_1 \rightarrow S_2$ provides a way to describe the mapping of signs in one system S_1 to signs in another system S_2. A good semiotic morphism should preserve as much of the structure in its sign system as possible, i.e., sorts and subsorts, operations, axioms, content, levels of sorts, and priority ordering on constructors. Empirical work showed that it is more important to preserve structure than content. Applications for algebraic semiotics include user interface design, cognitive linguists, metaphor theory, and cognitive poetics.

3.2 Wayfinding

In this section we review various aspects involved in wayfinding, stressing those concepts that are needed to construct the conceptual wayfinding model of the simulated agent. First we discuss the rough boundaries of the term ‘wayfinding’, further we look at cognitive models of space and the process of decision making during wayfinding. The final sub section is devoted to existing computational wayfinding models.

3.2.1 What is Wayfinding?

Within the task of this thesis to describe the wayfinding metaphor we explore if essential concepts of the term *wayfinding*, as it is used in the physical world, can also be found in the WWW. Thus we first need to focus on what *wayfinding* in the physical world means.

The American architect Kevin Lynch (1960) was the first to use the term *wayfinding* which replaced the term *spatial orientation* in the late 70s. Spatial orientation refers to a person’s ability to determine his or her location in a setting, thus, describes the static relationship of a person to his or her spatial setting. The term cannot encompass the dynamic aspects of people’s movements. In the late 70s the concept *wayfinding* filled this missing part of the *spatial orientation* concept. Wayfinding was used to account for people’s movement in space and their sense of being orientated, it described the process of reaching a destination, whether in a familiar or an unfamiliar environment. In the 80s, *wayfinding* was modeled as spatial problem solving (e.g., Downs and Stea 1977; Gärling, Böök et al. 1984), which within this framework comprises three specific but interrelated processes (Arthur and Passini 1992):

- Decision making and the development of a plan
- Decision execution, which transforms the plan into appropriate behavior
- Information processing, comprising environmental perception and cognition

The number of *wayfinding*-definitions that are found in the literature is extensive. Analyzing some of the definitions, the recurrence for specific terms is higher than for other terms. Thus, wayfinding is not uniquely defined and its boundaries seem to be unstable and rough. The terms that are more often used in the definitions define a central case of wayfinding (a kind of prototypical wayfinding), whereas more seldom mentioned features are extensions of the central meaning. We therefore assume that the concept of *wayfinding* represents a radial category (Rosch and Mervis 1975; Rosch 1978). It has gradations, and some wayfinding definitions represent better (more central) and worse (more peripheral) examples of the category. Members of the category do not possess inherent features as objects in traditional taxonomies do.

Due to the high number of wayfinding definitions, we cannot describe what the *correct* concepts are that define wayfinding in general. But as we need to express basic concepts of wayfinding (through axioms) for formalizing the wayfinding metaphor, we try to find the central features of wayfinding, and skip the more peripheral ones. The solar metaphor (Sutcliffe 1998) visualizes the idea of a radial category and expresses the relations of concepts that define wayfinding: The most central wayfinding features are visualized as sun, whereas more peripheral concepts of wayfinding are visualized as planets.

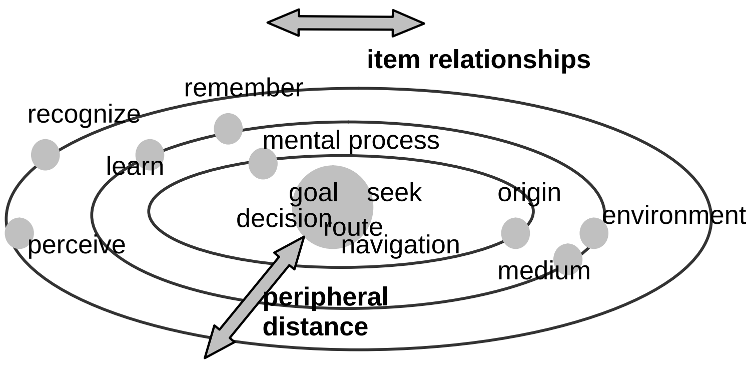


Figure 11: Schematic visualization of the Radial Category *Wayfinding* using the Solar metaphor

Before we look at existing wayfinding definitions, we should clarify the (small) differences between *wayfinding* and *navigation*. The difference seems to be based on properties of the surrounding environment: Human movement in *open spaces*, e.g., flying an aircraft, involves navigation Golledge (1999). It means to deliberately walk or make one's way through some space. Contrarily, wayfinding involves selecting paths from an *existing network* (Bovy and Stern 1990, Golledge, Jacobson et al. 2000). Allen (personal communication) claims that the terms *wayfinding* and *navigation* are similar in their meaning and that in most cases the two terms are interchangeable. The more similar a wayfinding situation is to plotting and executing a course, the better that analogy is.

To find the central concepts of wayfinding we look at following definitions (frequently used terms are written in italic font).

- Allen (1999) describes wayfinding as *purposeful movement* to a specific *destination* that is distal and, thus cannot be perceived directly by the traveler. Successful wayfinding is reflected in the traveler's ability to achieve a specific *destination* […] despite the uncertainty that exists.
- Golledge (1999) defines wayfinding as a process of *determining* and following a *path* or *route* between an *origin* and a *destination*.
- Wayfinding involves *selecting path segments* from an existing *network* and linking them as one travels a specific *path*. The process of wayfinding requires an ability to know *origins* and *seek a destination* […] (Golledge, Jacobson et al. 2000).
- Two critical characteristics of human wayfinding are *destination choice* and *path selection* (Golledge 1995).
- Bovy and Stern (1990) describe pathfinding and wayfinding as a process that involves *selecting paths* from a network.

- Blades (1991) defines wayfinding as the ability to learn and remember a *route* through the *environment*.
- Wayfinding is navigation that occurs both on and off known *routes* (Cornell and Heth 2000).
- Wayfinding is defined through the mental processes involved in *determining* a *route* between two points and then following that *route* (Mark, Freksa et al. 1999).
- Lynch (1960, p.3) defines wayfinding as based on “a consistent use and organization of definite sensory cues from the external *environment*.”

Outstanding terms of the listed definitions are *destination*, *path selection*, *determine*, *route*, *seek*, *environment*. These terms will be considered as part of the wayfinding axioms in chapter 5.

3.2.2 Cognitive Models of Space

Arthur and Passini (1992) claim that *cognitive mapping*, i.e., the mental structuring process leading to the creation of a cognitive map, is part of environmental perception and cognition. As perception and cognition are part of the wayfinding process, the use of a mental representation of the environment seems obvious for wayfinding tasks. The relative importance of a mental representation in the decision making process depends on the nature of the setting and on the wayfinding task.

The complexity of the physical world is reflected through various models that describe an environment or its mental representation. The models stress different features of the environment (e.g., topologic or metric relations between places; images of places) depending on the task of abstraction. For the construction of a precise map of an area, e.g., one needs *quantitative* knowledge. This allows the map user to predict precisely at which location an object will be encountered. Contrary, to describe a location to be *identified* in the real world, a limited amount of *qualitative* knowledge may suffice (Freksa 1991).

Lynch (1960) interviewed residents of three cities and found out that people build their mental model of a city based on five spatial elements:

(1) *Landmarks*, distinct points in a city that serve as reference to the user.

(2) *Paths*, channels of movement.

(3) *Nodes*, strategic spots in the city where the observer can enter.

(4) *Edges* are linear but do not facilitate movement. They form physical barriers.

(5) *Districts* are areas in a city that have some common characteristics such as a particular architectural style.

Siegel and White (1975) describe the stages in an individual's representation of spatial knowledge which are likely to come with increasing age or experience.

(1) *Landmark knowledge* comprises distinct, typically familiar points in the environment.

(2) *Route knowledge* is characterized by the knowledge of paths between landmarks (topological information).

(3) *Survey knowledge* allows people to locate landmarks and routes within a general frame of reference (i.e., incorporating metric measurements).

This model has been criticized for its strict developmental sequence (Montello 1998). For solving the problem of how an agent creates its spatial representation from its sensimotor experiences, Kuipers, Froom et al. (1993) and Remolina, Fernandez et al. (1999) use the computational theory *Spatial Semantic Hierarchy* (SSH). SSH is an ontological hierarchy of representations for knowledge of large scale space and comprises four levels: control, causal, topological, and metrical.

Some models of wayfinding reported in the literature stress the importance of topologic knowledge for the wayfinding process. The TOUR-model (Kuipers 1978), e.g., is based on the assumption that it is possible to store a topological relation between two places in the absence of any metrical information. Knowledge about particular environments is hereby classified into five categories (*route*, *topological structure*, *relative position*, *dividing boundaries*, *regions*). Freksa (1991) claims that topologic knowledge is relevant for wayfinding in the real world because movement in space is possible only between neighboring locations. Evidence for cognitive hierarchical organization of space was deduced in experiments from distance and direction judgments (Hirtle and Jonides 1985).

Several metaphors were introduced to express the characteristic of a mental representation. The term *cognitive map* was first used by Tolman (1948) who claimed that rats in a maze-learning task acquired knowledge of the spatial relation between start and goal. Other metaphors suggested are *cognitive collage*, *spatial mental model* (Tversky 1993), or *cognitive atlas* (Hirtle 1998).

Recent developments in cognitive science suggest that spatial relations do not exist in the real world but that they rather exist in mind (Mark and Frank 1996). Due to physiological similarities that exist among individual human beings, most people experience their environment in similar ways. Human experience in interacting the world leads to the most appropriate subdivision of continuous reality into objects (Frank 2001). Objects are typically formed in such way that many of their properties remain invariant over time. Johnson (1987) claims that experience from the environment and interaction with the environment uses recursive, imaginative patterns, so called *image schemata*. Many of the image schemata are inherently spatial or even graphical.

3.2.3 Epistemology—The Use of Affordances for Wayfinding in Airports and the WWW

Epistemology is the part of philosophy concerned with knowledge and knowledge representation. The task of epistemology is to derive observable consequences from theories. A representation is epistemologically adequate for a person or machine if it can be used practically to express the facts that one actually has about the aspects of the world (McCarthy and Hayes 1969). The elements of the ontology proposed in the case studies (sections 2.1.2 and 2.2.4) are mapped to the epistemology of the wayfinding subject.

In this thesis we use *affordances* to model the navigator's epistemology. We hereby add elements of cognition, situational aspects, and social constraints to Gibson's theory of perception. In this section we describe which affordances play a role for wayfinding in an airport environment and the WWW. The term *affordance* was created by Gibson (1977) when investigating how people perceive their environment. According to Gibson the environment consists of a *medium*, *substances*, and *surfaces* (see section 2.1.2). Gibson describes the process of perception as the extraction of invariants from the stimulus flux. Surfaces absorb or reflect light and Gibson's radical hypothesis is that the composition and layout of surfaces constitute what they afford. Affordances are therefore specific combinations of the properties of substances and surfaces taken with reference to an observer. Thus, animal and environment are modeled to be an inseparable pair. The theory of affordances is influenced by Koffka's work (Koffka 1935) on Gestalt psychology, where he states that each thing says what it is.

In his PhD work, Raubal (2001a) claims that both affordances and information are essential for people finding their ways in an unfamiliar environment. Affordances hereby provide possibilities for behavior and information helps people to choose between alternatives.

3.2.3.1 Affordances in the Airport-Environment

Most of this section follows Raubal (2001a). When performing a wayfinding task in a spatial environment people utilize a set of affordances. Some of the affordances are involved in the control of locomotion such as moving along a corridor, others are required for information acquisition such as reading and interpreting different signs, etc. Raubal distinguishes affordances concerning the realm they belong to, such as *social-institutional* affordances, *action* affordances, *physical* affordances, or *mental* affordances.

The most relevant *physical* affordance for a wayfinding simulation is the ‘go-to’ affordance of a path to move along it. Its utilization leads the agent from one node to another. Another physical affordance implemented is a sign’s affordance to reflect light, a decision point’s affordance to look around, or a doorway’s affordance to go through. *Mental* affordances are represented through the agent’s decision process. Sign information affords to be matched with the agent’s goal information, paths afford to be selected, and decision points afford searching, orienting, and deciding how to proceed. Information (such as from signs or from a map) is necessary for the agent to decide upon which affordances to utilize. Although a sign affords looking at it, only additional semantic information, such as letters, numbers, or symbols, allow the navigator to use the sign as navigation aid.

Social interaction is based on *social-institutional affordances*: Another traveler affords talking to, or asking. Physical and social-institutional affordances are the sources of *mental affordances*. In order to utilize a mental affordance, the agent needs to perform an internal operation. For example, letters displayed on a screen afford the navigator to perform the internal operation of matching the letters with his goal information.

What the navigator knows about the environment, partly depends on his physical and mental abilities, his life experience, and his task. This is due to the fact that utilizing affordances during the wayfinding process is user dependent. As an example, Raubal mentions the scenario of the case study where a mother is going on a flight with her 3-year-old son. Although mother and son perceive the same objects, objects afford to them different activities. Based on the task and the mother’s properties (e.g., being an adult), the check-in-counter, e.g., affords the mother to put her tickets on the counter so that the check-in agent can give her boarding passes. This is not afforded to the child as his properties (e.g., being too short) do not enable him to put something on the counter.

3.2.3.2 Affordances in the Web Navigation

Gibson’s theory states that affordances are based on the process of perception through the extraction of invariants from the stimulus flux. Despite this connection to visual perception, we propose that humans remember certain affordances of objects, i.e., humans assign affordances to pictures, symbols, or descriptions of an object (Hochmair and Frank 2001). In the case of web navigation, a link or icon displaying or describing an object, recalls a set of affordances to the user. This assumption of mentally stored affordances is an important concept to apply the proposed decision making model in the given task of the case study, as it connects displayed objects to affordances of a physical object (i.e., sneakers).

In the domain of a physical computer environment, which functions as the hardware platform for web navigation, Norman (1999) distinguishes between *physical* and *perceived* affordances. The *physical* affordances are carried by computer hardware, such as keyboard, computer screen, and mouse. These physical objects afford pressing a key, pointing, touching, looking, and clicking. Contrary, *perceived* affordances are provided by displayed objects on the user interface. Displayed objects *advertise* physical affordances. For example, the design of a hyperlink on the screen does not afford clicking, but provides a target and helps the user know where to click. Clicking on a perceived object on a screen with a pointing device is in Norman's view motivated by *cultural conventions* (shared by a cultural group), and not through affordances of the designed object on the screen itself.

In summary, a hyperlink (realized as text string or symbol) visualized on the user interface provides two layers of affordances (Figure 12), where the second layer will be shown to play a role for decision making.

- 1st layer *(perceived layer)*: advertises (*perceived affordance*) to click (*convention*) on the hyperlink or icon
- 2nd layer *(information layer)*: the information displayed as text or icon awakens the user's remembered *action affordances* of the object that is part of the web site behind the link (only if an object is displayed as link)

In the abstract model of the Web environment, links are expressed as edges between two nodes with semantic information attached. Due to the second layer of affordances provided by a hyperlink, we model the mental representation of a link in the abstract agent as the possibility to move to another web page of which the content is related to the information displayed on the link. The information of the link is not restricted to affordances of objects. Figure 12 demonstrates the distinction between the two layers provided by a hyperlink. The first layer of the displayed hyperlink *advertises* to the web navigator to move the displayed arrow over the hyperlink symbol which is carried out through moving the mouse (convention). Clicking on the symbol to reach the next web page is also convention. The user performs these activities as he expects that the web page behind the link is related to cycling (second layer) due to perceiving the picture. For example, the linked web page may contain the result of a cycling competition or be the Web page of a biking federation.

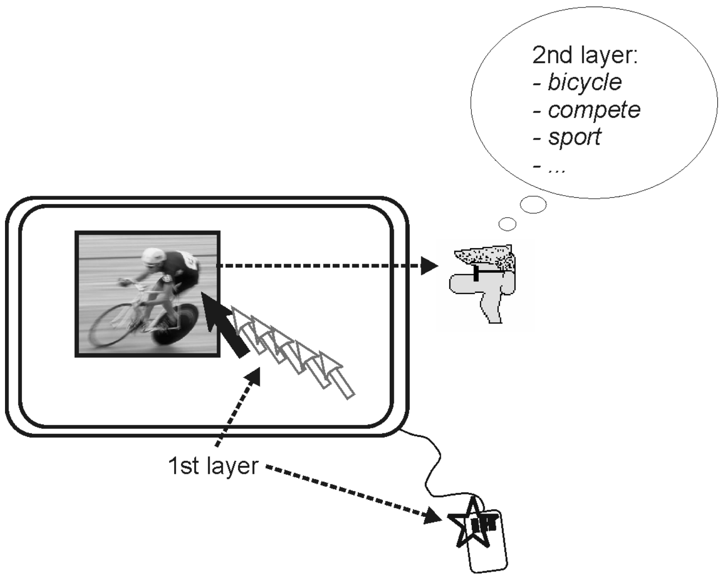


Figure 12: Perceived affordance, convention, and associations provided by a hyperlink

Table 4 relates objects in the abstract environments (of the physical world and the WWW) to the agent's epistemology. The table shows a potential approach of how objects may be reflected in the agent's behavior and beliefs.

Object in the abstract environment	**Agent's epistemology**
edge	possibility to move to endnode
signpost (airport and WWW)	read content, determine direction
letters and numbers (airport) or strings (WWW)	match with concepts in cognitive map
target-node	state that needs to be reached
intersection	make decision

Table 4: Agent's epistemology of objects in the simulated environment

3.2.4 Spatial Decision Making and Wayfinding Strategies

Decision theory covers a large range of models with different foci on describing how decisions could or should be made and on specifying decisions that are made (Golledge and Stimson 1997). We point out concepts of decision making that are general enough to hold for the physical world and the WWW.

Literature offers numerous decision making models. A classification that distinguishes between various types of outcomes of decisions—either optimal solutions or acceptable solutions—is given by Gärling and Golledge (2000). The classification discusses three approaches to the study of decision making:

- The *normative* approach, exemplified by economic choice theory, aims at defining optimal decisions.

- The goal of *prescriptive* approaches is to advise human decision-makers about how to make optimal decisions given that they possess a limited cognitive capacity to do so.
- *Descriptive* theories focus on approximate (heuristic) decision strategies.

Arthur and Passini (1992) distinguish between two decision making models: In the *optimizing model*, the person considers all options in the light of all subjectively relevant criteria and chooses an optimal solution whereas in the *satisficing model* an acceptable solution is retained without seeking the optimum (Downs and Stea 1973). The second model tends to be more popular for complex decision making. Different approaches may be applied to accomplish an acceptable solution:

- The rejection of all alternative options because of some unacceptable aspects.
- The choice of an option because some aspects are very desirable.
- A more nuanced comparison of aspects of options until one aspect clearly dominates and leads to the choice (or the rejection) of an option.

Gärling, Böök et al. (1984) propose the process of spatial decision making to consist of following stages:

- Retrieve information about the environment which is externally accessible or is accessible in a cognitive map.
- Represent decision alternatives in memory.
- Evaluate the decision alternatives.
- Apply a decision strategy.
- Implement the decision.

Studies have shown that heuristics for choosing the correct way at an intersection are influenced by configurational parameters of the spatial environment and by people’s perspectives during navigation (Janzen, Herrmann et al. 2000). Golledge (1995) found that decision criteria are influenced by changes in the structure of the environment. In his studies, subjects had to decide between three given routes in several, slightly changed environments on a map. He tested the subjects’ preferences for several criteria, such as fewest turns, longest leg first, preference for curves, shortest route, and most scenic route. The result showed for example that the preference to choose the shortest path criterion varied from 54% to 90% between slightly changed environments.

People can apply such criteria only when they are either familiar with the environment or have access to a map of the environment. For the wayfinding simulation in our case studies we presume that the agent moves in an unknown environment and has no map. Thus, the criteria applied in both environments are based on the agent's simulated life experience and utilization of affordances, and not on knowledge of the environment.

A place in the real world and in the WWW can be represented in several ways. Various cognitive models of space (section 3.2.2) reveal that a place—at least in the real world—includes metric, topologic, or semantic properties among others. Various properties of the place can be used for its definition. Depending on the type of the goal definition, different wayfinding strategies are applied (Hochmair and Raubal forthcoming). Besides the representation of the goal, other circumstances of the wayfinding situation have an effect on the decision criteria and wayfinding method applied, such as time restrictions (Stern and Portugali 1999) or emotions (Trappl, Petta et al. forthcoming).

The decision making process is strongly based on the individual's level of spatial knowledge (Bovy and Stern 1990), i.e., the familiarity with a given environment. Thus, a change in the relative use of knowledge components (i.e., the amount and type of information retrieved from one's associative memory) can be expected as a function of navigation frequency (Stern and Portugali 1999).

3.2.5 Simulating Human Wayfinding

There exist two approaches to simulate the interaction of humans with their environment. The first one is a behavior-based approach where autonomous robots perceive and act directly in the real world (Brooks 1986; Brooks 1991). The goal of this approach is to generate agents that behave intelligently, without any relation to human behavior. This approach is effective to build robots. The other approach is a computational computer model that simulates a human wayfinder as a cognizing agent in a simulated environment. The approach has the advantage that it allows simulating wayfinding behavior in various (simulated) environments and therefore is an effective tool to express the wayfinding metaphor on an abstract level. For this reason, the simulated wayfinding model in this thesis follows this approach.

Cognitively based computational models generally simulate a wayfinder that can solve route-planning tasks with the help of a cognitive-map-like representation. The focus of these models is to find how spatial knowledge is stored and used, and what cognitive processes operate upon it. The TOUR model (Kuipers 1978; Kuipers 1982) represents the first computational theory of wayfinding. The model copes with incomplete spatial knowledge of the environment and learning about the environment as more information is received. Several other cognitively based computational models, such as TRAVELLER (Leiser and Zilbershatz 1989), NAVIGATOR (Gopal, Klatzky et al. 1989) or ELMER (McCalla, Reid et al. 1982), simulate learning and problem solving in spatial networks. O'Neill (1991) designed a biologically based model of spatial cognition and wayfinding. Using the metaphor of a biological system, the model proposes a view of spatial cognition considering lower level, physiological mechanisms. For a more detailed description of computational models see (Gluck 1991).

3.3 Agent Theory

3.3.1 The Term 'agent'

In the literature, the term *agent* is used as a technical concept, a metaphor, or a design model (Nwana and Ndumu 1999). In this thesis we use the term as a conceptual paradigm to represent the human navigator. This allows us to elaborate the wayfinding problem on a more abstract and theoretical level, and to reduce the complexity of human navigation in the computational model. We consider the agent-based model as conceptual framework for the representation of the domain of interest—i.e., the wayfinding process. It applies the agent concept as a metaphor for the description of the active entities in some domain. The conceptualization will be realized in a computational language that must be expressible and understandable enough to allow the representation of the agent framework.

According several scientific research directions that are related to agent-theory, there exist many definitions of an agent. Russell and Norvig (1995, p.31) define an agent as "anything that can be viewed as perceiving its environment through sensors and acting upon the environment through effectors". *Anything* is related to the agent definition of Ferber (1998), who proposes that physical and virtual entities can be considered as agents. Another definition taken from Wooldridge and Jennings (1995, p.29), defines an agent as "…a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives". We create an agent as a software robot that simulates people's wayfinding behavior in the domain of the physical world and the WWW.

An *intelligent* agent is capable of autonomous action in order to meet its design objectives. Intelligent agents must operate robustly in rapidly changing, unpredictable environments, where there is the possibility that actions can fail (Wooldridge 1999). The intelligent agent reacts to changes in the environment, exhibits a goal-directed behavior, and has some sort of social ability which means that it can communicate with other agents in a multi-agent system (Weiss 1999).

There are two methods to describe the interaction between agent and environment. The first approach models an agent as *part of the environment*. In such a model, any of the agent's actions changes the state of the environment. This includes, besides actions visible from an external point of view, also actions in the agent's mind, e.g., making a decision. In this case, operations of the agent lead to changes in the environment which are represented by new environmental states after every performed operation. Formal specifications using this approach can be found in (Frank 1999; Bittner 2001).

The second approach *separates* the agent from the environment. This approach can be applied as a simplification if none of the agent's operations is assumed to change the state of the environment—as it is in the case of wayfinding. Using this approach, the agent does not act in the environment but interacts *with* the environment (Figure 13).

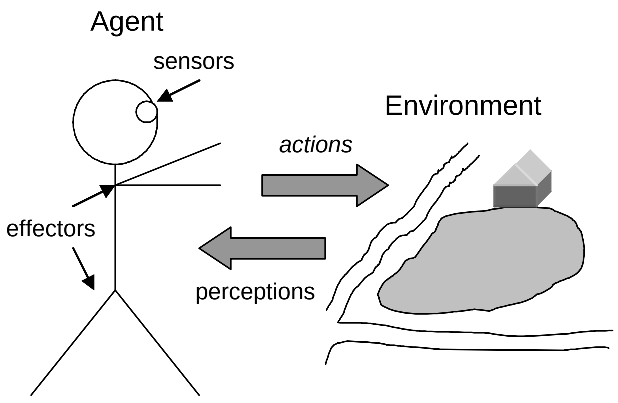


Figure 13: Agent interacts with environment through sensors and effectors

In this thesis, we use the second approach. No activities which would change the state of the environment are performed by the agent. Examples for such excluded actions are moving an object, blocking a road, or opening a door. The only activity that is 'visible' from an external point of view during the wayfinding process is a change of the agent's position after a step. As the agent is not part of the environment, changes due to the agent's wayfinding process are reflected in the agent, and *not* in the environment. To avoid a confusion of terms used in the computational model, we distinguish between *environment* and *world*. By *environment* we mean the static surrounding of the agent with which the agent interacts and that can be abstracted as a graph. The environmental surrounding and the agent together is called *world*.

3.3.2 Abstract Architectures of Agents

In (Wooldridge 1999) and (Russell and Norvig 1995) several abstract architectures of agents can be found. Ordered after their complexity (starting with the simplest architecture), the following architectures are discussed:

- Simple reflex agent
- Reflex agent with internal state
- Goal-based agent
- Utility-based agent

A *simple reflex agent* reacts based on condition-reaction rules. A condition rule can formally be written as an *if-then* condition. In a *reflex agent with internal state*, the current percept is combined with the old internal state to generate the updated internal state. A *goal-based agent* has—in addition to knowledge of the current state of the environment—some sort of goal information describing situations that are desirable. The architecture of the *utility-based* agent comprises an agent's goal and state. Whereas a goal-based agent only makes a rough distinction between 'happy' and 'unhappy' for decision alternatives, the utility-based agent uses a *utility function* that allows ordering decision alternatives for their utility. A utility function maps a state onto a real number, which describes the associated degree of happiness. In our simulation, the utility function does the following: If a decision node offers several signs that lead to the goal, a utility function makes the agent order the alternatives according to their utility and then choose the most preferable one.

A utility function allows coping with situations of conflicting goals where both goals cannot be achieved to the same extent. Problems, in which outcomes are characterized by two or more decision criteria that are evaluated at the same time, are solved by a multiattribute utility function (Russell and Norvig 1995). For example, siting a new airport requires consideration of the disruption caused by construction, the cost of land, the distance from centers of population, the noise of flight operations and so on.

Decision criteria can be classified into goal-related decision criteria and agent's preferences. The first ones represent parameters in a utility function which are represented as values depending on the decision situation. Preferences, on the other hand, are applied if no unique decision can be made based on the goal related decision criteria. As preferences are not related to the definition of the goal, they do not necessarily lead to a better decision. Including preferences in the decision model has the advantage that the decision behavior—compared to a random decision—can be predicted more precisely. As the simulated environments of the case studies are discrete, preferences come to application in several decision situations. An open question remains, namely, how often such an undecided situation that requires preferences appears in the real, continuous world. Summarizing, we can classify decision making into:

(1) multiattribute decision (Russell and Norvig 1995)

(2) Two-step decision with a sequential use of criteria

Case (1) describes a situation where one or more criteria are simultaneously applied to drive a single decision that leads to a unique result (Figure 14a). The second case also uses goal-independent preferences if step (1) does not lead to a unique result (Figure 14b).

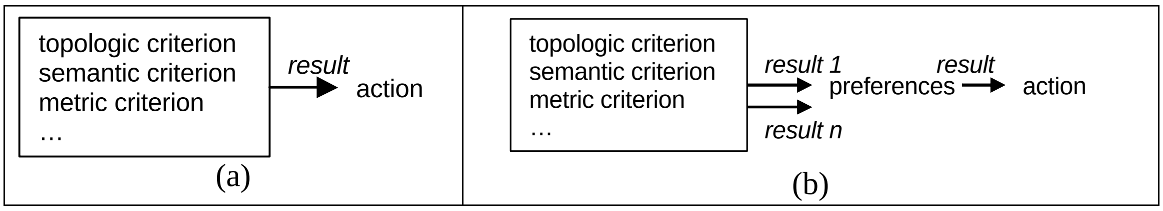


Figure 14: Principles of multiattribute decision (a) and 2-step decision (b) (Hochmair and Raubal forthcoming)

3.3.3 A Two-tiered Conceptual Model

Centuries ago, Immanuel Kant (1781) discussed the principle of *epistemological dualism*. Kant reasoned that we cannot actually experience the world itself as it is, but only get an internal perceptual replica of the world. Therefore two worlds of reality exist: the *nouminal* and the *phenomenal* world. The nouminal world is the objective *external* world, which is the source of the light that stimulates the retina. The phenomenal world is the *internal* perceptual world of conscious experience, which is a copy of the external world of objective reality constructed in our brain on the basis of the image received from the retina. Therefore the world we experience as external to our bodies is not actually the world itself, but only an internal virtual reality replica of that world generated by perceptual processes within our head.

This distinction of objective world and internal perceptual world is used in the AI tradition, where the term *belief* stresses the potential differences between reality and the agent's possibly erroneous beliefs about reality (Davis 1990). We use a two-tiered *reality and beliefs computational model* (Frank 2000) for both types of agent. The simulation separates the *representations* of the environment and the agent, and the agent's *knowledge* about himself and the environment. Each physical object in an environment can be described by perceptible properties, such as size, weight or color. Agents are objects and therefore have perceptible properties. In addition to non-living objects, an agent has beliefs about the world and himself. For example, an agent may believe that he has a certain weight or color (of which the values in the beliefs can be false).

To get a clear distinction between fact and beliefs in the agent's structure, we switch to an external perspective of the situation: We consider those parts of an agent as *facts* which are accessible from an external perspective. Those parts which are not accessible and perceptible from an external view, and therefore can be accessed by the agent only, are considered as *beliefs*. If the agent perceives facts of an object, the percepts are mapped to beliefs, and due to errors in cognition potentially distorted. Thus, fact and beliefs can be distinguished through their mode of access.

3.3.4 The Sense-Plan-Act Paradigm

A dominant view in the AI community concerning reactive architectures in the 1980s was a decomposition of an agent's control system into three functional elements: a *sensing* system, a *planning* system, and an *execution* system (Nilsson 1980). In the *Sense-Plan-Act* (SPA) approach, the flow of information among the components is unidirectional and linear (Figure 15). Information flows from sensors to the computing unit which plans the actions of the effectors.

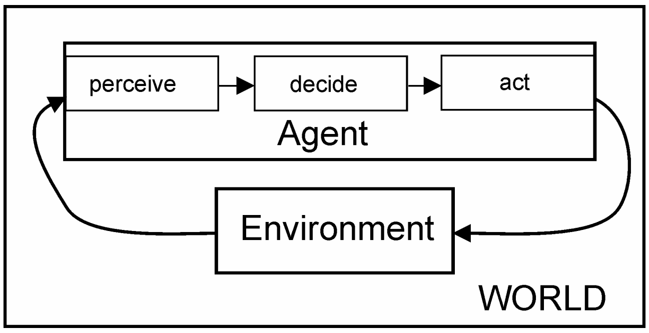


Figure 15: Basic operations of an agent system, following the Sense-Plan-Act-approach

A major disadvantage of the classic Sense-Plan-Act architecture is the time-consuming planning—which is not relevant for the topics of this thesis. Within the planning, the world may change in a way that invalidates the resulting plan. This disadvantage does not play a role in artificial environments, as they are deterministic, discrete and static.

3.3.5 Basic Operations of an Agent-System

3.3.5.1 External and Internal Operations

In this thesis, we follow the Sense-Plan-Act paradigm, i.e., the activity process of an agent can be divided into three components: perception, decision making, and action. All the simulated operations of the agent fall into one of two categories—*external* and *internal* operations (Table 5).

External Operations	**Simulation**
Perceive	add list of signposts into agent's state
Act	Change agent's position to next node
Internal Operations	
Decide	filter percepts after semantic criteria and apply metric criterion to get unique decision

Table 5: Basic external and internal activities as part of the Sense-Plan-Act approach in the agent simulation

The performance of *external operations* directly involves the agent's environment. We divide them into perception operations and action operations. The agent performs *perception operations* to receive input from the environment. In our simulation, this is done by means of simulated visual perception. *Perception* must not be mixed with *cognition* although these two processes are both components of information processing: Whereas perception is the process of obtaining information from the environment through senses, cognition means understanding of information. Thus, cognizing is part of the internal actions and connected to the perception.

Action operations execute decisions, i.e., the decisions have to be transformed into behavior (Arthur and Passini 1992). The important thing is that each decision must be transformed into the right activity at the right place. An action is characterized by two parts:

- an activity (behavior), such as turning or moving
- an environmental entity

To perform an action, the agent matches a *mental image* of something in the environment with what it *perceives* in the environment. If the corresponding part in the environment, e.g. sign, intersection, or landmark, is found, the action can be performed. As each step in wayfinding is connected to one single place in the environment, wayfinding actions are unique and ordered (see section 5.1.5).

Internal operations are performed on the agent's beliefs inside its mind and may in addition require specific information gained through perception. They do not have an immediate effect on the environment but change the state of the agent. Internal operations lead to decisions and subsequent external operations. The most important internal action in our conceptual model is the decision making process. Other internal actions include the extraction of semantic information from percepts, matching information with the predefined goal, updating the mental position after a step, or getting the mental goal from the cognitive map. Although human processing of information from perception through an assembly of neurons is a continued and parallel activity, we model discrete agent's activities in our simulation. We take this approach as the axioms are invariant under discrete or continuous activities and thus discrete, sequential activities can be modeled in a discrete environment.

3.3.5.2 Abstraction of External and Internal Operations

The function *perceive* represents the perception process of the agent. It maps the environment (E) to the percepts (P), which are part of the agent's beliefs about the environment.

```
perceive : E -> P
```

The realization of the *decision* function that represents the decision making process of the agent depends on the selected agent architecture. We distinguish between decision making of purely reactive agents and agents with internal state. A purely reactive agent directly maps input to output, i.e., percepts to actions (A). The decision function can be written as

```
decision : P -> A
```

Wooldridge (1999) gives a thermostat as an example for a purely reactive agent. If we assume that the thermostat's environment can be in one of two states—either temperature OK, or temperature too cold—the thermostat's decision function has the following form (a state 'too hot' is not modeled in the example):

```
decision T
  if T == OK = heater OFF
  if otherwise = heater ON
```

An agent with *internal state* (I) accesses his knowledge combined with his percepts to make a decision:

```
decision : P x I -> A
```

As a substep, an additional function *next* is applied, which maps internal state and percept to a new internal state.

```
next : I x P -> I
```

The *action* function requires two arguments for the input: the decision of the internal state, and the environment with that the agent interacts. The function results in an action of the agent. The environment is needed as part of the input, as a decision is related to a place.

```
action : I -> E -> A
```

3.3.6 Properties of an Environment

Russell and Norvig (1995) distinguish between artificial and real environments. Artificial environments have different properties than real environments and require different kinds of agents with different sensors and effectors. As real environments are too complex to be exactly represented in a computational model, and as wayfinding environments in the real world have a high degree of complexity (Raubal and Egenhofer 1998) one needs to apply mechanisms of abstraction.

Agents have to be coupled with an environment with which they interact. Different types of environments affect the design of agents. Normally, an agent has a repertoire of actions available to it, which describes the agent's ability to modify its environment (*effectoric capability*). Actions have pre-conditions associated with them, which define the possible situations and environments in which the actions can be applied. For example, the action to 'lift the table' is only applicable in situations where the weight of the table is small enough so that the table can be lifted by the agent.

The complexity of the decision making process can be affected by a number of different environmental properties. Russell and Norvig (1995) distinguish environments after following properties:

- Accessible vs. inaccessible
  In an accessible environment, the agent can obtain complete, up-to-date information about the environment's state. An environment is effectively accessible if the sensors detect all aspects that are relevant to the choice of action.
- Deterministic vs. non-deterministic
  An environment is deterministic if the next state of the environment is completely determined by the current state and the actions selected by the agents. The physical world can for all intents and purposes be regarded as non-deterministic.
- Episodic vs. non-episodic
  An environment is episodic if the result of an action depends only on the current perception-action cycle of an agent, i.e., an agent's action does not affect subsequent perception-action cycles.
- Static vs. dynamic
  If the environment can change while an agent is deliberating, the environment is dynamic for that agent, otherwise it is static. If the environment does not change with the passage of time but the agent's performance score does, the environment is called semidynamic.
- Discrete vs. continuous
  An environment is discrete if there are a fixed, finite numbers of actions and percepts in it. A chess game is an example of a discrete environment, whereas taxi driving is an example of a continuous one.

In our computational model, the environment is *inaccessible* to the agent as the agent can only obtain the information from the environment that it perceives at each decision point (and not the complete state of the environment). The *world* (i.e., the system of agent, environment, and time) is *episodic*, as an agent's decision does not affect future steps. We make following simplifications for the computational model:

- The environment (i.e., the *world* in our terminology) is *deterministic*: The next state of the agent is completely determined by the current agent's state. As learning is excluded in the agent's computational model, the simulated agent would redo the same action in the same decision situation.
- The environment is *static*. First, actions of the agent are not reflected in a change of the environmental state but in the agent's state (section 3.3.1), and second, we assume that no external impact changes the structure of the environment for the duration of the wayfinding process.
- As the agent can visit a limited number of nodes and perceive a limited number of objects, the abstract environment is *discrete*.

3.4 Summary

In this chapter we introduced various theories and methods that are helpful for understanding the basic features of the wayfinding metaphor, and that are part of our modeling concept of wayfinding. We discussed structures and classifications of metaphors found in the literature and saw that the role of metaphors in human everyday-life has changed over the years, i.e., that metaphoric imagination can create new unified wholes within human experience. Further we looked at formal models that describe the process of mapping semantics between two domains within a metaphor.

In the next sub section various wayfinding definitions were presented to help establish a possible central meaning for wayfinding. We looked at spatial mental models and saw a variety of features of the real world they focus on. In the field of decision making theories we showed some classifications of sub-processes of decision making and pointed out relevant factors that play a role in the human decision behavior. Further we described which affordances provided by the environment are integrated in the model of the wayfinding process in airport environments and the WWW. We gave an overview of simplifications for abstract environments.

We discussed features of agent theory that are required to develop the agent-based wayfinding model. The formalized agent is utility-based, has a state, and is separated from the environment. Facts and beliefs are separate in the agent’s structure. The agent’s operations follow the Sense-Plan-Act paradigm and can be divided into internal and external operations.

Chapter 4 Formal Tools

Within the formalized wayfinding model developed in this thesis we describe the mapping of semantics on an abstract level. Source and target domain hereby are formalized through algebraic specifications. The mapping process between these domains involves formal tools and theories (such as morphism, categories, polymorphism) to be discussed within this chapter. We describe the essential features of functional programming languages and show their advantages (for our task) compared to other programming languages. We further introduce the class structure of the functional programming language Haskell. The class structure describing mappings between domains. Graph theory is needed for the conversion of real, continuous environments into discrete, abstract environments of the formalized wayfinding model.

4.1 Algebraic Specifications

Our formal approach to describe the semantics of wayfinding in the real world and the WWW is based on algebraic specifications. Algebraic specifications describe an abstract class of objects and their behavior. Strictly speaking, algebraic specifications are language independent, but they require a formal language which is capable of expressing algebraic style specifications. The purpose of a formal specification is to formally describe the behavior of objects. Algebraic specifications were introduced to describe data abstractions in software design (Guttag, Horowitz et al. 1978). We use algebraic specifications to show that the behavior of data types—denoting objects in the real and WWW environment—satisfy a number of wayfinding axioms, so that this behavior can be called *wayfinding* in the computational model.

4.1.1 Definition

An algebra is a description of a set of connected operations that are applied to a set of types. This is the generalized definition of algebra, introduced as *universal algebra* (Birkhoff 1945). An algebraic specification consists of three parts (Ehrich, Gogolla et al. 1989):

- a set S of sorts, naming the object classes
- a set O of operations with their argument and result sorts

- a set E of axioms defining the behavior of these operations

An *algebraic specification* D is defined by the triple (S,O,E), which represents an algebraic structure. The set of sorts of the participating data and the set of operations declared on the sorts is called the *signature* of an algebraic specification, $\Sigma = (S, O)$.

A sort is a collection of objects of a particular type. If the set S contains sorts of only one type, we talk about *single-sorted* algebra, whereas a *multi-sorted* algebra includes sorts of different types.

The set O contains operations that are applicable to the sorts of S. Operations can be separated in constructors and observers (Liskov and Guttag 1986). *Constructors* are the operations that are used to define all possible states of the sort of the algebra. Their result is an object of the defined sort. *Observers* are operations to describe the functional behavior of the sort. They take objects from the primary carrier and relate them to other carriers. Their result is an object of another sort. A minimal set of operations that are sufficient to generate all values of a sort is a set of *basic constructors*, a minimal set of operations to retrieve these values as a set of *basic observers*.

Axioms can be thought of as a set of rules that describe the properties (behavior) of the operations. Axioms restrict the behavior of operations that are given through a signature. An axiom states that an operation can be reduced or rewritten as other operations while preserving its meaning. If a formal language is used to describe the axioms, the existence of a definition for operations and the consistent use of types can be checked.

Functional programming languages and algebraic specifications use a similar syntax and have similar mathematical foundations.

4.1.2 Example

We specify a stack of natural numbers. The notation is copied from Ehrich, Gogolla et al. (1989). After the keyword *Algebra* the name of the algebra is found. The keyword *Sorts* is followed by types and type parameters. In the algebra, the operations and constants are listed after the keyword *Ops* (operations). For an operation, the name of the operation, followed by '::', and the list of argument types and the return type is given. Constants are expressed through one type in the signature only. After the keyword *Eqs* ('equations') the axioms describing the behavior of the operations are listed.

As an example, a stack is a storage device where items are stored by the operation *push*. As the stack implements the 'last in, first out' principle, the operation *top* returns the topmost element. Access to lower items is only possible by first removing one by one the items above the item to be accessed which is done by the operation *pop*. The operation *empty* creates an empty stack (Figure 16).

```
Algebra Stack
Sorts Stack, Nat
Ops    empty :: Stack                                   -- constructor
       zero :: Nat                                      -- constructor
       push :: Nat -> Stack -> Stack                    -- constructor
       pop :: Stack -> Stack                            -- observer
       top :: Stack -> Nat                              -- observer
Eqs    top(push n s ) = n          -- a1
       pop(push n s ) = s          -- a2
       top(empty) = zero           -- a3
       pop(empty) = empty          -- a4
```

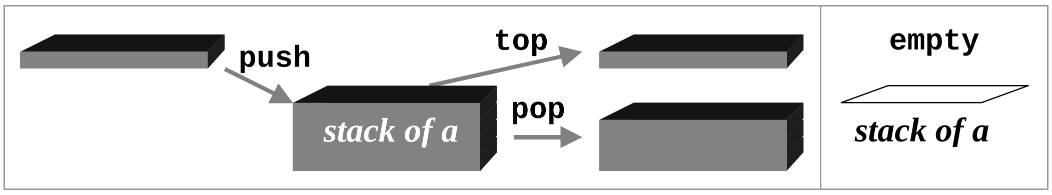


Figure 16: Functions of a stack

According to the formal notion in section 4.1.1, the given example defines an algebraic specification D = (S, O, E), where S = {Stack, Nat}, O = {empty, zero, push, pop, top}, and E = {a1, a2, a3, a4}. The behavior of the included operations is fully explained by the axioms a1, a2, a3, and a4.

4.2 Morphisms

Morphisms are structure preserving mappings of objects and operations from a source domain to a target domain. They are considered as basic concepts for metaphors (Fauconnier and Turner 1998; Goguen 1999). Metaphors are *partial* mappings between domains, where only a set of relevant operations on objects is maintained, and other operations are 'lost' in the mapping process or change their semantics. The mapped operations and functions that preserve their meaning are homomorph to the corresponding functions in the source domain. When abstracting the domain to an abstract level algebraically, morphisms describe the mapping between two algebras. Formalizing a domain as an algebra of *operations* (category), the forgetting of operations during the mapping process between categories is described by the concept of a *forgetful functor* (see section 4.4.2). Modeling the wayfinding process within a constructive programming language, we can show that the *structure* of the domains is preserved (through polymorphic functions and data types).

Morphisms can be graded after their strength, i.e., how similar the mapped objects and operations are to their corresponding counterparts in the source domain:

- signature morphism
- homomorphism
- isomorphism

The weakest kind of similarity is a *signature*-morphism which is a correspondence between the signatures defining two algebras (Ehrich, Gogolla et al. 1989). A stronger kind is the *homomorphism* which is a family of mappings from the sets of domain A to those of domain B preserving the semantics of operations. If the domains are categories, the homomorphism is provided by functors (see section 4.4.2). The *isomorphism* is a bijective homomorphism that allows a mapping in both directions without loss of information.

The concept of morphism is used to model *blending* of conceptual spaces (Fauconnier and Turner 1998), and to define similarities between the world and the cognitive model of the world within user interface theory and semiotic morphisms (Goguen 2001). Technical developments within category theory (MacLane 1971) have spurred further and deeper uses of morphisms within mathematics, and more recently in applied fields like computer science (Goguen 1999).

4.2.1 Signature-Morphism

A signature morphism is a structure-preserving mapping from a *signature* to another. Be $\Sigma_1 = \{S_1, O_1\}$ and $\Sigma_2 = \{S_2, O_2\}$ signatures. A signature morphism f: Σ_1 -> Σ_2 consists of two mappings:

(1) a mapping of the sorts: g: S_1 -> S_2, where $s_{2,1} = g(s_{1,1})$; $s_{2,2} = g(s_{1,2})$,…

(2) a mapping of operations signatures: h: O_1 -> O_2, where o_1: t_1 x t_2 … t_n -> s is mapped to a signature $f(o_1)$: $f(t_1)$ x $f(t_2)$ … $f(t_n)$ -> f(s).

As an example from Ehrich, Gogolla et al. (1989) we take the functions *push* (in the algebra *Natstack*) and *in* (in the algebra *Natqueue*). The operations are not semantically related one to each other except for the structure of the signatures.

```
Algebra Natstack
Sorts Nat, Stack
Ops push : Nat -> Stack -> Stack
```

```
Algebra Natqueue
Sorts Nat, Queue
Ops in : Nat -> Queue -> Queue
```

The signature of the operation *push* can be mapped to the signature of the operations *in* via the mappings

```
push -> in
Stack -> Queue
```

Signature morphisms can be applied to the signature of functions only. What they say about the semantics of the compared functions is that they have an equally structured data input, not more.

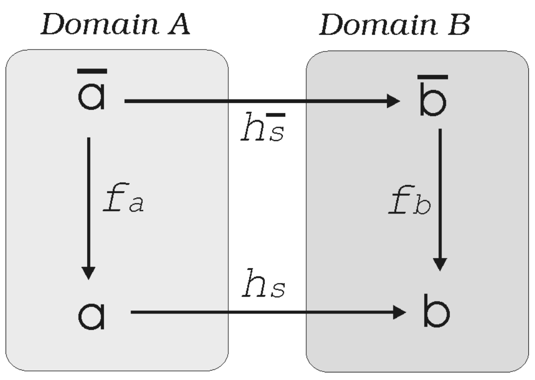
4.2.2 Homomorphism

A homomorphism h from an algebra A to an algebra B is a family of mappings {h_1, h_2, …h_n} from the set of objects S_A in domain A onto the set of objects S_B in domain B, where the *behavior* of the operations is preserved. This can be written as

$$h : A \rightarrow B$$
$$f_b(h_{\bar{S}}(\bar{a})) = h_s(f_a(\bar{a}))$$

where $h_{\bar{S}}(\bar{a}) = (h_{s1}(a_1),...h_{sn}(a_n))$, $\bar{s}$ = s_1,…s_n and $\bar{a}$ = (a_1,…a_n).

The concept of homomorphism can be visualized in a *commutativity* diagram (Figure 17).



$$\begin{tikzcd} \bar{a} \arrow[r, "h_{\bar{s}}"'] \arrow[d, "f_a"] & \bar{b} \arrow[d, "f_b"'] \\ a \arrow[r, "h_s"] & b \end{tikzcd}$$

Figure 17: Homomorphism diagram

Thus, when operations in one domain are performed and their results are mapped to another domain, the results are the same as when the arguments are mapped first to the second domain and then subjected to the corresponding operation in the second domain. The important point here is that the *semantics* of the operations must be preserved within the mapping process. If the morphism has an inverse, we talk about an *isomorphism* (section 4.2.3).

Horebeek and Lewi (1989) give an example for a homomorphism between two domains. The NAT domain consists of the set of natural numbers including zero {0, 1, 2, 3,…}, the *zero* function (creates a 0-value), and the *successor* function (increases the input by 1). The MOD2-algebra consists of the set of numbers modulo 2 {0,1}, the *zero* function, and the addition-modulo-2 function, denoted by *add2*. Axioms about the behavior of the *zero* function and the *succ* function are skipped here.

```
Algebra NaturalNumbers

Sorts Nat

Ops     zero :: -> Nat

        succ :: Nat -> Nat


Algebra Modulo2

Sorts Nat

Ops     zero :: -> Nat

        add2 :: Nat -> Nat
```

We look at the mapping of objects and operations between the two domains (see Figure 18):

- The mapping of natural numbers to numbers mod2 is the following: 0 and even numbers are mapped to 0_{MOD2}, and uneven numbers are mapped to 1_{MOD2}.
- The zero function can be mapped: $m(zero_{NAT}) = f(0) = 0_{MOD2} = zero_{MOD2}$
- The successor function can be mapped: $m((succ)(2n)) = (add2(m(2n))$ and $m((succ)(2n+1)) = (add2(m\ (2n+1))$

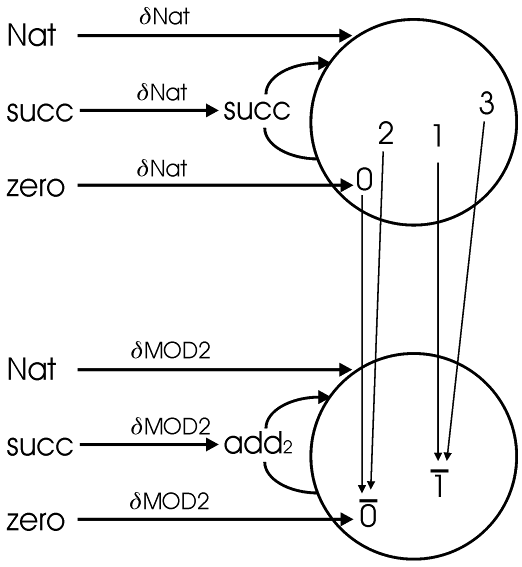


Figure 18: Homomorphism between natural numbers and numbers modulo 2 (after Horebeek and Lewi 1989)

In this example, no homomorphism exists from MOD2 to NAT as the mapping from the natural numbers to the numbers mod2 has no inverse: A unique mapping of objects from 0_{MOD2} and 1_{MOD2} to {0, 1, 2, 3, …} is not possible. Therefore, the homomorphism from NAT to MOD2 is not an isomorphism, i.e., it is irreversible.

The concept of homomorphism is not restricted to mathematical domains and computer science. There is a long tradition to compare two domains in respect to the semantics of the included functions. As an example from the past consider Ludwig Wittgenstein's picture theory. When he talks of pictures, Wittgenstein seems to have in mind pictures of the kind we would normally describe as 'pictures', but he seems to think that the basic logic of depiction applies to a much wider class of things than we would ordinarily count as pictures (Cashell 1998). Wittgenstein claims that for a picture to represent something which is actually the case,

- the elements of the picture must be correlated with elements in the situation which the picture represents, and
- they must be *related to each other* in the picture just as the elements of reality are *related to one another*.

4.2.3 Isomorphism

An isomorphism is a morphism *h : A->B*, for which exists an inverse morphism *g : B -> A*, so that $gh = id_A$ and $hg = id_B$. The domains A and B are then called *isomorphic.*

A common example for an isomorphism is the logarithm function. Similar to the given example with numbers modulo2 in section 4.2.2, it maps one domain of numbers and its functions onto another: The first domain (A) contains positive real numbers with multiplication and division to real numbers, the second domain (B) are real numbers with addition and subtraction. The multiplication and division operators (left side in Figure 19) are preserved as addition and subtraction (right side in Figure 19) through a mapping of multiplication (division) to the addition (subtraction) of powers to their base. An inverse mapping from the objects of B to A is possible using the inverse logarithm function. Therefore, the logarithm function is an isomorphism.

$$\log_a(u \cdot v) = \log_a u + \log_a v$$
$$\log_a(\frac{u}{v}) = \log_a u - \log_a v$$

Figure 19: Logarithm: an isomorphic function

4.2.4 Morphisms in Formalized Metaphors

Kuhn and Frank (1991) and Kuhn (1997) choose an algebraic approach for metaphorical mapping. The domains of a metaphor are abstracted as algebras in which the axioms define the behavior of objects and operations. The metaphorical mappings correspond to mappings between algebras which preserve the structure and the semantics of operations, i.e., homomorphism and isomorphism. Discussing the DESKTOP metaphor—a metaphor that is often used to organize graphical computer interfaces (e.g., Windows or Mac OS)—with the help of algebraic mappings, the authors show that (abstract) operations from the physical domain can be mapped to corresponding (abstract) operations in the electronic space. Depending on the amount of semantics preserved during the mapping process, the metaphor can be graded. As examples we give two formalized metaphors found in (Kuhn and Frank 1991) that represent a different grade of truth.

4.2.4.1 Metaphor with Algebraic Isomorphism

Metaphors can be graded, depending on their grade of truth (compare MacCormac 1985). The first metaphor printed here, is true in the sense of semantics of operations, i.e., representing an *isomorphism*. In its abstraction, a desktop-domain consists of the objects *Desktop*, *Folder*, and *Bool* (Kuhn and Frank 1991). It contains operators to create a new desktop (*new*), put a folder on a desktop (*put*), get a folder from a desktop (*get*), and check, whether a folder is on the desktop (*on*). In the axioms, the variables (*dt* for desktops, *f* for folders) are used for the corresponding sort.

```
Desktop
Sorts Desktop, Folder, Bool
Ops  new:                            -> Desktop
     put:    Desktop x Folder        -> Desktop
     get:    Desktop x Folder        -> Desktop
     on:     Desktop x Folder        -> Bool
Eqs  on(new,f) = false
     on(put(dt,f1),f2)    = if f1 == f2 then true
                            else on(dt,f2)
     get(put(dt,f1),f2)   = if f1 == f2 then dt
                            else put(get(dt,f2),f1)
```

Kuhn and Frank show that the desktop in the *electronic* space can be specified in an identical way (except for different sort and variable names). This means, that electronic desktops behave like real desktops with respect to the operations defined. In this example, four operations were considered to be salient and therefore included in the algebra. By including additional operators, dissimilarities could be shown, such as the fact that things fall from physical but not from electronic desktops. The designer has to decide which features of the source domain are relevant and which are not. This process of filtering functions can be described as a *forgetful functor* (see section 4.4.2). As the effects of the specified operators on office desktops correspond to the effects of the analogous operators on electronic desktops, and vice versa, the mapping between the two domains is an *isomorphism*.

4.2.4.2 Metaphor with Signature-morphism

In the following algebraic example, the signatures between the two algebras are the same, but the effects of some operations are different. Thus, the morphism between the algebras is a weaker one than in the previously given example. Kuhn and Frank (1991) compare the operations of real and electronic clipboards.

At the physical desktop, clips can be added to (*put*) and removed from the top (*get*), and it can be checked if a clip is on the board (*on*). The signature of operations is similar to the specification of desktops in the previous example, whereas the semantics of the operations is different.

```
Clipboard
Sorts Board, Clip, Bool
Ops  new:                -> Board
     put:   Board x Clip -> Board
     get:   Board        -> Board
     on:    Board x Clip -> Bool
Eqs  on(new,c)           = false
     on(put(b,c1),c2)    =  if c1 == c2 then true
                            else on(b,c2)
     get(put(b,c))       = b
```

Next, the electronic variety of clipboards, as provided by the Macintosh, is specified:

```
ElClipboard
Sorts ElBoard, ElClip, Bool
Ops  new:                          -> ElBoard
     put:   ElBoard x ElClip       -> ElBoard
     get:   ElBoard                -> ElBoard
     on:    ElBoard x ElClip       -> Bool
Eqs  on(new,elc) = false
```

```
    on(put(elb,elc1),elc2)  = if elc1 == elc2 then true
                                   else false
    get(put(elb,elc))       = put(elb,elc)
```

The underlined code indicates the two differences between physical and electronic clipboards:

- On a real board, several clips can be put on. Contrary, a clip remains on the electronic clipboard only until the next clip is put on (this has not been true since the use of *multi*-clipboards in operating systems).

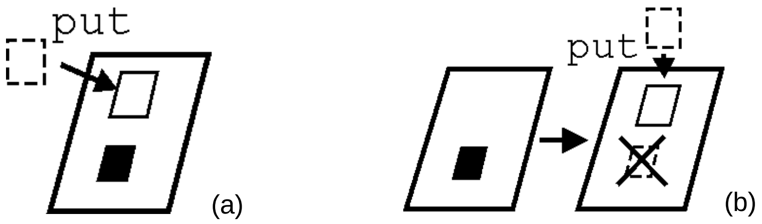


Figure 20: Different semantics of putting a clip on a real (a) and an electronic (b) clipboard

- A clip can be taken from a physical clipboard only once. In distinction, getting back a clip does not change the electronic clipboard, and the same clip can be retrieved several times.

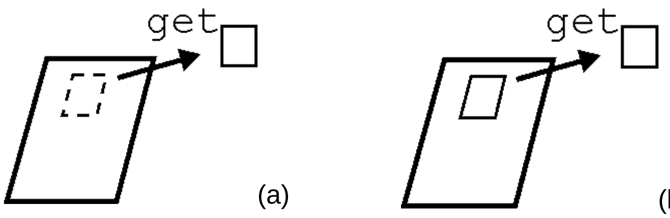


Figure 21: Different semantics of getting a clip from a real (a) and an electronic (b) clipboard

As signatures of operations are equal in both algebras, but the axioms show differences, the mapping from physical to electronic clipboards is a signature-morphism, and not a homomorphism. This statement could be refined if using an object-oriented notation of the algebra, i.e., using several classes. Then, one could distinguish between generic operations (section 4.7.4) and operations that are different in their instantiation.

The presented examples are simple in the respect that the source domains, i.e. desktop and clipboard in the real world, are compared with target domains of exactly the same algebraic structure (target and source domain have the same number of data types, and axioms). Thus, we can check the two algebras of the examples for isomorphism through line-by-line comparison.

Comparing two domains for similarity gets more complex if two abstract domains containing a different number of classes are involved. Demonstrating the similarity of source and target domain is then not possible through line-by-line comparison. We show a possible approach for this task when checking two domains for satisfying a set of wayfinding axioms.

4.3 Morphisms in the Wayfinding Model

In the computational model, the agent constructs simulated beliefs through simulated percepts of world states. The simulated beliefs may contain errors (similar to beliefs of a living human that may contain errors). Accordingly, a human's action in the environment is mapped to a simulated action (*act'*) which is the execution of a simulated decision (*decide'*). The environment is represented as a graph. The simulation contains objects and operations that are mapped from the real world to abstract domains, which then are formally compared.

Table 6 lists some of the features and operations we consider to be relevant for a simulated wayfinding process (i.e., which are mapped to the abstract domains). The sub components of objects (e.g., the agent's preferences) and sub-processes of operations (e.g., matching of perceived information with the goal definition) are not listed in the table.

Reality	**Simulation**
Physical environment or WWW	Graph
Person	Agent
Belief	Simulated belief (belief')
Percept	Simulated percept (perceive')
Decide	Simulated decision (decide')
Action	Simulation action (action')

Table 6: Mapping features and operations from the physical world and the Web to the simulated model

We assume that the effects of human activities correspond to the effects of the simulated activities in the abstract computer model. Taking this into account, the mapping from the real world domain (including human behavior in the WWW space) to the abstract domains are *assumed* to be homomorphic.

Figure 22 visualizes these mappings (labeled *h* and *g*) from the real world domain and the WWW domain to their counterparts in the computational model. These homomorphisms are independent of any wayfinding axioms. They informally expresses that objects and activities can be mapped from the source domains (i.e., airport and WWW in our simulation) to the abstract domains. Several properties of involved functions from the real world are assumed to be maintained in the abstract models. They are expressed through axioms (section 5). Depending on the class levels observed, the morphisms *m1* and *m2* between the instantiations are either inverse or not. The partiality and totality of morphisms and operations in the formal model is discussed in section 8.2.

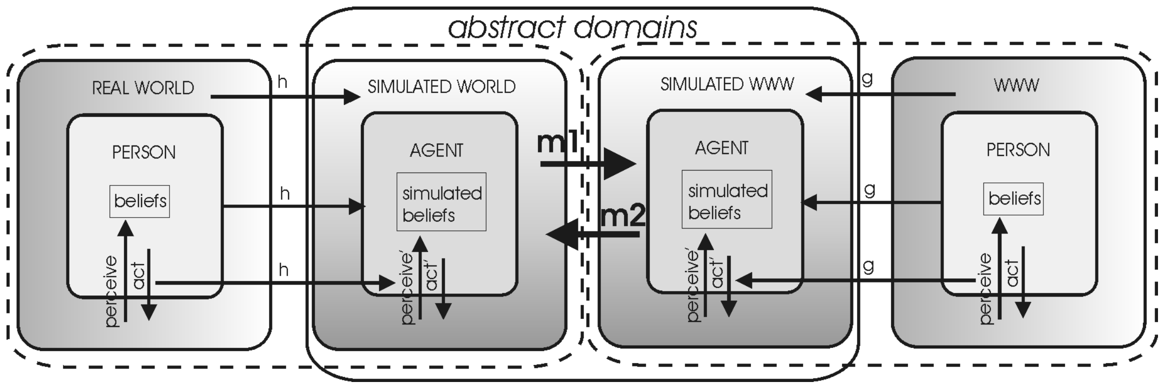


Figure 22: Mapping of objects and operations from the real world and the WWW to the abstract domains

4.4 Category Theory

In this section we introduce *mathematical* category theory (Eilenberg and Lane 1945) which is the algebra of functions. It is not related to the category theory of *cognitive science* where classes are formed by similar objects (e.g., Rosch and Mervis 1975; Rosch 1978).

Mathematical category theory deals with categories that consist of functions of some domains. One of its goals is to reveal the universal properties of structures of a given kind via their relationships with one another. Any immediate access to the internal structure of objects is prevented, and all properties of the objects must be specified by properties of morphisms. Instead of discussing the properties of individual objects, category theory focuses on the properties of the operations. Thus, properties of operations are described without reference to the objects the functions are applied to.

Category theory provides a uniform treatment of the notion of structure. This can be seen by considering the variety of examples of categories. The classical example is *Set* with sets as objects and functions as morphisms. Metric spaces form a category whose primitive elements are points and whose primitive operation is distance. The algebra of rings represents a category with rings as objects and ring homomorphisms as morphisms. Category theory is popular among algebraic topologists as it helps to assign algebraic invariants to topological structures. Thanks to its general nature, the language of category theory enables one to ‘transport’ problems from one area of mathematics, via suitable *functors*, to another area, where the solution may be easier to find. Invariants are of interest when discussing metaphors, as they are independent of any implementation and preserve functional axioms in different domains.

4.4.1 Definition

The following definition is taken from Bird and de Moor (1997) and Baez (1999):

A category **C** is an algebraic structure with a set of objects (A, B, C,…) and a set of morphisms (f, g, h,…) together with three total operations and one partial operation.

The first two *total* operations are called *source* and *target*. Both assign an object to a morphism. $f\colon A \to B$ indicates that the source of the morphism f is A and the target is B.

The third *total* operation takes an object A to a morphism *id*: $A \to A$, called the *identity* on A.

The *partial* operation is called *composition*. It combines two morphisms into another one. The composition *g.f* (pronounce "g after f") is defined if and only if f: $A \to B$ and $g\colon B \to C$ for some objects A, B, C, in which case $g.f\colon A \to C$. Composition needs

- to be associative, i.e., $f.(g.h) = (f.g).h$, and
- to have identity morphisms as units, i.e., $id_A.f = f = f.id_B$

A preordered set is an example of a category. Given two set elements p,q of the preordered set, there is a morphism f: $p \to q$ iff p is less than or equal to q. Thus, a preordered set is a category in which there is at most one morphism between any two objects.

Another example is the set of integer numbers with the operations *inc* (increases a number) and *dec* (decreases a number). Function composition of *inc* and *dec* gives the null-operations and therefore describes both morphisms as inverse to each other.

```
inc . dec = id
dec . inc = id
```

4.4.2 Functors

Abstractly defined, a functor is a homomorphism between categories. Given two categories **C** and **D**, a functor $F\colon \mathbf{C} \to \mathbf{D}$ consists of two mappings:

- mapping of objects to objects
- mapping of morphisms to morphisms

The two component mappings of a functor F are required to satisfy the property

$Ff\colon FC \to FD$ whenever $f\colon C \to D$

A functor is required to take identity morphisms to identities and composites to composites (called *functor laws*):

$F(id_A)=id_{FA}$ and $F(g.f)=Fg.Ff$

the latter condition holding whenever the composite morphism *g.f* is defined. For morphisms f: $C \to D$ and g: $D \to E$, these conditions may be visualized by commutative diagrams:

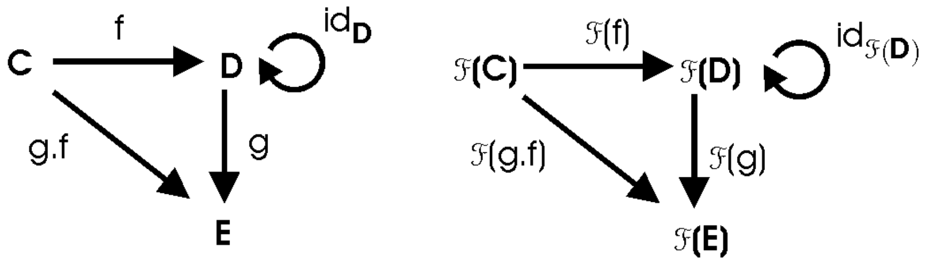


Figure 23: Composition and identity preserved by functor

Bird and de Moor (1997) give some examples for functors: identity functor, constant functor, squaring functor, product functor or the list functor.

A *forgetful* functor 'forgets' some or all of the underlying structure of an algebraic object. For example, the functor U: *Ring*→*Abelian* assigns to each ring *R* (e.g., **Z**;+;.) the additive Abelian group of *R* (i.e., **Z**;+) and to each morphism *f*:*R*→*R*′ of rings the same function *f*, regarded just as a morphism of addition. The multiplicative structure of the rings is 'forgotten'.

The idea of 'forgetting' operations between categories can be mapped to the concept of metaphor: Using a metaphor, not all operations from the source domain will be represented in the target domain. The forgetful functor keeps those *invariant* operations in the mapping process that are considered as important features of the source. The rest may be forgotten by the operator. We apply the concept of a forgetful functor when we define the wayfinding axioms and deliberately consider few operations in the real world to be essential parts of wayfinding (and worth being included in the computational model). Identity operation and function composition are assumed to be preserved in the computational model for both instances (Figure 24).

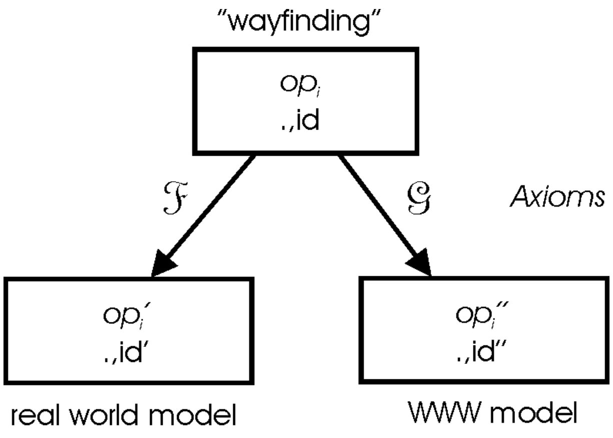


Figure 24: Forgetful functor: Abstracting wayfinding through a number of axioms

The 'forgetfulness' of the forgetful functors in the figure cannot be formalized in a computational model, as the input of the original category, i.e., wayfinding in the real world, is not completely accessible due to its complexity. What we do in the formal model, is to express a number of activities through parameterized operations that are instantiated with certain data types. Functors play a role in the definition of data types through constructor functions. This topic is discussed in section 4.5.5.

4.5 Type Systems and Polymorphism

4.5.1 Why Do We Need Type Systems?

Programming languages use data types to partition the untyped universe of values into organized collections. The purpose of a type system is to prevent the occurrence of execution errors during the running of a program (Cardelli 1997). In this thesis, we use type systems to demonstrate the concepts of polymorphism and homomorphism within the class system of the formalized wayfinding model.

We begin with the definition of an *untyped* universe. *Untyped* means that there is only one type in the universe. For example, in a computer memory, a bit string of fixed size is represented by the only type, called *word*. When looking at a piece of raw memory there is no way of telling what is being represented. Another example is the λ-calculus, where everything is a function. Yet there is only one type, i.e., the type of functions from values to values, where values are themselves functions.

Types arise informally in any domain to categorize objects according to their usage and behavior (Cardelli and Wegner 1985). In programming and mathematics, types impose constraints which help to enforce correctness. A type protects its underlying untyped representation from arbitrary and unintended use. Objects of a given type have a representation that respects the expected properties of the data type. The representation is chosen to make it easy to perform expected operations on data objects. Type systems cannot prevent execution errors, such as divide by zero and dereferencing *nil*.

4.5.2 Type Inference and Strong Typing

In a programming language, types are associated with constants, operators, variables, and function symbols. With the help of a *type inference* mechanism, types of expressions can be inferred, when little or no type information is given explicitly (Cardelli and Wegner 1985). That means that if some predefined types are given, the inference mechanism can logically deduce types of expressions that include the predefined types. Thus, a programmer is not forced to explicitly assign a type to each expression as it can be inferred. Type inference has a long tradition in functional programming languages (e.g., Milner 1978).

For *explicitly* typed languages, types are part of the syntax, where for *implicitly* typed ones it is not. No mainstream language is purely implicitly typed, but languages such as ML or Haskell support writing program parts where type information is omitted. The type inference mechanism of those languages automatically assigns types to such program fragments.

Languages in which all expressions are type-consistent are called *strongly typed* languages. This is typical for most functional programming languages, such as Miranda, ML, and Haskell. For strongly-typed languages, the compilers can guarantee that the programs it accepts will execute without type errors. Run-time tags or type checking are not required, since type checking occurs at compile-time (Goldberg 1991). All statically typed languages (e.g., Pascal or C), i.e., languages in which the type of every expression can be determined by static program analysis, are strongly typed, the converse is not necessarily true.

Weakly typed languages, e.g., BASIC, JavaScript, and Perl, enforce type rules with well-defined exceptions or an explicit type-violation mechanism. They are much more flexible about the data stored in the variables. Weak typing catches fewer errors at compile time than strong typing does.

4.5.3 Polymorphism

In *polymorphic languages*, values and variables may have more than one type. A *polymorphic function* is a function that can be applied to arguments of different types. *Polymorphic types* are types whose operations are applicable to values of more than one type. In contrast to polymorphic languages, *monomorphic* languages, such as Pascal, are based on the idea that functions and procedures, and hence their operands, have a unique type. Every variable can be interpreted to be of one and only one type.

Universal polymorphism can be classified into *parametric* polymorphism and *inclusion* polymorphism. Universally polymorphic functions work on an infinite number of types (all the types must have a given common structure). *Parametric polymorphism* is the purest form of polymorphism as the same object or function can be used uniformly in different type contexts without changes (Cardelli and Wegner 1985). An example of a function that exhibits parametric polymorphism is the *length* function (*length :: [a] -> Int*). This function calculates the length from a list of elements of arbitrary types (*[a]*), thus does its work independently of the argument type. *Inclusion polymorphism* models subtypes and inheritance, which allows the properties of one or more types to be reused in the definition of a new type. Subtypes and inheritance, in turn, are basic features of object-oriented programming. In Haskell, inheritance is modeled within the context of a class.

Ad-hoc polymorphism is obtained when a function works on several types but gives its operations a different meaning. In *overloading*—one of the two kinds of ad-hoc polymorphism—the same variable name is used to denote different functions, and the context is used to decide which function is denoted by a particular instance of the name. Hereby, the compiler resolves ambiguity at compile time and eliminates overloading by giving different names to the different functions; thus overloading is a purely syntactic way of using the same name for different semantic objects, and therefore some kind of *apparent* polymorphism (Cardelli and Wegner 1985). An example for overloading is given in section 4.7.2. A *coercion*—the second kind of ad-hoc polymorphism—is instead a semantic operation which is needed to convert an argument to the type expected by a function in a situation which would otherwise result in a type error. For example, the literals 1, 2, etc. are often used to represent both fixed and arbitrary precision integers; or numeric operators such as '+' are often defined to work on many different kinds of numbers.

4.5.4 Algebraic Data Types

Data type systems are widely used among functional programming languages. Data types can be divided into

- *base* types, whose values are given as primitive, and
- *composite* (or derived) types, whose values are constructed from those of other types.

More complex types (so called *user-defined* data types) can be created with a type constructor. We describe algebraic data types as used in the Haskell functional programming language.

4.5.4.1 Base Types and Composite Types

Haskell contains following pre-defined, built-in base types in the standard prelude file: fixed size integers (*Int*), arbitrary size integers (*Integer*), single precision floating point numbers (*Float*), double precision floating point numbers (*Double*), Boolean values (*Bool*), characters (*Char*). The symbol '::' can be read as 'is of type'. *Integer* and *Double* are not used in our simulation.

```
1 :: Int

1.0 :: Float

True, False :: Bool

'a' :: Char
```

Composite types are lists (*[t1]*) and tuples (*t1,t2,…tn*), which consist of several pre-defined base types (see also section 4.5.4.3). Lists can be arbitrarily long, but all elements must be of the same type. *Strings* are special kinds of lists, namely lists of characters. A finite list is denoted using square brackets and commas. The empty list is written as *[ ]* and a singleton list, containing just one element a is written *[a]*.

List comprehension provides a way to write down a list in terms of the elements of another list. The left side of the '|' symbol denotes an arbitrary expression, where on the right side there are one or several *qualifiers*. A qualifier is either a *generator* or a boolean-valued expression. The symbol '<-' in the generator denotes the mathematical symbol '$\in$' of being an element of a set. As an example for list comprehension we take a list *ls* which is [2,4,7]. The list comprehension

```
[ 2*n | n <- ls, n < 5]
```

results in [4,8]. It takes the list of value 2*n, where *n* is drawn from the list *ls* (generator) and n < 5 (boolean valued expression). If the boolean valued expression does not yield *True*, the element (i.e., the value 7) is not included in the result.

A *tuple* consists of a predefined number of objects. The type (*t1,t2*) corresponds to the cartesian product operation of set theory, where the notation '$t1 \times t2$' is more often seen. The number of elements is 2 or higher. Tuples represent a product type (see next section) of its base types.

```
(1, ‘a’) :: (Int, Char) -- pair
("Hugs", 1.5, 4) :: (String, Float, Int) -- triple
```

4.5.4.2 User-defined Data Types

User-defined data types are declared by the keyword *data*, followed by the name of the new type, an equals sign, followed by one or more *alternatives* separated by '|'. The alternatives each introduce a constructor function which takes 0 or more elements. Thus the general form of the algebraic type definition is

```
data Typename = Con1 t11 …t1n | Con2 t21…t2n …| Con3 …
```

The simplest algebraic type definitions are an enumeration of the elements or values of that new type. It is called *enumerated type* and represents the disjoint union of its elements. For example, the data type

```
data Temperature = Cold | Hot
```

introduces the data type *Temperature* which has two members. *Cold* and *Hot* are the constructors of the type *Temperature* which both have no arguments. The vertical bar (read "or") is interpreted as the operation of disjoint union. Thus, distinct constructors are associated with distinct values.

If the alternatives in a data type definition include other types, rather than being simple constants, this gives a *union type*. This defines a data type in terms of other data types. For example, the data type *Either* consists of 130 values: *B True*, *B False*, *C ascii0*, *C ascii1*,…*C ascii127*:

```
data Either = B Bool | C Char
```

In this example, the names *B* and *C* denote constructors for building values of type *Either*. Each constructor denotes a function of which the types are:

```
B :: Bool -> Either
C :: Char -> Either
```

A product type is a type that consists of at least two components which represent values from the two constituent data types. The following example from the abstract wayfinding model in this thesis introduces a new data type *Agent*. To construct an element of type *Agent*, one needs to supply two values: One of type *Fact*, and another of type *Beliefs*. A user-defined data type is called *n-ary* if it takes *n* numbers of arguments. In this example, the constructor function takes two argument types and is therefore called *binary*.

```
data Agent = Agent Fact Beliefs
```

Another notation for the data type *Agent* is (Fact × Belief). This notation expresses that the set of values for the product data type is the Cartesian product of values from the two constituent types. The number of values of the data type *Agent* is given by the product of the number of values in *Fact* and *Beliefs*.

4.5.4.3 Polymorphic and Recursive Data Types

It is also possible to define *polymorphic algebraic types*, where the constructor functions become polymorphic. Hence, data types can be defined without explicitly stating the type of its components. The type parameter must be instantiated with a type when the data is used, such as *EdgeEnv* and *EdgeMental* in the following example.

```
data E n = E n SignPost n SignPost
data EM n = EM n n
type EdgeEnv = E Node
type EdgeMental = EM NodeM
```

The data type *E n* represents an edge in the environmental graph. It consists of a start- and endnode (denoted by an *n*), and a sign (type *SignPost*) at the startnode and at the endnode.

The data type *EM n* represents an edge in the agent's cognitive map, and consists of a start and endnode. In both data types, the parameter *n* represents an arbitrary node, and can therefore be instantiated for example with the data types *Node* or *NodeM*.

Constructor functions can be used in *recursive* definitions, i.e., it is possible to use the algebraic type being defined in a data definition within its own definition. Lists are a common example of a recursive type. They are either empty or they consist of a head and a tail where the tail is also a list (this is a recursive union type). A polymorphic list can be written as:

```
data List a = NilList | Cons a (List a)
```

where the *Cons* constructor is equally to the ':' operator, which adds an element to the list: *(1:[2,3] = [1,2,3])*. The data type definition shows that all elements of a given list must have the same type. The same is true for the recursive data type *Tree*: A tree is either nil or given by combining a value and two sub-trees. A polymorphic tree is defined as

```
data Tree a = Nil | Node a (Tree a) (Tree a)
```

4.5.5 Data Types and Functors

In the domain of constructor functions a functor can be seen as a combination of a type constructor F of kind *->* and a mapping function that lifts a given function of type *a->b* (in its simplest case) to a function of type *f a -> f b*. Thus, the mapping function is a higher order function. In Haskell, the concept of a functor is captured by the *Functor* class definition:

```
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)
```

Instances of this class are supposed to satisfy the two functor laws (see also section 4.4.2):

```
fmap id = id
fmap (ϕ.ψ) = fmap ϕ . fmap ψ
```

MacLane and Birkhoff (1967, p.131) describe a functor as: "Many constructions of a new algebraic structure from a given one also construct suitable morphisms of the new algebraic system from morphisms between the given ones. These constructors will be called *functor* when they preserve identity morphism and composites morphism".

Typical examples of functors are recursive types such as lists or trees. In these cases, the mapping function applies to the first argument of the data type leaving its structure intact. For example, the *list* functor takes a set *a* to the set *[a]*, and a function *fmap f* that applies to each element of a list (which describes the functionality of the *map* function).

```
data List a = NilList | Cons a (List a)

instance Functor List where

  fmap f NilList = NilList

  fmap f (Cons t l) = Cons (f t) (fmap f l)
```

The first equation shows that the *Null* operation of the *List* function is preserved. To prove that the second functor axiom is satisfied, one needs to replace *f* with a composed function (*g.h*), and show that the equation

```
fmap (g.h) (Cons t l) == fmap g . fmap h (Cons t l)
```

yields true (which is not shown here). The Haskell prelude expresses the previously described semantics of the *map* function as the instantiation of the class *Functor* with the list operator:

```
instance Functor [] where

  fmap = map
```

and therefore

```
map :: (a -> b) -> [a] -> [b]
```

As an example for the *map* function let us consider *C* to be a category with a set of objects of the data type *a* (e.g., of type *Int*), and a morphism *f* between its objects (e.g., the ‘+’ operation). Mapping the set of objects from C to a list of objects *[a]* in a category *D*, and lifting the morphism *f* to each element of the list using the *fmap* function, represents a functor. The *id*-function (i.e., (+0)) and function composition are preserved. Figure 25 visualizes an example with concrete integer values.

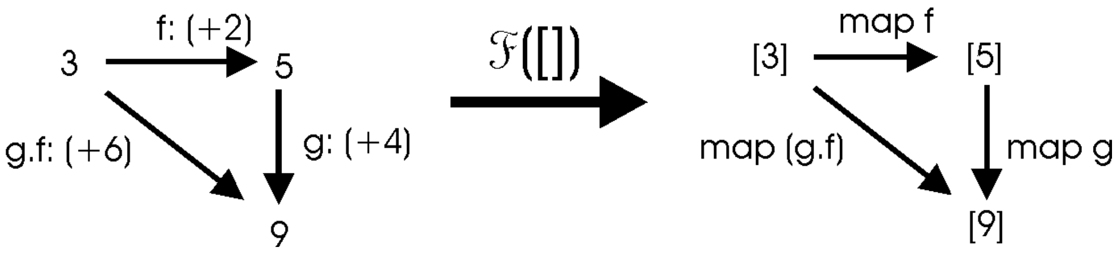


Figure 25: Mapping between two categories with a functor that is instantiated with the *List*-function

The *fmap* function can also be instantiated for non-recursive data types. This means that a function *a->b* can be lifted to non-recursive data types. Let us assume the following parameterized data type *D* and the instantiation of the class *Functor* with *D*:

```
data D n = NilD | D n n Int

instance Functor D where

  fmap f NilD = NilD

  fmap f (D n1 n2 i) = D (f n1) (f n2) i
```

Mapping the *length* function, e.g., onto the components of the parameterized data type *D* gives the following result:

```
> fmap length (D "Haskell" "code" 1)

D 7 4 1

> fmap length NilD

NilD
```

As the class *Functor* must be instantiated for data types of kind *->*, a type constructor with one parameter, such as *List* or *Tree* (section 4.5.4.3), is a valid functor, whereas the declaration

```
instance Functor List Int
```

would result in a kinding error (the data Type *Tree Int* is of kind *).

4.6 Functional Programming

Functional programming has some typical features that are not provided in procedural languages. We show the advantages of these features of functional programming for the task of this thesis. A comprehensive comparison of imperative and functional programming languages can be found in the Turing Award lecture by John Backus (1978). Referential transparency and strong typing, which are used in most functional languages, have been discussed in section 4.5.2.

4.6.1 Each Expression Is a Function

Programming in a functional language consists of building definitions and using the computer to evaluate expressions (Bird and Wadler 1988). In a functional programming language everything is a function. As functions in mathematics, these expressions give the same result for the same parameters. Programs are expressions which are evaluated and not a sequence of statements that are executed. Functional programming is a style of programming that emphasizes the evaluation of expressions, rather than execution of commands, and the interpreter works by replacing equals with equals, until no further replacements are given. Functional programming languages allow reasoning based on substitution, i.e., all values are assigned once and cannot change their value during the execution.

Contrary, structured programming languages, such as C, Pascal, or Modula, distinguish between constants, variables, and functions. Constants are static, whereas variables are dynamic. Functions are calls to pieces of code, diverting the flow of control from the calling function to the referenced function. Structured programming languages are executed line by line, with occasional jumps or functions calls. Constructs like *begin/end*, *while/do*, *repeat/until*, or *goto* are examples for sequencing. In functional languages, where every expression, even the main routine, is a function, there is no explicit flow of control. This is an advantage compared to structured programming languages, as code with loops is a regular source of programming errors.

4.6.2 No Side-Effects

Pure functional languages are free of side-effects, they compute only their result. A side effect is a construct that modifies the state of the system. The most common side-effects are assignment, input and output. Functions with side effects change a global state, which can influence the result. A typical example used in procedural languages for implicit storing of a state is the assignment to a counter (e.g., *i:=i+1*). Assignment is not possible in functional programming: Substitution of *i* on the right side with '*i+1*' gives a situation ('*i:=i+2*') that is different from the original line. Instead of loop, functional programming uses recursion. If a functional language is completely free of side effects, it is called a *pure* functional language, if some side effects exist, the language is *impure*.

4.6.3 Higher-order Functions

A higher-order function takes a function as an argument and returns a function as the result. A mathematical example is the derivation function, which takes a function as an argument and yields its derivative (which is a function, too) as the result. Higher-order functions are used to define axioms between functions in a category.

The most often cited example of higher-order functions in functional programming is the *functional composition*, denoted by the dot (.) operator. The composition of two functions *f* and *g* is the function *h* such that *h x = f (g x)*:

```
(f . g) x = f (g x)
```

The signature of functional composition is given by:

```
(.) :: (b->c) -> (a->b) -> (a->c)
```

Function composition plays an important role for the definition of a category (section 4.4.1). Further, functional composition expresses functions in a *point-free* style: A function can be formulated without reference to specific data types but described exclusively in terms of functional composition and algorithmic strategies (Bird and de Moor 1997). A *point-free* style of programming is free of the complications involved in manipulating formulae dealing with bound variables introduced by explicit quantifications.

The function *any* is a higher-order function that composes the *map* (section 4.5.5) and the *or* function. In a first step, *any* maps a boolean function to all objects of a list which results in a list of Boolean values. In a second step (*or*), the resultant list is checked for *any* value to be *True*. If this is the case, *any* results in *True*, otherwise the result is *False*.

```
any :: (a -> Bool) -> [a] -> Bool

any p = or . map p
```

For example, the execution of

```
any (>4) [1,3,5]
```

results in *True*. First, the function *(>4)* is mapped to all objects in the list which results in *[False, False, True]*, then function *or* checks the list for a *True* value, which results in *True*.

The *fold* function folds a function *f* into a list of objects *[a]*. The operation to be folded must be a binary function over the type *a*. The function *fold*, which is known as *foldr1* in Haskell, gives an error when applied to an empty list argument (the 'r' in the definition is for 'fold, bracketing to the right'). A modified definition (*foldr*) takes an extra argument that defines the value on the empty list.

```
foldr1 :: (a -> a -> a) -> [a] -> a

foldr :: (a -> b -> b) -> b -> [a] -> b
```

This is used in defining some of the standard functions of Haskell, such as:

```
or :: [Bool] -> Bool

or = foldr (||) False
```

Here, the '||' function on an empty list is defined as *False*. In the following example, folding the '+' function on an empty list yields 5.

```
> foldr (+) 5 []

5

> foldr (+) 5 [1,2,3]

11
```

4.6.4 Why We Use a Functional Programming Language

A major advantage of using a functional programming language compared to structural programming languages is the possibility to express parameterized functions. Parameterized functions are used as generic functions in the simulation, i.e., the semantics of these functions is equal for all applied data types. In functional programming languages, type inference is supported for parametric data types also and not only for a number of predefined data types. This is not true for traditional programming languages, such as Pascal where functions are defined for a specific data type. We use parameterized functions to show that certain operations involved in wayfinding are equal for both instantiations of the simulated wayfinding model, e.g., the *perceive* function. Parametric functions provide a means to discuss the wayfinding problem on a more abstract level, free of problems that might arise from a specific representation.

Higher-order functions—which are hardly provided by procedural programming languages—give the possibility to express functions in a point-free notation. Especially function composition, which is a basic operation of functional programming, makes it possible to view the wayfinding process as a category. Thus, the behavior of functions can be discussed, free of the implementation of specific data types.

4.7 Haskell

The lazy functional programming language Haskell is named after the logician Haskell Curry (1900-1982). His main work was in mathematical logic with particular interest in the theory of formal systems and processes. He formulated a logical calculus using inferential rules. The standardization of Haskell is supported by the scientific community (Peterson, Hammond et al. 1997). Haskell allows checking the syntax, type consistency and axioms of specifications already in an early phase of software development. It is a compiler that enables immediate execution of specifications. The static type system ensures that Haskell programs are type safe (Hudak 1989). Haskell's type system guarantees that all type errors are detected during the compilation process.

In this section we describe those features of Haskell which are important for this thesis. These are

- polymorphic functions and type inheritance (modeled through classes and instances)
- pattern matching

4.7.1 Classes

A class is used to model the behavior of a data type or a parameterized family of data types (Jones, Jones et al. 1997). The collection of types over which a function is defined is called type class or simply class (Thompson 1996). Classes allow us to express *polymorphic* functions (which are equally defined for all data types), and to *overload* functions (which use different definitions at different types).

A class consists of a set of operations expressed by functions applied to one or several data types. The class declaration (called the *header*) introduces the name of the class, lists the parameters, and may list conditions for the parameters (called *context*). In the lines after the class header, the signatures of operations are given, describing name, arguments and result of each operation.

The type class *Eq* as defined in Haskell's standard prelude is a simple and useful example. It takes one parameter *a* in its operations. The declaration of the class is given as follows:

```
class Eq a where
   (==),(/=) :: a -> a -> Bool
   x/=y = not (x==y)
```

The third line of the class declaration provides a default definition of the '/=' operator in terms of the '==' operator. If a method for a particular operation is omitted in an instance declaration, then the default one defined in the class declaration, if it exists, is used instead. Thus, in this example it is only necessary to give a definition for the '==' operator to define all of the member functions for the class *Eq*.

4.7.2 Instances

Haskell separates the abstract definition of an algebra on a parameterized type, from the instance which represents the implementation of an abstract data type. Thus, an instance describes how to apply operations of a class to a particular data type. Operations in an instance are given in form of executable equations.

In Haskell, the built-in instances of *Eq* include the base types *Int, Float, Bool, Char,* i.e., the function '==' is applicable for each of these types. It is possible to *override* the default member definitions by giving an alternative definition as appropriate for specific instances of the class. As an example from the simulated wayfinding model we take a data type *Edge* as instantiation for the parameterized data type *E n* (see section 4.5.4.3) with the data type *Node* and define the '==' function on it. Corresponding to the data type *E*, the data type synonym *Edge* consists of four components.

```
type Edge = E Node  -- Node SignPost Node SignPost
```

We can freely choose the way in which edges are compared for equality, for example just by testing for equality of the signposts of the start node:

```
instance Eq Edge where
   (==) (Edge sn1 ssp1 en1 esp1) (Edge sn2 ssp2 en2 esp2) = (==) ssp1 ssp2
```

4.7.3 Classes with Multiple Parameters

Classes can also be defined for multiple parameters, which allows the modeling of *multi-sorted* algebras. This feature hides implementation issues from the specification of functions. We can for example define an operation on edges (compare to section 4.5.4.3) without specifying how nodes *and* signposts are expressed:

```
class Edges e n s where
  startnode :: e n s -> n
  endnode :: e n s -> n
```

A representation (data type *E*) is also parameterized in a similar way (where the number of parameters is different to the definition given in section 4.5.4.3):

```
data E n s = E n s n s
```

An instantiation of the class *Edges* on the data type *E* is defined as follows:

```
instance Edges E n s where
  startnode (E n1 sp1 n2 sp2) = n1
  endnode (E n1 sp1 n2 sp2) = n2
```

The type of the result is not fixed and depends on the type of argument, i.e., of which data type the first parameter of *E* is.

4.7.4 Context

The concept of *inheritance* is modeled within the *context* of a class. This allows programming in an object-oriented style. The context of a class lists conditions for its parameters, and inherited behavior can be specified for each parameter.

Let us consider the function *getSignPostForNode* from the wayfinding simulation. This function takes a node and an edge of the environment as input and checks which node of the edge matches the input node. For the node of the edge where the boolean value is true, the function returns the attached signpost (Figure 26).

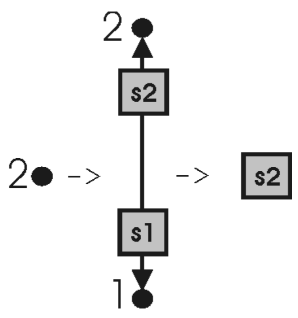


Figure 26: The function *getSignPostForNode*

The function returns all signposts that are perceivable from a node. It requires comparing nodes for equality using the '==' function. We must make a restriction on the arbitrary nodes for the class *Edges* which says that the equality over *n* is defined. Thus we must add a context to the class declaration, ensuring that for each implementation of the class *Edges* an implementation of the class *Eq* exists. The symbol for context is an arrow ('=>'). This symbol should not be read as implication; reverse implication would be a more accurate reading, the intention being that every instance of *Edges* is also an instance of *Eq*. Thus *Eq* plays the role of a superclass of *Edges*.

```
class (Eq n) => Edges e n s where
getSignPostForNode :: n -> e n s -> s
```

The implementation is defined as follows:

```
instance Edges E Node s where
getSignPostForNode n (E n1 sp1 n2 sp2)
   | (n == n1) = sp1
   | (n == n2) = sp2
   | otherwise = NoSign
```

In Haskell, polymorph operations are realized through parameterized operations in a class definition. The type system checks the instantiations of a parameterized function for the correctness of used data types. The definition of a function is not permitted to force any of its arguments to be polymorphic as a variable cannot have two types in a function. Parametric polymorphism as expressed in the class system does not guarantee that axioms of a function are identical and express the same *semantics* (this in general not possible for constructive programming languages), except if defined as a *generic* function *within* the class declaration.

The functions that exhibit parametric polymorphism are also called *generic* or *derived* functions. A generic function is a function that is defined by induction on the structure of user-defined data types. Such a function can be applied over all data types, i.e., it is a polymorph operation. A generic function expresses the same semantics for all parameterized data types of a class.

In creating the formal wayfinding model we tend to use several generic functions. This allows discussion of the semantics independent of any instantiation, i.e., both for the real world and the WWW.

4.7.5 Pattern Matching

A successful match in pattern matching binds the formal parameters in the pattern. In Haskell, there is a fixed set of different kinds of patterns, where matching among others is permitted using the constructors of any type, user-defined or not. Pattern matching provides a tool to define case expressions, which can be used if a function definition contains a number of equations, i.e., different semantics for different patterns. Each of these equations has a left-hand side in which the function is applied to a number of patterns. Haskell applies *sequential* pattern matching, i.e., it uses the first equation which applies. Failure of a pattern anywhere in one equation results in failure of the whole equation, and the next equation is tried. With pattern matching it is possible to define functions of which the result depends on the constructors of a data type, and each disjoint alternative of a union data type—which leads to a case statement in the processing—can give a different implementation. This plays a role when discussing the semantics of functions over disjoint parts of several instantiations.

Let us take the data type *Agent* that describes a product data type. The function *getFact* matches against the data type *Agent* and accesses the *Fact* component of a data type *Agent*:

```
getFact :: Agent -> Fact
getFact (Agent f b) = f
```

As an example where pattern matching is applied over different alternatives of a data type we take another function from the simulated wayfinding model. The function *dirNext* computes the agent's incoming direction in the agent's reference frame when the agent reaches a new node, and is part of the metric decision making process. The data type *IncomingDir* in the input represents the incoming direction in the local reference frame of a node and enumerates two alternatives. Only the first alternative (constructor *IDir*) contains information (for the airport navigating agent). On the contrary, a *WWW navigating* agent lacks an incoming direction (*NoIncDir*) as it does not locomote but moves *virtually*.

```
data IncomingDir = IDir Direction | NoIncDir
```

The function *dirNext* uses different equations for both types of agent: Matching the *dirNext* function with the constructor *NoIncDir* yields the 0-ary *NoIncDir* as result, whereas matching the constructor function *IDir* yields the equation for the real world case, resulting in an integer value that is used to describe the agent's orientation in space. Here, pattern matching expresses a different semantics for different types of agents.

```
class EnvAgentPos env where
  dirNext :: env -> Pos -> PrevPos -> IncomingDir -> IncomingDir


instance EnvAgentPos Environment where
dirNext env pos prev NoIncDir = NoIncDir        -- WWW case
dirNext env pos prev (IDir i)                   -- airport case
  | prev == unit0 = (IDir i)
```

```
    | otherwise = ...–- gives an integer value with the IDir constructor
```

4.8 Graph Theory

4.8.1 Definitions

In our formalized wayfinding models in the airport- and the WWW-domain, the environments are abstracted as finite graphs. This section explains basic concepts of graph theory that are relevant for the description for the computational model—including the agent's interaction with the abstract environment and the abstract environment itself. The section is based on definitions found in (Piff 1991; Black and Tanenbaum 2001).

A graph is a set of items (nodes, points, or vertices) connected by edges. It can be written as G=(V,E), where V expresses the set of nodes and E the set of edges. A graph is said to be *finite* if both the number of nodes and the number of edges are finite. In a graph, each edge is determined by the pair of vertices (called *endnodes*) that it links. If two nodes have a common edge they are said to be *adjacent*. An edge is said to be *incident* with a node if that node is one of its endnodes. A *loop* is an edge where the endnodes are identical.

The number of distinct edges incident with a node is called the *degree* of the node (Figure 27). A vertex of degree 0 is called an isolated vertex, whereas a vertex of degree 1 is called a pendant vertex.

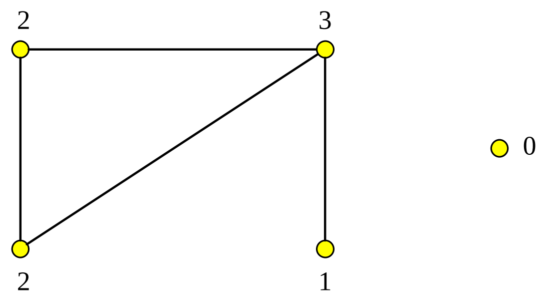


Figure 27: Vertex degrees

An *undirected* graph (Figure 28a) is a graph whose edges are *unordered* pairs of vertices. That is, each edge connects two vertices. In such a graph, the number of edges meeting at a node is the degree of that node.

A *directed* graph (or *digraph*) has directions assigned to its edges (Figure 28b) and edges are represented as arrows. In a directed graph, the *outdegree* of a node is the number of edges leaving the node, the *indegree* of a node is the number of incoming edges to that node.

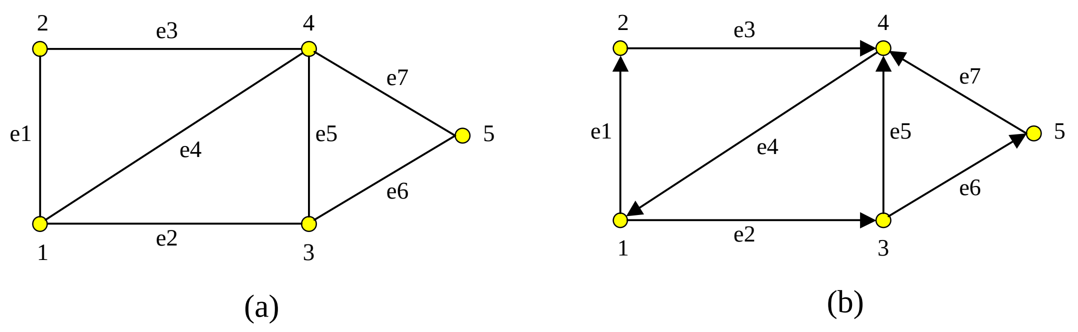


Figure 28: Undirected and Directed Graph

A graph that contains no parallel edges or loops is called *simple* graph. The alternative, where several edges can join the same two vertices, and a vertex can be joined to itself, is called a *multigraph* or *pseudograph*. A graph in which every vertex is adjacent to all others is called a *complete* graph. A graph is said to be *planar* if it can be drawn in a two-dimensional plane so that no two edges cross or intersect each other, i.e., edges can meet only at nodes.

A *walk* through a graph is a sequence of nodes <V_1, V_2, …V_n> for which any two adjacent nodes V_i and V_{i+1} are the endpoints of some edge (Sowa 1999). If the edges in a walk are all distinct it is called a *trail*. If the nodes in a walk are all distinct, apart from identity of the start and the end node, it is called a *path*. A walk, trail or path is called closed if v_n=v_1, otherwise it is *open*. These terms provide a hierarchy of concepts, depending on whether or whether not edges or vertices are repeated. Finally, a walk, trail or path is called *trivial* if it consists only of a single vertex, otherwise it is *nontrivial*. A nontrivial closed trail is called a *cycle*, a nontrivial closed path is called a *simple cycle*.

4.8.2 Shortest-Path Algorithms

There exist a number of algorithms to solve a variety of optimization problems in graphs. In our formalized model, the semantic decision process of the simulated agent uses the criterion of the shortest mental distance between concepts of the cognitive map. As the cognitive map is represented as a list of graphs, the shortest mental distance is computed with a shortest path algorithm. The abstracted cognitive map has no directed edges, thus it is sufficient to use the implementation for an undirected graph.

We give a definition of a *shortest path*: If each edge in a connected graph G = (V,E) is given a length function l, then the shortest path from u to v in G is a path P with edge set E', so that l(E') is as small as possible. For our formalized model we use *Dijkstra's shortest path algorithm*. The specification of Dijkstra's algorithm follows the interpretation and formal algorithm of Kirschenhofer (1995). For a detailed description of the substeps and visualized examples see (Kirschenhofer 1995; Car 1996). Other implementations that optimize performance for planar graphs are given by Frederickson (1987), and for sparse networks by

Johnson (1977). More details on different implementations can be found in (Ahuja, Magnanti et al. 1993).

4.9 Summary

In this chapter we presented the formalization method used in this thesis, i.e., algebraic specifications written in the functional programming language Haskell. In functional programming languages, every expression is treated as a function, there is no explicit flow of control, and there are no side-effects. We explained important features of functional programming concerning our task, including higher-order functions, type inference and polymorphism. We saw the structure of the class system in Haskell that enables programming in an object oriented style and instantiating operations for different data types. The class context hereby expresses inheritance between parameterized data types.

We discussed the role of morphism, category, and functor for the concept of metaphor, and gave a formal counterpart of these concepts within the explanation of type systems. We discussed how disjoint alternatives of a data type are differently treated with pattern matching, which is an important method used to compare the behavior of both instantiations of agents. We gave algebraic examples of metaphors and morphisms from the literature and discussed their strength of preserving semantics and structure during the mapping process.

The simulated wayfinding environment and the agent’s cognitive map are abstracted as undirected graphs. Shortest path algorithms are used to compute mental distances between concepts in the abstract agent’s cognitive map.

Chapter 5 The Wayfinding Axioms

In chapter 3.2 we gave a number wayfinding definitions reported in the literature. We saw that a unique, ‘correct’ definition of wayfinding does not exist but rather that wayfinding represents a radial category with a graduation of terms. Analyzing several wayfinding definitions we found some terms that seem to describe a kind of central meaning of wayfinding. For defining the axioms, we focus on these central properties of wayfinding and skip peculiarities of specific wayfinding strategies. We can say that the more general the term *wayfinding* is viewed, the fewer axioms are needed for its definition.

The axioms function as minimum requirements for environment, agent and the agent’s activities to call the activity ‘wayfinding’. The semantics expressed through the axioms needs to be mapped to another domain to give a metaphor. Thus, if a phrase in a natural language fails one of these axioms, the phrase (i.e., ‘wayfinding’) is not used in the correct sense. We begin with an informal definition of the wayfinding axioms and then formalize the axioms within an algebra.

5.1 Informal Description

5.1.1 First Axiom: Decision Points

The agent makes decisions during wayfinding. In a discrete environment—as it is the case in our computational model (we abstract both environments as an undirected graph, see section 2.1.3)—this enforces discrete decision points in the environment. In a more continuous environment (e.g., the desert or an ocean), decision making may occur permanently, without explicitly denoting decision points as such. We would use the term *navigating* for unstructured environments.

Decision points are those points where a navigator has the opportunity to select among different paths. Raubal and Egenhofer (1998) distinguish between points where the navigator has *one* obvious option to continue the wayfinding task (*enforced decision points*) and points where subjects have *more than one* choice to continue the wayfinding task (*decision points*). In an undirected graph, a decision point is a node that has a degree > 2: One edge is described as the incoming edge, and at least two other edges give the options from which the agent can choose. Thus, the axiom expresses the constraint on the abstract environment to include at least one node of degree > 2. This criterion is independent of the availability of information at a decision point. Thus a decision point is defined through its topology only. For a *directed* graph, the criterion would be formulated as the requirement that the *outdegree* of at least one node must be >1.

An environmental graph does not need to be checked for connectedness. Even, if the start and the goal node are not connected, the navigator can do wayfinding. The fact that the goal can potentially never be reached (e.g., through a damaged bridge along a street in a valley), does not play a role for the process to be defined as *wayfinding*.

5.1.2 Second Axiom: The Agent Has a Goal

The axiom says that the agent must have a goal. Both the representation of goal and mental position in the environment are reflected within the agent's mind. The goal is a desired state (i.e., a believed position) that the agent tries to reach. During the wayfinding process, goal and mental position have a different value. The goal is reached if these two components become equal.

5.1.3 Third Axiom: Moving Towards a Goal

The agent has a goal (as part of his beliefs) that he approaches with each step, i.e., the agent intends to reduce the mental distance between goal and mental position. This constraint of functional behavior defines the arbitrary component 'goal' in the conceptual model: If there is no intended movement towards a certain agent's state (the goal), an activity will not be called *wayfinding*, but rather described by related terms such as 'exploring' or 'sight-seeing'. In such case, the component 'goal' within the agent's beliefs loses its semantics.

There are many error sources in wayfinding, e.g., cognitive errors of perceived directions or angles, errors in reading and understanding text on signs, an erroneous or incomplete cognitive map, or unconventional rules of placing information signs in an environment. If the agent plans to move towards the goal, such errors may cause an incorrect decision behavior at decision points, i.e., lead to an action does *not* lead closer to the goal. Despite such potential errors, the agent's utility function (see section 3.3.2) stays unchanged, i.e., the *semantics* (and not the resultant values) of a utility function is independent of data used within a simulation. Thus, what counts for the axiom is the agent's *intention* to move towards the goal, and not, whether the agent *in fact* navigates towards the goal. Therefore the axiom is invariant under errors.

5.1.4 Fourth Axiom: No Impact on Environment

We consider the state of the environment invariant under the agent’s wayfinding activities (see 3.3.1). Potential impacts from *outside* the agent on the structure of the wayfinding environment would not be recognized by the agent (as it does not visit a node twice and the environment is unknown). For this reason, Allen’s claim (Allen 1999) that an element of uncertainty (e.g., in a changing environment) is a factor in every wayfinding effort, is not contradicted in this axiom. The axiom only states that the wayfinding activity does not intend to change the environment.

5.1.5 Fifth Axiom: Order of Actions

Actions in wayfinding are ordered. When executed in a sequence, the steps give a certain *path* of states. In wayfinding, making a step needs *one specific* state or pre-condition (i.e., a specific position in the environment) in order to be performed (section 3.3.5.1). Thus, a permutation of a sequence of actions does not give a path (the unique precondition for some of the permutated actions is not given). As decisions are executed (i.e., they are transformed into actions during the wayfinding process) this criterion can also be expressed in the formulation of Arthur and Passini (1992, p.27) who claim that during wayfinding and in other domains “Decisions are related one to each other; they are ordered”. The authors mention an example of opening a can of mushrooms which needs some very specific decisions:

- Get the can and the can opener
- Apply the can opener
- Activate the cutting device

One does not only have to make these three decisions, one has to execute them in a *certain order*. Chaining these actions gives a mental path that leads from the initial state (unopened can) to the mental goal (opened can). What distinguishes wayfinding from opening a can is that the latter changes the state of the environment (which contradicts axiom 4).

Some activities in everyday life are goal directed and have the *same* preconditions, i.e., *each* of these actions can be performed at the same initial state (they are *unordered*). Let us imagine the following situation: A person’s task is to move two bags of potatoes from building A into another building B. He or she can carry a bag one by one only. In Figure 29, *act1* denotes carrying the first bag to B, *act2* denotes carrying the second bag to B.

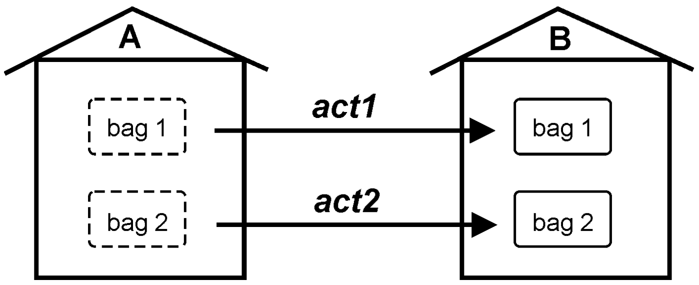


Figure 29:Unordered actions: Moving objects from one building to another

The person can start with carrying any of the two bags, the sequence of actions is interchangeable. The only precondition of *act1* is that *bag1* is in domain A, *act2* has a corresponding precondition with *bag2*. In the initial state both pre-conditions are fulfilled, therefore one can begin with either of the two actions. The state diagram of the agent's beliefs shows that both, the sequence *act1*-*act2* and the swapped sequence *act2*-*act1* lead to the same result (Figure 30).

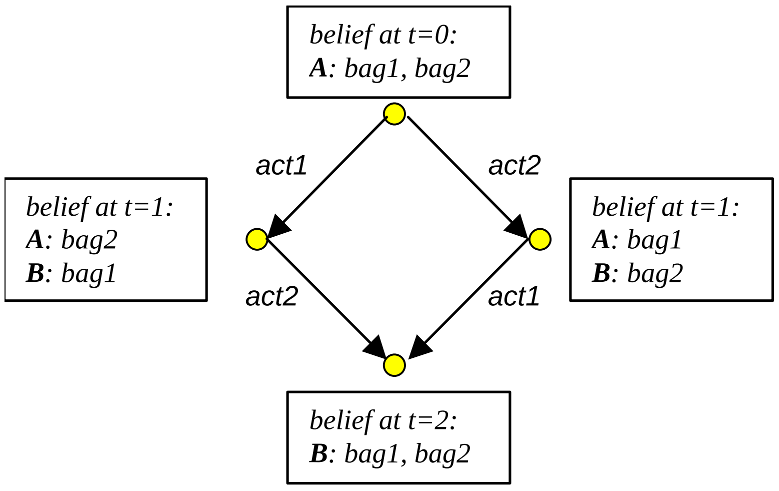


Figure 30: Agent's state diagram for unordered actions

Swapping operations is not possible for wayfinding, as is shown in the following simple example: The moves A->B, B->C give a path A-B-C. The sequence cannot be swapped as the sequence B->C, A->B cannot be executed (Figure 31). The operation A->B needs the precondition of the agent to be on node A, which is not given in the swapped sequence. In contrast to the previous example (Figure 30) where the precondition for *two* actions is given at the beginning (both bags are in building A), for a wayfinding sequence A->B->C the precondition is unique (to be at position A).

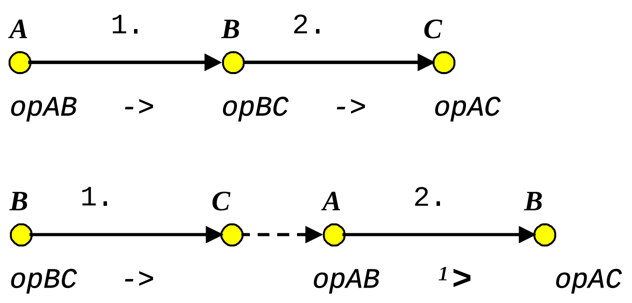


Figure 31: Wayfinding: an ordered activity

5.2 Formal Description

The first four axioms are formalized as algebraic specifications containing objects and operations. The objects hereby have no semantics but are formal stand-ins, i.e., they are used to describe the flow of information. The use of objects (i.e., data types) could be avoided if using a *point-free* notation (section 4.6.3). The disadvantage hereby would be a syntax that is more difficult to read. To convert *point-wise* axioms into *pointless style* would require currying of functions (see appendix) into functions in a single argument and then use function combinators. Methods to convert point-wise functions into a point-free style can be found in (Bird and de Moor 1997) and (Medak 1999). The fifth axiom is formalized as an algebra of *operations* (category) and describes the properties of its operations (morphisms). The strength of formalized axioms is their generality as they can be implemented into any system of any types of objects.

As we are interested in the behavior of objects in the wayfinding process, the internal structure of the used data types is not explicitly given, i.e., the structure of objects is arbitrary. Through the realization within the computational model, the arbitrary data types is given a representation. For example, the data type *Perceived* in the formalized wayfinding model is represented as a list of signs. In another model, the percepts may include further components, such as landmarks.

```
Algebra Wayfinding
Sorts:
   World, Environment, Agent,
   MentalPos, MentalGoal, CognitiveMap, Distance, Degree


Operations:
   worldStep :: World -> World
   dist :: MentalPos -> MentalGoal -> CognitiveMap -> Distance
   allDegrees :: Environment -> [Degree]
```

```
    sensePlanAct :: Environment -> Agent -> Agent


    getGoal:: Agent -> MentalGoal

    getPosM :: Agent -> MentalPos

    getCM :: Agent -> CognitiveMap

    getEnv :: World -> Environment


  Eqs:

    any (> 2) . allDegrees == True                          -- axiom 1

    goal /= pos where                                       -- axiom 2

         goal = getGoal agent

         pos = getPosM agent

    dist pos2 goal cm < dist pos1 goal cm where             -- axiom 3

         pos1 = getPosM agent

         pos2 = getPosM . sensePlanAct env $ agent

         goal = getGoal agent

         cm = getCM agent

    env1 == env2 where                                      -- axiom 4

         env1 = getEnv world

         env2 = getEnv . worldStep $ world

  ----------------

  category Wayfinding                 -- describes properties of operations

  Sorts: World

  Morphism: SensePlanAct

  Axiom:                                                    -- axiom 5

  SensePlanActAB . SensePlanActBC /= SensePlanActBC . SensePlanActAB
```

The first axiom checks if the degree of any node in the abstract environment > 2. The function *allDegrees* computes the degree of each node in the abstract environment and results in a list of integers (a degree is expressed as an integer value). This list together with the boolean function '> 2' is the input for the *any* function (see section 4.6.3). If none of the degrees is higher than 2, the *any* function results in *False*, i.e., the axiom is not satisfied.

The second axiom checks if the agent's goal and mental position have a different value. As these two elements are not semantically defined through their name (the algebra does not understand the meaning of *goal* and *pos*), the axiom only needs to check if the agent's beliefs has two components of same type and different value. In the representation of the axiom, the components *goal* and *pos* are local function definitions which are introduced by the keyword *where*. These local functions access the agent's goal and mental position through applying the observer functions *getGoal* and *getPosM*.

Mental position and goal are of the same data type. These components are modeled as parts of the agent's cognitive map. Therefore it is possible to compute a semantic distance between goal and mental position in the abstract cognitive map. In the notation of the third axiom, this is provided by the function *dist*. The agent's cognitive map is accessed by the observer function *getCM*. The local function *pos1* represent the agent's mental position *before,* and *pos2 after* a Sense-Plan-Act cycle. If the result of *dist* is smaller after a Sense-Plan-Act circle, i.e., for *pos2,* the agent has approached the goal, and the axiom is satisfied.

The fourth axiom compares the environment of the world states *before* and *after* a world step. If the two environments are equal, i.e., they stay unchanged under the triggered operations of the world step, the axiom is satisfied.

The formalization of the fifth axiom describes a property of the *sensePlanAct* function within the wayfinding category: The order of applying *sensePlanAct* functions between two nodes cannot be changed. As wayfinding is a category, the null-operation, i.e., an operation that keeps the state of the system unchanged, needs to exist. The null-operation is provided through a specific configuration of agent and environment, namely then, if the agent cannot use perceived information for making a decision. Then, the agent's state stays unchanged.

5.3 Excluded Features

In the axioms we included only a fraction of all the features of wayfinding found in the *wayfinding* definitions (section 3.2.1). We skipped those properties and features that are related to specific wayfinding strategies. Through this, the wayfinding axioms are kept general and can be mapped to abstract domains. The skipped features either relate to the behavior or the structure of the agent, or to particular properties of the environment. Examples for features that are part of the radial category 'wayfinding' but that were decided to ignore in the axioms are:

- An external map as navigation aid (Sheppard and Adams 1971)
- Path integration abilities (Loomis and Klatzky 1999)
- A marked trail in the environment (Allen 1999)
- Landmarks (Lynch 1960; Stern and Leiser 1987)
- The ability to determine turn angles and to maintain orientation (Golledge, Jacobson et al. 2000)

- Route learning (Cornell and Heth 2000)
- Communication with other navigators (Dieberger 1998; Höök and al. 1998)
- Short-term variations (e.g., temporary road detours) and long-term changes (e.g., suburban development) in the environment on which the navigator has to react (Allen 1999)

5.4 Summary

In this section we defined axioms that describe the semantics of the term *wayfinding*. The wayfinding axioms are based on terms and phrases used in wayfinding definitions reported in the literature. The axioms describe the behavior of objects and operations. They give constraints on the topologic structure of the environment, the structure of the wayfinding agent, and the operations applied in the wayfinding process. We excluded those features from the wayfinding axioms which are related to specific wayfinding strategies. The axioms will be demonstrated to be satisfied in both instantiations of the functional wayfinding model used in this thesis.

Chapter 6 Conceptual Features of The Wayfinding Agent

When introducing the case studies (chapter 2) and discussing agent theory (chapter 3.3) we already decided to adapt some of the explained concepts for the proposed wayfinding model (applied for both instances of agent and environment). The features discussed so far, concern

- the separation of the agent from the environment (section 3.3.1)
- properties and simplifications of the abstract environment (section 3.3.6)
- the distinction between fact and beliefs within the agent's structure (section 3.3.3)
- the use of the Sense-Plan-Act paradigm in the agent's sequence of operations (sections 3.3.4 and 3.3.5)

In this chapter we describe further features of the agent with a focus on details for both instantiated agents. We look at the agent's structure, including objects and the operations between the agent's components. As a part of the agent, we describe the agent's cognitive map concerning its structure and content. We will discuss how the content of a cognitive map depends on the agent's task. For the Web navigating agent we will demonstrate how to implement parts of a pre-existing ontology (*WordNet*) into the abstract cognitive map. At the end of this chapter we point out common features of the decision making strategy used for both types of agents.

6.1 The Structure of the Wayfinding Agent

The concepts defining the structure of both instantiations of the wayfinding agent are similar. Conceptual differences—either in parts of the structure or in the semantics of the operations—can only be found at a lower level of the data type hierarchy. The structure of the agent follows the architecture of a *utility-based agent with state* (section 3.3.2), its components fall into *Fact* and *Beliefs* (section 3.3.3), the beliefs denoting the agent's state. Figure 32 shows the basic components of the conceptualized agent and the operations involved in the wayfinding process. The basic operations of the Sense-Plan-Act approach are visualized in italic font, objects in regular font. For the representation we give the agent an id (added to the *facts* components). The facts of the agent are

- position
- previous position.

The state of the simulated, utility-based agent (section 3.3.2) consists of following objects:

- mental position
- cognitive map from which the goal can be constructed
- percepts, containing semantic and metric information
- decision for the next action
- metric preferences for the decision process
- the incoming direction

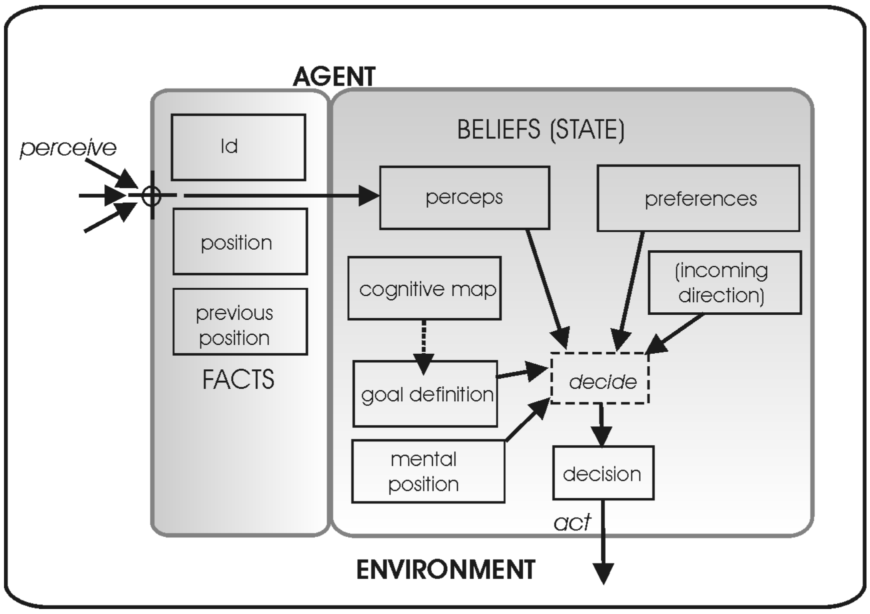


Figure 32: Basic components of a navigating agent

6.2 The Cognitive Map

6.2.1 The Role of a Cognitive Map for Wayfinding

Review of literature (section 3.2.2) revealed the outstanding role of cognitive maps in the wayfinding process. We assume that the type of how the goal is defined plays a role for the structure of the cognitive map and the information accessed in the cognitive map (section 3.2.4). We model the agent's goal to be part of the cognitive map, independent of the wayfinding strategy and the environment with that the agent interacts.

Although the goal of a wayfinding person exists mentally as part of the agent's cognitive map, such a goal does not essentially exist in the real world. Thus, the goal of a wayfinding person is *assumed* to exist. Approaching such assumed goals may be successful to a certain degree as the concepts *around* the assumed goal in the cognitive map may be matched with information perceived from the environment. An example for a goal that is *assumed* to exist (but does not exist) on the Vienna International Airport is gate ‘C399’. Reading a gate sign ‘C’ may lead the wayfinding person closer to his assumed goal ‘C399’, but the goal will never be reached. In the WWW, e.g., one may try to find a Web page where one can chat with an alien. A Web page providing this, cannot be found (except if it is a fake page). Although the second wayfinding axiom (section 5.1.2) forces the agent to have a mental goal, it does not say anything about the existence of the goal in the environment. Thus, even the process of trying to approach a *non-existing* goal can be considered as wayfinding (if the remaining wayfinding axioms are satisfied).

In the presented case studies we have a semantically defined goal (we classify topologic to be a kind of semantic here). For such a goal we model the cognitive map to consist of a semantic network (see section 2.2.4) that describes hierarchical relations between concepts of the domain—metric preference is hereby treated separately. Depending on the complexity of the goal, the number of networks that represent an abstract cognitive map varies. Goals that are defined through several features or attributes require a more complex structure in a cognitive map to be represented than goals that are expressed through one feature only.

6.2.2 The Cognitive Map of the Airport Navigating Agent

As described in the introducing case study (section 2.1.1), the task of the airport navigating agent is to find a gate labeled ‘C 54’ (which exists). The agent’s cognitive map represents a semantic network, where the semantic distance between information of perceived signposts and the mental goal is expressed through the number of edges in between (i.e., topologically). Therefore the concepts (i.e., nodes) in the cognitive map must be of the same type as the perceived information. In the airport environment, three types of gate signs can be found (section 2.1.3).

The rules of placing signs in an airport follow a simple rule, namely to make the navigator feel it is approaching a target gate when following a sequence of signs that match the name of the target gate. Structure and content of the abstract cognitive map follow this intuitive rule: The cognitive map reflects the distance between a position in the environment and the agent’s goal through the number of mental, hierarchical levels between the two corresponding concepts.

The gates are embedded within a hierarchical structure that can be described with the CONTAINER image schema. The higher the hierarchical level of the concept in the cognitive map is, the more instances (‘containers’) are included in the concept. For example, the goal ‘C 54’ is assumed to be in the containers ‘C53-C54’ or ‘C54-C55’, which in turn are in a container of all ‘C’-gates and so on (Figure 33a). Thus, if an agent perceived two signs at a decision point, for example ‘A,C’ and ‘C’, he would choose the edge connected to the sign ‘C’, when applying the criterion of the smallest semantic distance.

Metaphorically expressed, the agent’s goal is to get into one of the inner-most containers. The closer a sign at a decision point leads to the element ‘C54’ in the cognitive map, the higher is the utility of the edge connected to the perceived sign. In the simulation, the cognitive map can be simplified by dropping those hierarchical concepts that do not refer to signs in the environment (Figure 33b). Only information of those signs that are potentially perceived during the simulated wayfinding process is reflected in the cognitive map.

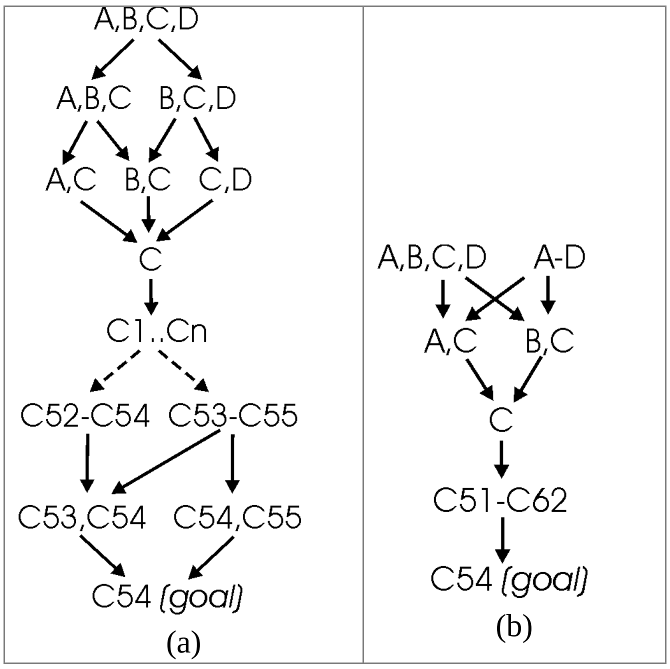


Figure 33: Hierarchical structure of airport signs (a) and simplified cognitive map for simulated agent (b)

6.2.3 The Cognitive Map of the WWW Navigating Agent

6.2.3.1 Structure of the Cognitive Map

As the task of the case study in the WWW is defined through several concepts, e.g., size and brand of an object that affords running with (section 2.2.3), the cognitive map of the Web-navigating agent is abstracted through *several* mental graphs. It is therefore more complex than the cognitive map needed for the case study in the airport environment. Different from the airport-navigating agent, concepts in the cognitive map of the WWW-navigating agent are modeled as strings, not as signs .

To find a suitable cognitive map for the given task in the WWW we go back to Aristotle's ontology that is based on substance and accident (section 2.1.2). Some objects involve both substances and accidental parts, so that objects are partially bearers of accidents. The theory of affordances (see section 3.2.3) connects objects with (potential) activities, too. Combining Aristotle's ontology with Gibson's affordance theory, we propose the WWW-agent's cognitive map to consist of following graphs (italic font indicating Aristotle's terms).

- action affordances (*events*)
- physical object hierarchy (*substances*)
- attributes (*qualities*)

Although listed in one single item, action affordances and events are not exactly the same: Events are *actual* activities and action affordances are *potential* activities. A fourth graph, named 'user intended actions', describes the activities a web user wants to perform on the desired web page. Figure 34 visualizes the basic structure of the proposed cognitive map.

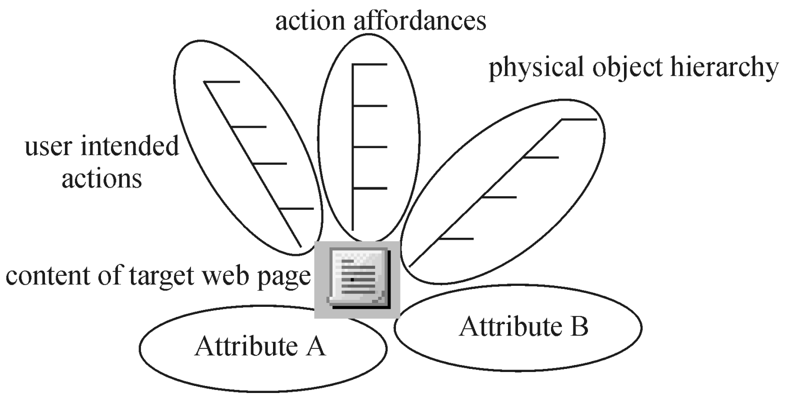


Figure 34: Structure of an agent's semantic map

User intended actions in the web are not limited to seeking (Ellis 1989; Ellis and Haugan 1997; Wilson 1997) and browsing (Marchionini 1995) but include all potential activities in the internet (e.g., purchasing, communicating, playing, advertising). All graphs except the attributes are structured as partonomy or taxonomy. The structure of the cognitive map as presented here, is sufficient for the WWW navigating agent to complete the task given in the case study. For other tasks, the structure may have to be adapted and extended with additional graphs. This is an epistemological detail which does not influence the wayfinding metaphor to be used for the Web space.

The element in the highest hierarchical layer in each graph (most distant from the target web page in Figure 34) expresses the most general term of a graph (except for the attribute-graph where the order of concepts follows subjective rules). Elements of a lower layer are either part of or kind of the term in an upper hierarchical level. The agent's goal consists of all the elements in the lowest hierarchy of each graph. The mental distance to the target page decreases with the similarity of the content of the actual web page and the defined mental goal.

6.2.3.2 Using a Predefined Ontology

The content of cognitive maps varies between individuals as their life experience is different. Therefore it is not possible to simulate the navigation behavior for each individual human. For the simulation we need one prototype agent with a prototype cognitive map. As we have not conducted experiments with human subjects, we have to rely on an existing ontology. We choose the ontology of *WordNet* (Miller 1990; Miller 1995), a database for the English language. One of the advantages of *WordNet* is its free availability in the internet. The online application can be visited at *http://www.cogsci.princeton.edu/cgi-bin/webwn*.

WordNet combines features of both a traditional dictionary and a thesaurus. All query results are given in form of synsets (Jones 1986), which describe sets of those words which can replace a particular word in a sentence without changing the way the sentence can be employed. The synsets are connected by a number of relations. Unlike in a thesaurus, the relations between concepts and words in WordNet are made explicit and labeled; users select the relation that guides them from one concept to the next and choose the direction of their navigation in the conceptual space. WordNet allows semantic queries between nouns, verbs, and adjectives.

For a query, the WordNet user enters a single term from which he wants to start the query. WordNet returns a description of the term (or a list of descriptions for a *polysemous* term). The user selects the intended meaning, and chooses the type of search (e.g., ‘synonyms’). WordNet then returns the related terms or hierarchies. Due to the fact that most terms are polysemous and the user must select the intended meaning of the term, the simulated cognitive map cannot be created automatically but requires several manual steps.

We stepwise fill the prototype cognitive map of a WWW-navigating agent that has the task defined in section 2.2.3. The structure of the cognitive map has been visualized in Figure 34. We use WordNet to fill the graphs ‘physical object hierarchy’, ‘user intended action’ and ‘action affordances’ in the semantic map. For attributes without IS-A or PART-OF relations, one needs to individually decide which values should be included in the cognitive map. How ‘happy’ a user is with an attribute-value that does not exactly correspond to the demanded value, cannot be generalized in an ontology. For example, consider a person that wants to buy red roses. If the store is out of red roses, the person may also be satisfied with white ones but not with yellow ones. This subjective gradation does not follow a general rule. Similarly, we use a subjective gradation for the values or the attributes *brand* and *size* in our simulated case study. As subjective gradations are not part of the WordNet ontology, we fill these two graphs according to subjective beliefs (see section 6.2.3.6).

6.2.3.3 Physical Object Hierarchy

A hierarchy of nouns is generated by hyponymy and hypernymy relations in WordNet. Usually a noun has only one hypernym but many hyponyms (Miller 1998). Available semantic queries for nouns among others are:

- coordinate terms (terms that have the same hypernym, 'sisters')
- hypernyms (generic term for a whole class)
- hyponyms (generic term used to designate a member of a class)
- synonyms

To fill the field 'physical object hierarchy' we request the hypernyms of 'shoe' as the physical part of sneakers. Bold terms in the result will be included in the prototype cognitive map. The polysemous term 'shoe' has four meanings in WordNet. Taking the first meaning ('a covering shaped to fit the foot'), we get following hypernyms:

shoe
=> footwear, footgear
    => **covering**
        => artifact, artefact
            => object, **physical object**
                => …

'Footwear' has two meanings in WordNet. Requesting the hypernyms for footwear in the sense of clothing gives the following result:

footwear
=> clothing, clothes, apparel, vesture, wearing apparel, wear
    => **covering**
        => artifact, artefact
                => …

For the field *physical object hierarchy,* we unite the results of the two queries (see Figure 35):

shoe
=> **footwear**
    => clothing, clothes, apparel, vesture, wearing apparel, wear
        => **covering**
            => artifact, artefact
                => …

6.2.3.4 User Intended Actions

Like nouns and adjectives in WordNet, verbs are grouped together as sets of synonyms (synsets). English has far fewer verbs than nouns, and verbs are approximately twice as polysemous as nouns (Fellbaum and Miller 1990). This has the consequence that the automated generation of a graph that denotes a taxonomy or partonomy of actions is even more complex than the generation of a noun graph.

The elements within the field ‘user intended actions’ are represented through verbs. The verb ‘purchase’ has one meaning in WordNet. It is described as:

buy, purchase: “obtain by purchase; acquire by means of a financial transaction”

This shows that buy and purchase are considered as *synonyms*. The verb ‘shop’ has four meanings in WordNet, one of them is defined as:

patronize, shop, shop at, buy at, frequent, sponsor: ”do one’s shopping at; do business with; be a customer or client of”

This, in turn, considers ‘buy’, ‘shop’, and ‘do business’ as synonyms so that the terms ‘shop’, ‘buy’, ‘purchase’, and ‘do business’ can be considered as synonyms. We drop the terms ‘purchase’ and ‘buy’ from the graph as they do not appear in the Yahoo categories but use the two others (‘do shopping’, ‘do business’) instead. They are represented at the same hierarchical level of the graph.

We add an additional term to the graph which denotes the activity of pressing a ‘purchase’ button on the user interface to confirm the purchasing process. If this action is offered to be performed, the Web page with the desired user intended action is reached. We represent this activity as term ‘confirm’ in the graph. In sum, we get three terms for the graph ‘user intended actions’.

confirm
=> **do shopping, do business**

6.2.3.5 Action Affordances

The term ‘running’ is both explained as a verb (42 meanings) and a noun (5 meanings) in WordNet. One of the explanations of the noun ‘running’ comes close to our intended meaning of doing sport: We search for hypernyms of the word ‘running’ in the sense of participating in an athletic competition involving running on a track. We get the following hypernyms that will be included in the graph ‘action affordances’:

track, **running**
=> **track and field**
    => sport, athletics

=> diversion, **recreation**
  => **…**

6.2.3.6 Attributes

A physical object has attributes, e.g., color or size. The attributes are expressed by nouns whereas attribute values are expressed by adjectives or values. Nouns serve as arguments for attributes as the value of an attribute changes, depending on the noun. What is realized in WordNet so far, is the connection between attribute nouns and adjectives which express values of that attribute. Examples are the noun *size* and the adjectives *large* and *small* or the connection between the noun *color* and the adjectives *red*, *yellow*, *green* and so on. WordNet has not implemented adjective-noun pairs so far, i.e., it is not possible to determine from WordNet the important attributes of a noun.

We deliberately decide to include two attributes for sneakers (*brand* and *size*) in the agent’s semantic map. Similar to the other graphs, the attribute-graph is hierarchically structured, but not necessarily through is-a or part-of relations. Its structural concept rather follows a subjective ordering. Besides the agent’s ‘ideal’ or ‘goal’ value for each attribute, alternative values that the agent would accept, are included in the cognitive map. All other attribute values are excluded from the graph.

Let us have a look at our simulated agent: The agent’s preferred value for the size of the sneakers is 9 1/2. Shoes that are a half size bigger (size 10) are acceptable but provide a lower happiness than 9 1/2; and sneakers of size 10 1/2 give the lowest happiness but still would be accepted. All other sizes, if offered through semantic information on a link, are not part of the mental map, and therefore no potential candidates for being selected at a decision point. Similar considerations are made for the attribute ‘brand’.

Combining all the fields we get the cognitive map of the prototype agent (Figure 35). It serves as basis for the decision process in our simulation.

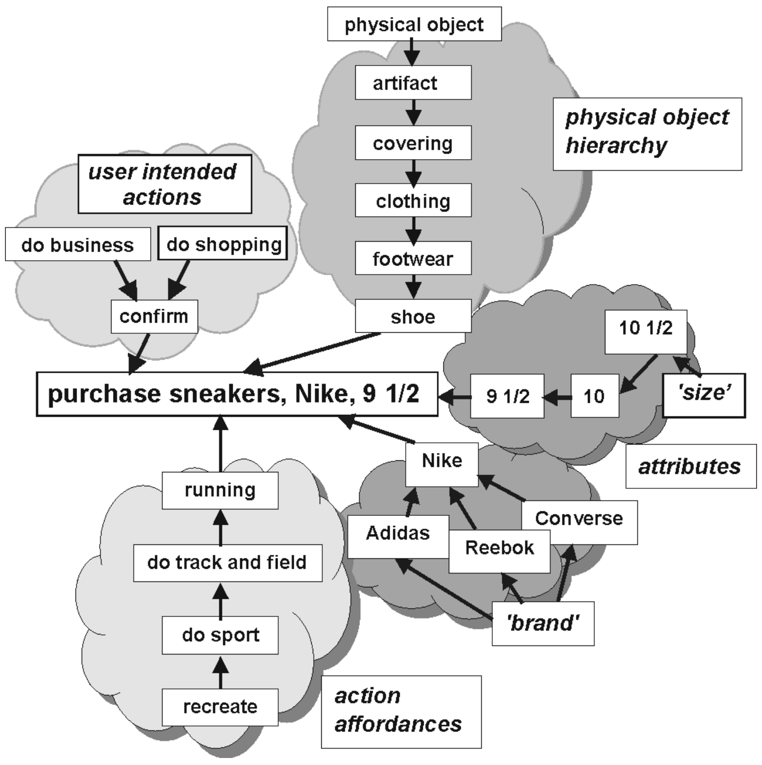


Figure 35: Cognitive map of the prototype agent, partly based on WordNet

When analyzing various web pages, we found that those elements of the mental goal that are not provided by the content of the actual web page, can usually be found after clicking the link that contains the concept of the lowest hierarchy in the graph ‘user intended actions’. Thus we make a simplification of the criterion that determines if the goal has been reached or not: The goal is modeled to be reached if the link labeled with the lowest element in the graph ‘user intended actions’ (i.e., the ‘confirm’ button in our example) can be perceived.

6.2.4 Comparing the Cognitive Maps of Both Agents

The cognitive maps of both types of agents show following commonalities:

- The cognitive map consists of hierarchically structured graphs.
- The semantic distance can be expressed through the number of mental edges between two concepts.
- The number of graphs depends on the complexity of the goal.
- The cognitive map implicitly defines the goal.

In contrast to the airport navigating model, where the goal is identified through a unique gate name, the goal of the WWW navigating agent is defined through a number of features of the target page (but not through a *unique* id). Thus, in our case study two or more different web pages of the environment may function as goal for the WWW navigating agent. It depends on the applied wayfinding strategy, which goal (if there exist more than one in the domain) will be reached.

6.3 The Decision Making Process

The decision making process as one of the internal operations plays a major role in the proposed wayfinding model. We presume that the environment is unknown to the agent. Therefore, assessing the utility of *several* sequential actions is not possible, and the decision behavior of the abstract agent is based on evaluating the *next* single action, which is called *single-shot decision* (Russell and Norvig 1995). The goals in both instances of the agent are defined semantically as part of the cognitive map (sections 6.2.2 and 6.2.3). Therefore, and for the fact that both environments offer semantic information on signs and links (we summarize signs and links as *signpost*), the decision making process stresses the use of semantic decision criteria. As both environments have metric properties (see sections 2.1.3 and 2.2.5), metric preference is also considered as part of the decision making model. Thus the model includes both, *multiattribute decision making* and a *two-step decision sequence* (section 3.3.2) within the decision process.

(1) The agent filters those signposts from the percepts which minimize the semantic distance to the mental goal (semantic, goal-related decision criterion).

(2) If step (1) does not result in a unique decision, he takes the edge with the most preferred direction (metric preference as part of a two-step decision sequence).

In chapter 7.5 we will describe the sub steps of semantic and metric decision making when introducing the formalized wayfinding model. We will show how the semantics of the operations is defined and where similarities and dissimilarities can be found between the two specified instances. A simple example for the principal method of semantic decision making has been given in section 6.2.2 (for the airport environment). In the following lines we explain how metric preference is realized in each of the two environments of the case study.

For the wayfinding agent in an airport environment, the metric bias is modeled as preferred directions within the agent's egocentric reference frame. This reference frame is represented through eight directions, i.e., front, back, left, right, and four directions in-between (Figure 36a), i.e., the directions are modeled in 45° steps within an egocentric reference frame. Each of the eight directions is given a corresponding preference value (bold numbers in the figure). The direction in front is assigned the highest preference.

For the WWW-navigating agent, preference for metric decision making is modeled through the position of the link on the screen (Figure 36b), giving the top-most link the best value. Compared to the airport navigation, metric decisions hereby are not influenced by a local reference frame.

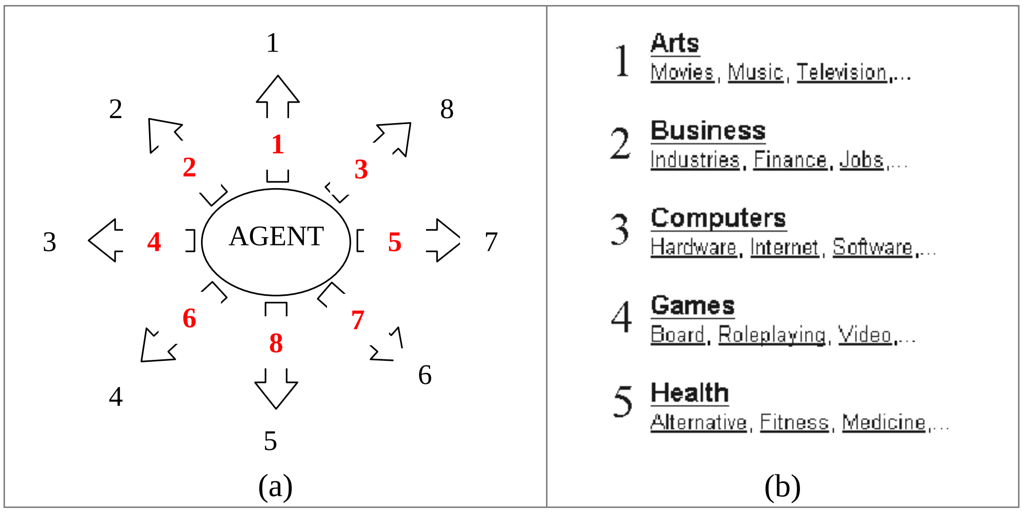

1
2
8
1
2
3
AGENT
3
4
5
7
6
7
8
4
6
5
(a)
1 Arts
Movies, Music, Television,...
2 Business
Industries, Finance, Jobs,...
3 Computers
Hardware, Internet, Software,...
4 Games
Board, Roleplaying, Video,...
5 Health
Alternative, Fitness, Medicine,...
(b)

Figure 36: (a) preference values (bold numbers) for directions within the agent's egocentric reference frame (from Raubal 2001b); (b) preference values for links on a user interface

6.4 Summary

At the beginning of this chapter we explained the structure of the wayfinding agent where fact and beliefs are separated. Further we showed which of the agent's components are involved in internal or external operations.

Both agents have a cognitive map that implicitly expresses the definition of the goal and permits them to assess the mental distance between the actual position and the goal. The cognitive map consists of a list of hierarchically structured graphs, and its complexity depends on the definition of the goal. We showed how to make use of the *WordNet*-ontology for developing the abstract cognitive map of the WWW-navigating agent.

We described the basic steps of the agent's decision behavior. Both instantiations take the semantic distance between mental position and mental goal as semantic decision criterion, and—in case of an undecided result—use metric preferences in addition. We showed how metric bias is defined for both instantiations of the agent.

Chapter 7
A Formal Model for Agent-Based Wayfinding in the Real World and the WWW

This section formalizes the conceptual model of the wayfinding agent using the syntax of Haskell (Thompson 1996). The result is an executable agent-based computational model. Our intention is to show that both the behavior of the abstract real world agent and the WWW agent can be described as *wayfinding*, i.e., that the wayfinding axioms are satisfied for both instances.

Haskell uses classes to structure the operations into semantically connected units. The semantics of operations in a class can either be expressed through derived functions (i.e., independent of instances), or through axioms within an instance. The first method uses parameterized data types in polymorph functions which are derived from other functions. The class context (see section 4.7.4) gives constraints for the parameterized data types. The second method uses instances that describe how to apply operations of a class to a particular data type. If such data type includes disjoint parts (formalized as *union data type*, see section 4.5.4.2), the instantiation of the operation may have to be overloaded for each of the disjoint parts (using pattern matching, see section 4.7.5).

In our simulation we use both methods to define the semantics of an operation, i.e., derived functions and instances. Those functions that we need for proving that the wayfinding axioms are satisfied for both instances, express the same semantics for both instances (i.e., no pattern matching over constructor functions of union data types is required). We explain those algebraic specifications and data type definitions that are needed for the verification of the axioms. All other details are left out.

In the description of the object hierarchy we use a top-down approach, starting with the highest level of objects and operations. Thus, we first explain data types and operations concerning the world, followed by a formal description of the environment and the agent.

7.1 World

7.1.1 Structure of the World

The world represents the highest level in the object hierarchy of the simulated system. It consists of three components: a time counter, an agent, and the environment (Figure 37). We separate the agent from the environment (see section 3.3.1) so that the state of the environment is not changed through the agent's operations. The world is defined through the following product data type:

```
data World = World Time Agent Environment
```

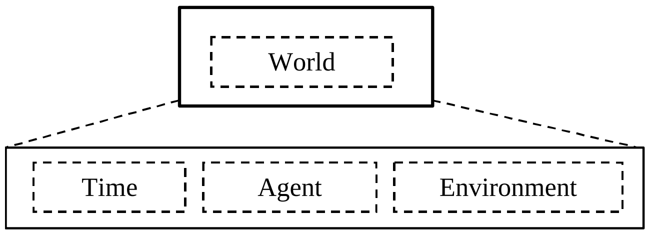


Figure 37: Structure of the data type *World*

7.1.2 Operations in the World

The operations of the world are defined within the class *WorldClass*. We discuss the two functions *worldStep* and *iterateWorldStep*. The class context gives constraints for the parameter *world*. It says that the class *CreateWorld* must be instantiated for the data types *Time Agent Environment world*. The class *CreateWorld* contains the *createWorld* function which constructs a world of the data type *World* from its components. This function is part of the *worldStep* function.

```
class (CreateWorld Time Agent Environment world) => WorldClass world where
  worldStep :: world -> world -- one sensePlanAct cycle
  iterateWorldStep :: world -> [world] -- complete wayfinding process

-- derived functions
worldStep world = createWorld (tick . getTime $ world)
  (sensePlanAct (getEnv world) (getAgent world)) (getEnv world)
iterateWorldStep world
  | (checkDegree (getEnv world) == True)
      = take (getNumberOfEdges world) (iterate worldStep world)
  | otherwise = error ("No decision points in the environment")
```

All activities of components of the world are triggered by the function *worldStep* that takes the world as input and gives a (changed) world as result. The function triggers two single events in the world:

- The time in the world is updated through calling the *tick* function. The *tick* function sets the time +1.
- The agent (accessed by the *getAgent* function) is triggered to perform a complete Sense-Plan-Act cycle through calling the *sensePlanAct* function.

The function *worldStep* shows that the environment stays unchanged within this operation: The environment is accessed through the *getEnv* function from the world, and then put back into the updated world within the *createWorld* function, in fact without applying any function on the environment. Thus, the fourth wayfinding axiom ('no impact on environment', see chapter 5) is satisfied.

The function *iterateWorldStep* uses an if-clause which checks if the degree of at least one node in the environment > 2, i.e., if there is a decision point in the environment. This testing routine is provided by the boolean *checkDegree* function. If the result of the condition is *True*, the *worldStep* function is iterated until the program is terminated within the Sense-Plan-Act cycle (e.g. through the agent reaching the goal). If the condition gives *False*, the program is terminated with an error message before triggering the Sense-Plan-Act cycles.

```
checkDegree :: g e n -> Bool
checkDegree =  any (> 2) . allDegrees
```

Figure 38 shows the hierarchy of sub functions of *iterateWorldStep* that are relevant for proving the wayfinding axioms.

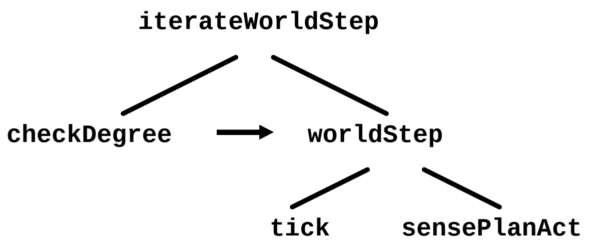


Figure 38: Sub functions of the *iterateWorldStep* function

7.2 The Environment

7.2.1 Nodes

Nodes represent places and decision points in the abstract environment. They are identified with an integer number:

```
data Node = Node NodeId

type NodeId = Int
```

7.2.2 Edges

An edge in the environment is constructed with the parameterized constructor function *E* (see section 4.5.4.3). An edge consists of a startnode with signpost, and the endnode with signpost.

```
data E n = E n SignPost n SignPost
```

A signpost contains metric information (the direction of the signpost) and semantic information (that is matched with concepts in the cognitive map). The direction of the signpost is given as an integer value. To express the case where no signpost is attached to an edge (see Figure 2, section 2.1.3), an alternative with the 0-ary constructor function *NoSign* is added to the data type.

```
data SignPost = S Direction Info | NoSign

type Direction = Int
```

The semantic information of a signpost (Figure 39) consists either of a gatesign (in the airport environment) or a text string (in the WWW). These two alternatives are denoted as the constructors *InfoR* and *InfoW* in the union data type *Info*. The data type *Info* is also used within further objects of agent and environment, thus, these objects have disjoint parts (one for the real world and one for the WWW). This affects among others the agent's cognitive map, mental position, mental goal, or decision.

```
data Info = Ir InfoR | Iw InfoW | NoInfo

type InfoR = GateSign

type InfoW = Text

type Text = String
```

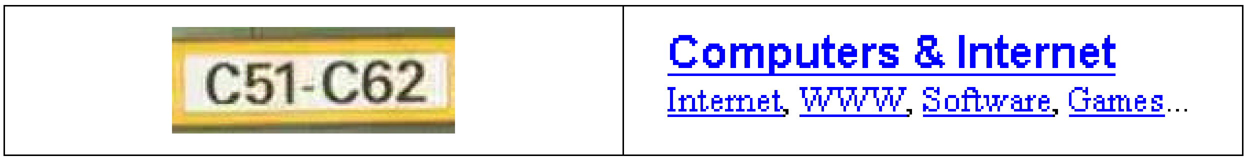


Figure 39: Semantic information of airport sign and hyperlink

The distinction of gatesigns into three types is specified through the data type *GateSign*. Corresponding to section 6.2.2, the components of its alternatives are the data types *GateSignSingle*, *GateSignList*, or *GateSignRange* (see also Raubal 2001b).

```
data GateSign = GateSign GateSignSingle | GateSign1 GateSignList | GateSign2
GateSignRange
```

7.2.3 Graphs

The environments are abstracted as graphs that consist of a list of edges (see sections 2.1.4 and 2.2.6). Using a polymorph constructor function *G* a graph is formalized as

```
data G e n = G [e n]
```

We use a data type synonym for the environment, instantiating the parameterized data type *e* with the constructor function *E*, and instantiating the parameter *n* with the data type *Node*.

```
type Environment = G E Node
```

In the visualization of the data type hierarchy of the environment (Figure 40), components of data types are visualized within dashed boxes, whereas 0-ary data type constructors are printed as text only.

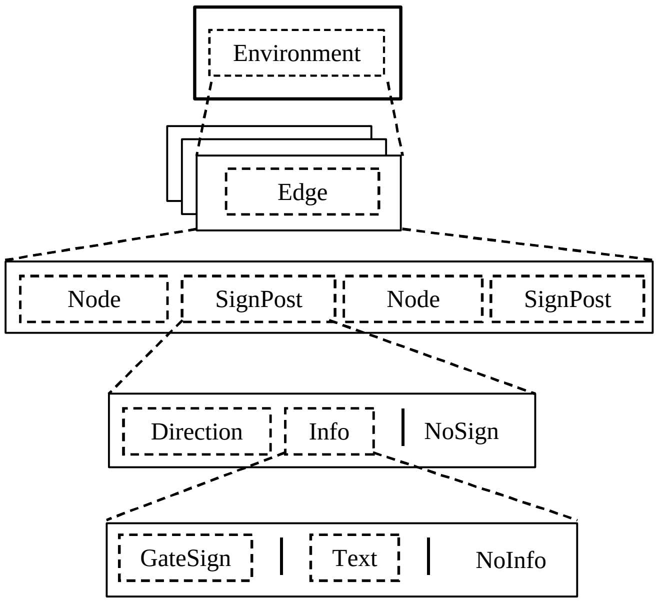


Figure 40: Structure of the data type *Environment*

7.3 Agent Structure

The structure of the formalized agent corresponds to the conceptual model discussed in section 6.1.

7.3.1 Fact and Beliefs

The data type hierarchy reflects the two-tiered conceptual model which separates fact and beliefs (Figure 41). The facts (*Fact*) consist of the agent's id (*AgentId*), position (*Pos*), and previous position (*PrevPos*). The beliefs (*Beliefs*) are mental position (*MentalPos*), cognitive map (*CognitiveMap*), percepts (*Perceived*), decision (*Decision*), preferences (*Preferences*), and incoming direction (*IncomingDir*).

```
data Agent = Agent Fact Beliefs
data Fact = Fact AgentId Pos PrevPos

data Beliefs = Beliefs MentalPos CognitiveMap Perceived Decision

                  Preferences IncomingDir
```

The data type synonym *AgentId* is an integer number, the data type *Pos* contains the type *Node* (section 7.2.1), and the data type *PrevPos* is a type synonym for *Pos*.

```
type AgentId = Int

data Pos = Pos Node

type PrevPos = Pos
```

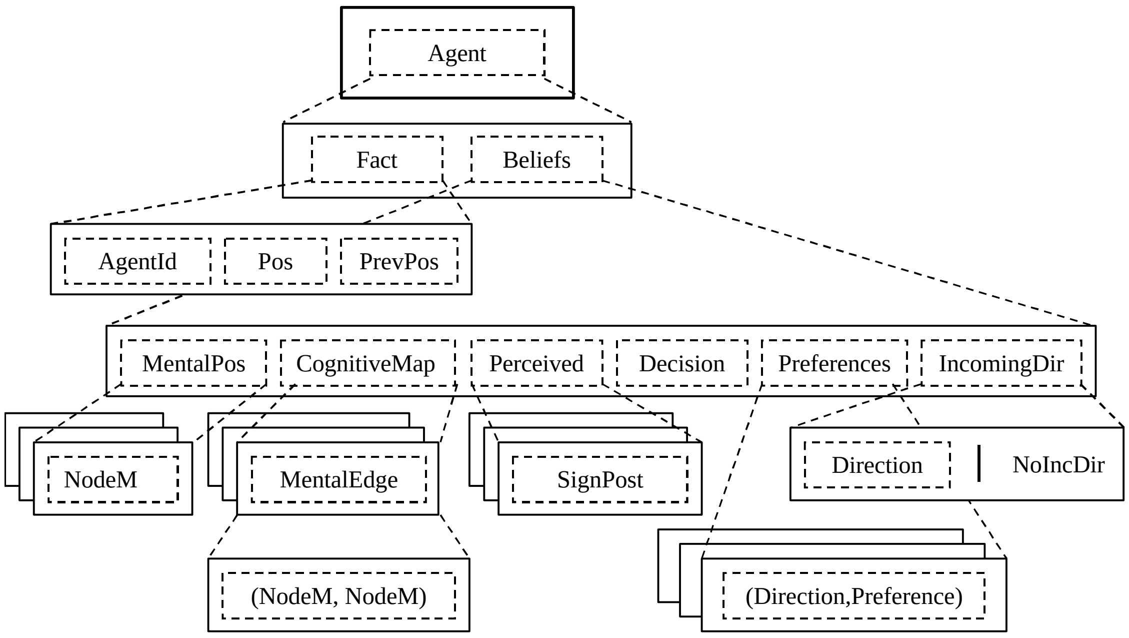


Figure 41: Structure of the data type *Agent*

7.3.2 Mental Position

The mental position (*MentalPos*) consists of none or exactly one concept for each of the graphs in the cognitive map, thus is abstracted as a list of mental nodes (*[NodeM]*). Hence, in a cognitive map that is abstracted as one graph, the mental position is defined through one single mental node, whereas in more complex cognitive maps, a list of several nodes describe the mental position. At the beginning of the wayfinding process the mental position is an empty list. The data type *NodeM* is a type synonym with the data type *Info* (see section 7.2.2), thus, contains disjoint alternatives.

```
type MentalPos = [NodeM]

type NodeM = Info
```

7.3.3 Cognitive Map

The data type structure for the agent (Figure 41) shows that mental position and cognitive map (including the mental goal) are separated, which is a condition for the second wayfinding axiom (‘the agent has a goal’). The cognitive map is abstracted as a list of graphs (section 6.2.1). Each of these graphs consists of a list of mental edges that are defined through the polymorph constructor function *EM*. A mental edge connects two arbitrary mental nodes *n*. In the type synonym *CognitiveMap*, the parameter *n* is instantiated with the type *NodeM*.

```
type CognitiveMap = [G EM NodeM]

data EM n = EM n n
```

7.3.4 Perception

The agent’s percepts are formalized through the data type *Perceive*. As the agent perceives signposts at a decision situation, the percepts are abstracted as a list of signposts (section 7.2.2).

```
type Perceived = [SignPost]
```

7.3.5 Decision

The data type *Decision* is used for the result of the decision function. In our model, a decision is defined by the semantic information of the signpost, which the agent chooses for the next step at a decision point. After a wayfinding step, the decision is reset and represented as the 0-ary constructor function (*NoInfo*). The data type *Decision* is a type synonym for the data type *Info*.

```
type Decision = Info
```

7.3.6 Preferences

The preferences evaluate the metric direction of signposts (section 6.3). They are given as a list of tuples, each consisting of direction and the corresponding preference value.

```
type Preferences = [(Direction,Preference)]

type Preference = Int
```

7.3.7 Incoming Direction

The incoming direction is represented through the data type *IncomingDir* (see section 4.7.5). It contains disjoint alternatives for the airport navigation and the WWW navigation.

```
data IncomingDir = IDir Direction | NoIncDir
```

7.4 External Operations

7.4.1 Class Definition and Derived Functions

The operations within a Sense-Plan-Act framework (sections 3.3.4 and 3.3.5) can be divided into external and internal operations. This distinction is reflected by the used class structure.

External operations are represented in the class *ExternalOps*. In its class header, the class includes two parameters, representing environment and agent. The class contains the functions *perceive* and *act,* which take the parameterized agent and environment as input and result in an agent with changed fact and beliefs. The *sensePlanAct* operation applies function composition to *perceive*, *decide* (an internal operation, see section 7.5), and *act*. The three functions are polymorph, and the semantics hereby is derived from other functions. The class context lists conditions for its parameters *env* and *agent*, thus the parameters inherit their behavior from other classes.

```
class (AgentPut Perceived agent, AgentPut Pos agent, AgentPut Fact agent,
AgentPut Int agent, AgentPut IncomingDir agent, AgentPut MentalPos agent,
EnvAgent env, InternalOps agent )
=> ExternalOps env agent where

perceive :: env -> agent -> agent
act :: env -> agent -> agent
sensePlanAct :: env -> agent -> agent

 -- derived functions
perceive env agent = putToAgent (perceiveAtPos env (getPos agent)) agent
act env a = putToAgent (setToZero (getDec a)) a5 where
   a5 = putToAgent (setToZero (getPerc a)) a4
   a4 = putToAgent (dirNext env (getPos a3) (getPrev a3) (getIncDir a3)) a3
   a3 = putToAgent (updatePosition env (getDec a2) (getPos a2)) a2
   a2 = putToAgent (updatePrev (getFact a1)) a1
   a1 = putToAgent (updateMentalPos (getPosM a) (getDec a) (getCM a)) a
sensePlanAct env = (act env) . decide . (perceive env)
```

7.4.2 The Class Context

The class context lists several classes that contain operations needed to execute the *perceive*, *act*, and *sensePlanAct* function.

The class *AgentPut* contains a function *putToAgent* that replaces a component of the agent, and therefore contains two parameters in its class header.

```
class AgentPut p agent where
  putToAgent :: p -> agent -> agent
```

The class *AgentGet* (which is found within the context of the class *InternalOps*, see section 7.5) contains *get*-functions that are used to navigate the agent's hierarchical data type structure. They allow us to access components of the agent. For example, *getCM* returns the agent's cognitive map:

```
class AgentGet agent where
  getCM :: agent -> CognitiveMap
```

The class *EnvAgent* contains operations that take the environment and some of the agent's components as input: The *perceiveAtPos* function results in all signposts that the agent can see from his actual position. The *dirNext* function gives the agent's incoming direction in the local reference frame if the agent enters a node; and *updatePosition* converts the agent's decision result into a movement to the next node.

```
class EnvAgent env where
  perceiveAtPos :: env -> Pos -> Perceived
  dirNext :: env -> Pos -> PrevPos -> IncomingDir -> IncomingDir
  updatePosition :: env -> Decision -> Pos -> Pos


instance EnvAgent Environment where
updatePosition env dec pos = sndNode p2 (edgeWithSignAndNode dec p2 env)
  where edgeWithSignAndNode dec p2 env =
            head ([e | e <- containNode pos env, containInfo e pos dec])
...
```

Besides the signatures of these three functions, we give the (simplified) instance for the *updatePosition* function. This instance is necessary for proving the fifth wayfinding axiom ('order of actions'). As proposed in section 5.1.5, the important part of a decision to be transformed into behavior is the *right place*. Within the transformation process, the agent matches a *mental image* of the environment (i.e., the semantic content of a signpost as result of the decision process) with the environment itself. Thus the environment is part of the input parameters (see also section 3.3.5.2). The function *updatePosition* executes the following steps:

- from the edges that are incident with the agent's position (function *containNode*) take the edge that contains a signpost with the information of the decision process (*containInfo*)
- get the other node (*sndNode*) of this edge for the new position

The instance shows that an action is uniquely defined through its preceding decision (i.e., through the semantic content of the chosen sign) and the agent's position. As a combination of node id and perceived information is unique in an environment, each movement is unique in the wayfinding process and cannot be replaced by another (i.e., the actions are ordered).

Further sub functions in *act* are the *setToZero* function and the *updatePrev* function. The first one resets the agent's percepts and decision after a performed move, whereas the second one replaces the previous position with the actual agent's position in the agent's facts.

7.4.3 The Semantics of the External Operations

The semantics of the external operations is expressed as derived functions within the class definition. Thus, for the airport case and the WWW case, all functions need to be formulated only once (and not instantiated separately).

The *perceive* function calls the *perceiveAtPos* function. This function gets those signs that can be perceived from a node. The principle of the *perceiveAtPos* function is visualized in Figure 42. Within the *perceive* function, the new percepts replace the existing percepts in the agent's beliefs after having reached a new node.

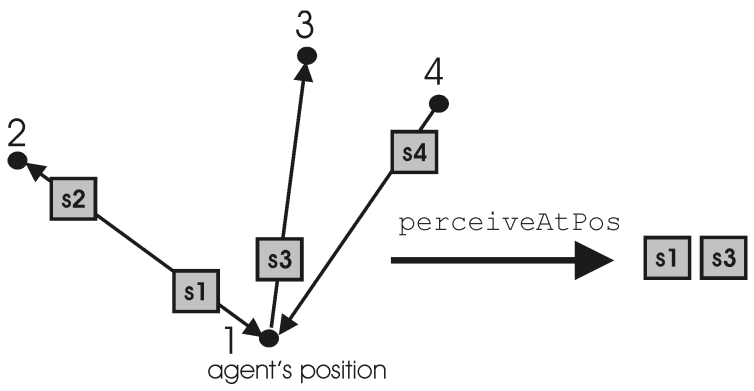


Figure 42: The function *perceiveAtPos*

The *act* function consists of six sub steps that change the agent's facts and beliefs. Some of the sub steps use environment and agent as argument (*dirNext*, *updatePosition*), the other sub steps are internal (*setToZero*, *updateMentalPos*, *updatePrev*) but also part of the agent's move.

(1) Within the *updateMentalPos* function, the agent's mental position is updated corresponding to the semantic information of the sign that is chosen for the next step.

(2) The *updatePrev* function sets the previous position in the agent's facts to the actual position.

(3) The *updatePosition* function changes the agent's position upon the decision (see section 7.4.2).

(4) The *dirNext* function computes the incoming direction of the agent in the local reference frame. For the WWW-navigating agent this function has no effect (see section 6.3).

(5) The percepts are reset using the *setToZero* function.

(6) The decision is reset using the *setToZero* function.

7.5 The Agent's Decision Process: An Internal Operation

The objective of this section is to demonstrate that the computational model satisfies the second axiom ('the agent has a goal') and the third axiom ('moving towards a goal'). We restrict the explanations to those sub steps needed to demonstrate this.

The *decide* function is defined within the class *InternalOps*. It is applied after the *perceive* function. The signature takes one parameter (denoting the agent) as input and output. The class context gives constraints for the parameterized data type *agent*.

```
(AgentGet agent, AgentPut Decision agent)
=> InternalOps agent where
class InternalOps agent where
  decide :: agent -> agent
  decide agent
    = if isAtGoal (getMM agent) (getPerc agent) then error ("GOAL REACHED!")
      else putToAgent (composedDecision (getMM agent) (getPerc agent)
           (getPos agent) (getPref agent) (getIncDir agent)) agent
```

The operation contains an if-clause which checks if the agent has reached his goal (*isAtGoal* function). The goal is reached if mental position and goal correspond. In this case, the simulation is terminated with a message. Otherwise the execution is continued and the *composedDecision* function is called. This function is defined in the class *ComposedDec* and triggers semantic (*semanticDecision*) and metric (*metricDecision*) decision making. The instance of the class *composedDec* shows that it takes some components of the agent's beliefs as input, thus represents an internal operation.

```
instance ComposedDec CognitiveMap Perceived Pos MentalPos Preferences
```

```
          IncomingDir where
    composedDecision cm perc pos mpos pref inc
          = metricDecision pref inc (semanticDecision cm pos mpos perc)
```

As the goal in our case study is semantically defined, the *semantic* decision making process is the interesting part for the proof. Metric decision making describes the agent's bias independently of the goal, therefore we skip its formal discussion. Figure 43 visualizes the so far mentioned sub functions of the *decide* function.

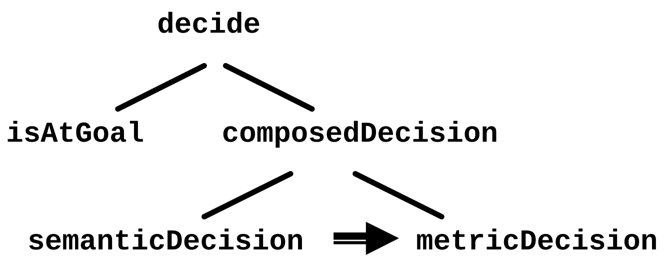


Figure 43: Hierarchy of the *decide* function

7.5.1 Sub-Processes of Semantic Decision Making

Semantic decision making consists of three steps. The term 'filter' in the explanation means to *keep* corresponding elements. The three steps are:

(1) filter those percepts of which the semantic information can be matched with a concept in the cognitive map

(2) exclude those percepts from the decision alternatives that would result in 'shifting up' the mental position in a hierarchical graph of the cognitive map, i.e., moving away from the goal

(3) filter those percepts that have a minimal semantic distance to the corresponding mental goal

Step (2) is the core for the proof of the third axiom ('moving towards a goal'). The implementation of step (3) shows that the goal can be extracted from the cognitive map, and therefore proves the second axiom ('the agent has a goal'). In our discussion we begin with step (2), continue with step (3), and finally show that all three steps together represent the complete semantic decision making process in the formal model.

7.5.2 Moving Towards the Goal

Step (2) of the above given enumeration filters those perceived signposts at a decision point that lead the agent mentally closer to his goal. Thus, taking the path connected to such signpost, decreases the semantic distance between the agent's mental goal and mental position. Further we find situations in environments where recurring signs confirm the agent to be on the correct path. The content of such recurring signs is not semantically closer to the agent's mental goal than the content of previously perceived signs. Rather the distance stays unchanged. Such a sign of constant, unchanged mental distance will then be chosen if other signs are not available. This is the functionality of step (2).

As the abstract cognitive map consists of a list of mental graphs, the mental position is defined through several concepts in the cognitive map—one for each graph (see section 7.3.2). As the mental position is an empty list at the beginning of the wayfinding process, the positions within the mental graphs are initially undefined. During the wayfinding process they are steadily filled through percepts from semantic information of signposts. The sub function *percHasNoMPos* checks if the perceived semantic information has already a position in the corresponding mental graph. If not, choosing a sign with such information for the next step means approaching the goal. The filtering of perceived signs that lead the agent *closer* to the goal is provided by the function *filterDowns*. It either chooses a sign that reduces the semantic distance in one of the mental graphs (through using the *filterDown* function), or it chooses a sign where there is no mental position in the corresponding graph so far (provided by the *percHasNoMPos* function).

The function *filterEquals* filters signs that express the *same* (and not only shorter) semantic distance to the goal compared to the actual mental position. The *filterDownBoth* function calls the *filterEquals* function if the *filterDowns* function gives an empty list (i.e., the agent does not move closer to the target but keeps the semantic distance). We summarize these three essential operations in Table 7:

operation	**semantics**
filterDowns	filter perceived signs that lead the agent closer to the goal
filterEquals	filter perceived signs that maintain semantic distance to goal
filterDownBoth	if non-empty result, call *filterDowns*, else *filterEquals*

Table 7: Functions providing that the agent approaches the goal

```
class SemanticDownFilter cm node sp where

  filterDown :: cm -> [node] -> sp -> Bool

  filterDowns :: cm -> [node] -> [sp] -> [sp]

  filterEqual :: cm -> [node] -> sp -> Bool

  filterEquals :: cm -> [node] -> [sp] -> [sp]

  filterDownBoth :: cm -> [node] -> [sp] -> [sp]
```

```
instance SemanticDownFilter CognitiveMap NodeM SignPost where
filterDown cm (n:ns) sp
   | (goalForNode infoSp cm == goalForNode n cm)
        = mentalDist cm sp < mentalDist cm n
   | otherwise = filterDown cm ns sp
        where infoSp = getInfoSignPost sp
filterDowns cm mpos sps = [s | s <- sps,
        filterDown cm mpos s == True || percHasNoMPos cm s mpos == True]

filterEqual cm (n:ns) sp
   | (goalForNode infoSp cm == goalForNode n cm)
        = mentalDist cm sp == mentalDist cm n
   | otherwise = filterDown cm ns sp
        where infoSp = getInfoSignPost sp
filterEquals cm mpos sps = [s | s <- sps, filterEqual cm mpos s == True]

filterDownBoth cm mpos perc
   | f == [] = filterEquals cm mpos perc
   | otherwise = f
        where f = filterDowns cm mpos perc
```

The *filterDown* and *FilterEqual* functions are recursively defined. The central parts of these functions are the boolean operators ‘<‘ and ‘==‘ applied on the mental distance. The *filterDown* function compares the mental distance (*mentalDist*) between a perceived sign (*sp*) and the goal, with the mental distance between the corresponding concept (*n*) in the semantic position and the goal. The result is *True* (i.e., the perceived signpost provides a smaller semantic distance to the goal than the actual position) if the first of the two values is the smaller one (expressed by the ‘<‘ operation). The *filterEqual* function is equally defined, except for replacing the ‘<‘ function with an equal sign.

7.5.3 The Semantic Distance to the Goal

We now have a closer look at how the mental distance is defined and computed. We show that the goal is gained from the cognitive map and therefore separated from the mental position (as claimed in the second wayfinding axiom). Hereby we look at the class

MentalPath that contains two functions, each with two parameterized data types in its signature.

The mental distance (function *mentalDist*) is defined as the length of the list of nodes (*length* function) that results from calculating the mental shortest path (using the *mentalShortPath* function) between the semantic content of a perceived signpost and the corresponding goal within the cognitive map. The *mentalShortPath* function in its instantiation takes the cognitive map and a perceived signpost as input. It calls the *getPathFromTo* function which calculates the shortest path between two nodes of a graph.

The function *getPathFromTo* uses three arguments: The first argument for the function is the information content *i* of the perceived signpost (which corresponds to a node the cognitive map). The second argument is the goal of the graph that contains the concept *i* (function *getGoalFromGraph*), and the third argument is the graph of the cognitive map where the shortest path is to be found, i.e., the graph that contains the concept *i* (function *graphWithNode*). The *getPathFromTo* function is realized through the shortest path algorithm by Dijkstra (see section 4.8.2).

```
class MentalPath cm sp where
  mentalShortPath :: cm -> sp -> [NodeM]
  mentalDist :: cm -> sp -> Int


instance MentalPath CognitiveMap Info where
  mentalShortPath  cm i = getPathFromTo i (getGoalFromGraph graph) graph
      where graph = graphWithNode i cm
  mentalDist cm i = length (mentalShortPath cm i)
```

Figure 44 shows in a simple example the interplay of the discussed functions for getting the utility of a perceived sign. The percept is the semantic content ‘a3’ of a signpost; the agent’s cognitive map is abstracted through two graphs A and B. Function *graphWithNode* matches the perceived ‘a3’ with the cognitive map and returns that the concept ‘a3’ is member of the graph A. The *getGoalFromGraph* function computes the goal of graph A (i.e., ‘a1’), the *getPathFromTo* function determines the shortest path between the perceived ‘a3’ and the goal ‘a1’, and *mentalDist* counts the number of elements in the resultant list of nodes (which is the value for the semantic distance). These functions together allow the agent to evaluate the utility of each sign perceived at an intersection. The functions show that the goal is calculated from the mental map (*getGoalFromGraph)*, and that therefore—together with the agent’s structure (section 6.1)—the second axiom is satisfied. How the goal is calculated from each graph (i.e., the semantics of the *getGoalFromGraph* function) is not relevant for the axiom.

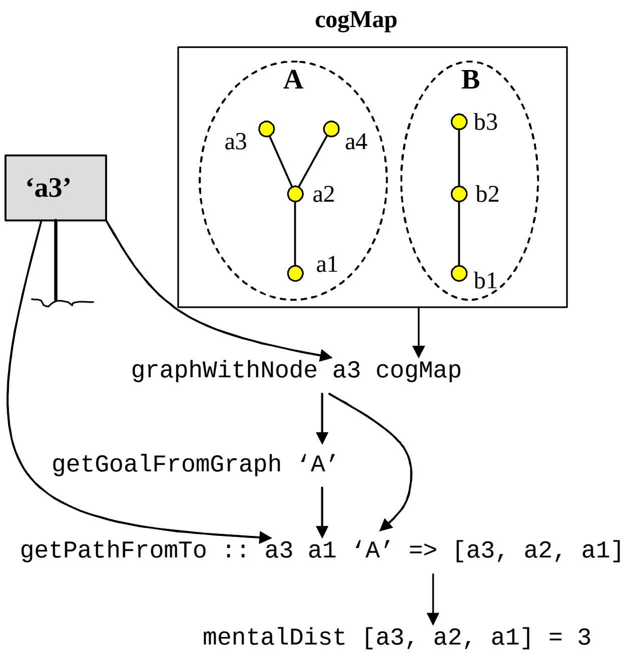

cogMap
A
B
a3
a4
a2
a1
b3
b2
b1
'a3'
graphWithNode a3 cogMap
getGoalFromGraph 'A'
getPathFromTo :: a3 a1 'A' => [a3, a2, a1]
mentalDist [a3, a2, a1] = 3

Figure 44: Computing the shortest mental path in the cognitive map for a perceived sign

7.5.4 Combining the Sub-steps

The function *semanticDecision* in the class *SemanticDec* unites steps (1), (2), and (3) of semantic decision making introduced in section 7.5.1. It takes the agent's cognitive map, mental position, and percepts as input and results in signposts (i.e., percepts) with a minimum semantic distance from their goal. Hereby *filterPercs* represents step (1), *filterDownBoth* represents step (2), and *semanticDistFilter* represents step (3).

```
class SemanticDec cm mpos perc where
   semanticDecision :: cm -> mpos -> perc -> perc
instance SemanticDec CognitiveMap MentalPos Perceived where
   semanticDecision cm mpos perc
   | f == [] = error ("NO SIGN INFORMATION FOUND")
   | otherwise = semanticDistFilter cm f
         where f = filterDownBoth cm mpos (filterPercs cm perc)
```

The *semanticDecision* function has following effects: If the list of signposts that satisfy criteria (1) and (2) is empty (*f == []*…), the program is terminated with an error message. In this case, none of the perceived signposts can be used for further decision making. If the list is not empty, the function *semanticDistFilter* is applied on matched (1) and 'down-filtered' (2) signposts. The semantics of the operations is equal for both instantiations of agent as no pattern matching on the constructor functions of disjoint alternatives is made.

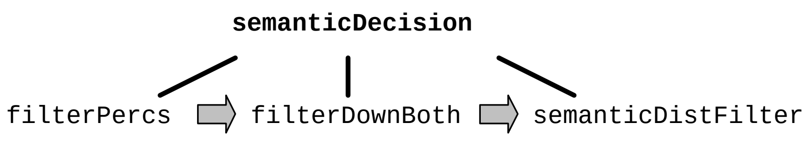


Figure 45: Functions involved in semantic decision making

7.6 Summary

In this section we formalized the conceptual model of a wayfinding agent. This resulted in a set of executable specifications, which will be used in section 9 to simulate the wayfinding process at the Vienna International Airport and in the Yahoo-domain.

We started with the definition of the world with its operations that trigger the agent's wayfinding process and the ticking of the world-time. Then we described the hierarchical structure of the environment that provides semantic and metric information for the agent.

Next we specified the data types for the cognizing agent according to the conceptual model, which distinguishes between fact and beliefs. Based on the Sense-Plan-Act framework and the agent's wayfinding strategies we defined the formal operations of the agent. These operations are applied on the data types introduced before. The agent's operations are hierarchically structured through the used class structure and the sequence of function calls. Differences in agent structure (expressed through union data types) and operations (realized through pattern matching on constructor functions) between both instantiations do not affect the proof that both abstract domains satisfy the wayfinding axioms. The parts needed for the formal proof are homomorph.

Chapter 8 Discussing the Computational Model

In this chapter we summarize the proofs that were given throughout the explanation of the formalized wayfinding model. Further, we review the mapping process between the domains as well as the functions within each domain concerning partiality and totality. Recalling the formalized wayfinding model we discuss which of the formal tools that may be used to express a metaphorical mapping have been implemented.

8.1 Verification of the Wayfinding Axioms

As the semantics of operations given in the functional model is either expressed through derived functions (i.e., independent of their instantiation), or defined for the airport-agent and the WWW-agent (without pattern matching over the constructor functions), proving instantiations to satisfy the axioms could be done in one step for both instantiations.

ad axiom 1 (‘Decision points’): This axiom expresses a condition for the environment with which the agent interacts, namely that the environment must include a node of degree > 2. In section 7.1.2 we have shown that the function *checkDegree* within the *iterateWorldStep* function checks the environment for this condition. If the condition is not satisfied the application will be terminated.

ad axiom 2 (‘The agent has a goal’): The agent’s beliefs separate the components ‘cognitive map’ and ‘mental position’ (section 7.3.1). As the goal is included in the cognitive map (section 7.5.3), the goal is separated from the mental position. Mental position and goal have different values throughout the wayfinding process (as claimed in the axiom). If they correspond, the wayfinding process is terminated (see the *decide* function).

ad axiom 3 (‘Moving Towards a Goal’): We showed that the *filterDownBoth* function (as part of semantic decision making) is applied within each decision process (section 7.5). This function guarantees that the agent chooses only among those perceived signs for the next step that either leads to a *reduction* of the mental distance between mental position and goal, or that—for the case of ‘confirming’ signs—results in an *unchanged* mental distance to the goal (section 7.5.2).

ad axiom 4 (‘No Impact on Environment’): The only operations where the environment is part of the result are the *worldStep* function and the *iterateWorldStep* function. The latter iterates the *worldStep* function. The *worldStep* function triggers both the ticking of the system-time, and the Sense-Plan-Act cycle of the agent (section 7.1.2). Within this function, the environment component is read out from the world (*getEnv world*) and recomposed (*createWorld*) unchanged with the other (changed) components into a new world. As no operation is performed on the environment component, the environment stays unchanged throughout the whole wayfinding process.

ad axiom 5 (‘order of actions’): In section 7.4.2 we discussed the *updatePosition* function (as part of the *act* function) which transforms an action decision into a step. We showed that each of the agent’s movements from one node to the other is *uniquely* defined through the agent’s position (i.e., node) and a sign (i.e., the agent’s decision). Thus, steps are ordered in the simulated wayfinding model and cannot be swapped.

8.2 Totality: Mappings and Functions

8.2.1 Totality of Mapping

We find several mappings in our approach: The wayfinding metaphor maps the semantics of wayfinding in the real world to other (e.g., artificial) domains. Mapping the radial category *wayfinding* to a set of axioms and thereby ‘forgetting’ some features of wayfinding in the real world represents the idea of a *forgetful functor*. The morphism *in* the functional model maps objects and operations between the two instantiations (Figure 22). A function, is said to be *total* if it is defined for all arguments of the appropriate type. The opposite is a *partial* function. What we are interested in, is whether the *morphism* in the functional model is total, i.e., if it maps all functions and operations from one domain to the other. Total functions are easy to handle because they can be combined with other functions without difficulties, whereas partial functions (or mappings) require a condition to check if there is a valid result before the result is used in another function.

The mapping of objects and operations between the two discussed domains is closely related one to each other: A mapped function can only be executed if the required objects are mapped. We discussed the object hierarchy of the world (section 7.1.1), the environment (7.2.3), and the agent (section 7.3). The data type hierarchy uses the same components in its upper levels for both instances (expressed through a *product* data type). Examples are the data types *Agent*, *Beliefs*, or *World*. All operations in classes that are instantiated with such data types are expressed through one single equation for both types of instances. Thus, the operations and objects of this hierarchical level are homomorph between the two domains, and the mapping is *total*. The mapping between instances of parameterized data types (e.g., the data types *E n* or *G e n*, section 7.2) is also total.

Union data types (section 4.5.2) were used to express disjoint unions between elements of a low hierarchical level. Examples are the data types *IncomingDir* or *Info*. The mapping of one alternative to the other is total on this level (with some potential loss of information, section 4.2.2). For operations that are accessing complete alternatives of a union data type (i.e., no single components of the alternatives), the instantiation of the operation is defined through pattern matching on the constructor of each alternative. Thus, the equations are also totally mapped between both instances. We must be aware of the fact that a total mapping function does not tell us anything about the *semantics* of the mapped function itself (except for using the same data type signature).

The mapping of functions that operate on *components* of disjoint alternatives is *partial*. For example, the *getGateLetter* function, which accesses a letter from the semantic information of an airport sign (the semantic information is represented by a union data type), is not defined for the Web environment, as the data type *Gate* is not part of the semantic information in WWW environment.

```
getGateLetter (Gate l n) = l
```

Table 8 summarizes the schema of partial and total mapping of data types and their functions. The first row expresses the case where a data type is equally defined for both instances of agent or environment (total mapping), the second row shows operations on a union data type (total mapping), and the third row describes the partial mapping of components of disjoint alternatives (and corresponding operations).

data type	mapping	function	mapping
`data A = A...`	`id`	`f a = ...`	`id`

`data C = C1...| C2...`	`C1...=> C2...`	`f(C1...) =`
		`f(C2...) =`
			`f(C1...)=>f(C2...)`

`data T = A a1 a2 a3 | B b1 b2`
`for a1 = b1, a2 /= b2`
	`a1 => b1`
	`a2 => b2`
	`a3: no mapping`
		`f1 a1,f2 a2, f3 a3,f4 b1, f5 b2`	`f1 a1 => f4 b1`
			`f2 a2, f3 a3: no mapping`

Table 8: Total and partial mapping of components and corresponding functions

8.2.2 Totality of Functions

The data type *Maybe* (defined within the Haskell prelude) is to provide a method of dealing with illegal or optional values without terminating the program, as would happen if *error* were used, i.e., *Maybe* allows to make a function *total*. A correct result is encapsulated by wrapping it in *Just*; an incorrect result is returned as *Nothing*. This data type is used in the computational model to read out the utility of a metric direction from the preference-direction pairs (section 7.3.6), i.e., for the case that a perceived direction of a sign is not included in the list of direction-utility-pairs (which is not the case in the data sets of the simulation).

```
data Maybe a = Nothing | Just a
```

The totality of all functions in the computational model can be checked through hierarchical deriving of the totality in bottom-up direction (we do not show this for our model here). One needs to start with checking the totality of basic Haskell functions that are used in user defined functions (e.g., *length* is total, whereas *head* is not). If the basic functions used within user-defined functions are partial, the user defined functions must provide an if-clause for input values that may cause an error within the basic function. This is possible through *pattern matching* (section 4.7.5). This concept is to be continued until the top-most function is checked to be total.

We use pattern matching, among other purposes, to define recursive functions with lists as input. An extra equation hereby treats the case of an empty list as input which would lead to a runtime error. Due to *sequential* pattern matching in Haskell, the equations for ‘special’ values must be placed before the general cases. An example for a recursive function that uses pattern matching is the *matchGoal* function. It checks if the information (*getInfoSignPost*) in any of the agent’s percepts (*p:ps*) is equal to the mental goal (*g*). Through the first line (and under the assumption that the applied sub functions are total, too) the *matchGoal* function is made total.

```
matchGoal g [] = False
matchGoal g (p:ps)
   | g == getInfoSignPost p = True
   | otherwise = matchGoal g ps
```

8.3 Comparing Theory and Implementation

Functors, i.e., the lifting of functions between categories, play a role in the theory of metaphors (section 4.4.2). In section 4.5.5 we gave formal examples implemented in Haskell. Analyzing the functional model we see that neither functors nor natural transformations play an important role in the proof of both instantiations satisfying the wayfinding axioms, or in the discussion of mapping of semantics between the two instantiations. Few functors can be found in the computational model as the actual code is more concrete than necessary. This means that more data types (theoretically) could have been defined through *parameterized* constructor functions (and therefore represent functors). The two reasons why we did not do so are: First, a fully parameterized code leads to a higher complexity of the representation of structures and corresponding operations. Proving the wayfinding axioms for two separate instantiations—one of the main goals of the thesis—would be hidden by the syntax of the generalization mechanisms. Second, the type inference system of the Haskell compiler has some limits in coping with generality.

The formal model does not give a functor- and category-based description of morphisms between two domains. Instead we made explicit use of polymorph functions in class declarations (derived functions), and instantiated operations that use one equation for both instances. We also used the class context to express type inheritance between parameterized data types—a step towards a fully parameterized type system. For proving the axioms, we made use of the formal homomorphism between the two domains so that proving did not need to be achieved separately for both instances.

8.4 Summary

We recalled the proofs that the wayfinding axioms are satisfied for both instantiations of agent and environment. Those operations and objects that are needed for the proofs are totally mapped between the two abstract domains. The mapping of operations which is defined on components of disjoint unions of an agent is only partial and does not contribute to the proofs given.

We also discussed partiality and totality on the level of operations. Haskell provides methods to make partial operations total, the most frequently used being pattern matching. When recalling the implemented formal methods, we saw that polymorphism and the class structure play a dominant role in the formal description of the metaphor, whereas the concept of functor is not found and therefore does not contribute to the formal description of a metaphor in our formal model. This task can be considered as part of the future work.

CHAPTER 9 Simulation of the Wayfinding Process in an Airport Environment and the WWW

In this section we test the formalized wayfinding model for being executable. The framework of the test situation is given through the cases studies introduced in section 2. We start with the construction of the simulated environments—parts of the Vienna International Airport and the Yahoo-Portal—and continue with the definition of the wayfinding agents. The abstract environments lack completeness of the real settings. Despite this, the data sets are extensive enough to demonstrate the agents' wayfinding behavior. For a better understanding of the wayfinding strategy we will analyze some decision situations in detail.

Due to the structural commonalities between objects of both instantiations, we use the same set of operations for creating the abstract world, environment, and agent in each of the instantiations. We use data sets that make the agent reach his goal (which is not crucial for the definition of 'wayfinding').

9.1 The Environments

The airport environment as well as the WWW environment are abstracted as static graphs with non-labeled, directed or undirected edges (section 2.3). The graphs are identical in their structure except for different data types denoting the semantic information on signposts (section 7.2.2).

9.1.1 Area of the Simulated Airport Environment

In section 2.2.6, we introduced the structure of the abstract airport environment and listed the semantic and metric properties of the included edges (Table 1). For creating the abstract graph in Haskell, the raw data of the table are represented as a list of 4-tuples (named *realStrings*), where each 4-tuple denotes an edge of the environment. A 4-tuple describes the id of the start node of the edge (data type *Int*), the signpost at this node (data type *SignPost*), the id of the end node, and the signpost at the end node. Directed edges (in the sense of being potentially entered from one side only) have a value 'NoSign' instead of one of their signposts.

The *makeE* function takes a single 4-tuple of the *realStrings* list and converts it into an edge; the *insertG* function inserts a list into a graph. The function *realGraph* represents the complete graph:

```
realGraph = foldr insertG (G []) (map makeE realStrings)


realStrings =

[(0,NoSign,1,NoSign),

(1,S 1 (Ir (GateSign2 (GateSignRange (LetterOnly ‘A’) (LetterOnly
‘D’)))),2,NoSign),

(2,S 0 (Ir (GateSign1 (GateSignList [(LetterOnly ‘A’),(LetterOnly
‘B’),(LetterOnly ‘C’),(LetterOnly ‘D’)]))),3, NoSign),

(3,S 1 (Ir (GateSign (GateSignSingle (LetterOnly ‘A’)))),4,NoSign),

(3,S 0 (Ir (GateSign1 (GateSignList [(LetterOnly ‘A’),(LetterOnly
‘C’)]))),5,NoSign),

(3,S 7 (Ir (GateSign1 (GateSignList [(LetterOnly ‘B’),(LetterOnly
‘C’)]))),6,NoSign),

(4,S 6 (Ir (GateSign (GateSignSingle (LetterOnly ‘C’)))),5, S 2 (Ir (GateSign
(GateSignSingle (LetterOnly ‘A’))))),

(5,S 6 (Ir (GateSign1 (GateSignList [(LetterOnly ‘B’),(LetterOnly ‘C’)]))),6,S
2 (Ir (GateSign (GateSignSingle (LetterOnly ‘A’))))),

(6,S 6 (Ir (GateSign1 (GateSignList [(LetterOnly ‘B’),(LetterOnly ‘C’)]))),7,
NoSign),

(7,S 6 (Ir (GateSign2 (GateSignRange1 (Gate ‘C’ 51) (Gate ‘C’
62)))),9,NoSign),

(7,S 5 (Ir (GateSign (GateSignSingle (LetterOnly ‘B’)))),8,NoSign),

(9,S 7 (Ir (GateSign (GateSignSingle1 (Gate ‘C’ 54)))),10,NoSign) ]
```

For a better understanding of how we use the local reference frame, we visualize the edge that is part of the given data set and printed in bold letters (see Figure 46). The edge has two nodes with the ids 4 and 5 (first and third component). The second component indicates a sign ‘C’ in direction 6 of the local reference frame of node 4, and the fourth component denotes a sign ‘A’ in the direction 2 of the local reference frame of node 5. Thus, the semantic and metric information of the edge is expressed within these four components.

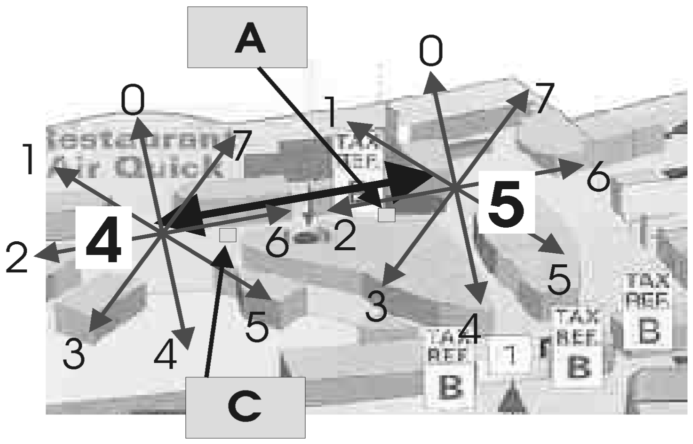


Figure 46: Information of an edge

9.1.2 Area of the Simulated WWW Environment

In section 2.2.6 we introduced the abstracted test area of the Yahoo domain which is represented as a graph with nodes and edges. We create the WWW graph similarly to the airport graph (section 9.1.1); the raw data (Table 2) are also given as 4-tuples (*webStrings*). The metric information of a signpost denotes the distance of the link to the upper margin of the user interface (section 2.2.5), i.e., the metric position of a link is absolutely defined and independent of a local reference frame. The complete graph is computed in the function *wwwGraph*.

```
wwwGraph = foldr insertG (G []) (map makeE webStrings)


webStrings =
[(1,S 2 (Iw "do business"),2,S 1 (Iw "Home")),
(1,S 9 (Iw "recreate"),3,S 1 (Iw "Home")),
(2,S 2 (Iw "do shopping"),4,S 1 (Iw "do business")),
(3,S 19 (Iw "do sport"),5,S 3 (Iw "recreate")),
...
(17,S 1 (Iw "confirm"),25, NoSign),
(18,S 1 (Iw "confirm"),22, NoSign ),
(19,S 1 (Iw "confirm"),23, NoSign),
(20,S 1 (Iw "confirm"), 24, NoSign) ]
```

9.2 The Agents

9.2.1 Creating the Agent’s Cognitive Map

An agent’s cognitive map contains a list of mental graphs. Each of these graphs consists of a list of mental edges with two mental nodes (section 7.3.3). As the data type of a mental node (*NodeM*) is a synonym for the data type *Info* (section 7.2.2), a mental node represents a gatesign for the airport navigating agent (Figure 33, section 6.2.2), and a text string for the WWW navigating agent (Figure 35, section 6.2.3.6). The raw data of a graph are given as tuples, consisting of either gatesigns (airport case) or text strings (WWW case). The function *makeEM* converts a tuple into a mental edge.

For the airport case, the complete graph is computed within the function *graphSigns*. As the cognitive map consists of one graph only, the list representing the cognitive map (*cmapR*) contains one element (i.e., *graphSigns*). The content of the cognitive map corresponds to Figure 33b (section 6.2.2).

```
cmapR :: CognitiveMap

cmapR = [graphSigns]

graphSigns = foldr insertG (G []) (map makeEM graphSignsS)


graphSignsS =

[(Ir (GateSign1 (GateSignList [(LetterOnly ‘A’),(LetterOnly ‘C’)])),
Ir (GateSign1 (GateSignList [(LetterOnly ‘A’),(LetterOnly ‘B’),(LetterOnly
‘C’),(LetterOnly ‘D’)]))),

(Ir (GateSign1 (GateSignList [(LetterOnly ‘B’),(LetterOnly ‘C’)])),
Ir (GateSign1 (GateSignList [(LetterOnly ‘A’),(LetterOnly ‘B’),(LetterOnly
‘C’),(LetterOnly ‘D’)]))),

(Ir (GateSign1 (GateSignList [(LetterOnly ‘A’),(LetterOnly ‘C’)])),
Ir (GateSign2 (GateSignRange (LetterOnly ‘A’) (LetterOnly ‘D’)))),

(Ir (GateSign1 (GateSignList [(LetterOnly ‘B’),(LetterOnly ‘C’)])),
Ir (GateSign2 (GateSignRange (LetterOnly ‘A’) (LetterOnly ‘D’)))),

(Ir (GateSign (GateSignSingle (LetterOnly ‘C’))),
Ir (GateSign1 (GateSignList [(LetterOnly ‘B’),(LetterOnly ‘C’)]))),

(Ir (GateSign (GateSignSingle (LetterOnly ‘C’))),
Ir (GateSign1 (GateSignList [(LetterOnly ‘A’),(LetterOnly ‘C’)]))),

(Ir (GateSign2 (GateSignRange1 (Gate ‘C’ 51) (Gate ‘C’ 62))),
Ir (GateSign (GateSignSingle (LetterOnly ‘C’)))),

(Ir (GateSign2 (GateSignRange1 (Gate ‘C’ 51) (Gate ‘C’ 62))),Ir (GateSign
(GateSignSingle1 (Gate ‘C’ 54))))]
```

Creating a mental graph of the WWW navigating agent is similar to the airport case. The cognitive map consists of five graphs (Figure 34), denoting user intended actions (*graphUia*), physical object hierarchy (*graphPhys*), action affordances (*graphAa*), brand (*graphBrand*),

and size (*graphSize*). The complete cognitive map (see Figure 35) is computed within the *cmapW* function.

```
cmapW :: CognitiveMap

cmapW = [graphUia, graphPhys, graphAa, graphBrand, graphSize]


graphUia = foldr insertG (G []) (map makeEM graphUiaS)

graphPhys = foldr insertG (G []) (map makeEM graphPhysS)

graphAa = foldr insertG (G []) (map makeEM graphAaS)

graphBrand = foldr insertG (G []) (map makeEM graphBrandS)

graphSize = foldr insertG (G []) (map makeEM graphSizeS)


graphUiaS = [(Iw "do shopping",Iw "confirm"), (Iw "do business",Iw "confirm")]

graphAaS = [(Iw "recreate",Iw "do sport" ), (Iw  "do sport",Iw "do track and
field" ), (Iw "do track and field",Iw  "running" )]

graphPhysS = [(Iw "physical object",Iw "artifact"), (Iw "artifact",Iw
"covering" ), (Iw "covering",Iw "clothing" ), (Iw "clothing",Iw "footwear" ),
(Iw "footwear",Iw "shoe" )]

graphBrandS = [(Iw "brand",Iw "Adidas"), (Iw "brand",Iw "Converse"), (Iw
"brand",Iw "Reebok"), (Iw "Adidas",Iw "Nike"), (Iw "Reebok",Iw "Nike"), (Iw
"Converse",Iw "Nike")]

graphSizeS = [ (Iw "size",Iw "10 1/2"),(Iw "10 1/2",Iw "10" ),(Iw "10",Iw "9
1/2")]
```

In both instances, the agent’s goal is not explicitly given but computed from the cognitive map through the *getGoalFromGraph* function (section 7.5.3).

9.2.2 Creating the Agent

This section shows how several components are composed into a whole simulated agent. For the agent’s definition we use the previously defined cognitive maps (section 9.2.1). Union data types for some of the agent’s components indicate disjoint unions between the two agents.

An agent is created through the user-defined data type *Agent* (section 7.3). The components of the agent are filled with the data listed behind the where clause. They represent the agent’s state at the beginning of the navigation process.

The airport-navigating agent (called *fred*) has an id 1, its position is at node 1, the previous position is the fictive node 0, the mental position is empty, his cognitive map (*cmapR*) has been defined in section 9.2.1, the percepts are empty, no decision is yet made (*NoInfo*), preferences correspond to Figure 36a (section 6.3), and the incoming direction is set to 0.

```
fred = Agent (Fact aid (Pos (Node pos)) (Pos (Node prev))) (Beliefs mpos cm
perc dec pref inc)

   where  aid = 1

          pos = 1

          prev = 0

          mpos = []

          cm = cmapR

          perc = []

          dec = NoInfo

          pref = [(1,1),(2,2),(3,4),(4,6),(5,8),(6,7),(7,5),(8,3)]

          inc = IDir 0
```

The wayfinding agent in the WWW (called *charly*) has identical components to the airport-navigating agent except for the cognitive map, preferences, and the incoming direction: The id is set to 1, the position is node 1, the previous position is node 0, the mental position is empty, the cognitive map (*cmapW*) has been defined in section 9.2.1, the percepts are empty, and no decision is yet made (*NoInfo*). The preferences reflect that upper elements on the user interface is given a higher utility (Figure 36b, section 6.3); the incoming direction is undefined (section 4.7.5).

```
charly = Agent (Fact aid (Pos (Node pos)) (Pos (Node prev))) (Beliefs mpos cm
perc dec pref inc)

   where  aid = 1

          pos = 1

          prev = 0

          mpos = []

          cm = cmapW

          perc = []

          dec = NoInfo

          pref =[(1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,8),(9,9),...
          ...(73,73),(74,74),(75,75),(76,76),(77,77),(78,78),(79,79)]

          inc = NoIncDir
```

9.3 The World

In the previous sections we discussed the creation of agent and environment for the simulation. Besides these two components, the system time is part of the world. The world is created through the user-defined data type *World* (section 7.1.1).

The world designed for the airport simulation consists of the real world navigating agent *fred* (section 9.2.2), the airport environment *realGraph* (section 9.1.1), and a timer which is initialized to the value 1.

```
worldAirport = World time agent g
   where  time = 1
          agent = fred
          g = realGraph
```

The world designed for the WWW simulation consists of the WWW navigating agent *charly* (section 9.2.2), the WWW environment *wwwGraph* (section 9.1.2), and a timer which is initialized to the value 1.

```
worldWWW = World time agent g
   where  time = 1
          agent = charly
          g = wwwGraph
```

9.4 Running the Simulation

The simulation of the navigation process is started with the *sim* function. It takes a data type *World* as input and results in the type *IO()* which stands for Input/Output.

```
sim :: World -> IO()
sim world = putStrLn (showTitle ++ concat (map text (iterateWorldStep world)))
```

The *sim* function allows us to modify the output with a formatting function (function *text*) in order to make it more readable for the user. The core of the function is the call of *iterateWorldStep* (section 7.1.2).

9.4.1 Wayfinding in the Airport

In the first simulation we follow the simulated agent's path through the abstract airport environment. We start the simulation with the function *sim* and give it the world *worldAirport* (see section 9.3) as input.

```
> sim worldAirport
TIME: 1
POSITION: 1
PREVNODE: 0
MENTAL POSTION:
INCOMINGDIR:  0
```

```
TIME: 2

POSITION: 2

PREVNODE: 1

MENTAL POSTION:   ‘A’ - ‘D’

INCOMINGDIR:  5


TIME: 3

POSITION: 3

PREVNODE: 2

MENTAL POSTION:   ‘A’  ‘B’  ‘C’  ‘D’

INCOMINGDIR:  4


TIME: 4

POSITION: 5

PREVNODE: 3

MENTAL POSTION:   ‘A’  ‘C’

INCOMINGDIR:  4


TIME: 5

POSITION: 6

PREVNODE: 5

MENTAL POSTION:   ‘B’  ‘C’

INCOMINGDIR:  2


TIME: 6

POSITION: 7

PREVNODE: 6

MENTAL POSTION:   ‘B’  ‘C’

INCOMINGDIR:  2
```

```
TIME: 7

POSITION: 9

PREVNODE: 7

MENTAL POSTION:  ‘C’51 - ‘C’62

INCOMINGDIR:  2


TIME: 8

POSITION: 10

PREVNODE: 9

MENTAL POSTION:  ‘C’54

INCOMINGDIR:  3

program error: AGENT HAS REACHED GOAL !
```

The output shows time, position, previous position, mental position, and the incoming direction of the agent for each step that is made during the wayfinding process. The last line says that the agent has successfully reached his goal, i.e., perceived the sign ‘C 54’ (section 2.1.1). The chosen path during the simulation is visualized in Figure 47 (bold line). The 2-step decision making strategy using semantic decision criteria and metric preferences is successfully applied for the given task.

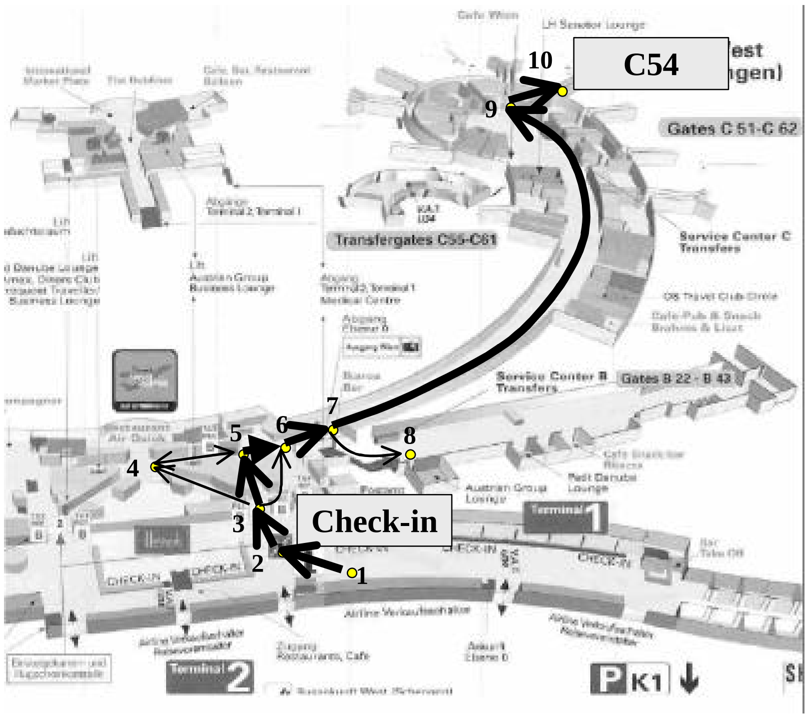


Figure 47: Visited nodes in the airport environment during a complete navigation process

Analyzing the agent’s mental position after each step and comparing it with the agent’s cognitive map (Figure 33b), we see that the mental position either changes towards the mental goal ‘C 54’ within a step (steps 1-2, 3-5, 7-9, 9-10), or keeps the mental distance from the goal (steps 2-3, 5-6, 6-7).

We give an example for both types of steps and start with the decision situation at node 3 in the airport graph (Figure 47). This situation requires the use of metric preferences in addition to semantic decision criteria. The agent’s mental position is at ‘A, B, C, D’ (see the simulation), visualized in Figure 48a in the cognitive map. In Figure 48b the agent’s position in the environment is denoted by a grey circle. The agent’s local reference frame is visualized as a star of arrows—direction 1 pointing to the agent’s front, direction 5 pointing to the agent’s previous position (backwards). The agent perceives the signs ‘A’, ‘A,C’ and ‘B,C’, each from a different direction.

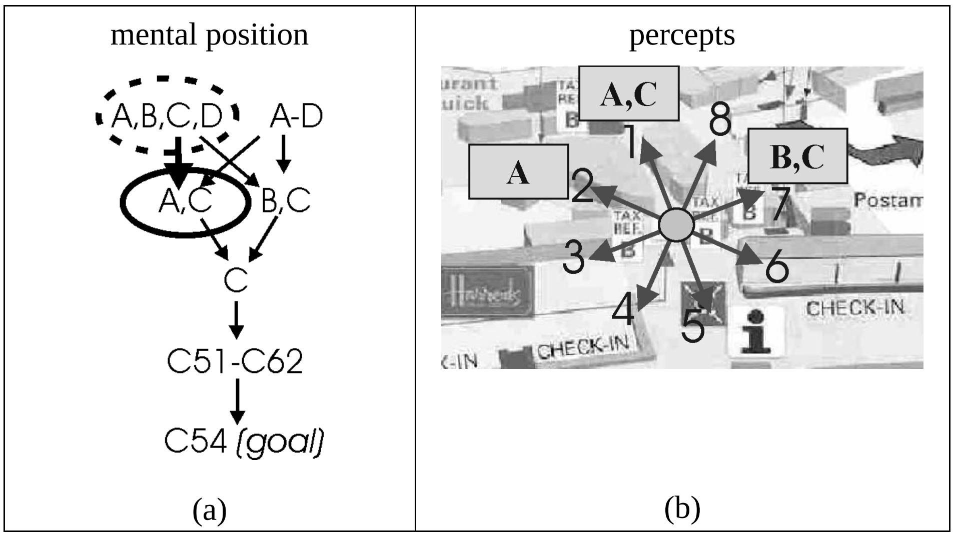


Figure 48: Agent’s mental position (a) and percepts in the agent’s local reference at node 3

The first step of the decision process (semantic decision making) filters the signs which do not lead the agent farer away from his goal (section 7.5.1). Analyzing the agent’s cognitive map (Figure 48a) we see that the signs ‘A,C’ and ‘B,C’ fulfill this criterion—both concepts are three edges distant from the mental goal—whereas the actual mental position is four steps from the goal. The concept ‘A’ is not included in the mental map and therefore the mental distance between ‘A’ and the goal is infinite. Due to the same utility of signs ‘A,C’ and ‘B,C’, metric bias is required for the decision making process. As the front direction 1 (coinciding with sign ‘A’C’) is given the highest preference value, the agent finally chooses the path indicated by the sign ‘A,C’. The mental position changes according to the content of the chosen sign (Figure 48a).

The next case describes the decision situation at node 6. When reaching this node, the agent’s mental position is at ‘B,C’ (Figure 49a). The only sign that can be perceived is ‘B, C’ (Figure 49b). This sign confirms the agent to be on the correct path. Thus, the agent chooses the path with this sign attached. The agent's mental position stays unchanged after the next step.

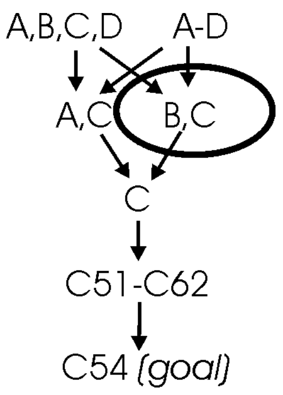
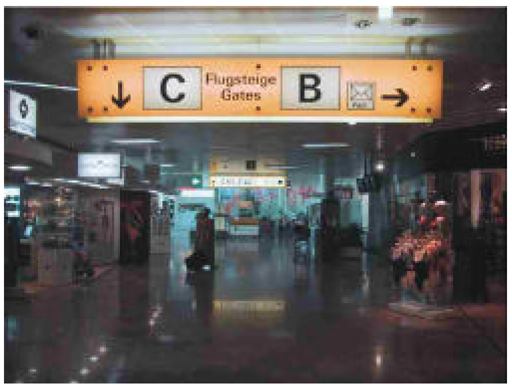
mental position	percepts
 A,B,C,D A-D A,C B,C C C51-C62 C54 *(goal)*	
(a)	(b)

Figure 49: Mental position (a) confirmed through percepts (b) (photo: M. Raubal)

9.4.2 Wayfinding in the WWW

The next test case simulates wayfinding in the Yahoo-domain. The simulation is started with the *sim* function which takes the world *worldWWW* (see section 9.3) as input.

```
> sim worldWWW

TIME: 1

POSITION: 1

PREVNODE: 0

MENTAL POSTION:

INCOMINGDIR:  No IncDir


TIME: 2

POSITION: 2

PREVNODE: 1

MENTAL POSTION:  do business;

INCOMINGDIR:  No IncDir
```

```
TIME: 3

POSITION: 4

PREVNODE: 2

MENTAL POSTION:  do shopping;

INCOMINGDIR:  No IncDir


TIME: 4

POSITION: 6

PREVNODE: 4

MENTAL POSTION:  clothing; do shopping;

INCOMINGDIR:  No IncDir


TIME: 5

POSITION: 11

PREVNODE: 6

MENTAL POSTION:  shoe; do shopping;

INCOMINGDIR:  No IncDir


TIME: 6

POSITION: 15

PREVNODE: 11

MENTAL POSTION:  do sport; shoe; do shopping;

INCOMINGDIR:  No IncDir


TIME: 7

POSITION: 17

PREVNODE: 15

MENTAL POSTION:  running; shoe; do shopping;

INCOMINGDIR:  No IncDir
```

```
TIME: 8

POSITION: 25

PREVNODE: 17

MENTAL POSTION:  running; shoe; confirm;

INCOMINGDIR:  No IncDir

Program error: AGENT HAS REACHED GOAL !
```

As for the airport navigating agent, each step is described through time-slot, position, previous node, mental position, and incoming direction where the incoming direction is undefined at each position (see section 4.7.5). The last line of the result shows that the agent has reached his goal. This means that the agent has perceived the ‘confirm’ link (section 6.2.3.5). The visited nodes during the wayfinding process are visualized in Figure 50.

Corresponding to the simulation of the airport navigating agent, the mental position is empty at the beginning of the wayfinding process. Except for the step 2-4, the mental position changes with *each* step towards the mental goal.

We have a closer look at a decision situation that applies a two-step decision sequence (semantic and metric decision making), similarly to the discussed situation at node 3 in the airport environment.

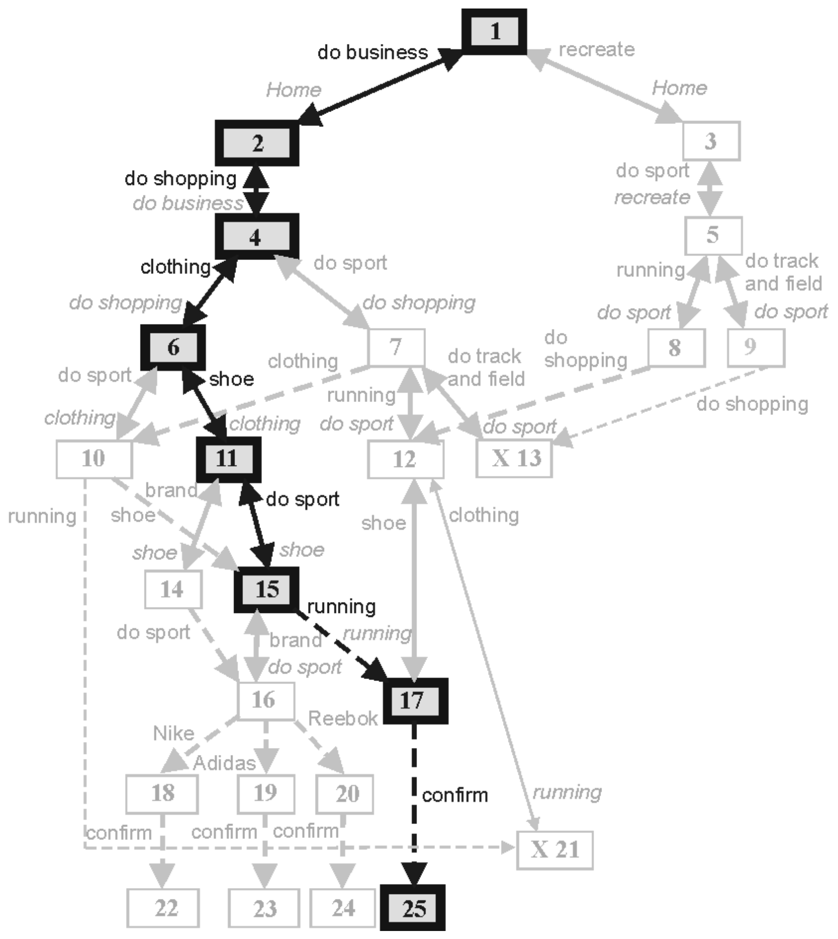
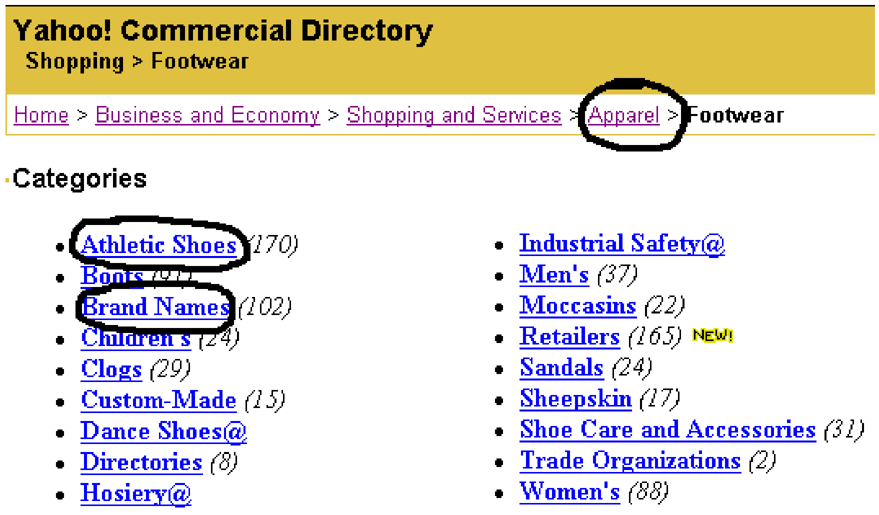


Figure 50: Visited nodes during the wayfinding simulation in the WWW

Let the agent's position be at the web page with the id 11 within the Web environment (Figure 50). The agent's mental position, which consists of the concept 'shoe' for the graph of the physical object hierarchy, and the concept 'do shopping' for the user intended actions is visualized through circles in the cognitive map (Figure 52). Out of the many links offered at node 11 (see the screen shot in Figure 51) we take three links that are related to the agent's goal (uplink 'clothing', and downlinks 'do sport' and 'brand'). We consider 'apparel' and 'clothing' as synonyms, as well as 'athletic shoes' and 'sport shoes'. Among the three links, 'clothing' is top most, and 'brand' is bottom-most on the user interface. The positions of the corresponding concepts in the cognitive map (Figure 52) are visualized as dashed circles.


Yahoo! Commercial Directory
Shopping > Footwear

Home > Business and Economy > Shopping and Services > Apparel > Footwear

Categories

- Athletic Shoes (170)
- Boots (97)
- Brand Names (102)
- Children's (24)
- Clogs (29)
- Custom-Made (15)
- Dance Shoes@
- Directories (8)
- Hosiery@
- Industrial Safety@
- Men's (37)
- Moccasins (22)
- Retailers (165) NEW!
- Sandals (24)
- Sheepskin (17)
- Shoe Care and Accessories (31)
- Trade Organizations (2)
- Women's (88)

Figure 51: Decision alternatives at node 11

During semantic decision making, the percept of the uplink ‘clothing’ is skipped from further considerations, as it would lead the agent away from the goal: The mental position ‘shoe’ in the cognitive map is closer to the target than ‘clothing’. The mental position in the graphs of the other two remaining concepts (‘do sport’ and ‘brand’) has not been defined so far. Thus, these concepts would lead the agent closer to the goal, and are therefore valid candidates for the further decision making process. The cognitive map shows that the semantic distance between each of the two concepts and their corresponding goals (i.e., the concepts ‘running’ and ‘Nike’) amounts to 2 steps, thus metric preference is required. As the link ‘do sport’ is more to the top than the link ‘brand’, the metric criterion gives a unique result, and the link ‘do sport’ is finally chosen for the next step.

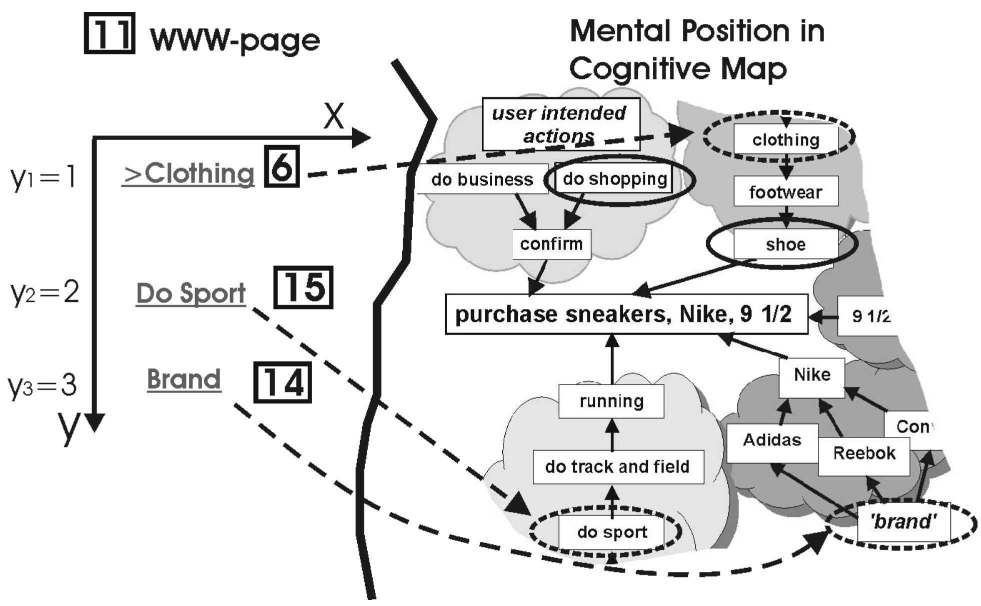


Figure 52: Applying semantic and metric decision making in a two-step decision sequence

9.5 Summary

This section simulated wayfinding at the Vienna International Airport and in the Yahoo-domain. The abstract agents completed the tasks given in the case studies (chapter 2). The simulation is an execution of the formal model that is hereby extended with two different data sets—one for each instantiation. The data sets describe the abstract settings (sections 2.1.4 and 2.2.6), the abstract cognitive maps (sections 6.2.2 and 6.2.3), and the agents' preferences (section 6.3).

After running the simulation, some decision points were analyzed in detail for a better understanding of the applied wayfinding strategies. The results show that—with the chosen data sets and strategy—the agent's mental position during each step either stays in equal distance from the mental goal or approaches the mental goal. We see that one single strategy (a combination of semantic and metric decision making) can be applied for both agents in different environments. The simulation demonstrates that the formal model presented in section 7 is executable.

Chapter 10 Conclusions and Future Work

At the beginning of this section we summarize the research done in this thesis. It describes all the stages we went through for

- defining the semantics of wayfinding
- formalizing the wayfinding metaphor
- simulating wayfinding in the real world and the WWW
- proving that the semantics of wayfinding can be mapped to the Web space.

We then present the results and major findings of our work. Finally, we propose various directions for future research.

10.1 Summary

The goal of the thesis was to find out whether the term ‘wayfinding’ can be used in the WWW. We hereby described the semantics of the term ‘wayfinding’ in its original domain, i.e., in the physical world, through a set of axioms. Within a formalized wayfinding model we could show that both instances—abstract wayfinding in an airport environment and a Web domain—satisfy these axioms. From this we conclude that the term *wayfinding* can be used for the WWW, and moreover in any other domain that fulfills the wayfinding axioms.

The case studies in chapter 2 introduced two specific settings of those domains for which the possibility of mapping the term *wayfinding* could be shown: As representative for the physical world, we explained peculiarities of airport environments and introduced the abstract structure of the Vienna International Airport. As representative of a test area in the WWW-simulation we visualized a part of the abstract directory structure of the Yahoo!-Search Engine-domain and gave a classification of searching methods in the WWW. We defined the agent’s task, looked at the two-layered ontology of both environments, and pointed out further conceptual similarities between the two testing areas. Although the wayfinding metaphor is invariant under the type of wayfinding environment, the two case studies are helpful for the explanation and illustration of the agent’s conceptual wayfinding model.

Describing the mapping of wayfinding semantics into abstract domains is an interdisciplinary endeavor. As a starting point we looked at theories and concepts from different scientific fields (chapter 3). We introduced metaphor theory where we focused especially on how the semantics of a domain is defined and how the semantics is mapped between domains in the various models. When discussing wayfinding theory we analyzed a number of wayfinding definitions in the literature in order to get an idea of the central meaning of the radial category *wayfinding*. For building the conceptual wayfinding model we further looked at cognitive models of space, epistemological models found for wayfinding in the real world and the Web space, and theories of spatial decision making. As we use an agent based approach for the wayfinding simulation, we looked at abstract architectures of agents, the Sense-Plan-Act paradigm, and properties of real and abstract environments.

In chapter 4 we introduced those tools and concepts that are needed to express the semantics of wayfinding and its metaphorical use in a formal manner. Algebraic specifications, type systems, and morphisms are the essential ingredients for a formal description of the wayfinding metaphor. The implementation is made in Haskell, a functional programming language. We used category theory and functors to describe the principle of metaphors rather than as the basis for a formal description within the computational model. Graph theory is needed for the abstraction of the settings chosen for the case studies, and for the description of the agent's wayfinding behavior on an abstract level.

In chapter 5 we gave informal and formal descriptions of the semantics of the term *wayfinding* with the help of a set of axioms. We tried to focus on the central and common meaning of the term—independent of decision strategies and particulars of the environment.

Chapter 6 discussed conceptual features of the wayfinding agent with a focus on the agent's structure and cognitive map, the connection between cognitive map and goal, and the influence of the goal definition on the applied decision making strategy.

In chapter 7 we formalized the conceptual model of wayfinding in two parallel instantiations—one for wayfinding in an airport environment, the other for wayfinding in a Web domain. The formal model reflects the conceptual commonalities between wayfinding in these two environments. The model gives a formal abstraction of a wayfinding agent that follows the Sense-Plan-Act paradigm, uses a two-tiered conceptual fact and beliefs computational model, interacts with the abstract environment, and applies semantic decision criteria combined with metric preferences for wayfinding decisions. With the help of the functional model we demonstrated the mapping of essential objects and operations between the two instantiations to be homomorph.

In chapter 8 we analyzed the formal model. We summarized the proofs that the wayfinding axioms are satisfied for both domains. We showed which parts of a formal model in Haskell are totally mapped between the two instantiations. These parts include functions and operations that behave homomorph and are relevant for the prove of the wayfinding axioms to be satisfied in both instances. We discussed which of the theories introduced in the 'formal tools'-chapter have been used for proving the hypothesis, and which provided theoretical background.

The simulations in chapter 9 demonstrate that the formalized code is executable for both types of agents and environments. Some decision points were discussed in detail to make the applied wayfinding strategies more intelligible to the reader.

10.2 Results and Major Findings

The major result of this thesis is providing a theory to explain why a term does or does not express metaphorical use. The semantics of the term in the source domain hereby is defined through the behavior of involved objects and operations. This behavior which represents minimum requirements of the source domain need to be transferred to the target domain so that the term can be used in a metaphorical sense. For the methodology of proving we suggested a formal approach (Figure 53): The axioms that define the semantics of the term in the source domain are formalized. Further the target-domain needs to be abstracted and checked for satisfying the formalized axioms. If the axioms are satisfied, the term or phrase can be used in the target domain.

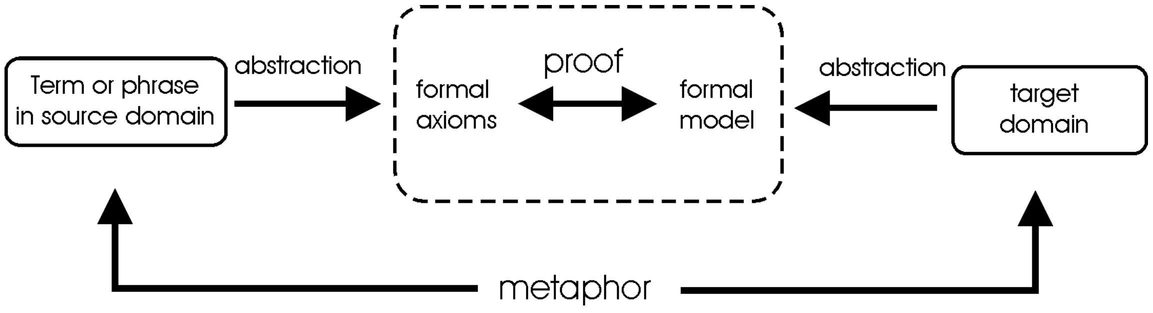


Figure 53: Method of proofing the metaphorical use of a term or phrase

Another major result is the 'product' created in this theses, i.e., a computational wayfinding model with two parallel instantiations. Using the model we could show that the proposed methodology of defining and abstracting the semantics of a term, abstracting source and target domain, and formally proving a set of axioms in both abstract domains, can be achieved. We utilized formal tools such as algebraic specifications, polymorphism, an object oriented style with the help of classes, and function composition as part of category theory.

10.2.1 The Semantics of Wayfinding

We expressed central features of the term *wayfinding* within the following five axioms:

(1) The environment contains decision points.

(2) The wayfinding person has a goal.

(3) The wayfinding person intends to move towards the goal.

(4) Wayfinding has no impact on the state of the environment.

(5) Wayfinding comprises ordered activities.

The axioms reveal that the semantics of wayfinding is independent of following particularities of the wayfinding process and the environment:

- wayfinding strategy
- representation of goal
- grade of familiarity with the environment
- ‘material’ of environment (e.g., physical, virtual, textual, semantic basis)
- reaching the goal
- existence of the goal

This generality of the wayfinding axioms allows mapping of these minimum requirements to other abstract domains.

Using the semantics of the wayfinding axioms we can now explain why the improperly used phrases in the introduction chapter (section 1.4) make problems in their metaphorical use.

ad “Sie findet ihren Weg durchs Leben”: As the phrase does not express that life has a goal (and this cannot be taken as a general assumption), the goal is missing here (fails axiom 2). Thus, the phrase can also not express the person’s intention to reach the (non-existent) goal (fails axiom 3).

ad “Ich finde meinen Weg durch das Turnier”: Here, the environment does not provide decision points (fails axiom 1), as the participant can (usually) not choose between his opponents.

10.2.2 Formalized Wayfinding Model

The formalized agent based model consists of algebraic specifications that allow for describing the structure of abstract data types and their operations. Thus, the algebraic specifications are particularly suited for the representation of change that is an essential part of the conceptual model.

The agent’s operations are separated into external and internal operations which all result in a change of the agent’s state. The model describes equal semantics for parts of both instantiations as the operations are expressed through the same equation for both agent types. These parts behave homomorph and are totally mapped. Disjoint unions between both instantiations are represented as union data types. Functions applied on these parts are totally mapped but may semantically be different one from each other. These differences were shown not to influence the proof that both instantiations satisfy the wayfinding axioms. We think that such an ‘incompleteness’ between both domains involved in a metaphor is typical: If source and target domain were completely equal, no metaphorical mapping would be needed.

The formal wayfinding model showed that wayfinding in the real world and the WWW have several common concepts. When abstracting both domains as formal models and generalizing the wayfinding process independent of particularities of both domains, conceptual similarities were found among others in the decision making process, the interplay of perceived information and knowledge, the definition of the goal, and the sequence of involved sub-processes. The result of the abstraction process allows one to determine those invariants of wayfinding that are not restricted to the physical world but can also be used in the Web space.

Within the formal model we showed how the agent’s epistemology and the ontology of the environment interact, and discussed the role of affordances in both domains. We found out that—due to differences in the physical basis of the two discussed domains—different affordances are offered by the two different types of environments. In the abstract model, these differences disappear (at least the hierarchical level of interest), and thus the affordances, i.e., the representation of objects in the agent’s behavior, are formalized equally as operations in both types of agent.

10.3 Future Work and Open Questions

We have restricted to one metaphor, i.e., the wayfinding metaphor. We proposed a method to define the semantics of the source domain through a set of axioms, and to check if these axioms are satisfied in the target domain. To evaluate the proposed method, future work includes testing this method—i.e., finding a set of axioms—for other metaphors. Then the proposed method may be improved, and potential methodological problems may be detected and discussed. It is of further interest, to find out if all types of metaphors (see section 3.1.2) can be formalized and characterized in the presented way.

In this respect, numerous open questions remain: Can the formalization of concepts in the source-domain be accomplished for both nouns and verbs in the source domain? Take for example the metaphor ‘the family is a nest’. The interpretation of the vehicle-concept ‘nest’ may be difficult to formalize through axioms as it represents static properties rather than operations (‘nest’ is no process but an object). The question that emerges here is if static properties can be expressed as operations and formalized as axioms. Moreover, associations with a vehicle concept may be of qualitative and emotional origin (such as ‘warmth’ or ‘well-being’ for ‘nest’) which may require another approach for their formalization. This question seems to be a relevant point for metaphor theory as physical and subjective experience with the external world is considered an essential part of metaphorical thought (Lakoff and Johnson 1980).

In metaphorical language, two concepts are combined so that they form a new concept. For example, in the metaphor ‘marriage is a nightmare’, both marriage and nightmare acquire a different meaning, where one reflects the nightmarish aspects of marriage and the other reflects the marriage-like quality of a nightmare (meaning unpleasant things that happen to you whether you want them or not). Can this melting process be described on an axiomatic level, similar to rules for blending of conceptual spaces (Fauconnier and Turner 1996; Fauconnier 1997)?

A further question is: Do axioms always have to be completely satisfied in the target domain to give a correct metaphor? Are there exceptions in specific contexts where only a part of the axioms is needed to make correct use of a term? Consider the examples ‘theories are buildings’ (see section 3.1.3) or ‘man is a wolf’. For the latter example, the feature ‘predator’, has to be reinstantiated in the semantic domain of the tenor, i.e., not all parts of being a predator can be mapped, depending on the context of the situation where the metaphor is used. The question (as for most metaphors) is: Which axioms have to be fulfilled in the context of the target domain?

Metaphor theory plays a role in user interface design and human computer interaction (Carroll and Rosson 1994; Goguen 1999). This holds specifically for spatial metaphors (e.g., Dieberger 1994; Sorrows and Hirtle 1999; Fabrikant 2000). Our method shows a direction that may help to clarify if concepts of the daily life are correctly mapped from the physical world to the computer environment, i.e., if their use corresponds to the idea of user interface metaphor. It is an open question and part of future work to determine to which extent the presented approach is useful to support related fields, e.g., the theory of sign systems or landmark theory.

Another challenge for future work is a formal one. We claimed category theory and functors to be useful tools to formulate the semantics of a domain and the mapping process of semantics between two domains. Category theory has a high level of abstractness, and it requires much experience to become familiar with its concepts so that one can use them productively. In this thesis, we presented a categorical point of view several times, and used it once within the axioms. We theoretically discussed potential advantages of categorical theory, but hardly made any use of this theory for proving the hypothesis. A part of future work therefore will be the attempt to utilize the concepts of category theory and functors within various tasks to describe the similarity of domains. For metaphor theory such approach may give more abstract and generalized results than those presented in this thesis.

In order to assess the results of the simulation applied to the case studies, one needs to compare them to the results of human subjects testing in the real world and the WWW—at least for the tasks given in the case studies. The formalized wayfinding model used in this thesis underlies simplifications and specific assumptions. Several human factors that may influence human decision behavior (e.g., emotions or stress) were left out, on the one side, as these factors are extremely difficult to handle in a model, and on the other side, to focus on the basic parts of a decision process. Despite this, human subjects testing may help to check the proposed wayfinding strategy for its actual use in the real world and the WWW, and it may be useful to find out if the utility function gives plausible results.

REFERENCES

Ahuja, R. K., Magnanti, T. L. and Orlin, J. B. (1993). *Network Flows: Theory, Algorithms, and Applications.* Englewood Cliffs, NJ, Prentice Hall.

Allen, G. L. (1999). Spatial Abilities, Cognitive Maps, and Wayfinding. *Wayfinding Behavior: Cognitive Mapping and Other Spatial Processes*. R. G. Golledge (ed.). Baltimore, MD, John Hopkins Press**:** 46-80.

Arthur, P. and Passini, R. (1992). *Wayfinding: People, Signs, and Architecture.* Toronto, McGraw-Hill Ryerson.

Backus, J. (1978). Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *CACM* 21: 613-641.

Baez, J. C. (1999). Higher-Dimensional Algebra and Planck-Scale Physics. *Physics Meets Philosophy at the Planck Scale*. C. Callender and N. Huggett (eds.), Cambridge U. Press.

Bird, R. and de Moor, O. (1997). *Algebra of Programming*. London, Prentice Hall Europe.

Bird, R. and Wadler, P. (1988). *Introduction to Functional Programming*. Exeter, BPC Wheatons Ltd.

Birkhoff, G. (1945). *Universal Algebra*. First Canadian Math. Congress, Toronto University Press.

Bittner, S. (2001). An agent-based model of reality in a cadastre. Institute for Geoinformation Vienna, Austria, Technical University Vienna. PhD Thesis.

Black, M. (1979). More about metaphor. *Metaphor and Thought.* A. Ortony (ed.). New York, Cambridge University Press**:** 19-43.

Black, P. E. and Tanenbaum, P. J. (2001). Dictionary of Algorithms, Data Structures, and Problems. *http://www.nist.gov/dads/*

Blades, M. (1991). Wayfinding Theory and Research: The Need for a New Approach. *Cognitive and Linguistic Aspects of Geographic Space*. D. Mark and A. Frank (eds.). Dordrecht, The Netherlands, Kluwer Academic Publishers. 63**:** 137-165.

Bovy, P. H. L. and Stern, E. (1990). *Route choice: Wayfinding in transport networks*. Dordrecht, Kluwer Academic.

Brooks, R. A. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation* 2: 14-23.

Brooks, R. A. (1991). *Intelligence without reason*. International Joint Conference on Artificial Intelligence.

Car, A. (1996). *Hierarchical Wayfinding - A Model and its Formalization*. ESF - GISDATA Summer Institute, Berlin.

Cardelli, L. (1997). Type Systems. *Handbook of Computer Science and Engineering*. A. B. Tucker (ed.), CRC Press**:** 2208-2236.

Cardelli, L. and Wegner, P. (1985). On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys* 17(4): 471 - 522.

Carroll, J. M., Mack, R. L. and Kellogg, W. A. (1988). Interface Metaphors and User Interface Design. *Handbook of Human-Computer Interaction*. M. Helander (ed.), Elsevier**:** 67-85.

Carroll, J. M. and Rosson, M. B. (1994). *Putting Metaphors to Work*. Graphics Interface ‘94, Banff, Alberta; 18-20 May 1994, Canadian Human-Computer Communications Society.

Casakin, H., Barkowsky, T., Klippel, A. and Freksa, C. (2000). Schematic Maps as Wayfinding Aids. *Spatial Cognition 2*. C. Freksa, W. Brauer and K. F. Wender (eds.). Berlin, Springer**:** 54-71.

Cashell, K. (1998). Attempt to Understand Wittgenstein’s picture of Theory of the Proposition. *Minerva - An Internet Journal of Philosophy* 2.

Chandrasekaran, B., Josephson, J. and Benjamins, R. (1999). What Are Ontologies, and Why Do We Need Them? *IEEE Intelligent Systems* 1: 20-26.

Cornell, E. and Heth, C. (2000). Route learning and wayfinding. *Cognitive Mapping - Past, present and future*. R. Kitchin and S. Freundschuh (eds.). London, Routledge**:** 66-83.

Davis, E. (1990). *Representations of Commonsense Knowledge*. San Mateo, California, Morgan Kaufmann Publishers.

Dieberger, A. (1994). Navigation in Textual Virtual Environments using a City Metaphor. Department for Design and Assessment of Technology Vienna, University of Technology. PhD-Thesis.

Dieberger, A. (1998). Social Connotations of Spatial Metaphors and Their Influence on (Direct) Social Navigation. *Workshop on Personalized and Social Navigation in Information Space*. K. Hook, A. Munro and D. Benyon (eds.). Kista, Sweden, Swedish Institute of Computer Science.

Downs, R. and Stea, D. (1977). *Maps in Minds: Reflections on Cognitive Mapping*. New York, Harper and Row.

Downs, R. M. and Stea, S. (1973). Cognitive maps and spatial behavior: process and product. *Image and Environment*. R. M. Downs and S. Stea (eds.). Chicago, Aldine.

Ehrich, H.-D., Gogolla, M. and Lipeck, U. W. (1989). *Algebraische Spezifikation abstrakter Datentypen*. Stuttgart, B.G. Teubner.

Eilenberg, S. and Lane, S. M. (1945). General theory of natural equivalences. *Transactions of the A.M.S.* 58: 231-294.

Ellis, D. (1989). A Behavioral Model for Information Retrieval System Design. *Journal of Information Science* 15(4/5): 237-247.

Ellis, D. and Haugan, M. (1997). Modelling the Information Seeking Patterns of Engineers and Research Scientists in an Industrial Environment. *Journal of Documentation* 53(4): 384-403.

Fabrikant, S. (2000). Spatial Metaphors for Browsing Large Data Archieves. Department of Geography, University of Colorado. PhD-Thesis.

Fauconnier, G. (1997). *Mappings in Thought and Language*. Cambridge, Cambridge University Press.

Fauconnier, G. and Turner, M. (1996). Blending as a central processof grammar. *Conceptual Structure, Discourse and Language*. A. E. Goldberg (ed.), CSLI.

Fauconnier, G. and Turner, M. (1998). Conceptual integration networks. *Cognitive Science* 22(2): 133-187.

Fellbaum, C. and Miller, G. A. (1990). Folk psychology or semantic entailment? A reply to Rips and Conrad. *The Psychological Review* 97: 565-570.

Ferber, J. (ed.) (1998). *Multi-Agent Systems - An Introduction to Distributed Artificial Intelligence*. J. New York, Addison-Wesley.

Frank, A. U. (1999). *Communication with Maps: A Formalized Model for Communication with Maps Using a Multi-Agent Formalism*. International Workshop on Maps and Diagrammatical Representations of the Environment, Hamburg, Germany.

Frank, A. U. (2000). Spatial Communication with Maps: Defining the Correctness of Maps Using Multi-Agent Simulation. *Spatial Cognition 2*. C. Freksa, W. Brauer and K. F. Wender (eds.). Berlin, Springer. 1849**:** 80-99.

Frank, A. U. (2001). Tiers of ontology and consistency constraints in geographic information systems. *IJGIS* 15(7): 667-678.

Frederickson, G. N. (1987). Fast Algorithms for Shortest Paths in Planar Graphs, With Applications in SIAM. *Journal of Computing* 16(6): 1004-1022.

Freksa, C. (1991). Qualitative Spatial Reasoning. *Cognitive and Linguistic Aspects of Geographic Space*. D. M. Mark and A. U. Frank (eds.). Dordrecht, The Netherlands, Kluwer Academic Press**:** 361-372.

Gärling, T., Böök, A. and Lindberg, E. (1984). Cognitive mapping of a large-scale environment: The interrelationships of action plans, aquisition, and orientation. *Environment and Behavior* 16: 3-34.

Gentner, D. (1983). Structure-Mapping: A Theoretical Framework for Analogy. *Cognitive Science* 7: 155-170.

Gibson, J. (1979). *The Ecological Approach to Visual Perception*. Boston, Houghton Mifflin Company.

Gibson, J. J. (1977). The Theory of Affordances. *J. Bransford*. R. E. Shaw (ed.). Hillsdale, NJ, Lawrence Erlbaum Associates.

Gluck, M. (1991). Making Sense of Human Wayfinding: Review of Cognitive and Linguistic Knowledge for Personal Navigation with a New Research Direction. *Cognitive and Linguistic Aspects of Geographic Space*. D. Mark and A. U. Frank (eds.). Dordecht/Boston/London, Kluwer Academic Publishers**:** 117-135.

Goguen, J. (1999). An Introduction to Algebraic Semiotics, with Application to User Interface Design. *Computation for Metaphor, Analogy and Agents - Lecture Notes in Artificial Intelligence*. C. Nehaniv (ed.), Springer. 1562**:** 242-291.

Goguen, J. (2001). *Towards a Design Theory for Virtual Worlds: Scientific Visualization and Algebraic Semiotics as a Case Study*. Virtual Worlds and Simulation Conference, Phoenix AZ.

Goldberg, B. (1991). *Tag-Free Garbage Collection for Strongly Typed Programming Languages*. ACM SIGPLAN '91 Conference on Programming Language Design and Implementation., Toronto, Canada.

Golledge, R. G. (1995). Path Selection and Route Preference in Human Navigation: A Progress Report. *Spatial Information Theory (COSIT'95)*. A. U. Frank and W. Kuhn (eds.). Berlin, Springer. 988**:** 207-222.

Golledge, R. G. (1999). Human wayfinding and cognitive maps. *Wayfinding Behavior: Cognitive Mapping and Other Spatial Processes*. R. G. Golledge (ed.). Baltimore, MD, John Hopkins Press**:** 5-45.

Golledge, R. G., Jacobson, R. D., Kitchin, R. and Blades, M. (2000). Cognitive Maps, Spatial Abilities, and Human Wayfinding. *Geographical Review of Japan* 73 (Ser. B)(2): 93-104.

Golledge, R. G. and Stimson, R. (1997). *Spatial Behavior: A Geographic Perspective*. New York, Guilford Press.

Gopal, S., Klatzky, R. L. and Smith, T. R. (1989). Navigator: A Psychologically Based Model of Environmental Learning Through Navigation. *Journal of Environmental Psychology* 9: 309-331.

Grady, J., Taub, S. and Morgan, P. (1996). Primitive and compund metaphors. *Conceptual structure, discourse, and language*. A. Goldberg (ed.). Cambridge, England, Cambridge University Press.

greatlook.com (2002). *http://www.greatlook.com/why.html*

Grey, W. (2000). Metaphor and Meaning. *Minerva - An Internet Journal of Philosophy* 4.

Gruber, T. R. (1993). *Toward Principles for the Design of Ontologies Used for Knowledge Sharing*, Knowledge Systems Laboratory, Stanford University

Guarino, N. (1997). Semantic Matching: Formal Ontological Distintions for Information Organization, Extraction, and Integration. *International Summer School, SCIE-97*. M. T. Pazienza (ed.), Springer. 1299**:** 139-170.

Guttag, J. V., Horowitz, E. and Musser, D. R. (1978). Abstract Data Types and Software Validation. *Comm. ACM* 21(12): 1048-1064.

Hirtle, S. (1998). The Cognitive Atlas: Using GIS as a Metaphor for Memory. *Spatial and Temporal Reasoning in Geographic Information Systems*. M. Egenhofer and R. Golledge (eds.), Oxford University Press**:** 267-276.

Hirtle, S. C. and Jonides, J. (1985). Evidence of hierarchies in cognitive maps. *Memory and Cognition* 13: 208-217.

Hochmair, H. and Frank, A. U. (2001). A Semantic Map as Basis for the Decision Process in the www Navigation. *Conference on Spatial Information Theory (COSIT'01)*. D. R. Montello (ed.). Berlin, Springer**:** 173-188.

Hochmair, H. and Raubal, M. (forthcoming). *Topologic and Metric Decision Criteria for Wayfinding in the Real World and the WWW*. SDH.

Höök, K., Benyon, D., Dahlbäck, N., McCall, R., Macaulay, C., Munro, A., Persson, P., Sjölinder, M. and Svensson, M. (1998). Introduction: A Framework for Information Space, Personal and Social Navigation. *Exploring Navigation; Towards a Framework for Design and Evaluation of Navigation in Electronic Spaces*. N. Dahlbäck (ed.). Kista, Sweden, Swedish Institute of Computer Science.

Horebeek, I. V. and Lewi, J. (1989). *Algebraic Specifications in Software Engineering - an introduction*. Berlin, Springer.

Hudak, P. (1989). Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys* 21(3): 359-411.

Janzen, G., Herrmann, T., Katz, S. and Schweizer, K. (2000). Oblique Angles Intersections and Barriers: Navigating through a Virtual Maze. *Spatial Cognition 2*. C. Freksa, W. Brauer and K. F. Wender (eds.). Berlin, Springer. 1849**:** 277-294.

Johnson, D. B. (1977). Efficient Algorithms for Shortest Paths in Sparse Networks. *Journal of the Association for Computing Machinery* 24(1): 1-13.

Johnson, M. (1987). *The Body in the Mind: The Bodily Basis of Meaning, Imagination, and Reason*. Chicago, University of Chicago Press.

Jones, K. S. (1986). *Synonymy and semantic classification*. Edinburgh, Edinburgh University Press.

Jones, S. L. P., Jones, M. P. and Meijer, E. (1997). *Type classes: exploring the design space*. Glasgow, Department of Computing Science, University of Glasgow

Jul, S. and Furnas, G. (1997). Navigation in Electronic Worlds. *SIGCHI* 29(4).

Kirschenhofer, P. (1995). The Mathematical Foundation of Graphs and Topology for GIS. *Geographic Information Systems - Materials for a Post Graduate Course*. A. U. Frank (ed.). Vienna, Department of Geoinformation, TU Vienna. 1**:** 155-176.

Koffka, K. (1935). *Principles of Gestalt Psychology*. New York, Harcourt Brace.

Kuhn, W. (1997). Approaching the Issue of Information Loss in Geographic Data Transfers. *Geographical Systems* 4(3): 261-276.

Kuhn, W. and Frank, A. U. (1991). A Formalization of Metaphors and Image-Schemas in User Interfaces. *Cognitive and Linguistic Aspects of Geographic Space*. D. M. Mark and A. U. Frank (eds.), Kluwer Academic Press**:** 419-434.

Kuipers, B. (1978). Modeling Spatial Knowledge. *Cognitive Science* 2.

Kuipers, B. (1982). The 'Map in the Head' Metaphor. *Environment and Behavior* 14: 202-220.

Kuipers, B., Froom, R., Lee, W. and Pierce, D. (1993). The Semantic Hierarchy in Robot Learning. *Robot Learning*. J. Connell and S. Mahadevan (eds.). Boston, Kluwer Academic Publishers**:** 141-170.

Lakoff, G. (1987). *Women, Fire, and Dangerous Things*. Chicago and London, The University of Chicago Press.

Lakoff, G. and Johnson, M. (1980). *Metaphors we live by*, University of Chicago Press.

Leiser, D. and Zilbershatz, A. (1989). The Traveller—A Computational Model of Spatial Network Learning. *Environment and Behavior* 21(4): 435-463.

Liskov, B. and Guttag, J. (1986). *Abstraction and Specification in Program Development*. Cambridge, Mass., The MIT Press.

Loomis, J. M. and Klatzky, R. L. (1999). Human Navigation by Path Integration. *Wayfinding Behavior: Cognitive Mapping and Other Spatial Processes*. R. G. Golledge (ed.). Baltimore, MD, John Hopkins Press**:** 125-151.

Lynch, K. (1960). *The image of the city*. Cambridge, Mass., MIT Press.

MacCormac, E. R. (1985). Metaphor as a knowledge process. *A Cognitive Theory of Metaphor*. E. R. MacCormac (ed.). Cambridge, MA/London, MIT Press.

MacLane, S. (1971). *Categories for the Working Mathematician*, Springer.

MacLane, S. and Birkhoff, G. (1967). *Algebra*. New York, Macmillan.

Maglio, P. and Matlock, T. (1998). Construcing Social Spaces in Virtual Environments: Mataphors We Surf the Web By. *Workshop on Personalized and Social Navigation in Information Space*. K. Hook, A. Munro and D. Benyon (eds.). Kista, Sweden, Swedish Institute of Computer Science**:** 138-147.

Marchionini, G. M. (1995). *Information Seeking in Electronic Environments*. Cambridge, England, Cambridge University Press.

Mark, D., Freksa, C., Hirtle, S., Lloyd, R. and Tversky, B. (1999). Cognitive models of geographical space. *IJGIS* 13(8): 747-774.

Mark, D. M. and Frank, A. U. (1996). Experiental and Formal Models of Geographic Space. *Environment & Planning B* 23: 3-24.

McCalla, G., Reid, L. and Schneider, P. (1982). Plan Creation, Plan Execution, and Knowledge Acquisition in a Dynamic Microworld. *International Journal of Man-Machine Studies* 16: 89-112.

McCarthy, J. and Hayes, P. J. (1969). Some Philosophical Problems from the Standpoint of Artificial Intelligence. *Machine Intelligence 4*. B. Meltzer and D. Michie (eds.). Edinburgh, Edinburgh University Press**:** 463-502.

Medak, D. (1999). Lifestyles - A Paradigm for the Description of Spatiotemporal Databases. Department of Geoinformation Vienna, Technical University Vienna. PhD Thesis.

Miller, G. (1990). Nouns in WordNet: A lexical Inheritance System. *International Journal of Lexicography* 3(4): 245-264.

Miller, G. A. (1995). WordNet: A Lexical Database for English. *Communications of the ACM* 38(11): 39-41.

Milner, R. (1978). A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences* 17: 348-375.

Montello, D. (1998). A New Framework for Understanding the Acquisition of Spatial Knowledge in Large-Scale Environments. *Spatial and Temporal Reasoning in Geographic Information Systems*. M. Egenhofer and R. Golledge (eds.). New York, Oxford University Press**:** 143-154.

Nilsson, N. J. (1980). *Principles of Artificial Intelligence*. Tioga, Palo Alto.

Norman, D. (1988). *The Design of Everyday Things*. New York, First Doubleday.

Norman, D. (1999). Affordances, Conventions, and Design. *interactions* 6(3).

Nückles, M. and Janetzko, D. (1997). *The role of semantic similarity in the comprehension of metaphor*. Nineteenth Annual Conference of the Cognitive Science Society, Stanford, CA, Erlbaum.

Nwana, H. S. and Ndumu, D. T. (1999). A perspective on software agents research. *The Knowledge Engineering Review* 14(2): 1-18.

O'Neill, M. (1991). A Biologically Based Model of Spatial Cognition and Wayfinding. *Journal of Environmental Psychology*(11): 299-320.

Ortony, A. (1979). Beyond Literal Similarity. *Psychological Review* 86: 161-180.

Pastore, M. (2001). Search Engines, Browsers Still Confusing Many Web Users. *http://cyberatlas.internet.com/big_picture/traffic_patterns/article/0,,5931_588851,00.html*

Peterson, J., Hammond, K., Augustsson, L., Boutel, B., Burton, W., Fasel, J., Gordon, A. D., Hughes, J., Hudak, P., Johnsson, T., Jones, M., Meijer, E., Jones, S. P., Reid, A. and Wadler, P. (1997). The Haskell 1.4 Report. *http://www.haskell.org/report/index.html*.

Piff, M. (1991). *Discrete Mathematics*. Cambridge, Camridge University Press.

Raubal, M. (2001a). Agent-based simulation of human wayfinding in unfamiliar buildings. Institute for Geoinformation Vienna, Technical University. PhD Thesis.

Raubal, M. (2001b). Ontology and Epistemology for Agent-based Wayfinding Simulation. *IJGIS* 15(7): 653-665.

Raubal, M. and Egenhofer, M. (1998). Comparing the complexity of wayfinding tasks in built environments. *Environment & Planning B* 25(6): 895-913.

Raubal, M., Egenhofer, M., Pfoser, D. and Tryfona, N. (1997). Structuring Space with Image Schemata: Wayfinding in Airports as a Case Study. *Spatial Information Theory-A Theoretical Basis for GIS (COSIT'97)*. S. Hirtle and A. Frank (eds.). Berlin, Springer. 1329**:** 85-102.

Remolina, E., Fernandez, J., Kuipers, B. and Gonzalez, J. (1999). Formalizing Regions in the Spatial Semantic Hierarchy: A AH-graphs Implementation Approach. *Spatial Information Theory (COSIT'99)*. C. Freksa and D. Mark (eds.), Springer. 1661**:** 37-50.

Rosch, E. (1978). Principles of categorization. *Cognition and categorization*. E. Rosch and B. B. Lloyd (eds.). Hillsdale, NJ, Erlbaum.

Rosch, E. and Mervis, C. B. (1975). Family resemblance: Studies in the internal structure of categories. *Cognitive Psychology*(7): 573-605.

Russell, S. J. and Norvig, P. (1995). *Artificial Intelligence - A Modern Approach*. London, Prentice-Hall International, Inc.

Scaruffi, P. (2001). Thinking About Thought. *http://www.thymos.com/tat/metaphor.html*

Sheppard, D. and Adams, J. M. (1971). A survey of drivers' opinions on maps for route finding. *The Cartographic Journal* 8: 105-114.

Siegel, A. W. and White, S. H. (1975). The development of spatial representations of large-scale environments. *Advances in child development and behaviour*. H. W. Reese (ed.). 10**:** 9-55.

Smith, B. (2001a). Objects and Their Environments: From Aristotle to Ecological Ontology. *The Life and Motion of Socioeconomic Units*. A. U. Frank, J. Raper and J.-P. Cheylan (eds.). London, Taylor and Francis.

Smith, B. (2001b). Ontology: Philosophical and Computational. *The Blackwell Guide to the Philosophy of Computing and Information*. L. Floridi (ed.). Oxford, Blackwell.

Sorrows, M. E. and Hirtle, S. C. (1999). The Nature of Landmarks for Real and Electronic Spaces. *Spatial Information Theory (COSIT'99)*. C. Freksa and D. Mark (eds.). Berlin, Springer. 1661**:** 37-50.

Sowa, J. (1999). Mathematical Background. *http://www.bestweb.net/~sowa/misc/mathw.htm*

Stern, E. and Leiser, D. (1987). Levels of Spatial Knowledge and Urban Travel Modeling. *Geographical Analysis* 20(2).

Stern, E. and Portugali, J. (1999). Environmental Cognition and Decision Making in Urban Navigation. *Wayfinding Behavior: Cognitive Mapping and Other Spatial Processes*. R. G. Golledge (ed.). Baltimore, MD, John Hopkins Press**:** 99-118.

Sutcliffe, A. (1998). Task-Related Navigation in Information Spaces. *Workshop on Personalized and Social Navigation in Information Space*. K. Hook, A. Munro and D. Benyon (eds.). Kista, Sweden, Swedish Institute of Computer Science**:** 58-65.

Sweetser, E. E. (1990). *From etymology to pragmatics - Metaphorical and cultural aspects of semantic structure*. Cambridge, UK, Cambridge University Press.

Thompson, S. (1996). *Haskell - The Craft of Functional Programming*. Harlow, England, Addison-Wesley.

Tolman, E. V. (1948). Cognitive maps in rats and men. *Psychological Review* 55: 189-208.

Trappl, R., Petta, P. and Payr, S. (eds.) (forthcoming). *Emotions in Humans and Artifacts*. Cambridge, MA, USA, MIT Press.

Tversky, B. (1993). Cognitive maps, cognitive collages, and spatial mental model. *Spatial Information Theory: Theoretical Basis for GIS (COSIT'93).* A. U. Frank and I. Campari (eds.). Berlin, Springer. 716**:** 14-24.

Weiss, G. (ed.) (1999). *Multiagent Systems - A Modern Approach to Distributed Artificial Intelligence*. Cambridge, MA, MIT Press.

whatis.com (2002). *http://searchwebmanagement.techtarget.com/sDefinition/0,,sid27_gci212301,00.html*

Wilson, T. D. (1997). Information Behavior: An Interdisciplinary Perspective. *Information Processing & Management* 33(4): 551-572.

Wooldridge, M. (1999). Intelligent Agents. *Multiagent Systems - A modern Approach to Distributed Artificial Intelligence*. G. Weiss (ed.). Cambridge, Massachusetts, The MIT Press.

Wooldridge, M. and Jennings, N. R. (1995). Intelligent agents: Theory and practice. *The Knowledge Engineering Review* 10(2): 115-152.

APPENDIX

Haskell Syntax

An identifier in Haskell begins with a letter of the alphabet optionally followed by a sequence of characters, each of which is either a letter, a digit, an apostrophe (‘) or an underbar (‘_’). Identifiers representing functions or variables begin with a lower case letter, whereas identifiers beginning with an upper case letter describe a data type or a constructor function.

Readability of Haskell code is improved by the layout-rule—the level of indentation indicates the structure of a program. Non-intended lines represent top levels of a Haskell program. Every indentation shows that the intended line actually continues a previous, less indented line. Equally indented lines share the same level in the structure.

Functions of more than one argument can be defined in two basic styles. Usually such functions are represented in the *curried* form, where they take their arguments one at a time. A function to multiply two integers, e.g., in the *curried* form is written as:

```
multiply :: Int -> Int -> Int

multiply x y = x*y
```

The other form, called *uncurried*, is defined by pairing the arguments:

```
multiplyUC :: (Int, Int) -> Int

multiplyUC (x,y) = x*y
```

The function *curry* (named after Haskell B. Curry) converts an uncurried function into a carried one:

```
curry :: ((a,b) -> c) -> (a -> b -> c)

curry :: f a b = f (a,b)
```

or written in the traditional syntax:

$$curry :: ((A \leftarrow C) \leftarrow B) \leftarrow (A \leftarrow (B \times C))$$

The Set of Specifications for the Wayfinding Model:

World

```
module WorldClass where

import ZeroOne
import Graphs
import AgentClass
import Strings

------------------ Data types --------------------

data World = World Time Agent Environment deriving Show
```

```
type Time = Int

------------------ Create a World -----------

class CreateWorld t a e w where
   createWorld :: t -> a -> e -> w

instance CreateWorld Time Agent Environment World where
   createWorld t a e = World t a e

------------------ Operations in the world ---------------

class (CreateWorld Time Agent Environment world) => WorldClass world where
   getTime :: world -> Time
   getAgent :: world -> Agent
   getEnv :: world -> Environment
   getNumberOfEdges :: world -> Int

   worldStep :: world -> world -- one sensePlanAct circle for agent in the world
   iterateWorldStep :: world -> [world] -- complete world steps

  -- derived functions

   getNumberOfEdges = numberOfEdges.getEnv
   worldStep world = createWorld (tick . getTime $ world)
      (sensePlanAct (getEnv world) (getAgent world)) (getEnv world)
   iterateWorldStep world
     | (checkDegree (getEnv world) == True) =
        take (getNumberOfEdges world) (iterate worldStep world)
     | otherwise = error ("No decision point in the environment")

instance WorldClass World where --(ExternalOps (G E Node) Environment) =>
   getAgent (World t a g) = a
   getTime (World t as g) = t
   getEnv (World t as g) = g

class (WorldClass world) => Analyze world where
   visitedNodes :: world -> [Pos]
   findCycle :: world -> [Pos]  -- find loop for one single agent

   -- axioms
   visitedNodes world = map (getPos . getAgent) (iterateWorldStep world)
        -- lists all positions of a navigating agent throughout the whole process

   findCycle world = findCycle1 (findRepeatingCycle positions) where
       positions = visitedNodes world   -- finds loop during navigation

------------------ Timer -------------

class Timer time where
   tick :: time -> time

instance Timer Time where
   tick t = t + 1


---------------------------- Text formatting and output functions ---------------

instance Strings World where
   text (World time agent world) = "\nTIME: " ++ text time ++ text agent ++ "\n"
   xtext (World time agent world) = putStrLn ("\nTIME: " ++ text time ++ text agent)

showTitle :: String
showTitle = "\n*********************************" ++
                  "\nAGENT-BASED WAYFINDING SIMULATION" ++
                  "\n*********************************\n"

showCycle :: [Pos] -> String
showCycle ps = "\n*********************************" ++
               "\nAgent has been caught in a loop."  ++
               "\n*********************************" ++
               "\nNodes: " ++ text ps ++ "\n"

sim :: World -> IO()
sim world = putStrLn (showTitle ++ concat (map text (iterateWorldStep world))
       ++ showCycle (findCycle world))
```

```
simFile :: World -> String
simFile world = output where
    output = showTitle  ++ concat (map text (iterateWorldStep world))
      ++ showCycle (findCycle world)

------------------------ Subfunctions for the findCycle function -------------------

matches :: Pos -> [Pos] -> [Pos]
matches pos listOfElts = filter (pos==) listOfElts
--this function picks out all occurences of an integer in a list;

findRepeatingCycle :: [Pos] -> [Pos]
findRepeatingCycle (a : as)
   = if (length (matches a as) > 1) then (a : as)
     else findRepeatingCycle as
   --gives a list with the repeating cycle (e.g., [1,2,3,1,2,3,1,2,3]);

findCycle1 :: [Pos] -> [Pos]
findCycle1 [] = []
findCycle1 (a : xa) = (a : takeWhile (a /=) (xa)) ++ (a : [])
   --gives a list with one occurence of the cycle;
```

Agent

```
module AgentClass where

import ShortestPath
import Graphs
import Strings
import ZeroOne

------------------ Data types ----------------------

data Agent= Agent Fact Beliefs deriving Show
data Fact = Fact AgentId Pos PrevPos deriving Show
type AgentId = Int
data Pos = Pos Node deriving (Show,Eq)
type PrevPos = Pos

data Beliefs = Beliefs MentalPos CognitiveMap Perceived Decision Preferences IncomingDir
deriving Show
type MentalPos = [NodeM]
type Perceived = [SignPost]
type Decision = Info
data IncomingDir = IDir Direction | NoIncDir deriving Show
type GoalCriterion = Info


------------------ Sense-Plan-Act operations ------------------

class (AgentPut Perceived agent, AgentPut Pos agent, AgentPut Fact agent, AgentPut Int agent,
       AgentPut IncomingDir agent, AgentPut MentalPos agent,
       EnvAgent env, InternalOps agent )
       => ExternalOps env agent where

   perceive :: env -> agent -> agent
   act :: env -> agent -> agent
   sensePlanAct :: env -> agent -> agent

   -- derived functions
   perceive env agent = putToAgent (perceiveAtPos env (getPos agent)) agent
   act env a = putToAgent (setToZero (getDec a)) a5 where
           a5 = putToAgent (setToZero (getPerc a4)) a4
           a4 = putToAgent (dirNext env (getPos a3) (getPrev a3) (getIncDir a3)) a3
           a3 = putToAgent (updatePosition env (getDec a2) (getPos a2)) a2
           a2 = putToAgent (updatePrev (getFact a1)) a1
           a1 = putToAgent (updateMentalPos (getPosM a) (getDec a) (getCM a)) a
   sensePlanAct env = (act env) . decide . (perceive env)

class (AgentGet agent, AgentPut Decision agent ) => InternalOps agent where
   decide :: agent -> agent
   decide agent
      = if isAtGoal (getCM agent) (getPerc agent) then error ("AGENT HAS REACHED GOAL ! ")
        else putToAgent (composedDecision (getCM agent) (getPerc agent) (getPos agent)
             (getPosM agent) (getPref agent) (getIncDir agent)) agent


------------------ Decision process --------------------
```

```haskell
class ComposedDec cm perc pos mpos pref inc where
   composedDecision :: cm -> perc -> pos -> mpos -> pref -> inc -> Decision

instance ComposedDec CognitiveMap Perceived Pos MentalPos Preferences IncomingDir where
   composedDecision cm perc pos mpos pref inc
       = metricDecision pref inc (semanticDecision cm pos mpos perc)

class SemanticDec cm pos mpos perc where
   semanticDecision :: cm -> pos -> mpos -> perc -> perc

instance SemanticDec CognitiveMap Pos MentalPos Perceived where -- 1st dec step
   semanticDecision cm pos mpos perc
      | f == [] = error ("NO SIGN INFORMATION FOUND at Node " ++ show (text pos))
      | otherwise = semanticDistFilter cm f
          where f = filterDownBoth cm mpos (filterPercs cm perc)

class MetricDec pref inc perc dec where
   metricDecision :: pref -> inc -> perc -> dec

instance MetricDec Preferences IncomingDir Perceived Decision where
   metricDecision pref (IDir i) perc
       = getInfoSignPost (head (sortLs (rotateAndUtil pref i perc))) -- real world
   metricDecision pref NoIncDir perc =
       = getInfoSignPost (head (sortLs (dirToUtility pref perc)))  -- www

    {- contains only step 2 (utility-function) for WWW-agent,
      and step 1 + 2 for real world agent (transformation of sign dirs and utility function-}

class SemanticFilter cm sp where
   filterPerc :: cm -> sp -> Bool    -- check if signpost is part of cogmap
   filterPercs :: cm -> [sp] -> [sp] -- filters percs that are part of cog map
   semanticDistFilter :: cm -> [sp] -> [sp]  -- find shortest paths with minimum distance

instance SemanticFilter CognitiveMap SignPost where
   filterPerc cm sp = isEltCM (getInfoSignPost sp) cm --check membership of signpost in cogmap
   filterPercs cm sps = [sp | sp <- sps, filterPerc cm sp == True]
   semanticDistFilter cm sps = [sp | sp <- sps, mentalDist cm sp == minLength cm sps]

class SemanticDownFilter cm node sp where
   filterDown :: cm -> [node] -> sp -> Bool
   filterDowns :: cm -> [node] -> [sp] -> [sp]
   filterEqual :: cm -> [node] -> sp -> Bool
   filterEquals :: cm -> [node] -> [sp] -> [sp]
   filterDownBoth :: cm -> [node] -> [sp] -> [sp]
   percHasNoMPosSingle :: cm -> sp -> node -> Bool
   percHasNoMPos :: cm -> sp -> [node] -> Bool

instance SemanticDownFilter CognitiveMap NodeM SignPost where
   percHasNoMPosSingle cm signpost nodem = goalForNode infoSp cm /= goalForNode nodem cm
      where infoSp = getInfoSignPost signpost

   percHasNoMPos cm sp [] = True
   percHasNoMPos cm sp mpos = and . map (percHasNoMPosSingle cm sp) $ mpos

   filterDown cm [] sp = False
   filterDown cm (n:ns) sp
     | (goalForNode infoSp cm == goalForNode n cm) = mentalDist cm sp < mentalDist cm n
     | otherwise = filterDown cm ns sp
         where infoSp = getInfoSignPost sp
   filterDowns cm mpos sps =
       [s | s <- sps, filterDown cm mpos s == True || percHasNoMPos cm s mpos == True]

   filterEqual cm [] sp = False
   filterEqual cm (n:ns) sp
     | (goalForNode infoSp cm == goalForNode n cm) = mentalDist cm sp == mentalDist cm n
     | otherwise = filterDown cm ns sp
         where infoSp = getInfoSignPost sp
   filterEquals cm mpos sps = [s | s <- sps, filterEqual cm mpos s == True]

   filterDownBoth cm mpos perc
       | f == [] = filterEquals cm mpos perc
       | otherwise = f
          where f = filterDowns cm mpos perc


------------------ Operations taking environment and components of agent ------
```

```
class EnvAgent env where
   perceiveAtPos :: env -> Pos -> Perceived
   dirNext :: env -> Pos -> PrevPos -> IncomingDir -> IncomingDir
   updatePosition :: env -> Decision -> Pos -> Pos

instance EnvAgent Environment where
   perceiveAtPos env pos =  percInfoAtNode (unPos pos) env

   dirNext env pos prev NoIncDir = NoIncDir -- WWW case, agent has no incoming dir
   dirNext env pos prev inc@(IDir i)  -- airport case
     | prev == unit0 = inc
     | otherwise
          = IDir (reverseDir (getDirSignPost (getSignPostForNode (unPos prev) incomingEdge)))
          where incomingEdge = head (filter (isAB (unPos pos) (unPos prev)) (getEdges env))
             --gives the direction where the agent comes from in the local reference frame

   updatePosition env dec pos = Pos (sndNode p2 (edgeWithSignAndNode dec p2 env))
       where edgeWithSignAndNode dec p2 env
                = head ([e | e <- containNode p2 env, containInfo e p2 dec])
             p2 = unPos pos


------------------ Operations referred to goal --------------

class GoalReached a perc where
   isAtGoal :: a -> perc -> Bool
   matchGoal :: a -> perc -> Bool

instance GoalReached GoalCriterion Perceived where
   matchGoal g [] = False
   matchGoal g (p:ps)
     | g == getInfoSignPost p = True
     | otherwise = matchGoal g ps

class GoalCrit cm where
   getGoalCrit :: cm -> GoalCriterion

instance GoalCrit CognitiveMap where
   getGoalCrit cmap@((G ((EM (Ir i1) (Ir i2)): es)):gs)
      = goalForNode (Ir (GateSign2 (GateSignRange (LetterOnly ‘A’) (LetterOnly ‘D’)))) cmap
   getGoalCrit cmap@((G ((EM (Iw i1) (Iw i2)): es)):gs) = goalForNode (Iw "do business") cmap

instance GoalReached CognitiveMap Perceived where
   isAtGoal cm perc = matchGoal (getGoalCrit cm) perc


------------------ Update agent’s components ---------

class Update1 a where
   setToZero :: a -> a  -- reset percept/decision
   updatePrev :: a -> a -- update previous node

instance Update1 Perceived where
   setToZero p = []

instance Update1 Decision where
   setToZero d = NoInfo

instance Update1 Fact where
   updatePrev (Fact a pos prev) = Fact a pos pos


------------------ Get-functions for agent ---------------

class AgentGet agent where
   getFact :: agent -> Fact
   getBel :: agent -> Beliefs
   getAId :: agent -> AgentId
   getPos :: agent -> Pos
   getPrev :: agent -> Pos
   getPosM :: agent -> MentalPos
   getCM :: agent -> CognitiveMap
   getPerc :: agent -> Perceived
   getDec :: agent -> Decision
   getPref :: agent -> Preferences
   getIncDir :: agent -> IncomingDir
```

```
instance AgentGet Agent where
   getFact (Agent f b) = f
   getBel (Agent f b) = b
   getAId = getIdFromFacts . getFact
   getPos = getPosFromFacts . getFact
   getPrev = getPrevFromFacts . getFact

   getPosM = getPosMFromBel . getBel
   getCM = getCMFromBel . getBel
   getPerc = getPercFromBel . getBel
   getDec = getDecFromBel . getBel
   getPref = getPrefFromBel . getBel
   getIncDir = getIncFromBel . getBel


------------------ Put-functions for agent --------------

class AgentPut p agent where
   putToAgent :: p -> agent -> agent

instance AgentPut Fact Agent where  -- put fact into agent
   putToAgent f2 (Agent f b) = Agent f2 b

instance AgentPut Beliefs Agent where -- put belief into agent
   putToAgent b2 (Agent f b) = Agent f b2

instance AgentPut Pos Agent where  -- put pos into agent
   putToAgent p a = putToAgent (putToFacts p (getFact a)) a

instance AgentPut MentalPos Agent where  -- put pos into agent
   putToAgent p a = putToAgent (putToBel p (getBel a)) a

instance AgentPut Perceived Agent where   -- put perceived to agent
   putToAgent p a = putToAgent (putToBel p (getBel a)) a

instance AgentPut Decision Agent where  -- put decision to agent
   putToAgent p a = putToAgent (putToBel p (getBel a)) a

instance AgentPut IncomingDir Agent where  -- put incDir to agent
   putToAgent p a = putToAgent (putToBel p (getBel a)) a


------------------ Put and get in agent structure ---------

class GetFromFacts f where
   getIdFromFacts :: f -> AgentId
   getPosFromFacts :: f -> Pos
   getPrevFromFacts :: f -> Pos

class PutToFacts p f where
   putToFacts :: p -> f -> f

class PutToFacts2 p f where
   putToFacts2 :: p -> f -> f

instance GetFromFacts Fact where
   getIdFromFacts (Fact aid pos prev) = aid
   getPosFromFacts (Fact aid pos prev) = pos
   getPrevFromFacts (Fact aid pos prev) = prev

instance PutToFacts Pos Fact where
   putToFacts pos2 (Fact a p pr) = Fact a pos2 pr

instance PutToFacts2 PrevPos Fact  where
   putToFacts2 pr2 (Fact a p pr) = Fact a p pr2

class GetFromBeliefs b where
   getPosMFromBel :: b -> MentalPos
   getCMFromBel :: b -> CognitiveMap
   getPercFromBel :: b -> Perceived
   getDecFromBel :: b -> Decision
   getPrefFromBel :: b -> Preferences
   getIncFromBel :: b -> IncomingDir

class PutToBeliefs b p where
   putToBel :: p -> b -> b

instance GetFromBeliefs Beliefs where
```

```
   getPosMFromBel (Beliefs mp cm perc dec pref inc) = mp
   getCMFromBel (Beliefs mp cm perc dec pref inc) = cm
   getPercFromBel (Beliefs mp cm perc dec pref inc) = perc
   getDecFromBel (Beliefs mp cm perc dec pref inc) = dec
   getPrefFromBel (Beliefs mp cm perc dec pref inc) = pref
   getIncFromBel (Beliefs mp cm perc dec pref inc) = inc

instance PutToBeliefs Beliefs MentalPos where
   putToBel mp2 (Beliefs mp cm perc dec pref inc) = Beliefs mp2 cm perc dec pref inc

instance PutToBeliefs Beliefs Perceived where
   putToBel perc2 (Beliefs mp cm perc dec pref inc) = Beliefs mp cm perc2 dec pref inc

instance PutToBeliefs Beliefs Decision where
   putToBel dec2 (Beliefs mp cm perc dec pref inc) = Beliefs mp cm perc dec2 pref inc

instance PutToBeliefs Beliefs IncomingDir where
   putToBel inc2 (Beliefs mp cm perc dec pref inc) = Beliefs mp cm perc dec pref inc2

class UnputPos p where
   unPos :: p -> Node

instance UnputPos Pos where
   unPos (Pos p) = p

class IncDirC i where
   unPutI :: i -> Direction

instance IncDirC IncomingDir where
   unPutI (IDir d) = d


--------------------- Instances for text output -------------

instance Strings Agent where
   text (Agent (Fact aid (Pos pos) (Pos prev)) (Beliefs mpos cmap per dec pref i ))
     = " \nPOSITION: " ++ text pos
         ++ " \nPREVNODE: " ++ text prev ++  "\nMENTAL POSTION: "
         ++ text mpos ++ " \nINCOMINGDIR: " ++ text i


   xtext (Agent (Fact aid (Pos pos) (Pos prev)) (Beliefs mpos cmap per dec pref i ))
     = putStrLn ("\nID:" ++ text aid ++ " \nPOSITION: " ++ text pos
         ++ " \nPREVPOS: " ++ text prev
         ++ " \nGOAL: " ++ text (getGoalFromCogMap cmap) ++" \nPERCEPTION: " ++ text per
         ++ " \nDECISION: " ++ text dec ++ " \nINCOMINGDIR: " ++ text i)

instance Strings [Agent] where
   text (a:alist) = text a ++ "\n\n" ++ text alist
   text [] = []

instance Strings IncomingDir where
   text (IDir i) = " " ++ text i
   text NoIncDir = " No IncDir "

instance Strings Pos where
   text (Pos w) = " " ++ text w

--------------------- Instances for ZeroOne class -------------

instance ZeroOne Pos where
   unit0 = Pos unit0
```

Operations on Graphs and Elements of Graphs

```
module Graphs where

import Strings
import List
import ZeroOne

------------------ DATA TYPES -------------

type NodeId = Int
type Text = String

data Node = Node NodeId deriving (Eq, Ord, Show)
data E n = E n SignPost n SignPost deriving (Eq,Show)
```

```haskell
data G e n = G [e n] deriving (Eq,Show)
type Environment = G E Node

data SignPost = S Direction Info | NoSign  deriving (Eq,Show)

type Direction = Int
data Info = Ir InfoR | Iw InfoW | NoInfo deriving (Show,Eq)
type InfoW = Text
type InfoR = GateSign

------------------ DATA TYPES for components of agent’s beliefs -----

type NodeM = Info
type Goal = [NodeM]

data EM n = EM n n deriving (Eq,Show)
type CognitiveMap = [G EM NodeM]

type Preferences   = [(Direction,Preference)]
type Preference    = Int

data Cost  = Cost Int deriving (Eq, Ord, Show)

------------------ DATA TYPES for signs and gates in airport environment -----

--specification of "gate-sign": 3 possibilities ->
--single: e.g., "A", "C51"
--list: e.g., "C52,C53", "A,B,D"
--range: e.g., "C54-C61", "A-D"

data GateSign = GateSign GateSignSingle | GateSign1 GateSignList | GateSign2 GateSignRange |
                NoGateSign deriving (Show, Eq)

data LetterOnly = LetterOnly Char deriving (Show,Eq)

data GateSignSingle = GateSignSingle LetterOnly | GateSignSingle1 Gate deriving (Show,Eq)
data GateSignList = GateSignList [LetterOnly] | GateSignList1 [Gate] deriving (Show,Eq)
data GateSignRange = GateSignRange LetterOnly LetterOnly | GateSignRange1 Gate Gate
                         deriving (Show,Eq)

data Gate = Gate Char Int deriving (Show,Eq)

------------------ Access elements of airport environment ----------

class LetterOnlys letterOnly where
   getLetterOnlyLetter :: letterOnly -> Char

instance LetterOnlys LetterOnly where
   getLetterOnlyLetter (LetterOnly l) = l

class Gates gate where
   getGateLetter :: gate -> Char
   getGateNumber :: gate -> Int
   getGate :: gate -> (Char,Int)

instance Gates Gate where
   getGateLetter (Gate l n) = l
   getGateNumber (Gate l n) = n
   getGate (Gate l n) = (l,n)

------------------- Access Information of signsposts -------

class SignPostC s where
   getInfoSignPost :: s -> Info
   getDirSignPost :: s -> Direction

instance SignPostC SignPost where
   getDirSignPost (S d i) = d
   getInfoSignPost (S d i) = i
   getInfoSignPost NoSign = NoInfo

class InfoC a where
   getInfoW :: a -> Text
   getInfoR :: a -> GateSign

instance InfoC Info where
   getInfoW (Iw i) = i
   getInfoR (Ir i) = i
```

```
------------------- Operations on nodes  ----------

class Nodes n where
   replace :: Int -> Info -> [n] -> [n] -- replaces n-th elt in [n] with info
   zeroNode :: n

------------------- Operations on edges ---------

class (Eq n) => Edges e n where
   endnode :: e n -> n
   startnode :: e n -> n
   getSignPostStart :: e n -> SignPost
   getSignPostEnd :: e n -> SignPost
   getSignPostForNode :: n -> e n -> SignPost -- signpost for edge at n
   getSignPostForNodeMap :: n -> [e n] -> [SignPost]
   getFstInfo :: e n -> Info
   getSndInfo :: e n -> Info
   isAB :: n -> n -> e n -> Bool
   sndNode :: n -> e n -> n  -- gets other node of an edge
   containInfo :: e n -> n -> Info -> Bool  -- check if edge has certain info at given node
   nodesE :: e n -> [n]
   costE :: n -> n -> e n  -> Cost
   equalEdge :: e n -> e n -> Bool
   isEltE :: n -> e n -> Bool
   hasNeighbour :: n -> e n -> Int
   makeE :: (Int, SignPost, Int, SignPost) -> e n

   makeEM :: (Info, Info) -> e n  -- functions to produce cmap

   -- derived function
   costE n m e = if isAB n m e then unit1 else unit100


---------------- Operations on graph --------------------

class (Edges e n) => Graphs g e n where
   insertG :: e n -> g e n -> g e n
   deleteG :: e n -> g e n -> g e n
   getEdges :: g e n -> [e n]
   isEltOfG :: n -> g e n -> Bool
   numberOfEdges :: g e n -> Int
   edgesOutWithSign :: n -> g e n -> [e n]
   edgesOutSignStart :: n -> g e n -> [e n]
   edgesOutSignEnd :: n -> g e n -> [e n]
   percInfoAtNode :: n -> g e n -> [SignPost]
   containNode :: n -> g e n -> [e n]
   nodes :: g e n -> [n]
   degree :: g e n -> n -> Int
   cost :: n -> n -> g e n -> Cost
   checkDegree :: g e n -> Bool
   allDegrees :: g e n -> [Int]
   getGoalFromGraph :: g e n -> n
   fieldPositionList :: [n] -> g e n -> [Bool]
   fieldPosition :: [n] -> g e n -> Int

   -- derived functions
   containNode n g = [e | e <- (getEdges g), (startnode e == n) || (endnode e == n)]
   numberOfEdges = length . getEdges
   nodes g = nub (concat [ nodesE e | e <- (getEdges g)])  --for shortest path


------------------ Operations on cognitive map ---------

class (Graphs g e n) => CognitiveMapC g e n where
   goalForNode :: n -> [g e n] -> n
   getGoalFromCogMap :: [g e n] -> [n]
   graphWithNode :: n -> [g e n] -> g e n
   nodeInCognitiveMap :: n -> [g e n] -> Bool
   updateMentalPos :: [n] -> n -> [g e n] -> [n]

   -- derived functions
   nodeInCognitiveMap n fs = or (map (isEltOfG n) fs) -- checks if mental map contains node n


------------------ Metric decision making ----------------

class DirectionC d where
```

```
   nodeToAgentDir :: d -> d -> d
   reverseDir :: d -> d

instance DirectionC Direction where
   nodeToAgentDir dir incDir = mod (dir -1 + mod (13 - incDir) 8 ) 8 + 1
           -- rotates sign directions at node to directions in agent’s egocentric ref frame
   reverseDir out = 1 + mod (out + 3) 8 -- makes outgoing direction to incoming direction

class Transformations signpost where
   rotateDirs :: Direction -> [signpost] -> [signpost]
   dirToUtility :: Preferences -> [signpost] -> [signpost]
   rotateAndUtil :: Preferences -> Direction -> [signpost] -> [signpost]
   orderSignPosts :: signpost -> signpost -> Bool
   sortLs :: [signpost] -> [signpost]

instance Transformations SignPost where  -- metric decision making (2nd step)
     {-1. function that translates info directions in local ref. frame of the node to
     directions in the  agent’s ref. frame; only required for real world case -}

   rotateDirs incDir signposts
      = map nodeToAgentDir’ signposts
            where nodeToAgentDir’ (S dir info) = S (nodeToAgentDir dir incDir) info

     {-2. function that converts directions in agent’s ref. frame to agent’s preferences;
     transforms a list of Infos to another list of Infos where the directions are changed to
     the agent’s preferences; lookup :: Eq a => a -> [(a,b)] -> Maybe b-}

   dirToUtility prefs signposts             -- utility function
      = map lookup1 signposts
            where lookup1 (S dir info)  = S (unMaybe (lookup dir prefs)) info

      --function composition 1. & 2.
   rotateAndUtil prefs incDir = dirToUtility prefs . rotateDirs incDir

     {-function that sorts a list of Infos according to order of preference-}
   orderSignPosts i1 i2 = (getDirSignPost i1) <= (getDirSignPost i2)

   sortLs [] = []
   sortLs (p:ps)
      = sortLs smaller ++ [p] ++ sortLs larger
        where
        smaller = [ q | q<-ps , orderSignPosts q p ]
        larger  = [ q | q<-ps , orderSignPosts p q ]
     --based on sortLs (Thompson, p189); sorts a list of percepts according to preference;


------------------ Instances for cognitive map --------------

instance Nodes NodeM where
   replace posIndex info mpos = genericTake (posIndex-1) mpos ++ [info]
                                                             ++ (genericDrop posIndex mpos)

instance Edges EM NodeM where
   startnode (EM sn en) = sn
   endnode (EM sn en) = en
   isAB n1 n2 e
      = ((n1==startnode e) && (n2==endnode e)) || ((n2==startnode e) && (n1==endnode e))
   nodesE (EM a b) = [a,b]
   equalEdge e1 e2 = (startnode e1 == startnode e2) && (endnode e1 == endnode e2)
   isEltE n e = startnode e == n || endnode e == n
   makeEM (a,b) = EM a b

instance Graphs G EM NodeM where
   insertG e (G es) = G (e:es)
   getEdges (G es) = es
   getGoalFromGraph = endnode . last . getEdges -- goal for given field
   fieldPositionList [] f = [False]
   fieldPositionList (p:ps) f = isEltOfG p f : fieldPositionList ps f
   fieldPosition ps f = 1 + head (elemIndices True (fieldPositionList ps f))
   cost n m (G es) = cost’ n m es
        where cost’ n m (e:es) = if isAB n m e then costE n m e
                else cost’ n m es
              cost’ n m [] = unit100
   isEltOfG n g = any (==n) (nodes g)

instance CognitiveMapC G EM NodeM where
   goalForNode n (g : gs)
     | isEltOfG n g == True = getGoalFromGraph g
     | otherwise = goalForNode n gs
   graphWithNode n [] = unit0
   graphWithNode n (field : fields)
     | isEltOfG n field == True = field
     | otherwise = graphWithNode n fields
   getGoalFromCogMap = map getGoalFromGraph
   updateMentalPos [] dec f = dec : []
   updateMentalPos mps dec (f:fs)
     | (isEltOfG dec f == True) && (or (fieldPositionList mps f) == True)
         = replace (fieldPosition mps f) dec mps
     | (isEltOfG dec f == True) && (or (fieldPositionList mps f) == False) = [dec] ++ mps
     | otherwise =  updateMentalPos mps dec fs


------------------ Instances for environment --------------

instance Edges E Node where
   startnode (E n1 sp1 n2 sp2) = n1
   endnode (E n1 sp1 n2 sp2) = n2
   sndNode n (E sn sp1 en sp2) = if n==sn then en else sn
   containInfo (E sn sp1 en sp2) n i
     | sp1 == NoSign && sp2 == NoSign
     | sp1 /= NoSign && sp2 == NoSign = (getInfoSignPost sp1 == i) && (n == sn)
     | sp1 == NoSign && sp2 /= NoSign = (getInfoSignPost sp2 == i) && (n == en)
     | otherwise = (getInfoSignPost sp1==i && (n==sn)|| (getInfoSignPost sp2==i) && (n==en))

   getSignPostStart (E n1 sp1 n2 sp2) = sp1
   getSignPostEnd (E n1 sp1 n2 sp2) = sp2
   getSignPostForNode n (E n1 sp1 n2 sp2)
     | (n == n1) = sp1
     | (n == n2) = sp2
     | otherwise = NoSign
   getSignPostForNodeMap n = map (getSignPostForNode n)
   getFstInfo (E n1 sp1 n2 sp2) = getInfoSignPost sp1
   getSndInfo (E n1 sp1 n2 sp2) = getInfoSignPost sp2
   isAB n1 n2 e = ((n1==startnode e) && (n2==endnode e))
                 || ((n2==startnode e) && (n1==endnode e))
   hasNeighbour n e@(E n1 sp1 n2 sp2)
     | (sndNode n e /= unit0) = 1
     | otherwise = 0
   makeE (n1, sp1, n2, sp2) = E (Node n1) sp1 (Node n2) sp2
   isEltE n e = (n==startnode e) || (n==endnode e)
   nodesE (E n1 sp1 n2 sp2) = [n1,n2]


instance Graphs G E Node where
   getEdges (G es) = es
   insertG e (G es) = G (e:es)
   degree g@(G (e:es)) n = sum (map (hasNeighbour n) (containNode n g))
   allDegrees g = map (degree g) (nodes g)
   checkDegree =  any (> 2) . allDegrees
   edgesOutSignStart sn g
        = [e | e <- (getEdges g), startnode e == sn, getSignPostStart e /= NoSign]
      -- edges that have startnode sn and have sign at this node
   edgesOutSignEnd sn g = [e | e <- (getEdges g), endnode e == sn, getSignPostEnd e /= NoSign]
      -- edges that have endnode sn and have sign at this node
   edgesOutWithSign sn g = concat [edgesOutSignStart sn g, edgesOutSignEnd sn g]
      -- edges that contain n and have sign at n
   percInfoAtNode sn g =  map (getSignPostForNode sn) (edgesOutWithSign sn g)
      -- list of perceived SignPosts at a node

------------------ Help function for Decision process  ----------

unMaybe :: Maybe a -> a
unMaybe (Just a) = a
unMaybe Nothing = error ("unMaybe of Nothing")

------------------ Instances for ZerOne class --------------------

instance ZeroOne Node where
   unit100 = Node 100
   unit0 = Node unit0

instance ZeroOne NodeM where
   unit0 = NoInfo

instance ZeroOne (G EM NodeM ) where
   unit0 = G []
```

```
instance ZeroOne CognitiveMap where
   unit0 = []

instance ZeroOne Cost where
   unit0 = Cost unit0
   unit1 = Cost 1
   unit100 = Cost 999999

------------------ Connectedness of nodes on edge  ---------------

instance Num Cost where
   (Cost f) + (Cost p) = Cost (p+f)

------------------ Instances for Strings class -------------------

instance Strings (EM NodeM) where
   text (EM n m) = " " ++ text n ++ " - " ++ text m

instance Strings (G EM NodeM) where
   text (G es) = " " ++  textRep es

instance Strings Node where
   text (Node n) =  text n

instance Strings Info where
   text (Iw i) = " " ++ text i
   text (Ir i) = " " ++ text i
   text NoInfo = " No info"

instance Strings SignPost where
   text (S dir info) = " SIGN " ++ text dir ++ text info
   text NoSign = " No sign "

instance Strings LetterOnly where
   text (LetterOnly l) = " " ++ text l

instance Strings GateSignSingle where
   text (GateSignSingle g) =  text g
   text (GateSignSingle1 g) = text g

instance Strings GateSignList where
   text (GateSignList ls) =  text ls
   text (GateSignList1 gs) = text gs

instance Strings GateSignRange where
   text (GateSignRange l1 l2) =  text l1 ++ " - " ++ text l2
   text (GateSignRange1 g1 g2) = text g1 ++ " - " ++ text g2

instance Strings GateSign where
   text (GateSign g) = " " ++ text g
   text (GateSign1 g) = " " ++ text g
   text (GateSign2 g) = " " ++ text g
   text NoGateSign = " No gatesign "

instance Strings Gate where
   text (Gate c i) = text c ++ text i
```

Shortest Path

```
{-       shortest path
         based on dijkstra as given by kirschenhofer
         implemented by andrew frank
          -}

module ShortestPath where

import Graphs
import Strings
import List
import ZeroOne

------------------ Data types for the Dijkstra code  ------------

data C  n = C n Cost n -- the node to which, cost, previous node on the shortest path
data SP n = SP [n] [n] [C n] [C n]  -- W, U, the active list,
                   -- the list of expanded (to keep, dijkstra does an update)
```

```
------------------ Operations in Dijkstra code  ------------

class NodesS n where
   getPathFromTo :: n -> n  -> G EM n -> [n]

class SPs sp n where
   makesp ::  n -> G EM n -> sp n
   step2 :: G EM n -> sp n -> sp n  -- expand one node
   endSteps, testConnected :: sp n -> Bool
   targetReached :: n -> sp n -> Bool
   getPathTo :: n -> G EM n -> sp n -> [n]
    -- the list of nodes to visit for shortest path to destination n

class CostTo c n where
   insertCL :: c n -> [c n] -> [c n]
   lu  :: n -> [c n] -> Cost
   prevShortest :: n -> [c n] -> n
   dropMax :: [c n] -> [c n]
   getPath :: n -> [c n] -> [n]
       -- find the sequence of nodes from start to destination

instance NodesS NodeM where      -- find shortest path between 2 nodes
   getPathFromTo start dest ges = getPathTo dest ges sp
     where sp::SP NodeM
           sp = (makesp start ges)

instance SPs SP NodeM where
   makesp x g@(G es) = SP [] (nodes g) [C x unit0 zeroNode] []   -- initialize
   endSteps (SP ws us _ _ ) = null us
   testConnected (SP ws us ls _) = not.null $ ls
   targetReached n (SP ws _ _ _) =  elem n ws
   step2 ges (SP ws us ls p) = SP ws’ us’ (sort(dropMax(map f us’))) p’
     where l = if null ls then error "SPs - not connected"
                          else head ls
           (C z luz _) = l             -- the minimum is at the head
           ws’ = z:ws
           us’ = filter (z/=) us
           luy y = lu y (ls++p) -- the previous cost to this node
           luy’ y = (lu z (ls++p)) + (cost z y ges)
           f y =  if (luy y) >  (luy’ y)
                  then C y (luy’ y) z
                  else C y (luy y)  (prevShortest y ls)
           p’ = l:p
   getPathTo dest ges sp = getPath dest costs
     where costs :: [C NodeM]
           costs = getLS lastSP
           getLS (SP ws us ls p) = p
           lastSP = head . dropWhile ((not.targetReached dest)
              &&& testConnected) . iterate (step2 ges) $ sp
           (&&&) cond1 cond2 a = (cond1 a) && (cond2 a)

instance CostTo C NodeM where
   insertCL cnc cl = insert cnc cl  -- this is the sorted insert
   lu m ((C n c _):cs) = if m==n then  c else lu m cs
   lu m [] = unit100  -- should not occur
   prevShortest m ((C n c ss):cs) = if m==n then  ss else prevShortest m cs
   prevShortest m [] = zeroNode --  should not occur

   dropMax  cs = filter notMax cs
      where notMax (C n c _) = c/= unit100
   getPath dest cs = reverse (getPath’ dest cs)
      where getPath’ dest [] = []
            getPath’ dest ((C n _ m):cs) = if dest==n then dest:getPath’ m cs
                                           else getPath’ dest cs

----------------- Functions based on the Dijkstra Algorithm  ---------------------

class MentalPath cm sp where
   mentalShortPath :: cm -> sp -> [NodeM]
   mentalShortPaths :: cm -> [sp] -> [[NodeM]]
   minLength :: cm -> [sp] -> Int
   mentalDist :: cm -> sp -> Int

instance MentalPath CognitiveMap SignPost where
   mentalShortPath  cm (S d i) = getPathFromTo i (getGoalFromGraph field) field
     where field = graphWithNode i cm
   mentalShortPaths mm [] = []
   mentalShortPaths mm (sp:sps) = mentalShortPath mm sp : mentalShortPaths mm sps
   minLength mm sps = minimum (map length (mentalShortPaths mm sps))
   mentalDist cm sp = length (mentalShortPath cm sp)

instance MentalPath CognitiveMap Info where
   mentalShortPath  cm i = getPathFromTo i (getGoalFromGraph graph) graph
       where graph = graphWithNode i cm
   mentalDist cm i = length (mentalShortPath cm i)

-------------------------------------------------------------------------------

instance Eq (C n) => Ord (C  n) where
        (<=) (C n c _) (C n2 c2 _) = c <= c2
instance Eq n => Eq (C n) where
        (==) (C n c _) (C n2 c2 _) = n==n2 && c==c2
```

Zero-One Elements

```
module ZeroOne where

class ZeroOne z where
   unit0, unit1, unit100 :: z

instance ZeroOne Int where
   unit0 = 0
   unit1 = 1
   unit100 = 100

instance (ZeroOne a) => ZeroOne [(a,a)] where
   unit0 = [(unit0, unit0)]
```

Output of Data types

```
{-
af july99
-}

module Strings where

infixr 5 ++., ++^, ++/, ++-

class Strings a where
         xtext :: a -> IO ()
         xtext = putStr . text

         text :: a -> String
--
         textRep :: a -> String                -- to show internal rep
         textRep = text
         text = textRep
         (++.), (++^), (++/), (++-) :: a -> a -> a   --  String -> String -> String

------------------ Instances ----------------------

--* added for convenient textual representation in Hugs
x :: Strings a => a -> IO ()
x = putStr . text
--*

instance Strings Int where
         text  = show --* was show’
instance Strings Float where
         text  = show
instance Strings Bool where
         text  = show
instance Strings Char where
         text = show

instance Strings String where
         text = id
         a ++. b = a ++ ", " ++ b
         a ++^ b = a ++ " " ++ b
         a ++/ b = a ++ "\n" ++ b
         a ++- b = a ++ "\t" ++ b

instance Strings t => Strings [t] where
```

```
    text =  foldr ((++).(++ ";").text) ""
    textRep  = foldr ((++).(++ ";").textRep) ""
              -- this adds a , after each element
```

Test data – Vienna International Airport

```
module VieData1 where

import WorldClass
import ZeroOne
import Graphs
import AgentClass
import Strings

------------------ Data for agent’s cognitive map -----------

cmapR :: CognitiveMap
cmapR = [graphSigns]

graphSigns :: G EM NodeM
graphSigns = foldr insertG (G []) (map makeEM graphSignsS)

graphSignsS =
 [(Ir (GateSign1 (GateSignList [(LetterOnly ‘A’),(LetterOnly ‘C’)])),
      Ir (GateSign1 (GateSignList [(LetterOnly ‘A’),(LetterOnly ‘B’),(LetterOnly
         ‘C’),(LetterOnly ‘D’)]))),
  (Ir (GateSign1 (GateSignList [(LetterOnly ‘B’),(LetterOnly ‘C’)])),
      Ir (GateSign1 (GateSignList [(LetterOnly ‘A’),(LetterOnly ‘B’),(LetterOnly
         ‘C’),(LetterOnly ‘D’)]))),
  (Ir (GateSign1 (GateSignList [(LetterOnly ‘A’),(LetterOnly ‘C’)])),
      Ir (GateSign2 (GateSignRange (LetterOnly ‘A’) (LetterOnly ‘D’)))),
  (Ir (GateSign1 (GateSignList [(LetterOnly ‘B’),(LetterOnly ‘C’)])),
      Ir (GateSign2 (GateSignRange (LetterOnly ‘A’) (LetterOnly ‘D’)))),
  (Ir (GateSign (GateSignSingle (LetterOnly ‘C’))),
      Ir (GateSign1 (GateSignList [(LetterOnly ‘B’),(LetterOnly ‘C’)]))),
  (Ir (GateSign (GateSignSingle (LetterOnly ‘C’))),
      Ir (GateSign1 (GateSignList [(LetterOnly ‘A’),(LetterOnly ‘C’)]))),
  (Ir (GateSign2 (GateSignRange1 (Gate ‘C’ 51) (Gate ‘C’ 62))),
      Ir (GateSign (GateSignSingle (LetterOnly ‘C’)))),
  (Ir (GateSign2 (GateSignRange1 (Gate ‘C’ 51) (Gate ‘C’ 62))),
      Ir (GateSign (GateSignSingle1 (Gate ‘C’ 54))))]


------------------ Data for airport graph -----------

realGraph :: Environment
realGraph = foldr insertG (G []) (map makeE realStrings)
realStrings :: [(NodeId, SignPost, NodeId, SignPost)]

realStrings =
 [(0,NoSign,1,NoSign), -- fictive edge for orientation at node 1
  (1,S 1 (Ir (GateSign2 (GateSignRange (LetterOnly ‘A’) (LetterOnly ‘D’)))),2,NoSign),
  (2,S 0 (Ir (GateSign1 (GateSignList [(LetterOnly ‘A’),(LetterOnly ‘B’),
     (LetterOnly ‘C’),(LetterOnly ‘D’)]))),3, NoSign),
  (3,S 1 (Ir (GateSign (GateSignSingle (LetterOnly ‘A’)))),4,NoSign),
  (3,S 0 (Ir (GateSign1 (GateSignList [(LetterOnly ‘A’),(LetterOnly ‘C’)]))),5,NoSign),
  (3,S 7 (Ir (GateSign1 (GateSignList [(LetterOnly ‘B’),(LetterOnly ‘C’)]))),6,NoSign),
  (4,S 6 (Ir (GateSign (GateSignSingle (LetterOnly ‘C’)))),
     5, S 2(Ir (GateSign (GateSignSingle (LetterOnly ‘A’))))),
  (5,S 6 (Ir (GateSign1 (GateSignList [(LetterOnly ‘B’),(LetterOnly ‘C’)]))),
     6,S 2 (Ir (GateSign (GateSignSingle (LetterOnly ‘A’))))),
  (6,S 6 (Ir (GateSign1 (GateSignList [(LetterOnly ‘B’),(LetterOnly ‘C’)]))),7, NoSign),
  (7,S 6 (Ir (GateSign2 (GateSignRange1 (Gate ‘C’ 51) (Gate ‘C’ 62)))),9,NoSign),
  (7,S 5 (Ir (GateSign (GateSignSingle (LetterOnly ‘B’)))),8,NoSign),
  (9,S 7 (Ir (GateSign (GateSignSingle1 (Gate ‘C’ 54)))),10,NoSign) ]

------------------ Creating the world -----------

worldAirport = World time agent g where
   time = 1
   agent = fred
   g = realGraph

------------------ Creating the agent ----------

fred = Agent (Fact aid (Pos (Node pos)) (Pos (Node prev))) (Beliefs mpos cm perc dec pref inc)
where
   aid = 1
   pos = 1
   prev = 0
   mpos = []
   cm = cmapR
   perc = []
   dec = NoInfo
   pref = [(1,1),(2,2),(3,4),(4,6),(5,8),(6,7),(7,5),(8,3)]
   inc = IDir 0
```

Test data – Yahoo-directories

```
module WebData1 where

import WorldClass
import ZeroOne
import Graphs
import AgentClass
import Strings

------------------ Data for agent’s cognitive map -----------

cmapW :: CognitiveMap
cmapW = [graphUia,graphPhys, graphAa, graphBrand, graphSize]

graphUia, graphPhys, graphAa, graphBrand, graphSize :: G EM NodeM
graphUia = foldr insertG (G []) (map makeEM graphUiaS)
graphPhys = foldr insertG (G []) (map makeEM graphPhysS)
graphAa = foldr insertG (G []) (map makeEM graphAaS)
graphBrand = foldr insertG (G []) (map makeEM graphBrandS)
graphSize = foldr insertG (G []) (map makeEM graphSizeS)

graphUiaS =
     [(Iw "do shopping",Iw "confirm"),
       Iw  "do business",Iw "confirm")]
graphAaS =
     [(Iw "recreate",Iw "do sport" ),
      (Iw  "do sport",Iw "do track and field" ),
      (Iw "do track and field",Iw  "running" )]
graphPhysS =
     [(Iw "physical object",Iw "artifact"),
      (Iw "artifact",Iw "covering" ),
      (Iw "covering",Iw "clothing" ),
      (Iw "clothing",Iw "footwear" ),
      (Iw "footwear",Iw "shoe" )]
graphBrandS =
     [(Iw "brand",Iw "Adidas"),
      (Iw "brand",Iw "Converse"),
      (Iw "brand",Iw "Reebok"),
      (Iw "Adidas",Iw "Nike"),
      (Iw "Reebok",Iw "Nike"),
      (Iw "Converse",Iw "Nike")]
graphSizeS =
     [(Iw "size",Iw "10 1/2"),
      (Iw "10 1/2",Iw "10" ),
      (Iw "10",Iw "9 1/2")]


------------------ Data for web graph -----------

wwwGraph :: Environment
wwwGraph = foldr insertG (G []) (map makeE webStrings)
webStrings :: [(NodeId, SignPost, NodeId, SignPost)]

webStrings =
 [(1,S 2 (Iw "do business"),2,S 1 (Iw "Home")),
  (1,S 9 (Iw "recreate"),3,S 1 (Iw "Home")),
  (2,S 2 (Iw "do shopping"),4,S 1 (Iw "do business")),
  (3,S 19 (Iw "do sport"),5,S 3 (Iw "recreate")),
  (4,S 3 (Iw "clothing"),6,S 1 (Iw "do shopping")),
  (4,S 61 (Iw "do sport"),7,S 1 (Iw "do shopping")),
  (5,S 107 (Iw "running"),8,S 1 (Iw "do sport")),
  (5,S 90 (Iw "do track and field"), 9,S 1 (Iw "do sport")),
  (6,S 2 (Iw "do sport"), 10,S 1 (Iw "clothing")),
  (6,S 7 (Iw "shoe"),11,S 1 (Iw "clothing")),
  (7,S 1 (Iw "clothing"), 10, NoSign),
  (7,S 59 (Iw "running"), 12,S 1 (Iw "do sport")),
  (7,S 76 (Iw "do track and field"), 13,S 1 (Iw "do sport")),
  (8,S 12 (Iw "do shopping"), 12, NoSign),
  (9,S 12 (Iw "do shopping"), 13, NoSign),
  (10,S 7 (Iw "shoe"), 15, NoSign),
  (10,S 21 (Iw "running"), 21, NoSign),
  (11,S 3 (Iw "brand"), 14,S 1 (Iw "shoe")),
  (11,S 2 (Iw "do sport"), 15,S 1 (Iw "shoe")),
  (12,S 5 (Iw "shoe"), 17,S 1 (Iw "running")),
  (12,S 1 (Iw "clothing"), 21,S 1 (Iw "running")),
  (14,S 1 (Iw "do sport"), 16, NoSign),
  (15,S 3 (Iw "brand"), 16,S 1 (Iw "do sport")),
  (15,S 9 (Iw "running"), 17, NoSign),
  (16,S 1 (Iw "Nike"), 18, NoSign),
  (16,S 2 (Iw "Adidas"), 19, NoSign),
  (16,S 6 (Iw "Reebok"), 20, NoSign),
  (17,S 1 (Iw "confirm"),25, NoSign),
  (18,S 1 (Iw "confirm"),22, NoSign ),
  (19,S 1 (Iw "confirm"),23, NoSign),
  (20,S 1 (Iw "confirm"), 24, NoSign) ]

------------------ Creating the world -----------

worldWWW = World time agent g where
   time = 1
   agent = charly
   g = wwwGraph

------------------ Creating the agent ----------

charly=Agent (Fact aid (Pos (Node pos)) (Pos (Node prev))) (Beliefs mpos cm perc dec pref inc)
where
   aid = 1
   pos = 1
   prev = 0
   mpos = []
   cm = cmapW
   perc = []
   dec = NoInfo
   pref = [(1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,8),(9,9),(10,10),(11,11),(12,12),
           (13,13),(14,14),(15,15),(16,16),(17,17),(18,18),(19,19),(20,20),
           (21,21),(22,22),(23,23),(24,24),(25,25),(26,26),(27,27),(28,28),(29,29),(30,30),
           (31,31),(32,32),(33,33),(34,34),(35,35),(36,36),(37,37),(38,38),(39,39),(40,40),
           (41,41),(42,42),(43,43),(44,44),(45,45),(46,46),(47,47),(48,48),(49,49),(50,50),
           (51,51),(52,52),(53,53),(54,54),(55,55),(56,56),(57,57),(58,58),(59,59),(60,60),
           (61,61),(62,62),(63,63),(64,64),(65,65),(66,66),(67,67),(68,68),(69,69),(70,70),
           (71,71),(72,72),(73,73),(74,74),(75,75),(76,76),(77,77),(78,78),(79,79)]
   inc = NoIncDir
```

BIOGRAPHY OF THE AUTHOR

14th August 1973	Born in Salzburg, Austria
1992	Graduated from high-school (Bundesrealgymnasium Zell am See)
1992-1993	Military service
Oct 1998	Master's degree in geodesy, Department of Applied and Engineering Geodesy, Technical University Vienna
Oct 1998 – Feb 1999	Worked as surveyor with Vermessungsbüro Hochmair, Zell am See
Mar 1999 -	Research and teaching assistant at the Institute for Geoinformation, Technical University Vienna
June 2000	Master's degree in electroacoustic composition, University of Music and Performing Arts Vienna

Grants:

Sep 1997	Invitation for the summer course „2. Komponistenforum Mittersill“ (scholarship by the Salzburger Landesregierung)
Sep 1998	Invitation for participation at the „6th international academy for composition and audio-art“ in Schwaz/Tirol (scholarship by the Stadtwerke Schwaz)
April 2000	Young researcher prize (GISRUK conference, York, UK)
Sep 2001	ESRI Student Award (COSIT 2001, Morro Bay, Ca, USA)