Constructing Mobile Agents using Transformations

Von der

Fakultät für Informatik

der

Universität der Bundeswehr München

zur Verleihung des akademischen Grades

Doktor der Naturwissenschaften (Dr. rer. nat.)

genehmigte Dissertation

von

Oliver Braun

Vorsitzender des Promotionsausschusses: Prof. Dr. Burkhard Stiller

1. Gutachter: Prof. Dr. Gunther Schmidt

2. Gutachter: Prof. Dr. Wolfram Kahl, McMaster Univ., Hamilton, Ontario, Kanada

Tag des Kolloquiums: 11. August 2004

Für Simone

Abstract

Although mobile agents are widely accepted as a useful and elegant way to develop software for distributed systems, they are not widely-used. One of the main reasons for this is the lack of provable and manageable protection of an agent platform.

We present a working prototype of a mobile agent programming and execution environment based on term graph transformation and the strongly typed functional programming language Haskell. The well-understood type system of Haskell can be used to guarantee that a mobile agent cannot do arbitrary I/O on an agent platform. Therefore the mobile agent migrates as Haskell source code and is type-checked on each platform.

The services a platform offers to a mobile agent are encapsulated in so-called possibly-provided functions (PPFs). A PPF is a function that returns a value of type `Maybe a` for arbitrary `a`. A PPF need not be available on all platforms. If it is not available, it will be temporarily replaced with a function returning the value `Nothing` of type `Maybe a` by a preprocessor on the agent platform. In order to be able to replace a PPF, the Haskell code is encapsulated in a list of code fragments.

A strongly typed mobile agent which consists of one complex function with a complex parameter which has to be encapsulated in a list of code fragments is a very awkward piece of software. The complex parameter is the data part of the mobile agent and is called its suitcase. Conventional programming using a normal text editor is not feasible. Therefore we use the Higher Object Programming System HOPS, a graphically interactive program development and transformation system based on term graphs, for developing a mobile agent.

When using HOPS we have the ability to define an easy-to-use domain-specific language for mobile agents and to derive the necessary shape of the mobile agent code using term graph transformation. Not only the encapsulation into the code fragment data type can be automated, but also the complex suitcase can be generated automatically from small manageable pieces.

With this thesis we demonstrate that the combination of a strongly typed functional language and a development process which is aided by term graph transformation together with our concept of PPFs is a very well suited approach towards a fully-fledged mobile agent system.

Contents

1 Introduction

A mobile agent is a code-containing object which is able to migrate under its own control from agent platform to agent platform in order to achieve a specific task. A representative example is an agent which searches on a couple of servers for the cheapest travelling arrangement, that fulfils some particular requirements. In traditional client/server computing all information about travelling arrangements is transmitted from the server to the client where the client process filters the data. By using mobile agents, the client process is transmitted to the server in order to process the available data. Subsequently, the mobile agent returns with the cheapest arrangement in its so-called suitcase, the data part of the agent. Other applications of mobile agents are, for instance, information dissemination or monitoring of remote resources.

Although the mobile agent paradigm provides an elegant way to develop software for distributed systems, mobile agents have seldom be used outside closed environments. One of the main reasons for this is the lack of provable and manageable protection of an agent platform. Nearly all mobile agent systems use imperative languages, most notably Java which uses a so-called sandbox for addressing security concerns. The weaknesses of the Java sandbox, especially in distributed environments, have already been pointed out by Zhong and Edwards (1998).

Our design of a mobile agent platform uses the strongly typed, purely functional language Haskell. A mobile agent, which migrates as Haskell source code, can be type-checked and compiled on the agent platform before its execution. On the one hand, this leads to higher loads on an agent platform, but, on the other hand, the well-understood type system of Haskell can be used to guarantee that a mobile agent cannot do arbitrary I/O.

An agent platform has to provide the functions defined in the Haskell 98 report (Peyton Jones et al., 2002) except for I/O functions, and it has to provide a set of five so-called platform functions, e.g., a function for migrating to another platform.

The additional functionality that a platform offers to a mobile agent, such as a function to access information provided by the platform, is encapsulated in special functions. We call those functions possibly-provided functions (PPF).

As mentioned before, a mobile agent is type-checked and compiled on each platform. This means that all PPFs the mobile agent code contains, have to be available on the platform on which the mobile agent should be executed. Providing every single function on all platforms is not manageable, not scalable, and not flexible. A much better solution is to remove all non-available PPFs temporarily from the mobile agent code. This is done by a preprocessor on each agent platform before the mobile agent is type-checked, compiled and executed. Furthermore, the mobile agent is able to transform its own code. This is useful, for instance, to simulate partial evaluation by replacing a part of the mobile agent with the value of this part calculated on a platform.

Since a mobile agent in Haskell is a function and a PPF is a fragment of that function, a PPF cannot simply be removed. It has to be replaced with some other code fragment. Therefore, we demand that all PPFs have a `Maybe` type. This means, the value `x` calculated by a PPF has to be returned as `Just x`. This way, a non-available possibly-provided function can be replaced by a function returning the value `Nothing` which is of type `Maybe a` for arbitrary `a`.

In order to be able to replace those non-available PPFs on an agent platform, the PPFs and their parameters have to be marked somehow. We decided to encapsulate the mobile agent code into a list of code fragments. A code fragment is a self-defined Haskell data type whose possible values include plain code chunks and PPFs. With the concept of replaceable PPFs, which cannot be found in any other mobile agent environment, it is furthermore possible to replace even bigger parts of a mobile agent depending on the availability of PPFs.

A strongly typed mobile agent which consists of one complex function with a complex suitcase as parameter and which has to be encapsulated into a list of code fragments is a very awkward piece of software. Conventional programming using a normal text editor is not feasible. Therefore, we use the Higher Object Programming System HOPS, a graphically interactive program development and transformation system based on term graphs, for developing a mobile agent. We were favourable to enjoy an environment in which we were able to employ higher-order transformations.

A term graph in HOPS is already type-checked during the development. By this means, it is ensured that the mobile agent developed in HOPS neither contains any

syntax error nor any type error. Furthermore, when using HOPS we have the ability to define an easy-to-use domain-specific language for mobile agents and to derive the necessary shape of the mobile agent code using term graph transformation. Not only the encapsulation into the code fragment data type can be automated, but also the complex suitcase can be generated automatically from small manageable pieces.

Aims

This thesis is a case study in program development with mathematical precision. It is a demonstration, how programs for the net that undergo frequent changes may be kept in a satisfactory status and do not migrate to junk status.

It is the aim of this thesis:

1. to develop a mobile agent execution environment which is secure by means of using a strongly typed standard programming language with a small, manageable set of possibly-provided functions, which can differ on each platform;

2. to develop a powerful, easy-to-use and extensible domain-specific language for mobile agents without the need to provide this domain-specific language on the agent platform;

3. to develop transformation rules and strategies that can be used to convert the domain-specific language into the standard programming language;

4. to demonstrate that term graph transformation aids the development process of software and, in particular, of mobile agents.

Looking at this thesis, one should separate the two main concerns. Firstly we tried to contribute to the design of mobile agents. Secondly, we have tried to develop the mobile agent we decided to use, with mathematical precision.

It is the combination of the two aims that is important. If someone proposes yet another item for the agents, we will probably be able to integrate it by transformation. On the other hand side, it may well be that we did not yet head for the utmost ideas to incorporate into the agents.

Haskell

We use the pure, strongly-typed, lazy, non-strict, functional programming language Haskell (Peyton Jones et al., 2002) for the implementation of the mobile agent execution environment.

A functional program is a single expression which is executed by evaluating this expression. Expressions are formed by using functions to combine basic values. The term "purely functional" is often used to describe languages that perform all their computations via function application. In a broader sense this means that languages might incorporate computational effects, but without altering the notion of function. Typically, the evaluation of an expression can yield a task which is then executed separately to cause computational effects. Haskell is a purely-functional language. Input and output is done within a monad abstraction.

Haskell is polymorphically typed. It supports a systematic form of overloading and a module system. Furthermore Haskell supports higher order functions which are functions where either one of its arguments or its result or both are functions. Lazy evaluation means that an argument to a function will only be evaluated if its value is needed to compute the overall result. If an argument is structured like a list or a tuple for instance only those parts of the argument which are needed will be examined. An argument is almost evaluated only once. This is done in the implementation by replacing expressions by graphs and term reduction by graph reduction. Lazy evaluation has consequences for the style of programs. With lazy evaluation it is possible to describe infinite structures.

In a strict language the arguments to a function are always evaluated before the function definition is invoked. This results in the fact that if the evaluation of an expression `e` does not terminate properly (this may happen when it generates a run-time error or enters an infinite loop), then neither will an expression of the form `f(e)`. In a non-strict language the arguments to a function are not evaluated until their values are actually required. For example evaluating an expression of the form `f(e)` may still terminate properly if the value of the parameter `e` is not used in the body of `f`, even if evaluation of `e` would not.

Until the mid-1980s there was no "standard" non-strict, purely-functional programming language. A language-design committee was set up in 1987, and the Haskell language is the result. We use Haskell 98 (Peyton Jones et al., 2002), the latest version of the language. For compiling the mobile agents and the mobile agent platform we use the

Glasgow Haskell Compiler (Peyton Jones, 1993), a robust, fully-featured, optimising compiler for Haskell 98.

Overview

There is an inherent problem in presenting a thesis like this. While a mathematician may develop his exposition with a formalised proof, e.g., presented in TEX, we here are formal in the same way but cannot present such details for lack of space. Once the mathematician has found his way to prove, he can easily present this proof with a simple sequence of rather complicated steps. In this thesis, however, the situation is reversed: We have a highly complicated (i.e., strategically determined) sequence of comparatively simple steps.

In the following chapter the Higher Object Programming System HOPS, a graphically interactive development and program transformation system based on term graphs, is introduced. Mobile agents are described in Chapter 3. Chapter 4 gives an overview of the mobile agent programming and execution environment based on HOPS and Haskell.

The development of a mobile agent is divided into three different phases. First of all, a mobile agent developer has to define a mobile agent in terms of the User Interface Domain-Specific Language (UI-DSL). UI-DSL is a combinator language and is introduced in Chapter 5. The second phase is the transformation of the UI-DSL mobile agent into an internal mobile agent using term graph transformation. I-DSL is the Internal Domain-Specific Language and is used to represent a mobile agent with an automatically generated suitcase. I-DSL is introduced in Chapter 6 and the transformation from UI-DSL to I-DSL is presented in Chapter 7. Some items of the strategies, e.g., in Chapter 7, are hard to communicate. The indications given in this exposition may seen sloppy, they nevertheless, refer to a representation greatly checked — against the HOPS system.

The third phase is the transformation of the internal mobile agent into a monadic form which can easily be used to create standard Haskell code using the simple code output facility of HOPS. The Haskell Mobile Agent Platform (HaMAP) which we have developed especially for this approach is introduced in Chapter 8. Transformation from I-DSL to the monadic form and code output is described in Chapter 9. In Chapter 10 different examples of mobile agents are presented in order to illustrate our approach.

The transformed versions of those agents and the generated Haskell code are shown in Appendix A. In Appendix B the parts of the Z-Notation that are used in Chapter 2 are introduced.

The effort required for this approach is considerable. It should however be compared to the gain in being able to maintaining the status. It is shown that with the help of these ideas very long-range transformation suites may safely be executed and also adapted to newly occurring situations.

Acknowledgements

Most of all I would like to thank my supervisor Prof. Dr. Gunther Schmidt. He gave me a lot of help and encouragement and provided the necessary freedom for my research. Furthermore, I would like to thank Dr. Frank Derichsweiler and Prof. Dr. Wolfram Kahl for the introduction to term graph transformation and HOPS, and for hours and hours of productive discussions about my ideas. Thanks also to Michael Ebert for taking over those discussions after Frank and Wolfram left the faculty. Last but not least, I would like to thank my colleagues at the Institute for Software Technology and at the Institute for Information Systems for the scientific environment that is an essential support for an endeavour like this.

2 Higher Object Programming System HOPS

As mentioned in Chapter 1, we use term graphs and term graph transformation for the development of mobile agents. Programming with term graphs is not yet widely used, but facilitates the development of programs which are correct by construction. Our tool was the Higher Object Programming System HOPS, which is a graphically interactive program development and transformation system based on term graphs. HOPS has been developed by a group led by Gunther Schmidt since the mid-eighties of the 20th century (Bayer et al., 1996) (Zierer et al., 1986) (Kahl, 1996) (Kahl, 1998) (Kahl, 1999). The current implementation of HOPS is theoretically founded on the work of Wolfram Kahl. Kahl (1996) seems to have presented the first algebraic approach to term graph rewriting encompassing the treatment of bound variables.

The term graphs used in HOPS are directed acyclic graphs (DAGs), where all the structure usually encoded via name and scope is made explicit. Variables in HOPS are nameless. An explicit variable identity edge can be used to denote which nodes stand for the same variable. Binding of variables is denoted by an explicit directed binding edge from the bound variable to its binder. All problems usually connected with name clashes and variable renaming are avoided this way.

HOPS uses term graph transformation for manipulating DAGs. Transformation rules are also given as term graphs with an additional arrow connecting the roots of the left- and right-hand sides. Application of rules is always possible while developing term graphs and can be automated by using strategies (Derichsweiler, 2002).

Although, it is not essential for programming in HOPS to know all formal definitions of its theoretical foundation, it is one of the strengths of HOPS that the underlying theory is mathematically sound. When programming in HOPS it is, for instance, possible to guarantee algebraic properties of the programs developed. So, we have decided to introduce a small part of the theoretical foundation by providing the formal definitions. In-between we give short explanations and examples to illustrate those definitions.

A mobile agent in this approach is a term graph which is manipulated using term graph transformation. As mentioned before, a transformation rule is also given as a term graph. In order to apply a rule to a term graph, a graph homomorphism from one rule side to the term graph has to be constructed. This homomorphism, which is also called matching homomorphism or matching, identifies the parts of the term graph representing the nodes of the appropriate side of the rule. If such a homomorphism exists, the rule can be applied to this term graph. In HOPS it is possible to use a rule in both directions from left-to-right and from right-to-left.

HOPS supports second-order term graphs, i.e., metavariables may have successors. The matching homomorphism maps a metavariable to a so-called interval. HOPS enforces term graph consistency and, therefore, ensures that the initial term graph as well as the transformed program does not contain any syntax error introduced by the user or by transformation. Furthermore, the utilisation of HOPS prevents type errors by using strong online term graph typing. Besides interactive application of a rule, a sequence of transformations can be automated with transformation strategies.

The present introduction to HOPS is limited to the concepts necessary for understanding the following chapters. Since term graphs are the main concept, they are formally introduced in Section 2.1. Homomorphy and typing are explained in Section 2.2, respectively in Section 2.3. All formal definitions presented in Section 2.1 and in Section 2.3, except for Definition 2.4, are verbally taken over from Kahl (1998). Transformation of term graphs is presented in Section 2.4. Term graph patterns and strategies can be used to automate term graph transformation. Those concepts are introduced in parts and in an informal manner in Section 2.5, which is sufficient for understanding the following chapters. More complete information and formal definitions can be found in Derichsweiler (2002), the doctoral thesis of Frank Derichsweiler, which essentially deals with those concepts. The formal definitions presented in this chapter are using the Z-Notation (Spivey, 1992) which is introduced in Appendix B.

2.1 Term Graphs

In HOPS typed, second-order term graphs are used to represent programs. A formal introduction is given by the following definitions.

Definition 2.1 (Term Graph Alphabet)
*A **term graph alphabet** is a tuple $(\mathcal{L}, A, \mathcal{C}, \mathcal{B}, \mathcal{M})$ with the set $\mathcal{L}$ of **node labels**, the **arity function** $A : \mathcal{L} \to \mathbb{N}$, and a partition of $\mathcal{L}$ into the sets $\mathcal{C}$ of labels for **constant constructors**, $\mathcal{B}$ for **bindable variables**, and $\mathcal{M}$ for **metavariables**.* □

The structure of a term graph alphabet is essentially the same as the structure of second-order terms (Klop, 1980), but there is no separate class of binders, and the "constant constructor" refers to what is usually called "function symbol". In the following a fixed term graph alphabet $(\mathcal{L}, A, \mathcal{C}, \mathcal{B}, \mathcal{M})$ is assumed.

Definition 2.2 (Term Graph)
*A **term graph** is a tuple $G = (\mathcal{N}, L, S, D, B, W, T)$ with*

- *$\mathcal{N}$, the finite **node set**,*
- *$L : \mathcal{N} \to \mathcal{L}$, the **node labelling** function,*
- *$S : \mathcal{N} \to \mathcal{N}^*$ the **successor** function with $L \fatsemi A = S \fatsemi \mathrm{len}$, i.e., the length of the successor list of each node has to be the arity of its label,*
- *$D : \mathcal{N} \leftrightarrow \mathcal{N}$, the **associated relation**, $D := \{(x, l) : S;\ y : \mathrm{ran}\, l \bullet (x, y)\}$,*
- *$T : \mathcal{N} \nrightarrow \mathcal{N}$, the partial **typing** function, where $(D \cup T)$ has to be acyclic,*
- *$B : \mathcal{N} \nrightarrow \mathcal{N}$, the **binding** function, where for $(x, b) : B$ the bound variable x has to have a label in $\mathcal{B}$, the binder b a label in $\mathcal{C}$, and b dominates x in the graph induced by $(D \cup T)$,*
- *$W : \mathcal{N} \leftrightarrow \mathcal{N}$, the **variable identity**, a partial equivalence relation defined exactly on variables, i.e., on nodes with labels from $B \cup M$. The variable identity has to be compatible with the labelling: $W \fatsemi L \subseteq L$, and with the binding: $W \fatsemi B \subseteq B$.*

*Roots are considered with respect to $(D \cup T)$, and the **type part** of a typed term graph is the set $\mathrm{ran}.(T \fatsemi (D \cup T)^*)$ containing all nodes reachable from typing nodes.* □

The term graphs used in HOPS are directed and acyclic. In the following they are therefore also called **DAG**, which is the abbreviation for **directed acyclic graph**. In Figure 2.1 three example DAGs are shown. The left one corresponds to the term $(\lambda\, x.x + 1)\ 1$, the DAG in the middle to $(\lambda\, x.f\ x)\ g$, and the right one to $(f\ 1) + (f\ 2)$.

The black arrows are the successor edges with the sequence indicated by their left-to-right order. The red respectively thick, dark grey arrow is the binding edge from the bound variable to the binder and the blue respectively thick, medium grey line is the variable identity. The green respectively bright grey arrows represent the typing

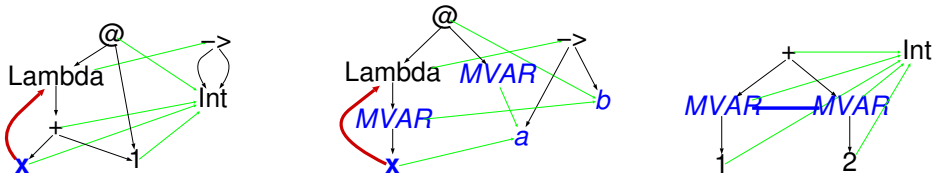


Figure 2.1: Example DAGs

function. The black labelled nodes are constant constructors, the blue respectively grey labelled nodes with normal font are metavariables, the blue respectively grey labelled nodes with bold font are bindable variables. The nodes which are source of a typing arrow, i.e., the nodes which are element of $\mathrm{dom}\,.\,T$, are in the **object layer** of the DAG. The nodes which are sink of a typing arrow or reachable from such a sink, i.e., $\mathrm{ran}\,.(T \fatsemi (D \cup T)^*)$, are building the **type layer** of the DAG. The object layer and the type layer are disjoint.

Definition 2.3 (Free Variable, Encapsulation)
*A variable node x is **free below** a node a, if there is a $(D \cup T)$-path from a to x such that no binder of x lies on that path; if in this constellation x is bound by b, then b **encapsulates** a. The **encapsulation** $C : \mathcal{N} \leftrightarrow \mathcal{N}$ relates b with a exactly when b encapsulates a.* □

In the DAG shown on the left-hand side of Figure 2.1 the variable `x` is encapsulated by the node `Lambda` and `x` is free below the node `+`.

In the following chapters, we sometimes refer to only part of a DAG. Therefore, we define the term sub-DAG.

Definition 2.4 (Sub-DAG, Induced Sub-DAG)
*A term graph $G' = (\mathcal{N}', L', S', D', B', W', T')$ is called **sub-DAG** of a DAG $G = (\mathcal{N}, L, S, D, B, W, T)$ if*

- *$\mathcal{N}' \subseteq \mathcal{N}$,*
- *$\mathcal{N}'$ is closed with respect to S and T,*
- *all components restricted to $\mathcal{N}'$ are corresponding to the components of G', i.e., $L' = \mathcal{N}' \lhd L$, $S' = \mathcal{N}' \lhd S$, $D' = \mathcal{N}' \lhd D$, $T' = \mathcal{N}' \lhd T$, $B' = \mathcal{N}' \lhd B \rhd \mathcal{N}'$, and $W' = \mathcal{N}' \lhd W \rhd \mathcal{N}'$.*

*For a node $n \in \mathcal{N}$ the sub-DAG which contains n and has the smallest set of nodes with respect to set inclusion is called **sub-DAG induced by** n.* □

2.2 Homomorphy

In second-order term graphs metavariables may have successors. Therefore the images of metavariables have to *stop* before the image nodes of their successors.

In HOPS the image of a metavariable is called **(image) interval**. An interval consists of a top node and a partial node sequence, the lower border. The lower border represents the images resp. the top nodes of the image intervals of the successors of the metavariable.

All nodes that are reachable from the top node via paths on which there lies no node of the lower border are called **inner nodes**. If for an interval the top node is element of the lower border the interval contains no inner node and is called empty.

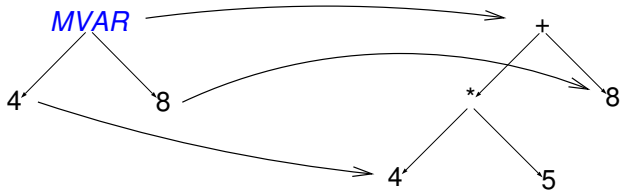


Figure 2.2: Two graphs with homomorphism

In Figure 2.2 two graphs and a function from one graph to the other is shown. The function, illustrated by the arrows from the left-hand side to the right-hand side, maps `MVAR` to `+`, `4` to `4`, and `8` to `8`. Since `MVAR` is a metavariable node, `+` is the top node of its image interval. The lower border contains the nodes `4` and `8`, the inner nodes are `*` and `5`.

The function shown in Figure 2.2 is called **term graph homomorphism**. Term graph homomorphisms will later mainly be used to serve as matchings from rule sides into application graphs. Rules are introduced in Section 2.4.

2.3 Typing

Term graphs in HOPS have to be well-formed as described in the preceding sections and they have to be well-typed as described below.

Definition 2.5 (Typing Element)
*A **typing element** is a typed term graph G which either is rooted or has all its sources related to each other by the variable identity, and where all successors of the*

source nodes are meta variables and all successors of those metavariables are bound by the root node. Such a typing element is said to be ***for*** *the label of its root node.* □

Typing elements closely correspond to typing rules in type derivation systems, e.g., presented in Barendregt (1992). Typing elements are also called **bricks** in the context of HOPS.

Definition 2.6 (Term Graph Language)
A ***term graph language*** *is a set $\mathcal{T}$ of typing elements, such that $\mathcal{T}$ contains at most one typing element for each node label $l : \mathcal{C}$.* □

The available implementations of HOPS do not support any kind of overloading of bricks. Thus, Definition 2.6 is not really a restriction in this context. A term graph language in the context of HOPS is also called **domain-specific language (DSL)**, since it is a language which has been designed for a specific domain.

Definition 2.7 (Well-Typed Term Graph)
A typed term graph G is ***well-typed*** *with respect to a term graph language $\mathcal{T}$ if for every node $n : \mathcal{N}$ of G there is a typing element τ for $L.n$ and a homomorphism from τ into G that maps a source node of τ to n.* □

In Figure 2.3 a typing element representing function abstraction is shown. This brick declares that λ is of type `a->b`, if the successor `MVAR` is of type `b` and the bound variable `x` is of type `a`.

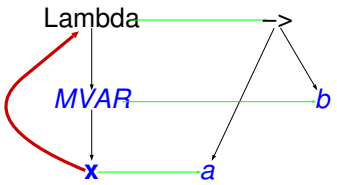


Figure 2.3: Typing element for the function abstraction

In HOPS only three bricks were "hard-coded": zero-ary bindable variables and n-ary ($n : \mathbb{N}$) metavariables in the object layer, and zero-ary metavariables in the type layer. During the development of our mobile agents we have added two additional hard-coded bricks to the object layer: the node metavariable `NVAR` and the distinct edges metavariable `DEMVAR`. The node metavariable is a metavariable with an image interval consisting of exactly one node, the top node, and no inner nodes.

The distinct edges metavariable is a metavariable with an image interval where all edges to successors of the `DEMVAR` are represented with edges that are distinct from

each other in the image of the term graph homomorphism. In Figure 2.4 three term graphs are shown. Since the left one uses the `DEMVAR`, it is only possible to find a matching homomorphism from the left one to the one in the middle, but not to the graph on the right-hand side of Figure 2.4, since the right graph does not contain two different edges from the matching of the metavariable to the node with the label `4`.

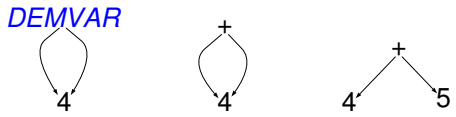


Figure 2.4: Three graphs for the illustration of the `DEMVAR`

When using a normal metavariable instead of the distinct edges metavariable in the graph on the left-hand side of Figure 2.4 a matching homomorphism from this term graph to the right one can be constructed, too. In this case, both outgoing edges are represented by the single edge from `+` to `4`.

Kahl (1998) has shown that for every DAG we are able to use in HOPS there is a principal type in the following sense: Among all well-typed graphs with isomorphic object layers there is always one from which there is a homomorphism to every other. In HOPS the principal type of a DAG is calculated automatically at each modification of the graph.

2.4 Term Graph Transformation

HOPS supports two different operations on term graphs: Application of a transformation rule and (maximal-)identification. In the following we only describe the application of a rule from left to right. A rule is applied from right to left in the analogous way. Maximal-identification is the recognition of common sub-DAGs and their identification and is introduced at the end of this section.

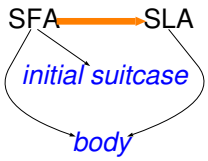


Figure 2.5: Example rule

As mentioned before, a transformation rule is a term graph with an additional arrow connecting the roots of the left- and right-hand sides. In Figure 2.5 an example of a

rule is shown. The left-hand side is the source of the orange respectively thick, medium grey arrow with the label `SFA`. The right-hand side is the sink of this arrow with the label `SLA`. The rule can be used to transform a stateful agent (`SFA`) into a stateless agent (`SLA`) and vice versa. It is explained in Section 5.1.

A rule is applicable from left to right to a node in an application graph, if and only if there exists a homomorphism from the left rule side to the application graph. If and only if there exists a homomorphism from the right-hand side of the rule to the application graph the rule is applicable from right to left. Such a homomorphism is also called matching-homomorphism or matching. If there is more than one matching for a rule side, the user has to choose one of them interactively.

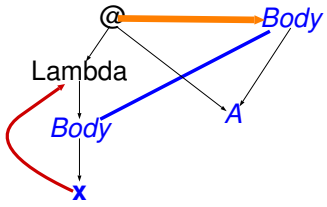


Figure 2.6: Rule for beta reduction

In Figure 2.6 the rule for beta reduction in the λ calculus is shown. In Figure 2.7 the matching homomorphism from the left-hand side of this rule to an application graph representing the term $(\lambda\, x.x + 3)\ 1$ is shown. The term graph homomorphism is illustrated by the black arrows from left to right.

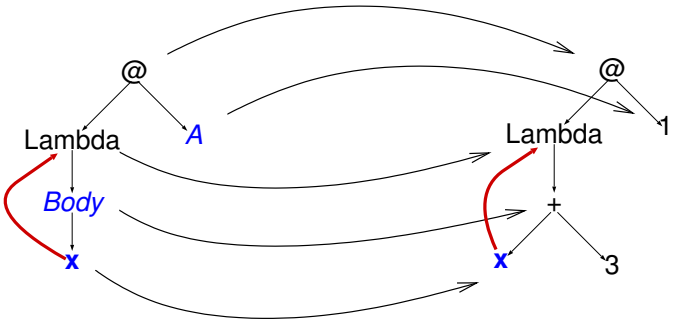


Figure 2.7: Term graph homomorphism from the left-hand side of the beta reduction rule to an application graph

The matching homomorphism shown in Figure 2.7 maps the constant nodes, namely λ and @, to the corresponding constant nodes in the application graph. The metavariable `A` is mapped to the interval consisting of the top node `1` with an empty lower border and no inner nodes. The top node of the interval on which the metavariable `Body` is mapped, consists of the top node `+` and the lower border consisting of the bound

variable `x`. The inner node is the node with the label `3`. The bound variable `x` is mapped to the bound variable in the application graph. Since the above matching homomorphism exists, the rule is applicable from left to right to the application graph. Applying the rule means replacing the image of the left-hand side of the rule with the "corresponding" right-hand side of the rule. By the corresponding right-hand side of the rule we mean the constant parts of the right-hand side of the rule and the images of all metavariables which also are part of the left-hand side of the rule or have a variable identity to one of the metavariables of the left-hand side. If a metavariable only occurs in the right-hand side without any variable identity relation to any metavariable which is part of the left-hand side, this metavariable will be inserted as it is into the term graph.

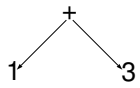


Figure 2.8: Application graph after applying beta reduction rule

In Figure 2.8 a term graph is shown, which is the application graph from Figure 2.7 after applying the rule shown in Figure 2.6 from left to right. The image of `Body` is the interval consisting of the top node `+` and the inner node `3`. The bound variable `x` has been replaced by the image of the metavariable `A`.

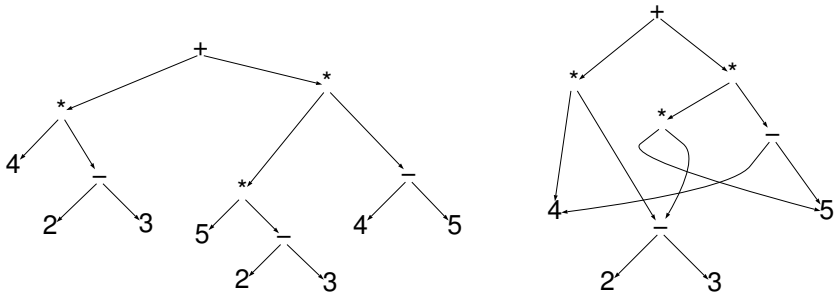


Figure 2.9: Two graphs representing the same term, one without sharing and one as maximal-identified graph

As mentioned above, maximal-identification is the recognition of common sub-DAGs and their identification. In Figure 2.9 two term graphs representing the same term are shown. In the graph on left-hand side common sub-DAGs, e.g., the sub-DAGs representing the term $2 - 3$, are not shared. The graph on the right-hand side uses sharing for all common sub-DAGs. Since it is not possible to find more common sub-DAGs which are not already shared, the graph on the right-hand side of Figure 2.9 is called **maximal-identified**.

2.5 Transformation Strategies and Term Graph Patterns

Strategies and term graph patterns can be used for automating term graph transformations. In this section, only those parts of these concepts that are necessary for understanding the following chapters are introduced in an informal manner. Defining them formally goes beyond the scope of this introduction to HOPS and is not essential for this thesis. Furthermore, the syntax used for strategies in this approach is simplified in contrast to the syntax introduced in Derichsweiler (2002), where a more complete introduction and formal definitions of those concepts can be found.

A **transformation strategy** is a root-DAG consisting of nodes with the following labels:

- `Units` *no pat*, consisting of
  - *no* $\in$ {`NO`, `TD`, `TDLR`, `BU`}; the navigation order which is *node-only*, *top-down*, *top-down-local-restart*, or *bottom-up*, and
  - *pat*, the set of term graph patterns used in this strategy.

  `Units` *no pat* is only allowed to be the label of the root of the strategy DAG. Furthermore, the root has to be a node with this label.
- `Ref` *sn*, the reference to another strategy, where *sn* is the name of that strategy.
- `Rule` *rs dir*, consisting of
  - *rs*, the set of rules with the label *rs*
  - *dir* $\in$ {*LR*, *RL*, *LRRL*}, the allowed directions of the applications of the rules in *rs*, which can be *left-to-right*, *right-to-left*, or *left-to-right-and-right-to-left*.
- `Search` *nl dir*, consisting of
  - *nl*, the node label, and
  - *dir* $\in$ {*Up*, *Down*, *Node*}, the direction in which the node label is searched.
- `SeqComp`, the sequential composition of its two successors,
- `Alternative`, which uses its second successor if and only if the first successor has not led to any transformation of the DAG.
- `IfThen`, which uses its second successor if and only if the first successor has led to a transformation of the DAG.
- `MaxId`, the maximal-identification,
- `Star`, which repeats its successor infinitely.

- `Finish`, which can be used to stop the strategy immediately.

A transformation strategy is applied to a node of a term graph and is started at its root. The strategies used in Chapter 7 and Chapter 9 are all explained explicitly in those chapters.

A **term graph pattern** is a term graph which can be used to specify where a particular transformation has to be done. In this approach the only purpose where term graph patterns are used, is to be able to apply a transformation strategy to the top-node, but transform only a part of the DAG. Term graph patterns are used in Section 9.4 and in Section 9.6.

2.6 Code Output

The implementation of HOPS used for the prototypical implementation of the approach presented in this thesis has a simple code output mechanism. The code output mechanism can be applied to a node of a DAG in order to generate code for the sub-DAG induced by this node. Therefore, a code string has to be defined for each brick that occurs in this sub-DAG. Besides constant parts of a string it is possible to use special strings to denote the code generated for successors and the like. In Table 2.1 all special strings are shown.

Code string	Denotation
`$<number>$`	code generated for successor `<number>`
`$B<number>$`	code generated for bound variable `<number>`
`$DL<name>$`	generates a unique number which is bound to `<name>`
`$L<name>$`	uses previously generated unique number bound to `<name>`
`$T$`	code generated for the type of the brick

Table 2.1: Special strings for the code generation

For example, the code in the programming language Haskell which is generated for the node λ in the DAGs shown in Figure 2.1 can be denoted by "`\$B1$ -> $1$`". This means, the code for the first bindable variable and for the first successor is generated, and is used to replace `$B1$` respectivley `$1$` in the above code string.

3 Mobile Agents

At least two categories of agents exist: Intelligent agents and mobile agents. While intelligent agents (Wooldridge and Jennings, 1994) have been studied mainly by the AI[1] community, mobile agents are originated in distributed computing and programming language research communities. Mobility and intelligence in the context of agents are considered to be orthogonal. An intelligent agent may be mobile or not and a mobile agent may be intelligent or not. The focus of the approach presented in this thesis is on agents that are mobile following the definition presented in Gray (1997):

> *A mobile agent is an autonomous program that can migrate under its own control from machine to machine in a heterogeneous network.*

Autonomous program means that the program does not depend on any user interaction while executing. All machines have to provide a mobile agent execution environment, which is often called agent platform. Obviously, it is possible to add code to a mobile agent which makes the mobile agent intelligent in any way. Nevertheless, this approach does not provide any special feature to aid the development of intelligence.

Thomsen and Thomsen (1997) stated that mobile agents are the new paradigm in computing and there will be "one of the most important paradigm shifts in computing since object oriented methods and client/server based distributed systems". Although, no killer application[2] for mobile agents has been identified until now, there are plenty of applications that benefit from using mobile agents, e.g., distributed information retrieval, information dissemination, monitoring and notification, as well as personal assistance.

[1] Artificial intelligence (AI) is the science and engineering of making intelligent machines, especially intelligent computer programs.

[2] Killer application or "killer app" is a buzzword that describes a software application that surpasses all of its competitors.

3.1 Alternatives to Mobile Agents

Alternatives to mobile agents for client/server interaction can be divided into two classes (Chess et al., 1994), asynchronous protocols and synchronous protocols. An example for asynchronous protocols is messaging, an example for synchronous protocols is remote procedure call. In both cases, only data is transferred between the client and the server. The procedures that handle the data are stationary on the server respectively on the client.

The remote procedure call (RPC) (Birrell and Nelson, 1983) is an extension of the traditional procedure call mechanism. The client application opens a communication channel to the server process and passes the parameters to the server process. Interface routines are used to marshal the parameters into a form that is suitable for transmission and to unpack them after the transmission. The server processes the parameters and returns the result through the communication channel to the waiting client application. High efficiency and low latency are the strengths of remote procedure calls.

Messaging is an outgrowth of electronic mail systems and earlier distributing computing schemes with communications done via pipes or files. A message composed by the client application typically consists of tagged or structured text. This message is delivered to a software processor which is appropriate for this kind of message and is mostly indicated in the message header. A popular example is the Simple Mail Transport Protocol (SMTP) (Postel, 1982). Messaging is asynchronous, i.e., a client which has handed of the message continues its execution. The main advantage of messaging is its robustness, particularly over wide area networks (WANs).

3.2 Properties of Mobile Agents

Individual advantages of mobile agents have been already pointed out by Chess et al. (1994):

- *Reduction of network traffic.* In traditional client/server environments the client fetches a possibly huge amount of data to filter out a particular needed small part. With the utilisation of mobile agents filtering is already done at the server.
- *Better support for mobile clients.* Mobile devices are not permanently connected to the internet. A mobile client can launch the agent, disconnect, and receive the returning agent during a subsequent connection session. Mobile clients are often

connected through a relatively low bandwidth. Furthermore, mobile clients have limited storage and processing capacity.

- *Asynchronous interaction.* Although all message-based systems, e.g., the Simple Mail Transport Protocol (Postel, 1982), provide asynchronous interaction, the utilisation of mobile agents allows to process arbitrary computations asynchronously.

- *Queries and transactions can be more robust.* RPC computation was developed for LAN[3]-based systems with strong assumptions about the integrity of communications and the availability of the server. Nowadays, RPC is often realised over WANs which results in less reliable connections. Using mobile agents moves the computations over the WAN to the server and thus leads to more robust queries and transactions.

- *Avoid the need to preserve process state.* A mobile agent migrates with its explicit state from platform to platform. This explicit state is often called its suitcase. The mobile agent, and not the agent platform or the underlying operating system, is responsible for its own state.

- *Remote real-time control.* If the transmission latency in a network is to high to achieve real-time constraints, a mobile agent can be executed locally on the remote machine.

Each of the above properties, except for the remote real-time control, can be accomplished with systems that do not use mobile agents, but a mobile agent framework addresses all of them at once (Chess et al., 1994).

Obviously, a mobile agent environment has also some disadvantages. The main disadvantage, besides the application-specific disadvantages of mobile agents, e.g. transmission efficiency of a courier agent compared to a simple SMTP mail message, is the need for a highly secure agent execution environment. Two main security concerns have been identified (Gray et al., 1998): Protecting the mobile agent and protecting the agent platform. While this first approach does not focus on security, the minimal requirement, protecting the agent platform, has been satisfied by using a strongly typed language, namely Haskell. A mobile agent migrates as Haskell source code and thus will be type-checked and compiled before execution.

[3]LAN: Local Area Network

Usual protection techniques are the sandbox model, code signing, and proof-carrying code. The sandbox model (Fritzinger and Mueller, 1996) provides strict limitations on what system resources the mobile agents can request or access. The compliance with those limitations is verified before the mobile code will be executed. Interactions with the outside world are mediated by a monitor. Code signing is a concept which is orthogonal to the sandbox model. Code is divided into two classes, trusted and untrusted. Trusted code has to be signed by a trusted entity and is allowed to run outside the restrictions of a sandbox. The proof-carrying code (Necula and Lee, 1997) technique constructs a proof that guarantees that the code does not violate some safety policies. The proof which is delivered with the code can be verified before the code will be executed. The utilisation of proof-carrying code within a heterogeneous environment is not possible, because this technique is tied to the hardware and the operating system of the agent platform.

3.3 Applications for Mobile Agents

Applications that benefit from using mobile agents have already been identified by several authors (Sanneck et al., 2002) (Kotz and Gray, 1999) (Lange and Oshima, 1999). Examples of those applications are:

- *Distributed information retrieval.* Move the computation which searches for data and filters data to the servers instead of fetching a large amount of data to the client.
- *E-commerce.* Real-time access to remote resources such as stock quotes or agent-to-agent negotiation.
- *Personal assistance.* Perform tasks independently from network connectivity of users.
- *Telecommunication network services.* Dynamic network reconfiguration and user customisation.
- *Workflow applications and groupware.* Mobile agents may contain workflow items.
- *Monitoring and notification.* Mobile agents can monitor a given information source and notify the user if certain kinds of information become available.

- *Information dissemination.* Mobile agents can be used to disseminate news or automatic software updates.

3.4 Standardisation

There are two standards for mobile agent technology: The Mobile Agent System Interoperability Facility (MASIF) developed by the Object Management Group (OMG) and the specifications published by the Foundation for Intelligent Physical Agents (FIPA). OMG is a non-profit organisation which was formed in 1989. OMG's work includes CORBA, a standard for distributed software systems. FIPA is also a non-profit organisation with a focus especially on standards for agent technology.

MASIF is based on agent platforms and enables agents to migrate from one platform to another. Mobile agents have to migrate via CORBA interfaces. FIPA aims at enabling the intelligent agents interoperability via standardised agent communication and content languages. The approach presented in this thesis is an academic prototype which does not use CORBA for migration, in particular because it is not available in Haskell yet. Since this approach focuses on mobile agents that are not intelligent per se the FIPA standard has also not been taken into consideration.

3.5 Mobile Agent Languages

Nearly all mobile agent systems use imperative programming languages, most notably Java (Tripathi et al., 2002) (Tripathi et al., 1999) (Karjoth et al., 1997), C/C++ (Lucco et al., 1995) (Peine and Stolpmann, 1997) (Johansen et al., 1996), and various scripting languages (White, 1994) (Gray, 1997). Functional programming languages are used only in a few systems (Knabe, 1995) (Gray, 1997).

Frederick Knabe distinguishes in his PhD thesis (Knabe, 1995) mobile agent programming languages from other programming languages by naming some essential properties which are a minimum for agent applications in real distributed environments:

- Support for manipulating, transmitting, receiving, and executing code-containing objects
- Support for heterogeneous computer systems

- Performance sufficient to meet the needs of applications

Knabe (1995) also adds some desirable properties for improving agent programming, namely remote resource access, strong typing, automatic memory management, stand-alone execution, independent compilation, and security. Strong static typing is very important when programming with agents, since it is extremely difficult to debug distributed programs. His proposed language is based on Facile (Thomsen et al., 1993), a higher-order, mostly functional language that integrates support for concurrency and distribution. In Facile it is possible to send and receive function closures and communicate through channels.

The modifications done by Knabe (1995) include a code representation that can be executed on heterogeneous architectures, dynamic linking on the execution site and lazy compilation of received agents. Since remote resource access is strongly typed, the type of those remote resources has to be known during agent programming, and every platform has to provide all remote resources with this type. In an open environment this is too restrictive.

The approach presented in this thesis does not only use a strongly typed functional programming language, namely Haskell, for the execution environment, it uses a typed representation already at the time of developing a mobile agent in HOPS (see Chapter 2). Remote resources, which we call possibly-provided functions, may be available on a platform, but they do not have to be available on all platforms. Our approach provides support for removing possibly-provided functions temporarily from the mobile agent code on an agent platform which does not provide the particular function. Furthermore, it is also possible to migrate only to those platforms that provide the necessary functions.

In order to be able to remove non-available functions temporarily the mobile agents migrate as Haskell source-code. This means each mobile agent is type-checked and compiled on any agent platform before it is executed. Beneath the drawback of generating some overhead in agent execution this makes it possible to use the type system of Haskell for the protection of the agent platform. Therefore, the function representing the mobile agent is encapsulated in the `runAgent` function which uses rank-2 polymorphism in one of its arguments. This makes it impossible to use arbitrary I/O functions in the mobile agent code. More detailed information can be found in Section 8.2.1.

Our mobile agents use so-called weak mobility, i.e., a mobile agent does not resume its execution from the instruction following the migration — this would be called strong

mobility — it always "restarts" the entire mobile agent function. We use weak mobility for our first approach, since it appears in the functional programming context to be the more natural way to express mobility. Bettini and Nicola (2001) have introduced a purely syntactic translation from strong mobility to weak mobility. This translation can be integrated into a future version of our development system.

4 Design Overview

The mobile agent programming and execution environment is based on the Higher Object Programming System HOPS and the purely functional programming language Haskell. The main objective is to develop a working prototype of a mobile agent programming and execution environment based on term-graph transformation and functional programming. This work is considered to be a case study and a first step towards a fully-fledged mobile agent programming and execution environment. In the following two types of notation can be found: Declarations and expressions written in Haskell (Peyton Jones et al., 2002) and directed acyclic graphs (DAGs) as used in HOPS (see Chapter 2).

In Section 4.1 the terms which are used in the following chapters are defined. The terms agent platform, home platform, agent, mobile agent, and suitcase can also be found in other approaches whereas the distinction between the local and the global suitcase, the terms primitive agent, stateful agent, stateless agent, agent combinator, platform agent, and value agent are unique in the present approach. Furthermore, the concept of possibly-provided functions is an outcome of this work and cannot be found in any other mobile agent environment. In Section 4.2 the objectives of the approach towards the first mobile agent development and execution environment based on term graph transformation and functional programming are described. Section 4.3 gives a sketch of the realisation. The author has developed both, the mobile agent programming environment based on HOPS and the mobile agent execution environment based on Haskell, from scratch in a joint development.

4.1 Definitions

As already mentioned in Chapter 3, the following definition for a mobile agent, taken from Gray (1997), is used as a starting point for this approach:

> *A mobile agent is an autonomous program that can migrate under its own control from machine to machine in a heterogeneous network.*

Obviously, a mobile agent needs a run-time environment on each machine: The agent platform.

Definition 4.1 (Agent Platform, Home Platform)
*An **agent platform**, or platform for short, is a mobile agent execution environment. The platform where a user starts a mobile agent is called the **home platform** of the mobile agent.* □

Since an agent platform is implemented as server process on a machine, it is possible to run more than one platform on each machine. On the other hand, each agent platform runs on exactly one machine. The implementation of agent platforms is called the Haskell Mobile Agent Platform (HaMAP) and is introduced in Chapter 8.

Definition 4.2 (Primitive Agent)
*A **primitive agent** of type* `Agent a` *is a function returning a value of type* `a`*, where* `a` *is a Standard Haskell Type (Peyton Jones et al., 2002) excluding I/O types.* □

A primitive agent consists only of functions and values defined in the Haskell 98 report (Peyton Jones et al., 2002) except for functions and values with an I/O type. An agent cannot use I/O functions in the usual way. The only facility to do I/O within a mobile agent is by using platform functions or possibly-provided functions (see Definition 4.4). The mechanism to ensure that a mobile agent cannot do arbitrary I/O is described in Section 8.2.1.

Definition 4.3 (Stateless Agent, Stateful Agent, Local Suitcase)
*A primitive agent calculating a value which is independent from the value calculated on the previous platform is called **stateless agent**. A primitive agent calculating a value depending on the previously calculated value is called a **stateful agent**. The value of a stateful agent is also called its **suitcase** or **local suitcase**.* □

The services and information a platform provides to the mobile agent are encapsulated in possibly-provided functions and platform functions.

Definition 4.4 (Possibly-Provided Function, Platform Function)
*A **possibly-provided function** is a function that may be available on a platform; a **platform function** is a function that has to be available on each platform. A possibly-provided function returns a value of type* `Maybe a` *for arbitrary* `a`*. The set of*

platform functions consists of five functions, namely `migrate`, `getPFID`, `getPPFInfo`, `hasPPFs`, *and* `replaceCF`. □

Since a possibly-provided function returns a value of type `Maybe a`, it is possible to replace a function that is not available on a platform by a function returning the value `Nothing`. This is done by a preprocessor on the Haskell Mobile Agent Platform (see Section 8.2.1). With this concept, a possibly-provided function can be used within a mobile agent even if it is not available on an agent platform visited by the mobile agent.

With the concept of meta agent combinators, which is introduced in Section 5.6, it is possible to transform away parts of the agent's code depending on the availability of possibly-provided functions on the current platform. Without using meta agent combinators it is not possible to distinguish between the result `Nothing` caused by the fact that the possibly-provided function is not available and the same result caused by a temporary or request-dependend failure. However, this design decision has been made in order to provide possibly-provided functions following the principle of least astonishment by using a type of `a -> Maybe b` for a function for which the user expects a type of `a -> b`. The requirement to distinguish between non-availability and a failure would lead, for instance, to a function of type `Maybe (a -> Maybe b)`.

The platform function `migrate` is needed to migrate from the current agent platform to another platform. The platform identifier of the current platform is returned by the function `getPFID`. The function `getPPFInfo` returns a list of pairs consisting of platform identifiers and possibly-provided functions which are available on that platform. The function `hasPPFs` can be used to filter all platform identifiers of platforms providing a list of possibly-provided functions from a list of platform-identifiers, and the fifth function, `replaceCF` can be used to replace a code fragment of the mobile agent code with another code fragment. More detailed information about those functions can be found in Section 5.2 and in Section 8.1.2.

Definition 4.5 (Agent Combinator)
An n-ary function returning a value of type `Agent a` *for arbitrary* `a` *is called an (n-ary)* ***agent combinator****. At least, one argument has to be of type* `Agent b` *for arbitrary* `b`. □

The second sentence of Definition 4.5 is needed to ensure that primitive agents are not considered to be agent combinators. An example of an agent combinator is the pair agent combinator.

```
agentPair (agent1 :: Agent a) (agent2 :: Agent b) :: Agent (a,b)
```

The pair agent combinator combines two agents, one returning a value `x` of type `a`, the other returning a value `y` of type `b`, to an agent returning the pair `(x,y)`.

We now introduce the agent.

Definition 4.6 (Agent, Value of the Agent)

1. *A primitive agent is an agent.*
2. *The application of an n-ary agent combinator to n appropriately chosen arguments is an agent.*
3. *A platform function is an agent.*
4. *A possibly-provided function is an agent.*

An agent which returns a value `v` of type `a` is of type `Agent a`, `v` is called the value of the agent. □

Mobility is introduced by the mobile agent combinator.

Definition 4.7 (Mobile Agent, Platform Agent, Value Agent)
*A **mobile agent** consists of two agents: The **platform agent** and the **value agent**. The platform agent is an agent returning a value of type `[PFID]`. The value of the value agent is called the the value of the mobile agent.* □

A value of type `PFID` is called **platform identifier**. The platform agent is responsible for the calculation of the next platform to which the agent migrates. The platform agent returns a list of platforms where the mobile agent tries to migrate to. If the migration to the head of the list fails, the mobile agent tries to migrate to the next platform in the list, until the migration to a platform succeeds or the attempt to migrate to all platforms fails. If migration has failed for all platforms in the list, the current agent platform will send the mobile agent back to the home platform.

The value agent is responsible for the value the mobile agent calculates on each platform.

Definition 4.8 (Global Suitcase)
*A mobile agent has exactly one **global suitcase**. The global suitcase contains all local suitcases and the value of the value agent.* □

The mobile agent programmer has to define only local suitcases. The term-graph transformation in HOPS generates a global suitcase containing the local suitcases. Furthermore, it will be ensured by the transformations that the value of the mobile agent is part of the global suitcase.

For a mobile agent with a value agent calculating a value `val` of type `a` and a global suitcase `sc` of type `b`, there exists a function `valueFromSuitcase` of type `b -> a` for which the following equation holds:

```
valueFromSuitcase sc == val
```

This function to extract the value from the suitcase will be generated by the transformations in HOPS. It can be used to present the value of the mobile agent to the user once the mobile agent has returned to the home platform.

4.2 Objectives

The main objectives of the approach to develop a mobile agent system with HOPS and Haskell are presented below.

Separation of Concerns

Separation of concerns is a concept to encapsulate those parts of software that are relevant to a particular concern. Furthermore, it refers to the ability to identify and manipulate those parts. With an appropriate separation of concerns software complexity can be reduced and reusability of software can be improved.

A kind of separation of concerns used in functional programming languages are combinator libraries, for instance parser combinators (Swierstra and Duponcheel, 1996) (Leijen, 2000). A main advantage of combinator libraries is compositionality, and thus greater modularity. With combinators it is possible to "combine" manageable pieces of code to a complex function. A manageable piece of code should be small enough to review and should be limited to only one concern, or even some part of a concern.

In the context of this approach the following aspects help to separate the concerns and make programming of mobile agents less error-prone:

1. The calculation on a platform and the calculation of the next platforms to visit is split into the value and the platform agent. Both parts will be transformed into one function.

2. An agent can be decomposed into smaller agents. These smaller agents can be composed using agent combinators.

3. Only local suitcases have to be defined by the mobile agent programmer. The global suitcase and the function to extract the value from the global suitcase are generated automatically.

Possibly-Provided Functions

An agent platform has to provide the set of platform functions (see Definition 4.4) beneath Haskell 98 (Peyton Jones et al., 2002) without I/O. Additional functionality a platform offers to a mobile agent is encapsulated in so-called possibly-provided functions (PPF). The concept of possibly-provided functions has been developed by the author and cannot be found in any other mobile agent environment. Since possibly-provided functions are Haskell 98 functions everything that is programmable in Haskell can be used in possibly-provided functions. Although, the main purpose of possibly-provided functions is to encapsulate functions that return some information, it is possible to define functions that change the state of the agent platform. Such a function can be, for instance, a function for booking a journey. All possibly-provided functions return their value `x` of type `a` encapsulated in the value `Just x` of type `Maybe a`. This way, it is possible to use a strongly typed execution environment without the need to provide every possibly-provided function on each agent platform. If a possibly-provided function `ppf` is not available a platform `pf`, the preprocessor of `pf` replaces `ppf` with a function returning the value `Nothing`.

Furthermore, with the concept of possibly-provided functions it is possible to migrate only to platforms that provide the set of possibly-provided functions needed. With the meta agent combinators introduced in Section 5.6 it is also possible to calculate values only if a specific set of possibly-provided functions is available, or to use an alternative code fragment if not all necessary possibly-provided functions are available. Moreover, the utilisation of a preprocessor to replace possibly-provided functions provides the opportunity to replace a non-available possibly-provided function `ppf1` with a possibly-provided function `ppf2` returning the appropriate value. As the same behaviour can be achieved by defining a possibly-provided function `ppf2` which uses `ppf1`, this allows to

minimise the number of possibly-provided functions and thus to make the code of the platform more manageable. Since all I/O is done using possibly-provided functions, those functions are the only parts of the agent platform that may contain security holes.

Modularity, Extensibility and Reusability

In the majority of cases, a mobile agent is a piece of software which is used only a few times or even only once. Therefore, a mobile agent programming environment should provide a facility to adapt existing agents for new tasks and to easily reuse parts of existing mobile agents when developing a new one. In the context of functional programming, this requirement is almost fulfilled by using a combinator library. With term graph transformation it is even possible to remove unwanted parts of a reused agent automatically.

By using HOPS it is also straight-forward to extend the domain-specific language used with some new agent combinators and the transformation rules needed for converting the new combinator into accurate Haskell. A mobile agent in Haskell is not a complete Haskell module, it is only one function containing only Haskell 98 without I/O, platform functions and possibly-provided functions. By this means, the Haskell code of a mobile agent can be very complex, because it is not possible to define small functions and use them by name in the mobile agent code. Nevertheless, it is possible to develop an agent in small pieces in HOPS and using appropriate transformation rules to inline[1] these functions.

Strong Typing

The functional language Haskell, which is used in this approach, is strongly typed. This eliminates a huge class of easy-to-make errors at compile time. Since a mobile agent migrates in its Haskell source code representation, it has to be compiled on each platform. Using a strongly typed language, which is type-checked and compiled before executions on each platform, adds security for the agent platforms (see below).

For the mobile agent programmer type-checking just before compilation is not sufficient, because the mobile agent will not be compiled and type-checked on the home platform. Obviously, it is possible to compile the mobile agent on the home platform

[1] Inlining functions is an optimisation technique used by most compilers.

before migrating it to an agent platform. But since the mobile agent code which will be compiled and executed on an agent platform depends on the availability of possibly-provided functions on this particular platform, it would be necessary to implement prototypes of all possibly-provided functions appearing in the mobile agent code and to type-check the mobile agent code in all different forms.

Fortunately, HOPS provides online type-checking while programming. In particular this means that every mobile agent DAG which can be built and transformed in HOPS is well-typed at any time. It is simply not possible in HOPS to create a DAG containing a type-error.

To sum up, the utilisation of HOPS and Haskell provides a strongly typed development and execution environment which is essential in the context of mobile agent programming. A mobile agent failing on an agent platform due to errors which might be detected by type-checking is not acceptable.

Secure Execution

In the context of mobile agents, two main security concerns have been identified (Gray et al., 1998): Protecting the mobile agent and protecting the agent platform. While this first approach does not focus on security, the minimal requirement, protecting the agent platform, has been satisfied by using a strongly typed language. In particular this means, that rank-2 polymorphism in the type of a function which encapsulates mobile agent execution ensures that the mobile agent cannot do arbitrary I/O. More detailed information about this can be found in Section 8.2.1.

4.3 Realisation

The development and execution environment for mobile agents based on HOPS and Haskell has been developed from scratch by the author. It is the first approach which uses term graph transformation and functional programming for mobile agents. The development of a mobile agent can be divided into different phases which are illustrated in Figure 4.1.

First of all, a mobile agent developer has to define a mobile agent in terms of the User Interface Domain-Specific Language (UI-DSL). UI-DSL is a combinator language and is introduced in Chapter 5. In UI-DSL primitive agents are combined to complex

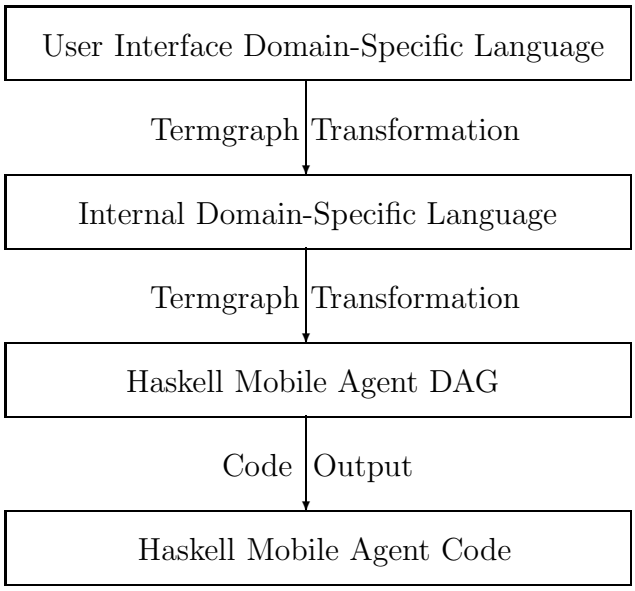


Figure 4.1: Steps from UI-DSL to Haskell Code

agents using agent combinators. An UI-DSL mobile agent is transformed into an internal mobile agent. I-DSL is the Internal Domain-Specific Language and is used to represent a mobile agent with a global suitcase containing the value of the value agent and all local suitcases of the UI-DSL representation of the mobile agent. The transformation from UI-DSL to I-DSL creates this global suitcase. Furthermore, all self-defined functions used in the UI-DSL mobile agent are expanded in the I-DSL mobile agent. I-DSL is introduced in Chapter 6 and the transformation from UI-DSL to I-DSL is presented in Chapter 7.

The internal mobile agent is then transformed into a monadic form which can be easily used to create standard Haskell code using the simple code output facility of HOPS. It is also conceivable to transform the I-DSL mobile agent into another form to create, e.g., Java code which can be used within existing mobile agent execution environments. Nevertheless, this thesis focuses on a purely functional development and execution environment. Furthermore, design aspects like possibly-provided functions are not available in this form in any other existing mobile agent environment known to the author. The Haskell Mobile Agent Platform (HaMAP) which has been developed especially for this approach is introduced in Chapter 8. Transformation from I-DSL to the monadic form and code output is described in Chapter 9.

The generated Haskell code is not in a form which can be compiled directly. The code has to be processed by a preprocessor on each mobile agent platform. This preprocessor replaces all non-available possibly-provided functions temporarily with a

function returning `Nothing`, respectively transforms the Haskell code in dependence on the available possibly-provided functions. The home platform adds some information like its identifier and an identifier for the mobile agent to the code before migrating the agent to the first agent platform. After the mobile agent gets back on the home platform a function, which is generated while transforming UI-DSL into I-DSL, is used to extract the value of the value agent from the global suitcase. By this means, the utilisation of the global suitcase is transparent for the user of the mobile agent.

5 User Interface Domain-Specific Language

The User Interface Domain-Specific Language (UI-DSL) for mobile agents is a term graph language, which has been developed in HOPS by the author. UI-DSL is a combinator language consisting of primitive agents (see Section 5.1) and agent combinators (see Section 5.4, 5.5, and 5.6). A mobile agent can interact with the agent platform only through a fixed set of platform functions and a variable set of possibly-provided functions (see Section 5.2), which can be different on each platform. The meta data of a mobile agent (see Section 5.3) is its run-time information, e.g., its home platform.

5.1 Primitive Agents

A primitive agent of type `Agent a` is a function returning a value of type `a`. The UI-DSL provides two slightly different primitive agents: The stateful agent (see Section 5.1.1) and the stateless agent (see Section 5.1.2).

5.1.1 Stateful Agent

The stateful agent `SFA` of type `Agent a` consists of a function `body` returning a value of type `a` which depends on the value of the evaluation of `body` on the previous platform. Therefore, the stateful agent has a local suitcase where the result of `body` is stored during the migration to another platform. This local suitcase is the state of the stateful agent. For the first evaluation of `body` an initial value of type `a` is needed. This initial value is called the initial suitcase of the stateful agent. The stateful agent can also be regarded as a recursive function with one recursive call on each platform. Figure 5.1 shows the HOPS declaration of the stateful agent.

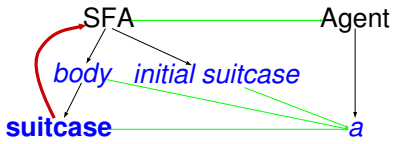


Figure 5.1: HOPS declaration of the stateful agent `SFA`

Figure 5.2 shows an example of a stateful agent which counts the number of visits to platforms. For this purpose it simply adds the integer 1 to the current suitcase starting with an initial suitcase containing the integer 0. Since this function will be evaluated once on each visit to a platform the suitcase always contains the number of visits to platforms so far.

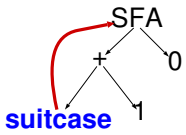


Figure 5.2: Stateful agent that counts the number of visits to platforms

5.1.2 Stateless Agent

A stateless agent `SLA` consists of a function `body` that does not depend on the previous calculated value. Therefore, a stateless agent does not need a suitcase and has no state. Figure 5.3 shows the HOPS declaration of the stateless agent.

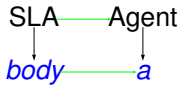


Figure 5.3: HOPS declaration of the stateless agent `SLA`

A stateless agent can be useful, for instance, to calculate a value that will be passed as an argument to another agent using the $\lambda_{\texttt{Agent}}$ combinator (see Section 5.4.2).

Obviously, a stateful agent which is not using its suitcase is equivalent to a stateless agent with the same `body`. Therefore, all stateful agents without a bound variable will be transformed to stateless agents by the rule shown in Figure 5.4.

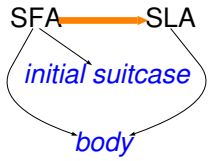


Figure 5.4: Rule to transform a SFA which does not use its suitcase into a SLA

5.2 Possibly-Provided Functions

A mobile agent that cannot interact with an agent platform is almost useless but, on the other hand, a mobile agent which is able to use arbitrary I/O functions on an agent platform is a high security risk. In this approach, where a mobile agent is migrated as Haskell source code which will be type-checked and compiled on each platform, it is possible to restrict the interaction with a platform to a set of special functions: The platform functions and the possibly-provided functions.

As already mentioned in Definition 4.4, a platform function must be available on each platform, whereas possibly-provided functions may be available only on a few platforms.

Two platform functions, namely `getPFID` and `getPPFInfo`, are available in the UI-DSL. The function `getPFID` returns the platform identifier of the current platform; `getPPFInfo` returns a list of pairs consisting of a platform identifier and a list containing information about the possibly-provided functions available on this platform. Figure 5.5 shows the HOPS declaration of `getPFID` and `getPPFInfo`. The other platform functions, namely `migrate`, `hasPPFs`, and `replaceCF`, are not intended to be used directly in UI-DSL. They are automatically inserted by the transformation of an UI-DSL mobile agent into an internal mobile agent (see Chapter 7).

Figure 5.5: HOPS declaration of `getPFID` and `getPPFInfo`

A possibly-provided function returns a value of type `Maybe a`. By this means, it is possible to replace all possibly-provided functions which are not available on the current platform automatically with a function returning the value `Nothing`. Thus, a mobile

agent programmer does not need to think about whether a particular possibly-provided function is available on each platform or not; he can simply use the possibly-provided function in his DAG. Since a possibly-provided function can be an arbitrary function, the author decided to abstain from introducing any possibly-provided function at this point. Particular possibly-provided functions are introduced when they are used.

5.3 Meta Data

The mobile agents meta data is a set of information like the home platform identifier and the mobile agent identifier. These meta data will not be generated by HOPS, they will be generated on the home platform when starting the mobile agent. Therefore, it is also called the mobile agents run-time information. Since this information is generated on the home platform, it is possible to use the generated code more than once with different agent identifiers and even from different home platforms. Although the meta data is not generated by HOPS it is possible to use access-functions in the UI-DSL, i.e., there are bricks like `homePF`, `agentID`, etc. in the UI-DSL.

5.4 Basic Agent Combinators

Two sets of basic agent combinators are part of the UI-DSL: Agent pairs and agent functions. Both sets are described in this section.

5.4.1 Agent Pairs

The first set of agent combinators consists of three basic combinators: $(,)_{\texttt{Agent}}$, $\pi_{\texttt{Agent}}$, and $\rho_{\texttt{Agent}}$. The agent pair combinator $(,)_{\texttt{Agent}}$ can be used to build an agent which returns the pair `(x,y)` from an agent returning `x` and an agent returning `y`. Figure 5.6 shows the HOPS declaration of $(,)_{\texttt{Agent}}$.

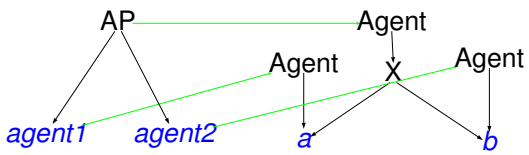


Figure 5.6: HOPS declaration of the agent pair combinator $(,)_{\texttt{Agent}}$

The agent pi combinator $\pi_{\texttt{Agent}}$ and the agent rho combinator $\rho_{\texttt{Agent}}$ are the usual projections. The following equations will hold for arbitrary `agent1 :: Agent a` and `agent2 :: Agent b`:

$$\pi_{\texttt{Agent}}(\texttt{agent1}, \texttt{agent2})_{\texttt{Agent}} \equiv \texttt{agent1}$$

$$\rho_{\texttt{Agent}}(\texttt{agent1}, \texttt{agent2})_{\texttt{Agent}} \equiv \texttt{agent2}$$

Rules to transform the left side to the right side of the above equations are automatically applied to make the mobile agents code smaller in size. Figure 5.7 and Figure 5.8 show the HOPS declaration of the $\pi_{\texttt{Agent}}$ and the $\rho_{\texttt{Agent}}$ combinator.

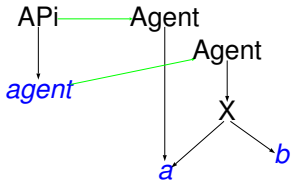


Figure 5.7: HOPS declaration of the agent pi combinator $\pi_{\texttt{Agent}}$

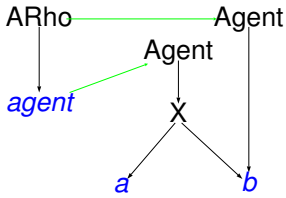


Figure 5.8: HOPS declaration of the agent rho combinator $\rho_{\texttt{Agent}}$

5.4.2 Agent Functions

The second set of combinators consists of two combinators: $\lambda_{\texttt{Agent}}$ and $@_{\texttt{Agent}}$. The agent abstraction combinator $\lambda_{\texttt{Agent}}$ can be used to build an agent which needs an input value of type `a` to calculate a value of type `b`. Figure 5.9 shows the HOPS declaration of $\lambda_{\texttt{Agent}}$.

In order to apply an agent abstraction of type `a -> Agent b` to an agent of type `Agent a` the agent application combinator $@_{\texttt{Agent}}$ has to be used. Figure 5.10 shows the HOPS declaration of $@_{\texttt{Agent}}$.

Figure 5.11 shows an example agent which calculates a list of pairs consisting of a consecutive number and a platform identifier. The consecutive number is calculated by the

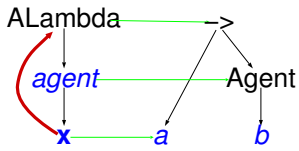


Figure 5.9: HOPS declaration of the agent abstraction combinator $\lambda_{\texttt{Agent}}$

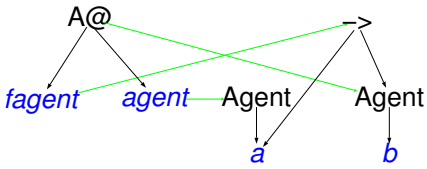


Figure 5.10: HOPS declaration of the agent application combinator $@_{\texttt{Agent}}$

agent shown in Figure 5.2; the platform identifier is returned by the platform function `getPFID`. The stateful agent on the left hand side prefixes a list with a value which is bound by the $\lambda_{\texttt{Agent}}$ combinator. This $\lambda_{\texttt{Agent}}$ is applied to the $(,)_{\texttt{Agent}}$ combinator.

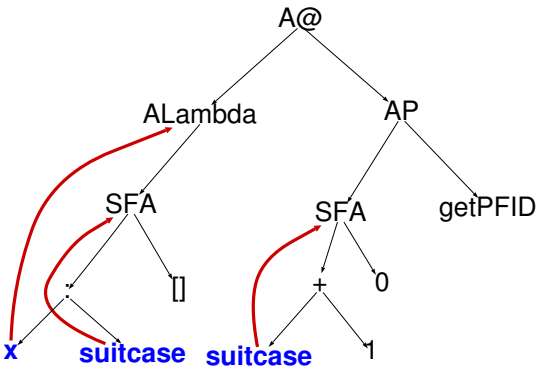


Figure 5.11: Agent which calculates a list of pairs consisting of a consecutive number and a platform identifier of a visited platform

5.5 Mobile Agents

As mentioned in Definition 4.7, a mobile agent consists of two agents: The value agent and the platform agent. The value agent is of type `Agent a` where `a` is not an I/O type. The platform agent is of type `Agent [PFID]`. `PFID` is the type of a platform identifier, thus a platform agent has to calculate a list of platform identifiers. This list of platforms is not necessarily the list of platforms which will be visited by the mobile agent before returning to the home platform. In fact the list calculated on the current platform specifies only where to migrate to from this platform. The

mobile agent tries to migrate to each platform beginning with the head of the list until migration to a platform succeeds. Execution of the mobile agent on the current platform immediately stops after a successful migration. The HOPS declaration of the mobile agent combinator is shown in Figure 5.12.

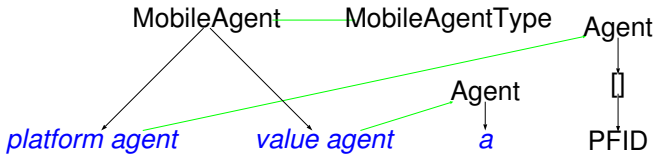


Figure 5.12: HOPS declaration of the mobile agent combinator

A mobile agent is of type `MobileAgentType`. A mobile agent is not of type `Agent a` to ensure that it cannot be used within other agent combinators. The mobile agent brick has to be the top node of a DAG representing a mobile agent. This design decision has been made because of the semantics of a mobile agent with its platform agent and its value agent. All agent combinators combine values of agents. Since the value of a mobile agent is the value of its value agent, there is no advantage in allowing mobile agents to be used within agent combinators. In addition, the UI-DSL does not provide any special combinator for combining mobile agents, because this does not lead to any benefit.

Figure 5.13 shows an agent which can be used as platform agent. The agent consists of two primitive agents, a stateless and a stateful agent, and uses two platform functions, namely `getPFID` and `getPPFInfo`. The stateful agent calculates a list of already visited platforms. The platform identifier of the current platform — the value of `getPFID` — will be prefixed to the list calculated on the previous platform. The stateless agent extracts all platform identifiers from the list returned by `getPPFInfo` and returns only those platform identifiers not included in the list of already visited platforms.

As already mentioned in Chapter 2, variables in HOPS are nameless, because variable binding, variable identity and scope is encoded explicitly. In particular, this means the node-names representing different variables in the DAGs may be identical. In Figure 5.13, for instance, there are four nodes represented by the string `x`, but each of them stands for a different variable since there is no variable identity edge between them. In Figure 5.14 a mobile agent is shown, which migrates from platform to platform until `getPPFInfo` returns information about already visited platforms only. On each platform the mobile agent prefixes a list of pairs with a pair consisting of a consecutive number and the platform identifier of the current platform. The platform agent is the agent shown in Figure 5.13; the value agent is the agent shown in Figure 5.11.

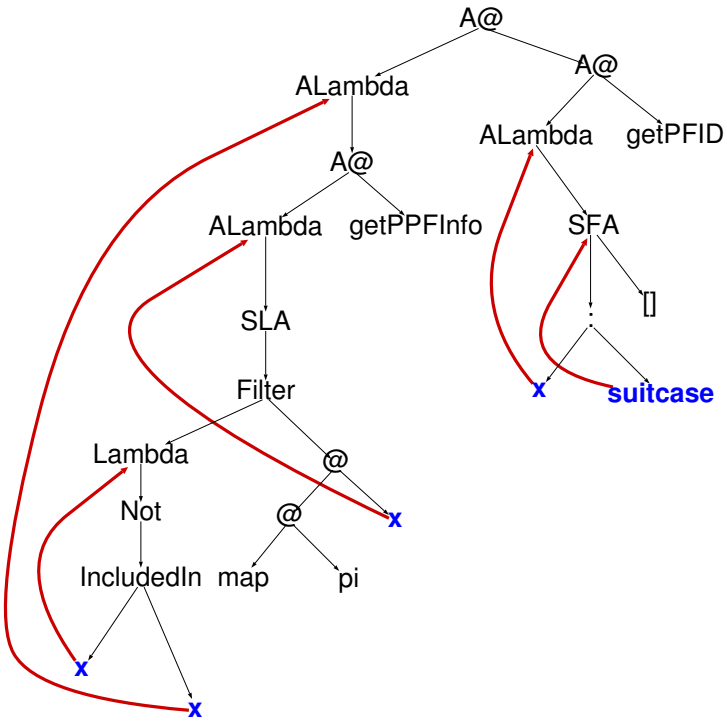


Figure 5.13: Agent which returns a list of not yet visited platforms

5.6 Meta Agent Combinators

With the concept of possibly-provided functions, which could be available on a particular agent platform or not, arises the need for being able to formulate calculations depending on the availability of possibly-provided functions. Within the approach presented here this is done in an additional layer, called the meta layer, with meta agent combinators. This means that in the Haskell implementation of the agent platform the functions of the meta layer are not part of the compilable agent code, which is called the code layer. These functions are embedded in the mobile agent code during migration and they are used to transform the code before type-checking and compilation. This transformation is done by the agent platform through a preprocessor. During the execution of a mobile agent, the mobile agent is also able to transform its code. These transformations will not take effect before migrating to the next platform.

In the UI-DSL meta combinators can be used directly in the DAG like any other agent combinator. In the remainder of this section the following UI-DSL meta agent combinators are introduced: The $\texttt{ALL}_{\texttt{PPF}}$ combinator, the $\texttt{OR}_{\texttt{PPF}}$ combinator, the `ValueMarker` and the `OldValueMarker` combinator, and the `Replace` combinator.

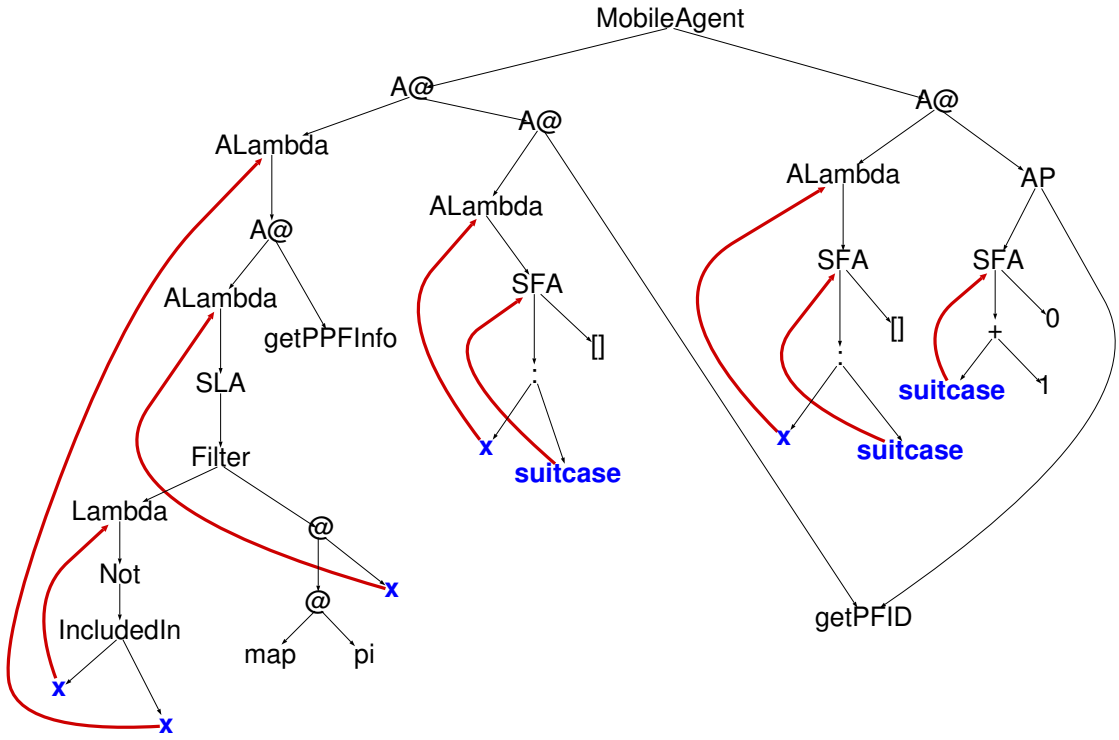


Figure 5.14: Mobile agent which calculates a list of pairs consisting of a consecutive number and the platform identifier of a visited platform while visiting platforms that are known by any other platform

The $\texttt{ALL}_{\texttt{PPF}}$ combinator can be used to calculate a value only if all needed possibly-provided functions in the sub-DAG induced by the $\texttt{ALL}_{\texttt{PPF}}$ brick are available on the current agent platform. For an agent `agent` of type `Agent a`, $\texttt{ALL}_{\texttt{PPF}}$ (`agent`) is of type `Agent (Maybe a)`. By this means, $\texttt{ALL}_{\texttt{PPF}}$ (`agent`) will be replaced by a function which returns the value `val` calculated by `agent` as `Just val` if all needed possibly-provided functions in `agent` are available. Otherwise, it will be replaced by a function returning the value `Nothing`. Figure 5.15 shows the HOPS declaration of the $\texttt{ALL}_{\texttt{PPF}}$ combinator.

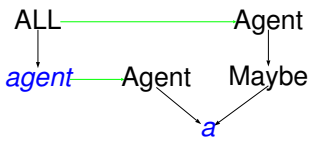


Figure 5.15: HOPS declaration of the $\texttt{ALL}_{\texttt{PPF}}$ combinator

With the $\texttt{OR}_{\texttt{PPF}}$ combinator it is possible to specify two alternatives for calculating a value. For `agent1` of type `Agent a` and `agent2` of type `Agent b` in $\texttt{OR}_{\texttt{PPF}}$ (`agent1`, `agent2`) the equation `a` $\equiv$ `b` must hold. $\texttt{OR}_{\texttt{PPF}}$ (`agent1`, `agent2`) will be replaced with `agent1` if and only if all possibly-provided functions needed in `agent1` are available.

Otherwise, it will be replaced with `agent2`. Figure 5.16 shows the HOPS declaration of the $\texttt{OR}_{\texttt{PPF}}$ combinator.

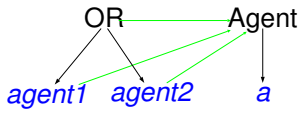


Figure 5.16: HOPS declaration of the $\texttt{OR}_{\texttt{PPF}}$ combinator

As mentioned before, in the UI-DSL there are only local suitcases, which are part of stateful agents. Thus, it is not intended to provide something like a "shared suitcase" directly for the $\texttt{OR}_{\texttt{PPF}}$ combinator. Nevertheless, it is possible to use a shared suitcase for both alternatives as the examples in Figure 5.17 and Figure 5.18 show. The agent shown in Figure 5.17 uses a shared suitcase whereupon the old suitcase value is not used by the agents in the $\texttt{OR}_{\texttt{PPF}}$ combinator. The value calculated by either `agent1` or `agent2` is used as input value for the $\lambda_{\texttt{Agent}}$ combinator. In this case the computations done in `agent1` and `agent2` are independent of the current suitcase value.

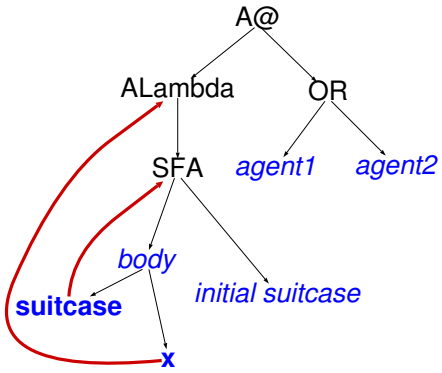


Figure 5.17: Example DAG with $\texttt{OR}_{\texttt{PPF}}$ using a shared suitcase

If the functions in `agent1` or `agent2` depend on the current suitcase value the combination of the `ValueMarker` and the `OldValueMarker` combinator has to be used. The combination of both is needed to avoid problems with bound variables which always have to be dominated by their binder. An edge from `agent1` to `ValueMarker` in Figure 5.18, for example, would be a domination violation with respect to the variable `x` bound by $\lambda_{\texttt{Agent}}$.

The semantics of `ValueMarker` and `OldValueMarker` is as follows: The value calculated by the first successor of `ValueMarker` on the current platform will be used to replace the `OldValueMarker` — the second successor of `ValueMarker` — in the mobile agent code and, thus, is used as old suitcase value on the next platform. The successor of

`OldValueMarker` is the initial value. In the example shown in Figure 5.18 this initial value is the initial suitcase of the stateless agent.

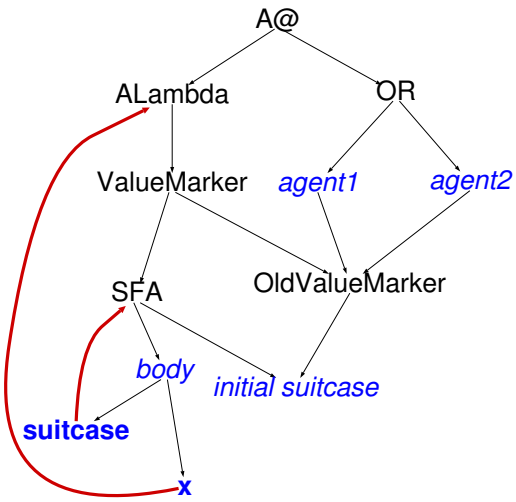


Figure 5.18: Example DAG with $\texttt{OR}_{\texttt{PPF}}$ calculating a value which depends on the shared suitcase value

The initial value can be automatically generated by HOPS using the initial suitcases of the involved stateful agents. Since it is possible to use the `ValueMarker` and the `OldValueMarker` at arbitrary positions inside a mobile agent DAG, the generation of the initial value does not necessarily result in a proper initial value. If, e.g., a stateless agent is part of the DAG induced by `ValueMarker`, the transformation would insert a polymorphic `undefined` brick which has to be replaced with a proper value by the mobile agent programmer. If the undefined part of the value is not used in any calculation, it is appropriate to leave the `undefined` brick in the DAG untouched. Figure 5.19 shows an example DAG with an automatically generated initial value including an `undefined` brick.

All meta agent combinators introduced so far are transforming the mobile agent code automatically. The `Replace` combinator can be used to transform the mobile agent code only if an arbitrary condition has been fulfilled. Therefore, the `Replace` combinator, which is shown in Figure 5.20, has three successors: The condition `cond`, the old code `old`, and the new code `new`. The old code is an agent of type `Agent a`. The value calculated by this agent is passed to the condition which results in an agent of type `Agent Bool`. The third successor, namely `new`, is also a function which will be applied to the value calculated by `old`. The result of `new` is an agent of type `Agent (ReplacementType (Agent a))`. A value of type `Replacement b` for arbitrary

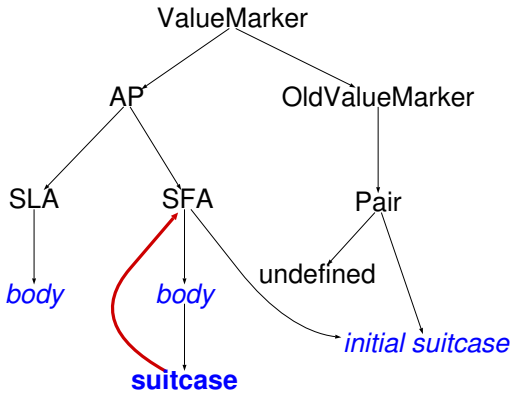


Figure 5.19: Example DAG with `ValueMarker` and generated initial value

`b` is called a replacement, and can be used to replace a code fragment of type `b`. By this means, the value of type `Agent (ReplacementType (Agent a))` is an agent returning a replacement to replace an agent of type `Agent a`, which is `old` in the context of the `Replace` combinator.

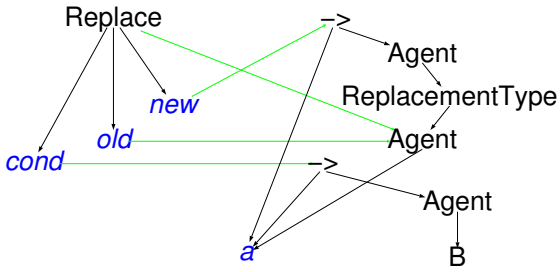


Figure 5.20: HOPS declaration of the `Replace` combinator

A replacement is a DAG induced by one of the three bricks shown in Figure 5.21. The `ReplacementValue` brick has to be used if the replacement is the value which is calculated by the successor of this brick. If the code fragment which is represented by the first successor should be used as replacement without being evaluated first, the `Replacement` brick has to be used. If only a part of the replacement should be calculated, the `Replacement@` brick can be used. The first successor of `Replacement@` of type `b -> a` will not be evaluated whereas the second successor will be evaluated. This results in a replacement consisting of the application of the function represented by the first successor to the value calculated by the second successor.

An example application of the `Replace` combinator is the `EvaluateOnce` combinator, which is defined by the HOPS rule shown in Figure 5.22. The `agent` is the agent that should be evaluated only once, i.e., once the value of `agent` is available, `agent` should be replaced with the value. The availability of the value is indicated by the value

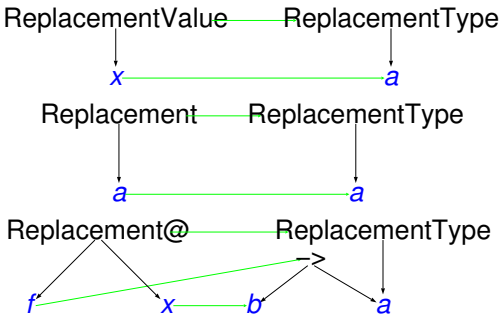


Figure 5.21: HOPS declaration of the `Replace` combinator

`Just x` for some `x`. Therefore, the condition agent is a stateless agent that applies the function `isJust`[1] to the value calculated by `agent`.

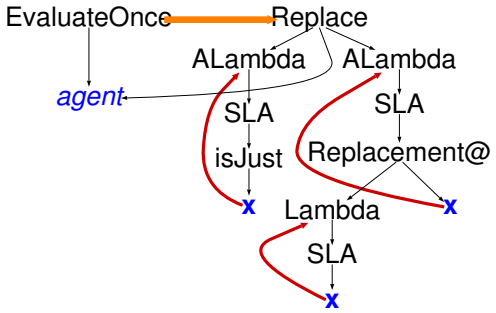


Figure 5.22: Rule for `EvaluateOnce`

The replacement agent consists of a stateless agent with a `Replacement@` brick as its successor. By this means, the replacement is a stateless agent returning the value which has been calculated by `agent`. An example agent using the `EvaluateOnce` combinator can be found in Chapter 10.

[1] `isJust :: Maybe a -> Bool`

6 Internal Domain-Specific Language

The Internal Domain-Specific Language (I-DSL) is used as interface between the UI-DSL (see Chapter 5) and the target programming language which will be generated. The purely functional programming language Haskell (Peyton Jones et al., 2002) is used here as target language. The main difference between an UI-DSL mobile agent and an I-DSL mobile agent is, that the former has only local suitcases whereas the latter has one global suitcase. The global suitcase contains all local suitcases and the value of the value agent.

The I-DSL agents and agent combinators are described in the following sections. The I-DSL primitive agent is the internal agent (see Section 6.1). All UI-DSL basic agent combinators are also available in the I-DSL. The `letInA` combinator, which is also introduced in Section 6.1 is used temporarily during the transformation from UI-DSL to I-DSL. In Section 6.2 the I-DSL meta combinators are presented. The suitcase handler (see Section 6.3) is used for the step-by-step generation of the global suitcase. In Section 6.4 the internal mobile agent is introduced.

6.1 Internal Agents

The I-DSL provides only one primitive agent: The internal agent. The internal agent of type `Agent a` consists of a body of type `a`. Thus, the internal agent is identical to the stateless agent (see Section 5.1.2). Although the internal agent and the SLA are identical, the internal agent brick is needed to indicate whether all primitive agents have been taken into account when transforming UI-DSL to I-DSL. If the mobile agent DAG still contains a stateless agent, the transformation to I-DSL has not been finished yet. In I-DSL the mobile agent has a global suitcase which is generated from the local suitcases. By this means, the state of an I-DSL mobile agent corresponds to its global suitcase. Since local suitcases do not exist in I-DSL an I-DSL primitive agent cannot have a state, anyway. Figure 6.1 shows the HOPS declaration of the internal agent.

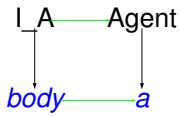


Figure 6.1: HOPS declaration of the internal agent

All UI-DSL basic agent combinators introduced in Section 5.4 are available in I-DSL as well. The `letInA` combinator is an I-DSL combinator which is only used during the transformation from UI-DSL to I-DSL. In Figure 6.2 the HOPS declaration of the `letInA` combinator is shown.

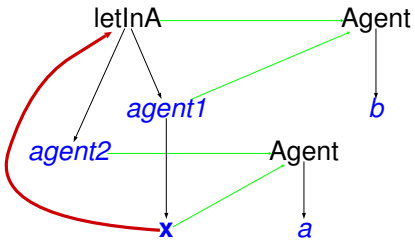


Figure 6.2: HOPS declaration of the `letInA` combinator

The semantics of the `letInA` combinator is similar to the semantics of the let expression in Haskell, `let x = agent1 in agent2`.

6.2 Possibly-Provided Functions and Meta Agent Combinators

For all UI-DSL meta agent combinators which are dependent on the availability of possibly-provided functions, there exist corresponding I-DSL meta agent combinators. The difference between the UI-DSL version and the I-DSL version of a meta combinator is that the latter has an additional successor of type `PPFName` which represents the necessary possibly-provided functions for this meta agent combinator. This information is required, because a possibly-provided function which is needed for a particular meta agent combinator is not necessarily part of a successor of this meta agent combinator after all transformations in HOPS took place. For instance, a possibly-provided function which is also used in another part of a mobile agent will be moved to a position above both occurrences before the Haskell code is generated. Figure 6.3 shows the HOPS declaration of the $\texttt{I\_ALL}_{\texttt{PPF}}$ and the $\texttt{I\_OR}_{\texttt{PPF}}$ combinator.

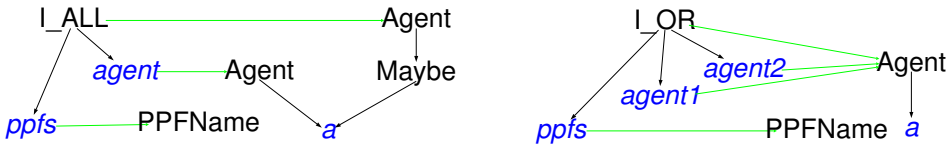


Figure 6.3: HOPS declaration of the `I_ALL`$_{\texttt{PPF}}$ and the `I_OR`$_{\texttt{PPF}}$ combinator

6.3 Suitcase Handler

As mentioned before, an I-DSL mobile agent has one global suitcase and contains no local ones. During the generation of the global suitcase, which is described in Chapter 7, an additional agent combinator is needed to handle the suitcase of an arbitrary agent. This agent combinator is called the suitcase handler. The HOPS declaration of the suitcase handler is shown in Figure 6.4.

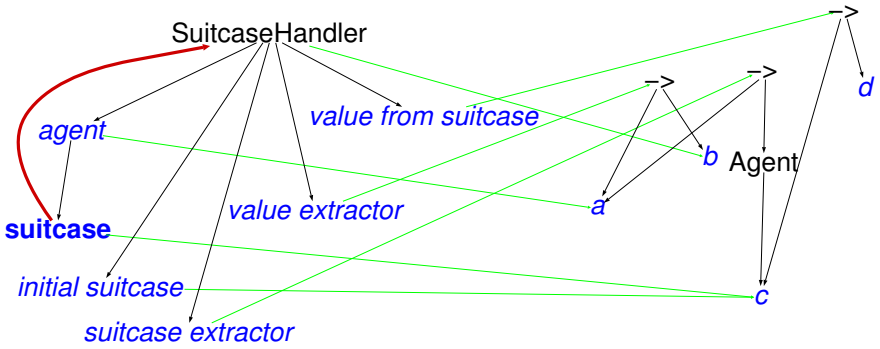


Figure 6.4: HOPS declaration of the suitcase handler

The main part of the suitcase handler is the `agent` which depends on the `suitcase`. The `initial suitcase` is the initial value of the `suitcase`. So far, the suitcase handler is quite similar to the UI-DSL stateful agent (see Section 5.1.1). The value calculated by `agent` contains the new suitcase and the value of the suitcase handler. Since `agent` is an arbitrary agent consisting of primitive agents and agent combinators, the suitcase and the value can differ from each other. To extract the suitcase and the value of the suitcase handler from the value of `agent` the `suitcase extractor` and the `value extractor` are used. Finally, `value from suitcase` is a function to extract the value from the suitcase. For an arbitrary `agent` it is possible that `value from suitcase` may not be defined, because the value of the suitcase handler is not part of its suitcase. However, this function is needed for the mobile agent to extract the value of the mobile agent from its suitcase after it has returned to its home platform.

During the transformation from UI-DSL to I-DSL it is ensured, that the value of the entire mobile agent is part of its global suitcase (see Section 7.3).

6.4 Internal Mobile Agent

The counterpart of the UI-DSL mobile agent (see Section 5.5) in I-DSL is the internal mobile agent. The UI-DSL mobile agent consists only of the platform agent and the value agent, whereas the internal mobile agent is the suitcase handler for its global suitcase. The internal mobile agent and the suitcase handler only differ concerning the extraction functions. The `suitcase extractor` and the `platform extractor` are used to calculate the suitcase respectively the list of platforms from the value calculated by `agent`. Both values are needed for the migration to the next platform.

The `value extractor` is a function to calculate the value of the mobile agent from its global suitcase. This function is not needed until the mobile agent has returned to its home platform. Therefore, the Haskell Mobile Agent Platform (see Chapter 8) provides the possibility to leave this function on the home platform when starting the mobile agent. Obviously, the `value extractor` can be used only if the value is part of the global suitcase, which, in fact, is guaranteed by the transformations from UI-DSL to I-DSL (see Chapter 7).

The `agent`, the `suitcase`, and the `initial suitcase` have the same semantics as the appropriate parts of the suitcase handler. Figure 6.5 shows the HOPS declaration of the internal mobile agent.

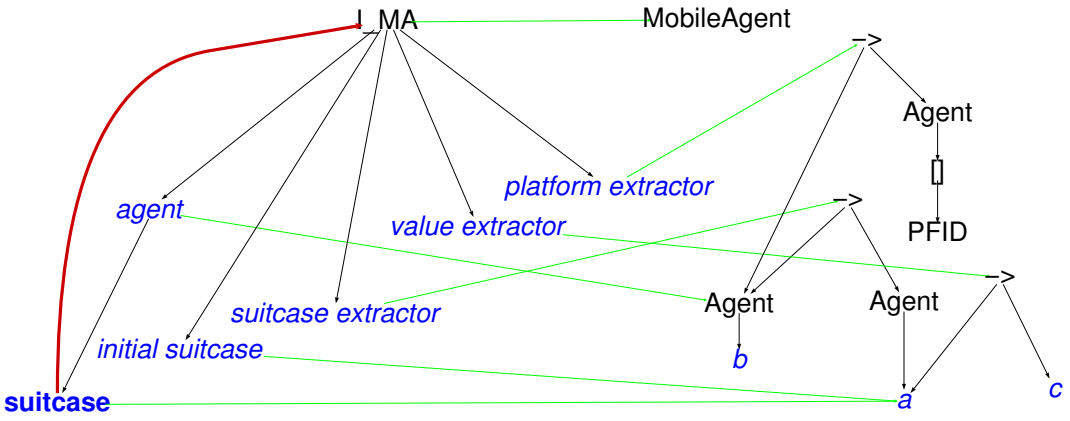


Figure 6.5: HOPS declaration of the internal mobile agent

7 Transforming UI-DSL to I-DSL

The transformation from the User-Interface Domain-Specific Language to the Internal Domain-Specific Language is the second step in the development of a mobile agent. In Figure 7.1 the HOPS transformation strategy is shown, which basically consists of the sequential application of five strategies, namely the `MetaAgent` strategy, the `Sharing` strategy, the `VinSC` strategy, the `Suitcase_Handler` strategy, and the `Cleanup` strategy. Before applying the `Sharing` strategy respectively the `Cleanup` strategy the DAG has to be maximal-identified.

The `MetaAgent` strategy is described in Section 7.1 and is used to replace the UI-DSL meta agent combinators with I-DSL meta agent combinators. The `Sharing` strategy is described in Section 7.2 and is used to replace the sharing of agents with the `letInA` combinator. The `VinSC` strategy (see Section 7.3) is used to ensure that the value of the mobile agent is part of its suitcase. The global suitcase of the mobile agent is generated with the `Suitcase_Handler` strategy which is introduced in Section 7.4. The last steps of the transformation of an UI-DSL mobile agent to an I-DSL mobile agent are the maximal-identification and the application of the `Cleanup` strategy which is described in Section 7.5.

7.1 Meta Agent Combinators

With the `MetaAgent` strategy the UI-DSL meta agent combinators, which are dependent on the availability of possibly-provided functions, are transformed into the corresponding I-DSL meta agent combinators. Furthermore the `MetaAgent` strategy calculates the list of necessary possibly-provided functions for the entire mobile agent. This list is used to replace the `neededPPFs` brick (see Chapter 10). The `MetaAgent` strategy is shown in Figure 7.2. This strategy is a sequential application of a few steps which are described below.

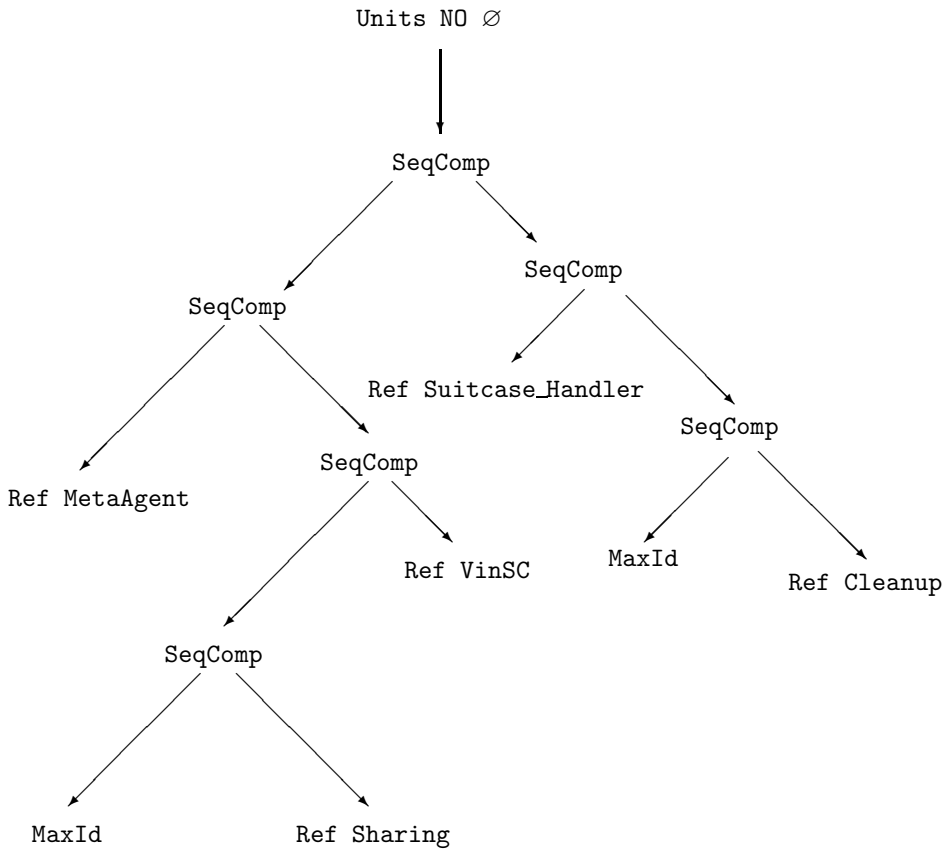


Figure 7.1: Strategy to transform an UI-DSL mobile agent into an I-DSL mobile agent

The `ppfname` strategy, which is referenced from the `MetaAgent` strategy, is shown in Figure 7.3. Within this strategy all possibly-provided functions will be marked with the `PPF_Marker`. The $\texttt{ALL}_{\texttt{PPF}}$ combinators and the $\texttt{OR}_{\texttt{PPF}}$ combinators are replaced by the $\texttt{I\_ALL}_{\texttt{PPF}}$ respectively the $\texttt{I\_OR}_{\texttt{PPF}}$ combinators (see Figure 6.3). Moreover, the UI-DSL mobile agent combinator will be replaced with the $\texttt{MA}_{\texttt{PPF}}$ (see Figure 7.4) combinator. The first successor of the $\texttt{MA}_{\texttt{PPF}}$ combinator is the list of necessary possibly-provided functions of type `[PPFName]`.

Figure 7.5 shows the HOPS declaration of the `PPF_Marker`; `ppf` is the possibly-provided function and `ppfname` is a value of type `PPFName` which contains a string with the name of the possibly-provided function.

In order to insert the `PPF_Marker` into the mobile agent DAG, a transformation rule has to be defined for each possibly-provided function. This is necessary since each possibly-provided function is represented as unique constant brick which has to be used in the corresponding transformation rule. Figure 7.6 shows an example of such a rule: The rule for the possibly-provided function `GetHotel`. The function `GetHotel` has three arguments: The town `in`, the arrival date `fromdate`, and the departure date

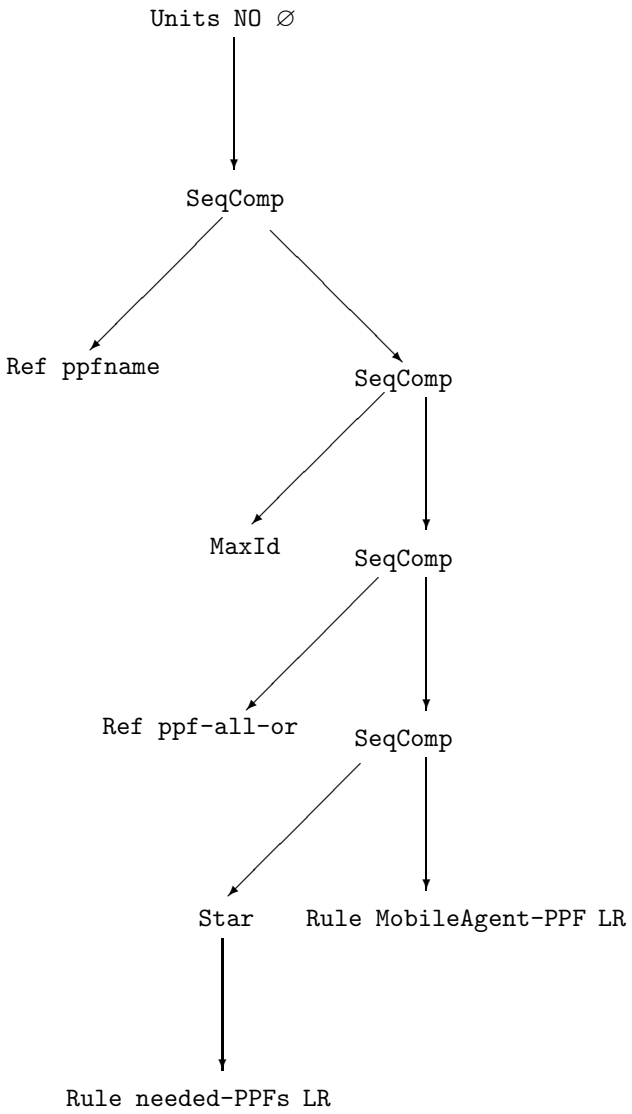


Figure 7.2: `MetaAgent` strategy to transform UI-DSL meta agent combinators into an I-DSL meta agent combinator

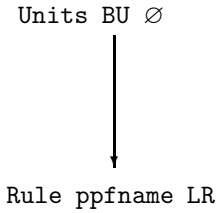


Figure 7.3: `ppfname` strategy

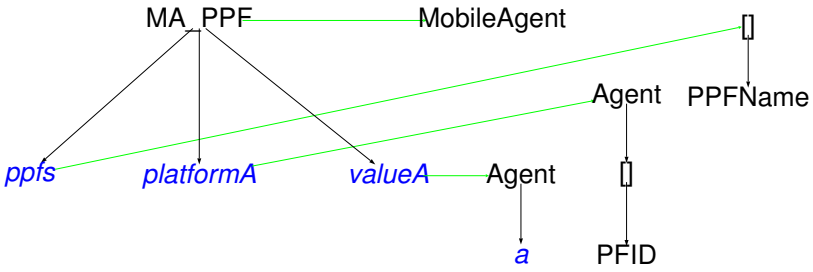


Figure 7.4: HOPS declaration of the $\texttt{MA}_{\texttt{PPF}}$ combinator

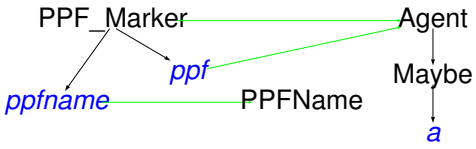


Figure 7.5: HOPS declaration of the `PPF_Marker`

`todate`. It returns a list of pairs consisting of the price and a description of available hotels encapsulated in the `Just` data constructor.

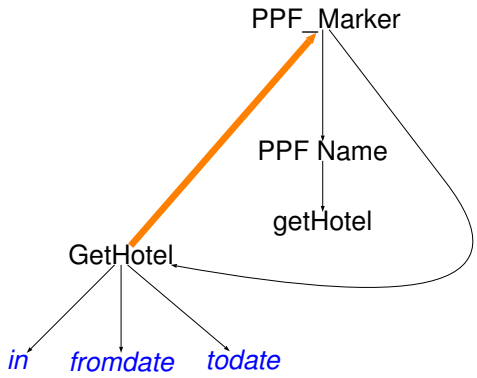


Figure 7.6: Transformation rule to insert `PPF_Marker` for the possibly-provided function `GetHotel`

This is the only step where a special transformation rule is needed for each possibly-provided function. After applying these rules, the `PPF_Marker` brick is used as an abstraction from the particular possibly-provided function.

All $\texttt{ALL}_{\texttt{PPF}}$ combinators are replaced with $\texttt{I\_ALL}_{\texttt{PPF}}$ combinators and all $\texttt{OR}_{\texttt{PPF}}$ combinators are replaced with $\texttt{I\_OR}_{\texttt{PPF}}$ combinators using the transformation rules shown in Figure 7.7. This has to be done, because the possibly-provided functions, which are now contained in the sub-DAG induced by the meta agent combinators, are not necessarily contained in the induced sub-DAGs after all transformation strategies have been applied, e.g., a possibly-provided function which is also used outside the induced sub-

DAG will be transformed out of this sub-DAG by the `FlattenBind` strategy, which is described in Section 9.3. The `ppfname` parts of those internal meta agent combinators include only empty lists after applying the rules shown in Figure 7.7. The elements of these lists are generated as a next step of the `MetaAgent` strategy.

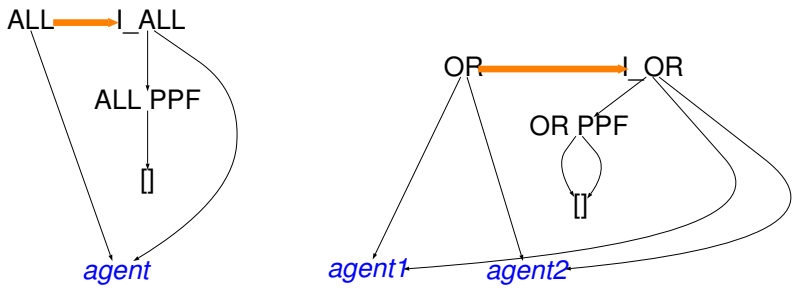


Figure 7.7: Transformation rules to replace $\texttt{ALL}_{\texttt{PPF}}$ with $\texttt{I\_ALL}_{\texttt{PPF}}$ and $\texttt{OR}_{\texttt{PPF}}$ with $\texttt{I\_OR}_{\texttt{PPF}}$

Before applying the `ppf-all-or` strategy the DAG will be maximal-identified with the `MaxId` step in order to avoid duplicates in the generated lists of necessary possibly-provided functions. The `ppf-all-or` strategy is shown in Figure 7.8.

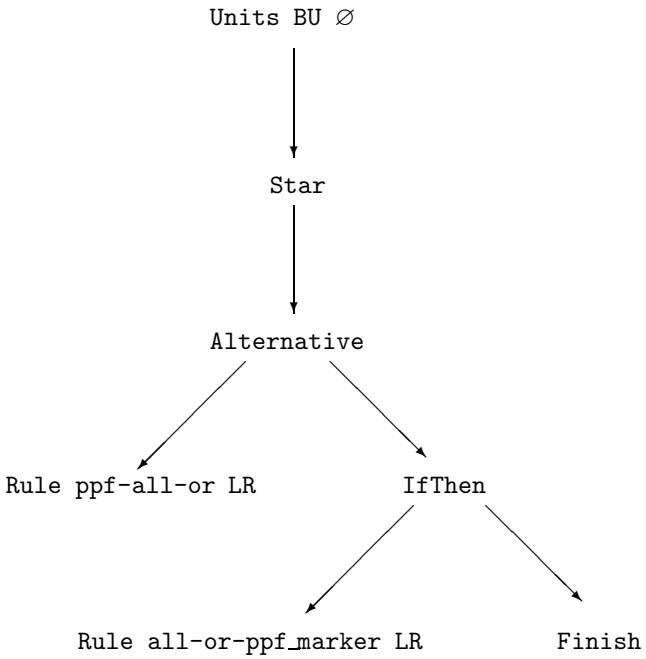


Figure 7.8: `ppf-all-or` strategy

With the `ppf-all-or` strategy the lists in the `ppfname` part of the $\texttt{I\_ALL}_{\texttt{PPF}}$ combinator and the $\texttt{I\_OR}_{\texttt{PPF}}$ combinator are generated. Furthermore the list of needed possibly-provided functions for the entire mobile agent is calculated. The strategy works as follows: The focus moves bottom-up until an $\texttt{I\_ALL}_{\texttt{PPF}}$, an $\texttt{I\_OR}_{\texttt{PPF}}$ or the $\texttt{MA}_{\texttt{PPF}}$ brick is

found. The rules in the rule set labelled with `ppf-all-or` are applied until no more `PPF_Marker` is found in the sub-DAG induced by the focused brick. Subsequently, the `PPF_Marker` is inserted above the focused brick using one of the rules labelled by `all-or-ppf_marker` and the step is finished. Afterwards, the focus is moved upwards to the next $\texttt{I\_ALL}_{\texttt{PPF}}$, $\texttt{I\_OR}_{\texttt{PPF}}$ or $\texttt{MA}_{\texttt{PPF}}$ brick and the same steps are performed. The strategy ends after the list for the $\texttt{MA}_{\texttt{PPF}}$ combinator has been generated.

Figure 7.9 shows the rules for the $\texttt{I\_ALL}_{\texttt{PPF}}$ combinator. The rule on the left hand side is the rule labelled with `ppf-all-or` which is used to generate the list of needed possibly-provided functions. The rule on the right hand side is used to insert the `PPF_Marker` atop of the $\texttt{I\_ALL}_{\texttt{PPF}}$ combinator and is contained in the rule-set labelled with `all-or-ppf_marker`.

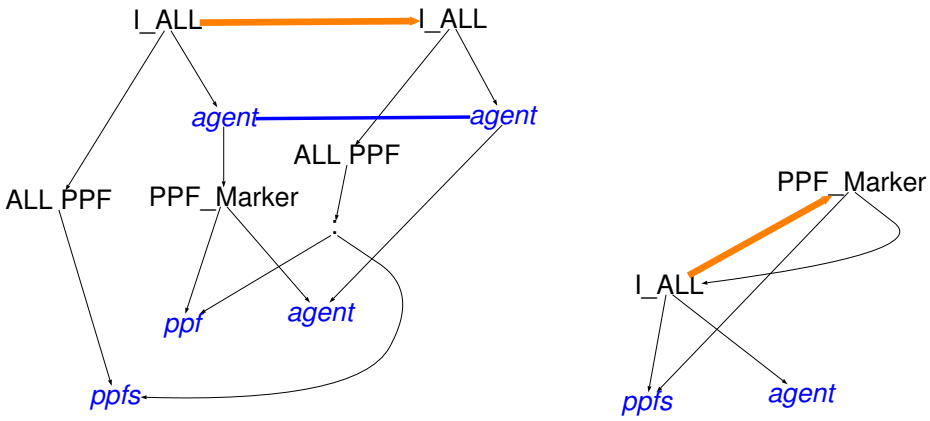


Figure 7.9: A rule to generate the list of needed possibly-provided functions for the $\texttt{I\_ALL}_{\texttt{PPF}}$ combinator and a rule to insert the `PPF_Marker` above the $\texttt{I\_ALL}_{\texttt{PPF}}$ combinator

The rules appropriate for the $\texttt{I\_OR}_{\texttt{PPF}}$ combinator and the $\texttt{MA}_{\texttt{PPF}}$ combinator are analog to the rules shown in Figure 7.9 and are, therefore, not presented here.

The last two sequential steps of the `MetaAgent` strategy are the application of the rules labelled with `needed-PPFs` and the rules labelled with `MobileAgent-PPF`. The rules labelled with `needed-PPFs` are used to replace all `neededPPFs` bricks with the list of needed possibly-provided functions. The rule labelled with `MobileAgent-PPF` reverses the replacement of the UI-DSL mobile agent combinator with the $\texttt{MA}_{\texttt{PPF}}$ brick.

7.2 Sharing

The `Sharing` strategy is used to replace the sharing of agents with the `letInA` combinator (see Section 6.1). This replacement is needed to ensure that the suitcase of a

shared agent is included in the generated global suitcase only once. Before applying the `Sharing` strategy the mobile agent DAG will be maximal-identified (see Figure 7.1). The `Sharing` strategy traverses the DAG top-down and uses only one rule (see Figure 7.10) to transform the sharing of an agent into the `letInA` combinator.

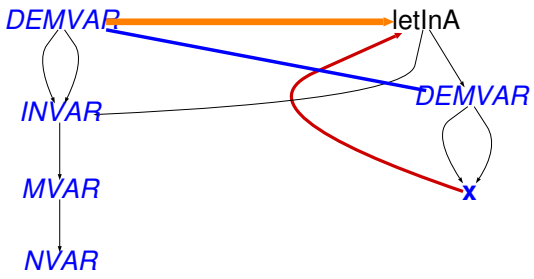


Figure 7.10: Rule to transform a shared agent into a `letInA` combinator

In this rule some special hard-coded meta variables are used, namely the `DEMVAR`, the `NVAR`, and the `INVAR`. The distinct edges meta variable `DEMVAR` is a meta variable where all outgoing edges have to be distinct from each other. The node variable `NVAR` matches on exactly one node. The interval node variable `INVAR` matches on at least one node and the matching nodes of all successors of `INVAR` have to be immediate successors of the matching node of `INVAR`. The left side of the rule shown in Figure 7.10 matches if there is a sub-DAG consisting of at least two nodes with at least two distinct edges from the matching of `DEMVAR` to the top-node of this sub-DAG.

7.3 Value in Suitcase

The `VinSC` strategy ensures that the value of the mobile agent is part of its suitcase which will be generated with the `Suitcase_Handler` strategy (see Section 7.4). The `VinSC` strategy is shown in Figure 7.12.

The `VinSC` strategy is divided into three sub-strategies: The `VinSC-pre` strategy, the `VinSC-main` strategy, and the `VinSC-post` strategy. The `VinSC-pre` strategy prepares a DAG for the `VinSC-main` strategy in the following way:

- All possibly-provided functions are marked with the `PPF_Marker` (see Figure 7.5).
- All stateful agents which do not use their suitcase, i.e. stateful agents without a bound variable, are replaced by a stateless agent with the same body using the rule shown in Figure 5.4.
- An initial `VinSC-Marker` is inserted using the rule shown in Figure 7.13.

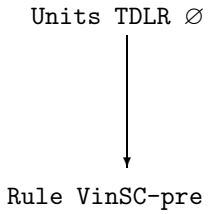


Figure 7.11: `VinSC-pre` strategy

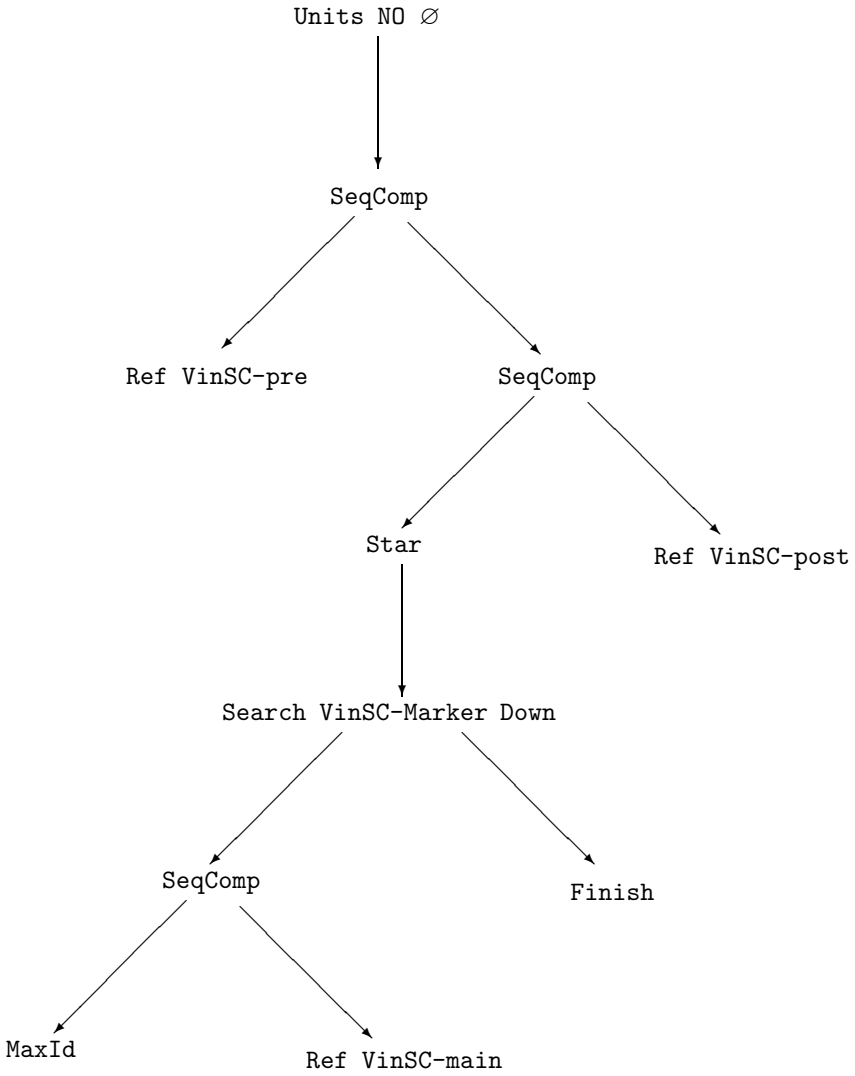


Figure 7.12: `VinSC` strategy

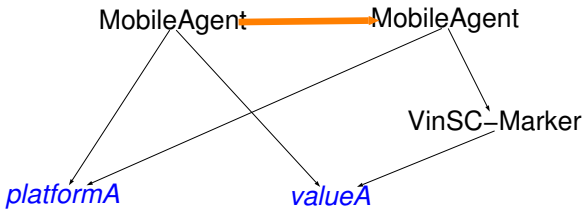


Figure 7.13: Rule to insert the initial `VinSC-Marker`

The main part of the `VinSC` strategy is the `VinSC-main` strategy which is shown in Figure 7.14. This strategy will be applied to the mobile agent as long as a `VinSC-Marker` is found in the DAG (see Figure 7.12). Prior to any application of the `VinSC-main` strategy the mobile agent DAG will be maximal-identified. The reason for both, the search of the `VinSC-Marker` and the maximal-identification, is the `letInA` combinator.

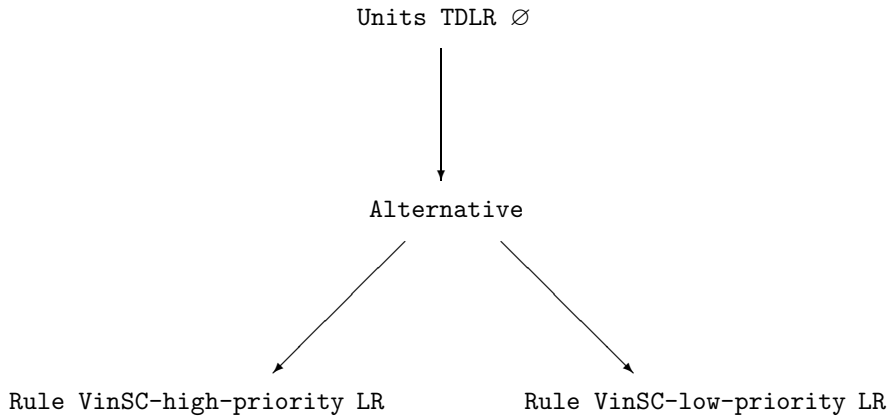


Figure 7.14: `VinSC-main` strategy

There are two rules for the `letInA` combinator (see Figure 7.15). The left rule in Figure 7.15 is used to move the `VinSC-Marker` under the `letInA` combinator, but only above its second successor. The rule on the right-hand side of Figure 7.15 is used to move a `VinSC-Marker` immediately above the bound variable to the first successor of the `letInA` combinator. But with the otherwise needed navigation top-down left-right of the `VinSC-main` strategy this rule can never be applied. Therefore the transformation strategy starts again from the top node if a `VinSC-Marker` is still in the DAG. With the maximal-identification all `VinSC-Marker` immediately above the bound variable are identified and the rule on the right hand side of Figure 7.15 is applicable at most once for each `letInA` combinator.

Within the `VinSC-main` strategy two sets of rules are used, one set with a higher priority and one set with a lower priority. The different priority is given by the `Alternative` node of the `VinSC-main` strategy. Only if no rule labelled with `VinSC-high-priority`

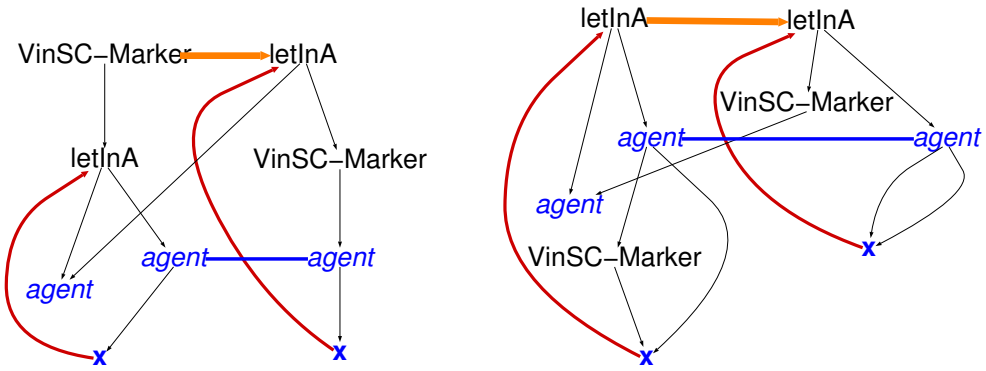


Figure 7.15: Rules for the `VinSC-Marker` in conjunction with the `letInA` combinator

is applicable the rules labelled with `VinSC-low-priority` are used. By this means, it is, for instance, possible to use two different rules for the stateless agent, one which is always applicable and the other one which can be used only if the stateless agent is of type `Agent (Maybe a)` (see Figure 7.17).

The idea of the `VinSC-main` strategy is to move the `VinSC-Marker` top-down across the agent combinators until a platform function, a possibly-provided function, or a primitive agent — a stateful or a stateless agent — has been reached. The rules for the agent combinators are almost straightforward. All those rules, except for the rule for the $@_{\texttt{Agent}}$ combinator which is shown in Figure 7.16 and the rules for the `letInA` combinator (see Figure 7.15), are moving the `VinSC-Marker` from above the particular agent combinator directly above all successors of the agent combinator.

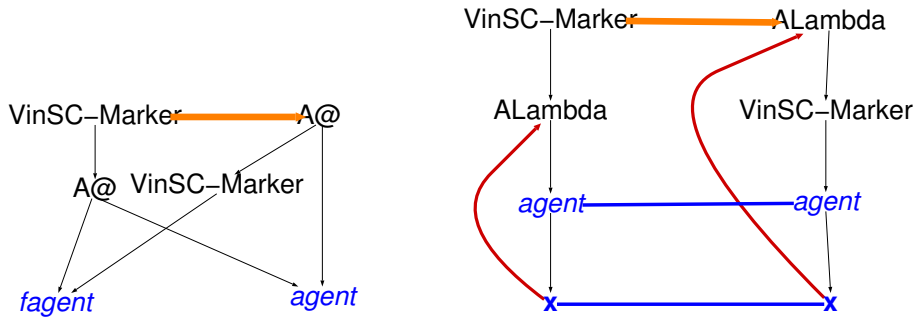


Figure 7.16: Rule to move the `VinSC-Marker` over the $@_{\texttt{Agent}}$ combinator and the $\lambda_{\texttt{Agent}}$ combinator

The value of the $@_{\texttt{Agent}}$ combinator is calculated by the function agent `fagent`. Therefore, the `VinSC-Marker` is only moved above the `fagent`.

Figure 7.17 shows the rules for the stateless agent. The rule on the left hand side is in the set of rules with the higher priority. This rule can only be applied if the

stateless agent is of type `Agent (Maybe a)`. In this case the stateless agent can be transformed into a stateful agent. Since the suitcase is of type `Maybe a` the value `Nothing` can be used as initial suitcase. The rule on the right hand side is in the set of rules with the lower priority. This rule can be applied to each stateless agent. It transforms a stateless agent into a `stateless agent VinSC`, a temporarily used brick which will be replaced within the `Suitcase_Handler` strategy (see Section 7.4). The platform functions are transformed similarly into special temporary bricks which are also replaced in the `Suitcase_Handler` strategy.

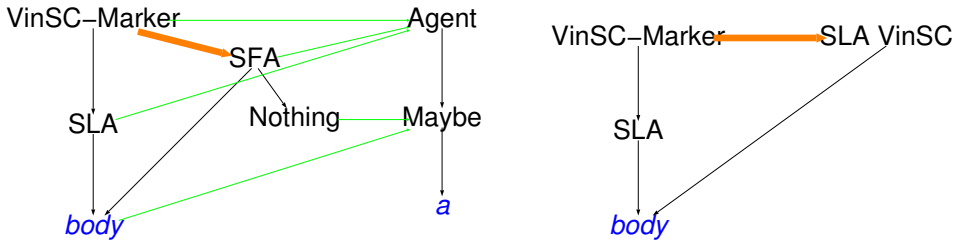


Figure 7.17: Rules to apply the `VinSC-Marker` to the stateless agent

Since the value of the stateful agent is already contained in its local suitcase, the `VinSC-Marker` atop of a stateful agent is just removed with the rule shown in Figure 7.18. In Figure 7.19 the rule for a possibly-provided function represented by the `PPF_Marker` is shown. As mentioned before, the value of a possibly-provided function is always of type `Maybe a`. By this means, this value can easily be stored in the local suitcase of a stateful agent with the value `Nothing` as its initial suitcase.

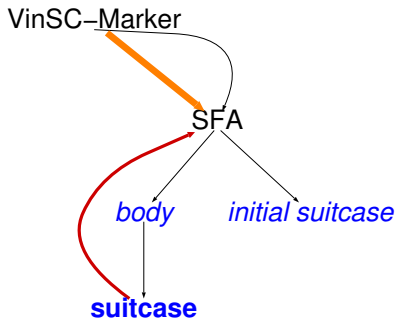


Figure 7.18: Rule to apply the `VinSC-Marker` to the stateful agent

The `VinSC-post` strategy is used to remove any remaining `PPF_Marker` which have been inserted by the `VinSC-pre` strategy but have not yet been removed by the `VinSC-main` strategy. Furthermore, all `letInA` combinators are replaced with $@_{\texttt{Agent}}$ combinators using the rule shown in Figure 7.20.

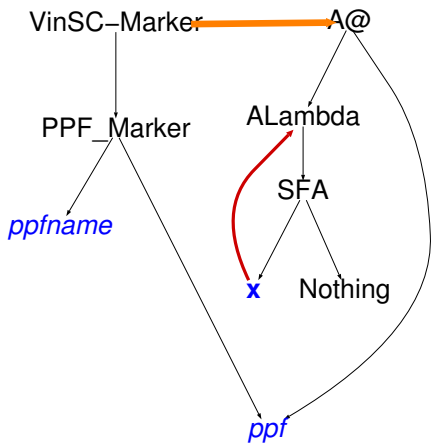


Figure 7.19: Rule to apply the `VinSC-Marker` to a possibly-provided function

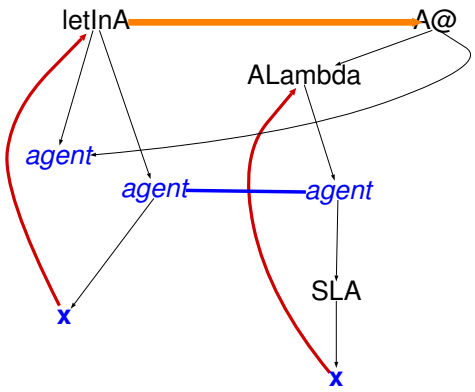


Figure 7.20: Rule to replace the `letInA` combinator with the $@_{\texttt{Agent}}$ combinator

7.4 Suitcase Handler

The `Suitcase_Handler` strategy is used to generate the global suitcase of the mobile agent. After applying the `VinSC` strategy it is ensured that the value of the mobile agent will be part of the generated global suitcase. In Figure 7.21 the `Suitcase_Handler` strategy is shown. The `Suitcase_Handler` strategy assembles the suitcases of sub-DAGs step-by-step from the bottom to the top of the mobile agent DAG using two sets of rules, one set with a high priority and one set with a low priority. The set with the high priority contains all rules where all successors of the matching node are suitcase handlers.

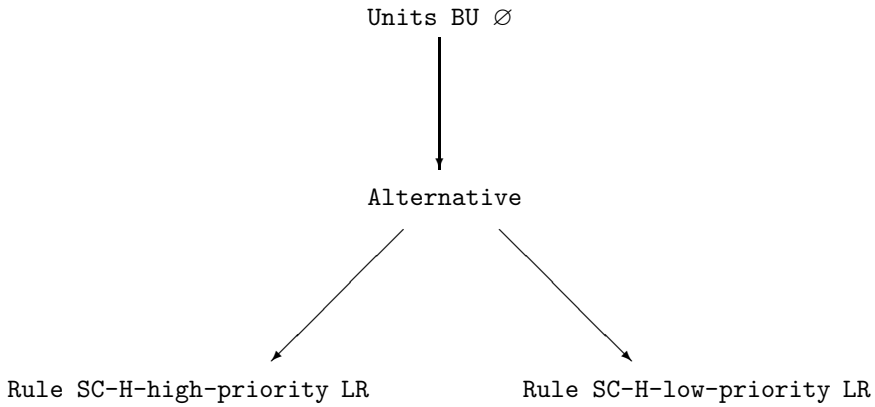


Figure 7.21: `Suitcase_Handler` strategy

A suitcase of a sub-DAG is handled by the suitcase handler (see Section 6.3). As mentioned in Chapter 6, the I-DSL has only one primitive agent, the internal agent (see Section 6.1). All stateful agents are transformed by the rule shown in Figure 7.22 to a suitcase handler and an internal agent.

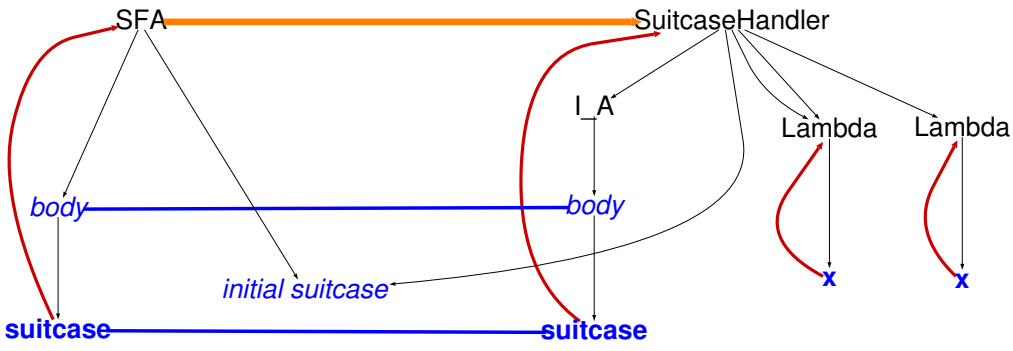


Figure 7.22: Rule to transform a stateful agent into a suitcase handler

Since the value of a stateful agent is identical to its suitcase, the `suitcase extractor` and the `value extractor` both coincide with the identity. Furthermore, the `value`

`from suitcase` function is the identity. The particular type of the identity is the reason why only the `suitcase extractor` and the `value extractor` can be represented as one shared DAG. The type of the `suitcase extractor` respectively the `value extractor` is `Agent a -> Agent a`, whereas the `value from suitcase` function is of type `a -> a`.

The identity is frequently drawn as $\lambda x.x$ (e.g. in Figure 7.22) and not represented by a single node `id`, since the pattern consisting of λ and the bound variable is needed by most of the rules for transforming agent combinators (e.g. the rule shown in Figure 7.25).

Figure 7.23 shows the rule to transform a stateless agent into an internal agent. The transformation rule for the `stateless agent VinSC` is shown in Figure 7.24. This agent is of type `Agent a` with `a` $\neq$ `Maybe b` for all `b`, since all stateless agents of type `Agent (Maybe b)` for an arbitrary `b` have been transformed already to a stateful agent by the rule with higher priority shown in Figure 7.17.

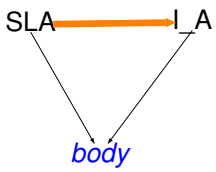


Figure 7.23: Rule to transform a stateless agent into an internal agent

The initial suitcase is the main issue of the transformation of the `stateless agent VinSC` into a suitcase handler. Since the initial suitcase is never used, one possibility is to use an arbitrary value of the particular type. For this approach, however, it is necessary to define a special rule for every possible type. Another solution is to use a polymorphic `undefined :: a` brick. The drawback of this idea is that the `undefined` value can not be read and shown by the standard Haskell functions `read` and `show`. Since the suitcase is of an arbitrary type, the Haskell Mobile Agent Platform only handles a string representation of the suitcase and the mobile agent uses `read` and `show` for converting the suitcase from this string representation to the real value and vice versa. More detailed information about this can be found in Section 8.1.

A more elegant solution is to encapsulate the value `val` of type `a` calculated by the stateless agent into the value `Just val` of type `Maybe a`. By this means, it is possible to use the value `Nothing` as initial suitcase. The `suitcase extractor` is the identity, but the `value extractor` and the `value from suitcase` function need to extract the value `val` from the value `Just val` using the function `fromJust :: Maybe a -> a`.

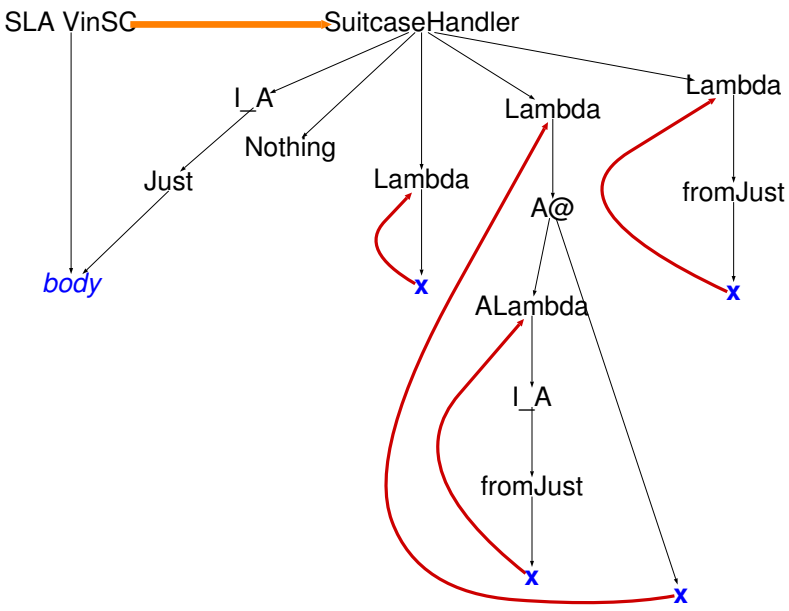


Figure 7.24: Rule to transform a `stateless agent VinSC` into a suitcase handler

The rule to transform a $\lambda_{\texttt{Agent}}$ combinator with a suitcase handler as successor into a suitcase handler is shown in Figure 7.25. The `undefined` brick in the `suitcase extractor` can be used without any difficulty, since it will be removed when applying the rule for the appropriate $@_{\texttt{Agent}}$ combinator.

Three different rules are used for the $@_{\texttt{Agent}}$ combinator depending on whether the first successor, the second successor, or both successors of the $@_{\texttt{Agent}}$ combinator are suitcase handlers. Since it is not possible in HOPS to define "negative" constraints for matchings, like "*this node should be anything but a suitcase handler*", the rule for a $@_{\texttt{Agent}}$ where both successors are suitcase handlers has to be used with a higher priority. Therefore, this rule is element of the set of rules with high priority whereas the other rules for the $@_{\texttt{Agent}}$ combinator are in the set of rules with low priority (see Figure 7.21).

In Figure 7.26 the rule to transform an $@_{\texttt{Agent}}$ combinator with a suitcase handler as first successor into a suitcase handler is shown. In Figure 7.27 the rule to transform an $@_{\texttt{Agent}}$ combinator with a suitcase handler as second successor into a suitcase handler is shown. The rule in Figure 7.27 ignores the `value from suitcase` function and uses the `undefined` brick as new `value from suitcase` function, because the value of the $@_{\texttt{Agent}}$ combinator is not part of the suitcase. Otherwise, the first successor has to be a suitcase handler.

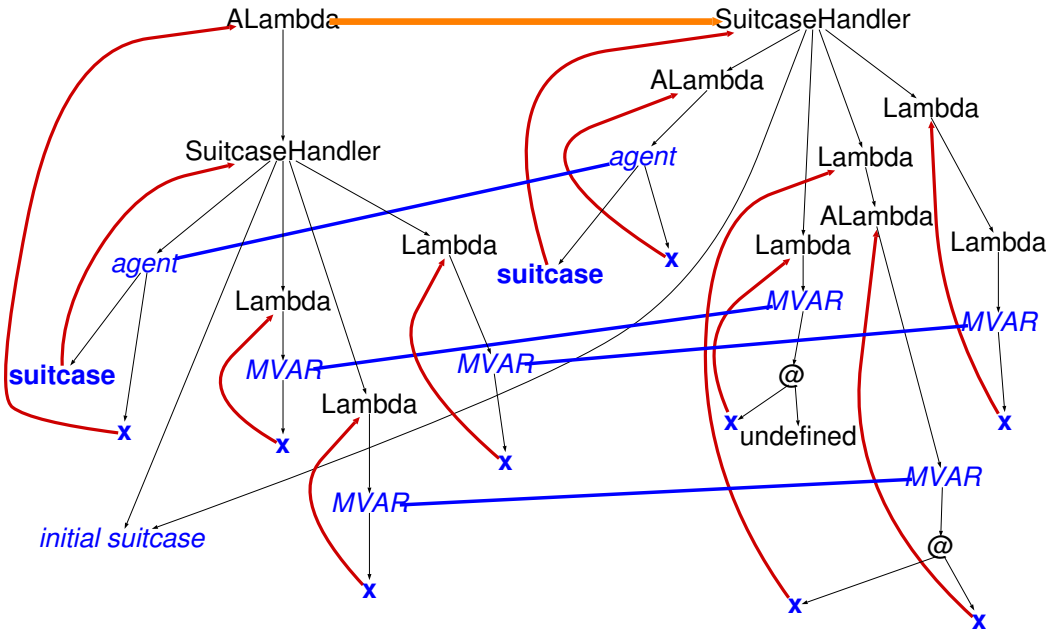


Figure 7.25: Rule to transform a $\lambda_{\texttt{Agent}}$ combinator with a suitcase handler as successor into a suitcase handler

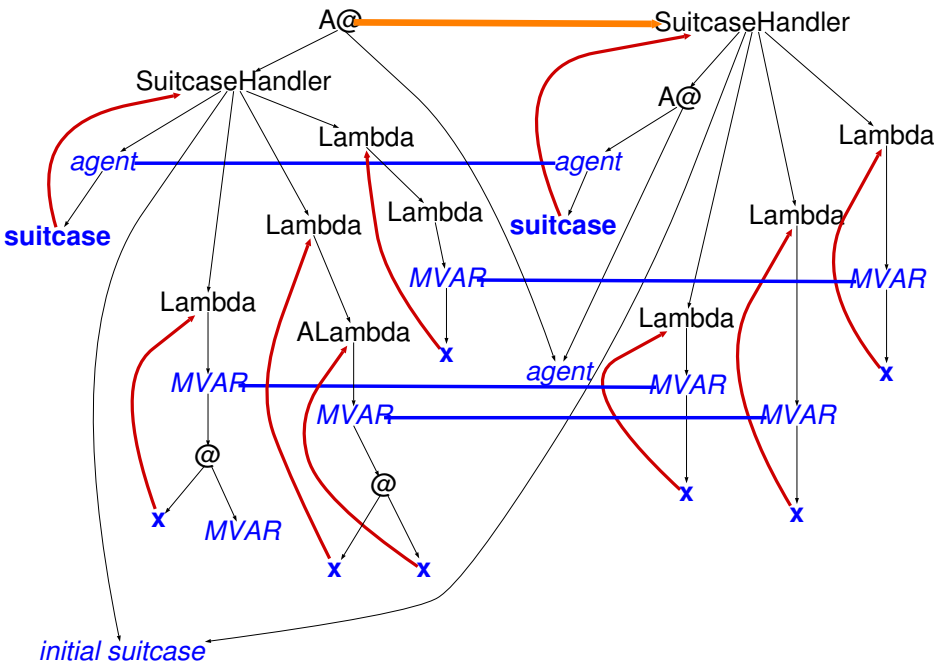


Figure 7.26: Rule to transform an $@_{\texttt{Agent}}$ combinator with a suitcase handler as first successor into a suitcase handler

Since the value and the suitcase of the $@_{\texttt{Agent}}$ combinator shown in Figure 7.27 are different from each other, a $(,)_{\texttt{Agent}}$ combinator is used to combine the agent calculating the value and the agent calculating the suitcase. The first successor of the $(,)_{\texttt{Agent}}$ combinator is an $@_{\texttt{Agent}}$ combinator with the old `fagent` as first successor and the application of the old `suitcase extractor` to the old argument, the `agent`, as second successor. The second successor of the $(,)_{\texttt{Agent}}$ combinator is the application of the old `value extractor` to the `agent`. The new `suitcase extractor` and the new `value extractor` are the projections to the particular component using the $\pi_{\texttt{Agent}}$ respectively the $\rho_{\texttt{Agent}}$ combinator.

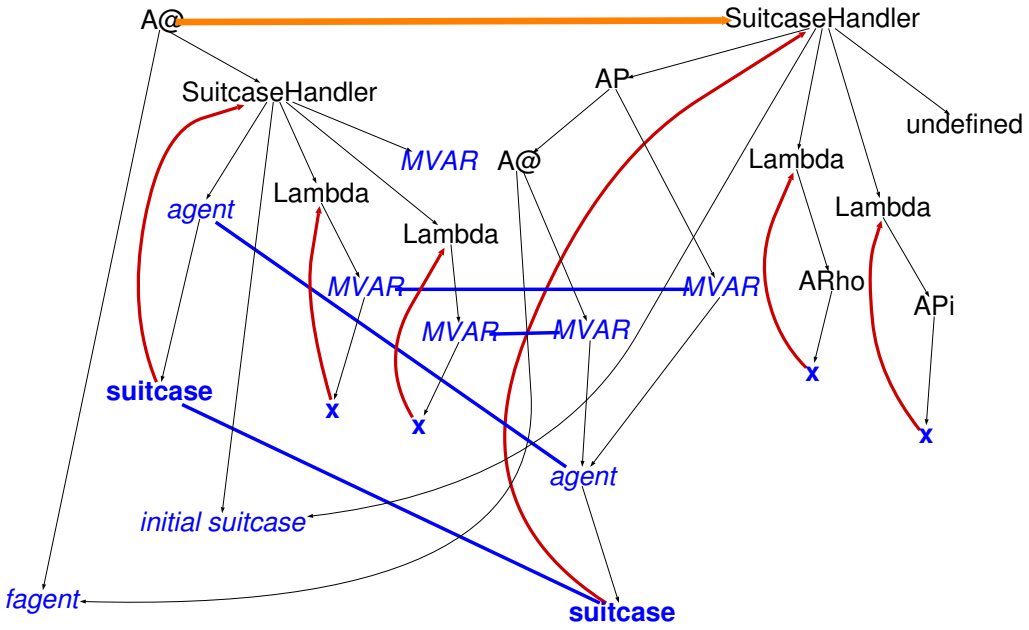


Figure 7.27: Rule to transform an $@_{\texttt{Agent}}$ combinator with a suitcase handler as second successor into a suitcase handler

In Figure 7.28 the rule to transform an $@_{\texttt{Agent}}$ combinator with a suitcase handler as first and as second successor into a suitcase handler is shown.

The rule to transform a mobile agent with a suitcase handler as a first as well as a second successor is shown in Figure 7.29. The `agent` of the internal mobile agent is an agent pair consisting of the platform agent and the value agent. The `suitcase extractor` of the internal mobile agent is the combination of the platform agent `suitcase extractor` and the value agent `suitcase extractor`. The `value extractor` of the internal mobile agent is the composition of the projection on the second component and the `value from suitcase` function of the value agent. The `platform extractor` is the composition of the $\pi_{\texttt{Agent}}$ combinator and the `value extractor` of

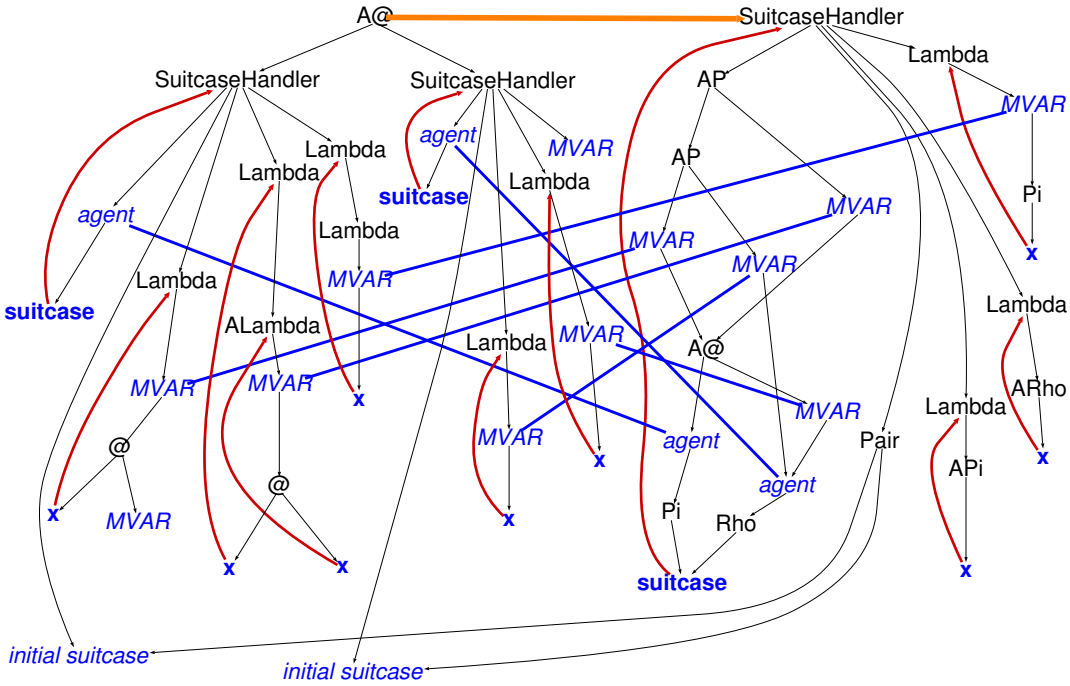


Figure 7.28: Rule to transform an $@_{\texttt{Agent}}$ combinator with a suitcase handler as first and as second successor into a suitcase handler

the platform agent. The `value extractor` of the value agent as well as the `value from suitcase` function of the platform agent is not needed in the internal mobile agent.

The second rule to transform a mobile agent to an internal mobile agent is shown in Figure 7.30. In this case only the value agent is considered to be a suitcase handler. Obviously, the rule shown in Figure 7.29 has to be element of the set of rules with the higher priority.

A rule to transform a mobile agent to an internal mobile agent where only the platform agent is a suitcase is not necessary, because the value agent is always a suitcase handler containing at least its value in its suitcase. The rules for the $(,)_{\texttt{Agent}}$ combinator, the $\pi_{\texttt{Agent}}$ combinator, and the $\rho_{\texttt{Agent}}$ combinator are straightforward and, therefore, the author decided to abstain from presenting these rules.

7.5 Cleanup

The `Cleanup` strategy is used to simplify reducible parts of the internal mobile agent which have been generated by the transformations described in this chapter. Before

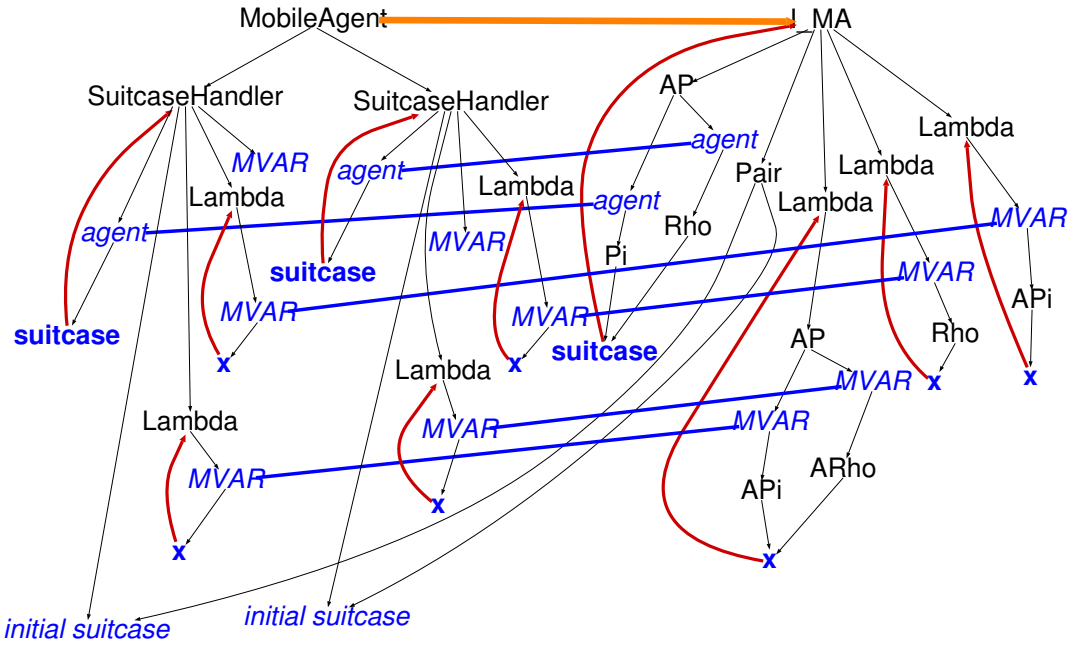


Figure 7.29: Rule to transform a mobile agent with suitcase handlers as successors into a internal mobile agent

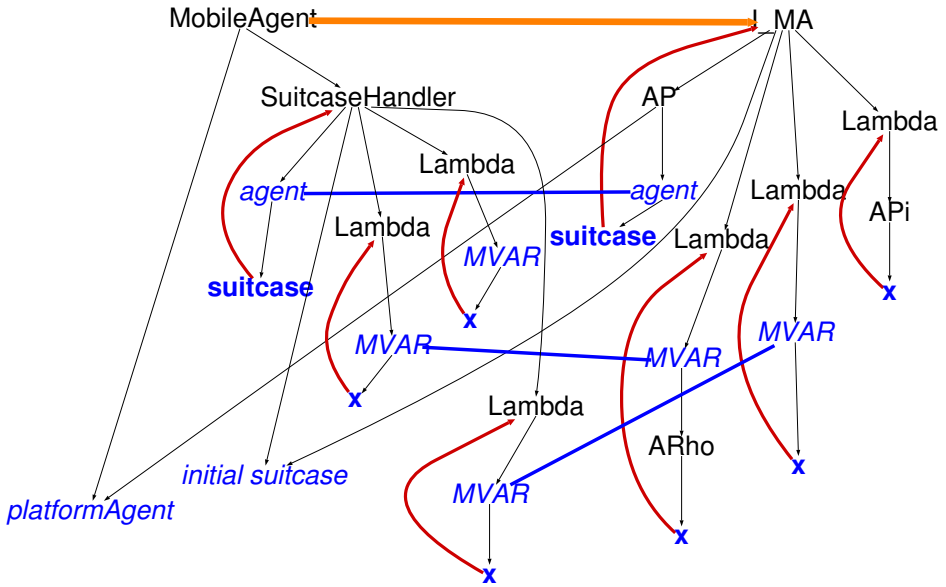


Figure 7.30: Rule to transform a mobile agent with a suitcase handlers as second successor into a internal mobile agent

applying the `Cleanup` strategy the mobile agent DAG will be maximal-identified (see Figure 7.1). The `Cleanup` strategy is shown in Figure 7.31.

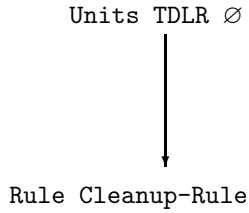


Figure 7.31: `Cleanup` strategy

In Figure 7.32 two rules are shown to exemplify the rules with the label `Cleanup-Rules`. The rule on the left-hand side is used to transform an agent of the form $(\pi_{\texttt{Agent}}(\texttt{agent}), \rho_{\texttt{Agent}}(\texttt{agent}))_{\texttt{Agent}}$ into `agent`. The rule on the left-hand side transforms `fromJust (Just x)` into `x`.

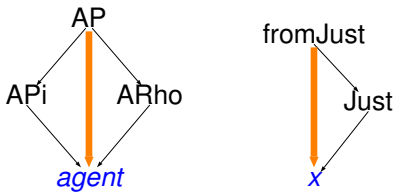


Figure 7.32: Two rules which are element of the set of rules labelled with `Cleanup-Rules`

8 Haskell Mobile Agent Platform

The Haskell Mobile Agent Platform (HaMAP) has been created in a joint development with the mobile agent development system based on the Higher Object Programming System HOPS. The main focus in the context of this thesis is to provide a test bed for mobile agents developed, transformed, and generated with HOPS. The core of HaMAP has been designed and implemented with the option of extensibility. Since the development of a fully-fledged mobile agent execution environment goes beyond the scope of this thesis, scalability, efficiency and interoperability with existing environments have not been taken into consideration. HaMAP is written in the purely functional language Haskell (Peyton Jones et al., 2002). A first working prototype of an agent platform written in Haskell has been presented by Morbach (2001).

The design and the implementation of the Haskell Mobile Agent Platform are described in Section 8.1 respectively in Section 8.2. In Section 8.3 migration and communication between mobile agent platforms are introduced.

8.0 Haskell Prerequisites

In the following we use the socalled `Agent` monad. A monad is a mathematical structure which comes from a mathematical discipline, called the category theory (Asperti and Longo, 1991). A characteristic of a monad are the following three laws which will hold for an arbitrary monad:

```
(return x) >>= f == f x
m >>= return     == m
(f >>= g) >>= h  == f >>= (\x -> g x >>= h)
```

The first law means that `return` is left-identity with respect to `>>=`. The second law states that `return` is also right-identity with respect to `>>=`. The third law is the associativity law for `>>=`.

8.1 Design Overview

The design of the HaMAP components is outlined in the following sections. The Haskell Mobile Agent is introduced in Section 8.1.1. Section 8.1.2 describes the Agent Monad and the minimal set of monadic functions a mobile agent platform has to provide. The different mobile agent platforms and the different migration functions will be explained in Section 8.1.3.

8.1.1 Haskell Mobile Agent

A Haskell Mobile Agent consists of three parts: The code, the suitcase, and the meta data. The code and the suitcase are generated by the code output facility of HOPS (see Section 9.6).

```
data MobileAgent = MobileAgent { code     :: [CodeFragment]
                               , suitcase :: Suitcase
                               , metadata :: MetaData
                               }
```

The code part is a list of code fragments. In order to execute the mobile agent, the code part has to be transformed to a code string by the mobile agent platform. The objective of this transformation is to replace all possibly-provided functions which are not available on the particular platform with a function returning the value `Nothing :: Maybe a`. Therefore, all possibly-provided functions have to return a value of type `Maybe a` for arbitrary `a`. Each code fragment will be transformed into a string, and these strings will be concatenated to one string, which is the compilable mobile agent code. The implementation of the transformation function will be introduced in Section 8.2.1.

The code fragment data type consists of seven constructors as alternatives, namely `Code`, `PPF`, `ALL`, `OR`, `Replaceable`, `Replacement`, and `CFList`. Those constructors with an extreme nesting are necessary, since the code fragments are used for controlling the preprocessor, and the meta agent combinators are all expressed in terms of code fragment values.

```
data CodeFragment = Code String
                  | PPF PPFName [CodeFragment]
                  | ALL PPFName [CodeFragment]
```

```
                | OR  PPFName [CodeFragment] [CodeFragment]
                | Replaceable Integer [CodeFragment]
                | Replacement [CodeFragment]
                | CFList [CodeFragment]
```

The code fragment `Code string` is a plain code chunk. A possibly-provided function is represented as `PPF ppfname ppfargs`, where `ppfname` contains the name of the possibly-provided function and `ppfargs` is a list of code fragments representing the arguments of the possibly-provided function.

The `PPFName` data type is used to denote the needed possibly-provided functions. The value, which is used for the `ALL` and the `OR` code fragment, is generated already while transforming the UI-DSL mobile agent to an I-DSL mobile agent (see Section 7.1).

```
data PPFName = PPFName String
             | ALLPPF [PPFName]
             | ORPPF  [PPFName] [PPFName]
             | ReplaceablePPF Integer [PPFName]
             | EmptyPPF
```

A code fragment `ALL (ALLPPF ppfnames) codefrags` has to be replaced, unless all possibly-provided functions needed[1] in `ppfnames` are available. A code fragment `OR (ORPPF ppfnames1 ppfnames2) codefrags1 codefrags2` consists of two alternative lists of code fragments. The first alternative `codefrags1` will be used, if and only if all possibly-provided functions needed in `ppfnames1` are available. Otherwise, `codefrags2` will be used.

A code fragment `Replaceable int codefrags` can be replaced with another list of code fragments using the function `replaceCF` (see Section 8.1.2). The function application `replaceCF pos new old` replaces the code fragment `Replaceable int codefrags` in `old` with the list of code fragments `new` if `int == pos`. If `codefrags` contains possibly-provided functions, these functions are represented as `ReplaceablePPF int' ppfnames`. The function `replaceCF` replaces this value with `EmptyPPF` if `int' == pos`.

With the `Replacement` constructor it is possible to use a list of code fragments that will not be modified by the preprocessor. In particular, this is useful for the list of

[1] The term "needed" is used, since not all functions which are found in a list of `PPFName`'s are needed. Functions which occur only in the alternative of the `OR` constructor which is not used, are obviously not needed.

code fragments which are used to replace a `Replaceable` code fragment. The `CFList` constructor represents a list of code fragments. This is necessary for replacing a code fragment with a list of code fragments, because a value of type `CodeFragment` cannot be replaced with a value of type `[CodeFragment]`.

A mobile agent has a suitcase, which has been built by HOPS using term graph transformation. Such a suitcase can be different for each mobile agent. Since Haskell is a strongly typed language, it is not possible to use such a value with an arbitrary type directly. But in this approach, it is possible to use an "untyped" suitcase without losing the type-safety, since the suitcase has been type-checked already by HOPS. The "untyped" suitcase is the string representation of the structured suitcase value.

```
newtype Suitcase = Suitcase String
```

Once the mobile agent is executed, the string representation has to be converted into the structured suitcase value. This conversion can be done by the mobile agent, since the type of its suitcase is known in the context of the particular mobile agent.

Haskell provides a way to convert a value of type `a` to a string and vice versa, using the Haskell Standard Prelude functions `read` and `show`.

```
read :: (Read a) => String -> a
show :: (Show a) => a -> String
```

`Read a` and `Show a` in the type signatures above are called context. This means that the type `a` has to be an instance of the Haskell class `Read` to be read, and an instance of the Haskell class `Show` to be convertible to a string. All Standard Haskell Types except function types and I/O types are instances of `Read` and `Show`. Using the transformation mechanisms of HOPS it can be ensured that the type of the suitcase is an instance of `Read` and `Show`. Values with function or I/O types are not allowed in the suitcase. The functions of the mobile agent are part of the code, not of the data of a mobile agent. Values with an `IO` type are not very useful in the mobile agent, since the agent cannot do I/O, anyway. The mechanism to ensure that the mobile agent cannot do I/O is explained in Section 8.2.1.

The fact that the mobile agent platform cannot simply[2] access the suitcase leads to the introduction of the third part of the mobile agent, the meta data.

[2]Note: It is possible for the mobile agent platform to access the structured suitcase value. The `read` function application including the type of the suitcase is part of the mobile agent code. Thus, the suitcase is not secure in terms of privacy.

```
data MetaData = MetaData { agentID :: MAID
                         , homePF  :: PFID
                         , ...
                         , logs    :: [(PFID,[String])]
                         }
```

The meta data is a second suitcase, which is accessible by the mobile agent platform. It contains among other things the mobile agent identifier, the home platform identifier and log information from all platforms visited so far. All mobile agent platforms have to be able to process at least three different kinds of platform identifiers: The socket platform identifier, the electronic mail platform identifier and the common gateway interface platform identifier. The socket platform identifier consists of the Internet Protocol address or the host-name, and the socket number. The electronic mail platform identifier contains the electronic mail address, and the common gateway platform identifier contains the uniform resource locator. The mobile agent identifier consists of an integer, which has to be unique on the home platform. Since cloning of mobile agents is not yet included in HaMAP, the home platform identifier together with the mobile agent identifier builds a unique attribute of a mobile agent.

With the meta data suitcase it will be possible, for example, to send the mobile agent back to the home platform and add some information to the logs, if the mobile agent fails during execution or even during compilation. The meta data is not part of the mobile agent generated by HOPS. It has to be created on the home platform. This will be explained further in Section 8.2.2.

8.1.2 Agent Monad

For the purpose of providing information on the execution of a mobile agent, e.g. used possibly-provided functions or error messages, the mobile agent platform maintains a state. The exact type and contents of the state depends on the platform only and not on any mobile agent. The usual way to handle this in Haskell is using a monad. With the concept of monads it is possible to encapsulate a state transforming function into a pure calculation.

A monad in Haskell has to be an instance of the class `Monad`, and has to implement at least the following two functions:

```
return :: Monad m => a -> m a
(>>=)  :: Monad m => m a -> (a -> m b) -> m b
```

The first function is the identity function of the monad. The function application `return x` lifts the value `x` into the monad `m` and returns `x`. The second function is pronounced "bind" and is used for the composition of monadic computations. With the bind function it is also possible to sequence calculations. Therefore, bind is also called "and then". I/O in Haskell is done in a monad, where e.g., an input is read "and then" some calculations are made "and then" the calculated value is printed. In this thesis the term "evaluating `g` **after** evaluating `f`" means `f >>= g`. In this context "the code **after** `f`" is `g`. The instance of monad used in this approach is called the Agent Monad.

```
class Monad m => Agent m
```

This agent monad can be implemented in different ways on different platforms. Five monadic functions, namely `migrate`, `getPFID`, `getPPFInfo`, `hasPPFs`, and `replaceCF`, have to be implemented on every mobile agent platform, besides `return` and `(>>=)`.

```
migrate    :: Agent m
           => [CodeFragment] -> Suitcase -> MetaData -> PFID -> m ()
getPFID    :: Agent m => m PFID
getPPFInfo :: Agent m => m [(PFID,PPFInfo)]
hasPPFs    :: Agent m => [PFID] -> [PPFName] -> m [PFID]
replaceCF  :: Agent m
           => Integer -> [CodeFragment] -> [CodeFragment] -> m [CodeFragment]
```

The monadic function `migrate` provides the option to migrate to another platform. The first argument is the code, the second the suitcase and the third the meta data of the mobile agent. The fourth argument is the platform identifier of the platform to migrate to. If the migration was successful, `migrate` will exit using the Haskell function `exitWith ExitSuccess`. In this case, the code of the mobile agent after the `migrate` function will not be evaluated. If the migration was not successful, `migrate` returns the trivial value `()`. In that case the code of the mobile agent after the `migrate` function can react on the failure and for instance can try to migrate to another platform. By this means, the code of a mobile agent trying to migrate to one of a list of platforms, can be written in the following way:

```
sequence_ (map (migrate code suitcase metadata) [pf1,pf2,pf3,pf4,pf5])
```

With this code the mobile agent tries to migrate to `pf1`, and if that fails to `pf2`, and if that fails to `pf3` and so on. Information about the Haskell Standard Prelude functions

`sequence_` and `map` can be found in (Peyton Jones et al., 2002). If the mobile agent code does not react on the failure or the reaction fails, the mobile agent platform has to react (see Section 8.2).

The function `getPFID` returns the platform identifier of the current platform. The function `getPPFInfo` returns a list of pairs, consisting of platform identifiers and information about the possibly-provided functions on this platform. The maintenance of this list will be introduced in Section 8.3.1. The first pair is always the information about the current platform, i.e. the following equation must hold:

```
getPFID == getPPFInfo >>= return . fst . head
```

Splitting `getPPFInfo` into two functions, `getThisPPFInfo` returning the information of the current platform only and `getOthersPPFInfo` returning the list of pairs containing the information about other platforms, would be a safer approach. Nevertheless, we use `getPPFInfo` only, in order to keep the number of required platform functions a little bit smaller. Obviously, a concrete implementation can simply provide `getPPFInfo` based on `getThisPPFInfo` and `getOthersPPFInfo`.

The function application `hasPPFs pfids ppfnames` returns a list of platform identifiers. A platform identifier is element of this list if it is element of `pfids` and all needed possibly-provided functions which are part of `ppfnames` are available on that particular platform.

Obviously, it is possible to define `getPFID`, `getPPFInfo`, and `hasPPFs` in a non-monadic style assuming there is nothing really useful to be mentioned in the state about a mobile agent using one of these functions. As mentioned above, I/O in Haskell is also monadic, and by using the agent monad for `getPFID`, `getPPFInfo`, and `hasPPFs` it will be possible to encapsulate I/O. That way a mobile agent platform can for instance query a database for information about the possibly-provided functions.

The fifth monadic function, namely `replaceCF`, can be used to replace a part of the mobile agent code with a list of code fragments. In the function application `replaceCF pos old cfs` the part of `old` which is selected by `pos` will be replaced with `cfs`.

8.1.3 Mobile Agent Platform

The Haskell Mobile Agent Platform consists of three different types of platforms: The Agent Platform, the Home Platform, and the Proxy Platform. For further details on the implementation of the different platforms see Section 8.2.1, 8.2.2, and 8.2.3.

The life-cycle of a mobile agent starts and ends on the home platform. The mobile agent will be sent to an agent platform and migrates from agent platform to agent platform under its own control. If the mobile agent has finished its work, it will migrate back to the home platform.

In the above scenario the home platform has to be online continuously. Since this is not very practicable, particularly when the home platform is on a mobile device or on a personal computer, the approach presented in this thesis uses also proxy platforms. The main objective of a proxy platform is to store the mobile agent after its work has been done and provide an opportunity to download the mobile agent to the home platform. Furthermore, it is possible to send the agent directly from the home platform to the proxy platform.

Figure 8.1 shows the three different types of mobile agent platforms and the paths pictured as arrows to migrate from one to another.

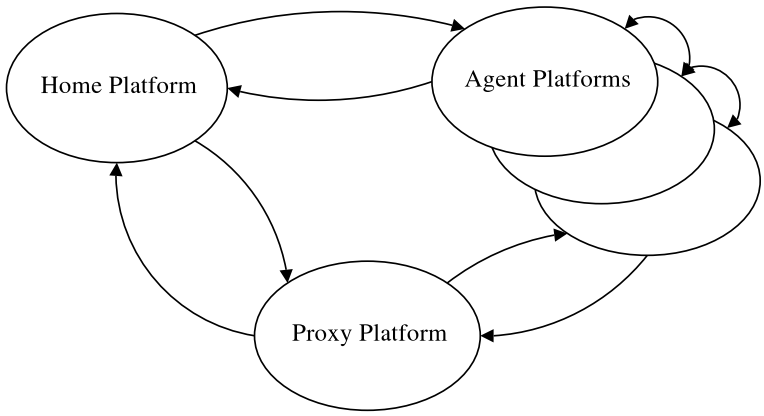


Figure 8.1: The mobile agent platforms and migration paths.

Mobile agent migration can be done in different ways. Three basic approaches are used in the prototypical implementation. The standard migration procedure is using a network connection through sockets. The messages used for migration are part of the Haskell Mobile Agent Platform Protocol which will be introduced in Section 8.3.1. In order to provide a secure transfer it is possible to use pre-connected tunnels based on the tunnelling mechanisms of SSH (Barrett and Silverman, 2001).

Another way to migrate is through electronic mail (see Section 8.3.2, (Crocker, 1982)). The agent platform sends the agent by electronic mail to a mailbox. This mailbox can either be read by a human user, who injects the agent to the next platform or by a mobile agent platform which handles the agent itself. The third way to migrate used in this approach is uploading a mobile agent through the Common Gateway Interface

(CGI, see Section 8.3.3, (Robinson and Coar, 2003)) onto a web-server that passes the mobile agent to a mobile agent platform.

8.2 Implementation of Mobile Agent Platforms

A part of the prototypical implementation of HaMAP is described below. Section 8.2.1 introduces the agent platform and explains the mechanism to prevent the mobile agent from using arbitrary I/O functions. Furthermore, the integration of security levels is presented. Section 8.2.2 and 8.2.3 give a short explanation of the implementation of the home platform and the proxy platform.

8.2.1 Agent Platform

The mobile agent migrates from platform to platform and does some calculations on each platform. After the mobile agent has migrated to an agent platform, this platform has to transform, compile, and execute the mobile agent. Figure 8.2 shows this procedure.

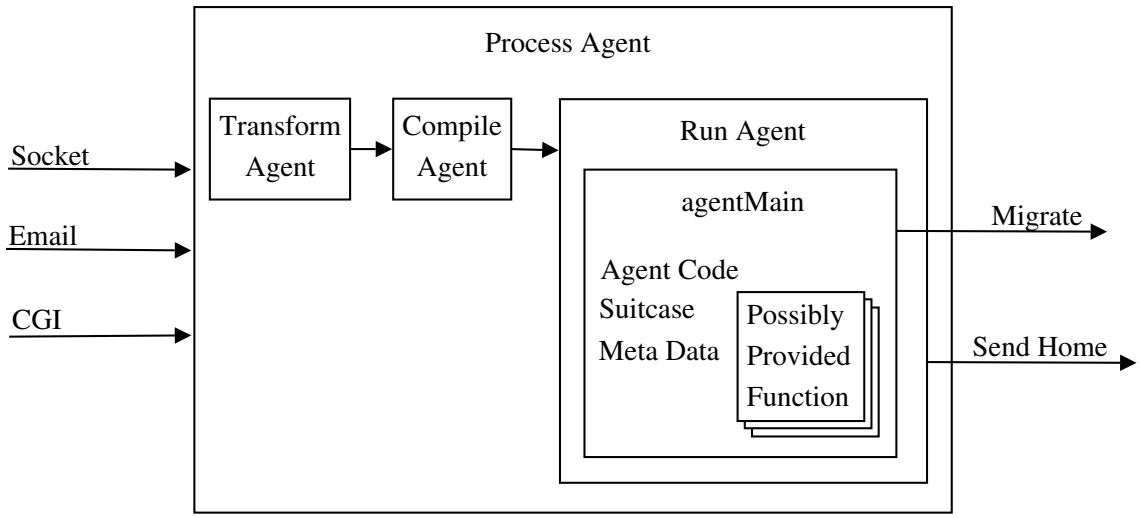


Figure 8.2: Processing a mobile agent on an agent platform.

Transformation, compilation and execution of a mobile agent is done under the control of the `processAgent` program. After the migration to a platform `processAgent` is executed. The first step is the transformation of the mobile agent code. This is done using the function `transformAgentCode`.

```
transformAgentCode :: Platform -> [CodeFragment] -> String
```

The first argument is a value of type `Platform`. This value is used to encapsulate all platform-dependent information and configuration parameters. Not only the value, but also the data type can be different on each platform, to fit the needs for the particular platform. Usually it contains the platform identifier and information about the possibly-provided functions. The second argument of `transformAgentCode` is the list of code fragments from the mobile agent. The result is the string representation of the compilable code.

As already mentioned, the objective of `transformAgentCode` is to remove all non-available possibly-provided functions. Therefore, all possibly-provided functions have to be encapsulated into a code fragment of the form `PPF ppfname ppfargs`. If a possibly-provided function occurs in a code fragment of the form `Code string`, it will not be possible to remove it when it is not available on the particular platform. All possibly-provided functions return their calculated value `val` of type `a` encapsulated in the value `Just val` of type `Maybe a`. By this means non-available possibly-provided functions can easily be replaced with a code chunk returning `Nothing`. In the prototypical implementation the following function is used:

```
ppfNotAvailable :: Agent m => String -> m (Maybe a)
```

The function application `ppfNotAvailable ppfname` adds an entry about `ppfname` being not available to the mobile agent logs and returns `Nothing`. With this design it is necessary that each possibly-provided function is used in the following way:

```
ppf args
 >>= maybe ( ... )          -- do something if ppf is not available
           (\ val -> ... ) -- do something with the value returned by ppf
```

While this requirement would be hard to demand for "normal" development of mobile agents, it is easy to ensure for mobile agents developed and transformed into Haskell code with HOPS.

The implementation of the transformation of the code fragments `Code string`, `ALL ppfnames codefragments`, and `OR ppfnames codefragments1 codefragments2` is straightforward, and therefore further explanation will be omitted at this point.

The transformation is done using a preprocessor, but it is principally also possible to use Template Haskell (Sheard and Peyton Jones, 2002). Template Haskell is an extension to Haskell that supports compile-time meta-programming. But in the context of

mobile agents the execution of arbitrary code while compiling is absolutely unacceptable. Therefore Template Haskell is not used in the prototypical implementation.

The transformed mobile agent code plus the suitcase and the meta data will be written into a file in order to compile the mobile agent. The transformed mobile agent code has to be encapsulated into a function, which returns a value of type `IO ()`, in order to be able to compile it into an executable program. The function used for the encapsulation is called `runAgent`.

```
runAgent :: (forall m .
              Agent m => [CodeFragment] -> Suitcase -> MetaData -> m ())
         -> [CodeFragment] -> Suitcase -> MetaData -> Platform -> IO ()
```

The type of `runAgent` is not a Hindley-Milner type, since rank-2 polymorphism is used in the first argument. The idea is based on a paper about encapsulating state in Haskell (Launchbury and Jones, 1994). John Launchbury and Simon Peyton Jones have pointed out in this paper that rank-2 polymorphism in the type of their state encapsulating function makes it impossible to use a state reference from one stateful thread in another. In the context of mobile agents the type of `runAgent` ensures that a mobile agent cannot do arbitrary I/O. In order to be able to explain how this works, one choice of a concrete agent monad is introduced below.

The agent monad on an agent platform manages a state, the agent platform state `APFState`. The `APFState` contains amongst other things information about the platform and logs about possibly-provided functions used by the mobile agent. The agent platform type `APF` and the inverse function `unAPF` are defined in the following way:

```
newtype APF a = APF (APFState -> IO (a, APFState))
unAPF (APF x) = x
```

This means `APF a` is a function type which takes an agent platform state of type `APFState`, does some I/O, and returns a pair consisting of the calculated value of type `a` and the new agent platform state. In order to be used for the mobile agent code, `APF` has to be an instance of the class `Monad`, i.e. `return` and `(>>=)` have to be defined for `APF`.

```
instance Monad APF where
  return a = APF $ curry return a
  f >>= g  = APF $ flip (>>=) (uncurry (unAPF . g)) . unAPF f
```

The function application `return a` takes an agent platform state and returns a pair consisting of `a` and the unchanged agent platform state. In the definition of `(>>=)` first `unAPF f` is applied to the platform state resulting in a value of type `IO (a,APFState)`. The function `flip (>>=) (uncurry (unAPF . g))` is then applied to this result, whereas the first argument of `flip` is the bind function of the `IO` monad. By this means `f` will be evaluated "and then" `g`.

For the illustration of the need of rank-2 polymorphism in the type of `runAgent` to ensure that a mobile agent cannot do arbitrary I/O, the function `runAgent'` without rank-2 polymorphism in the type and a example code chunk will be used. The implementation of the function `runAgent'` is exactly the same as the definition of the function `runAgent`, only the type has no rank-2 polymorphism.

```
runAgent' :: ([CodeFragment] -> Suitcase -> MetaData -> APF ())
          -> [CodeFragment] -> Suitcase -> MetaData -> Platform -> IO ()
```

When using the mobile agent code with `runAgent'` the mobile agent code containing the following code chunk can be type-checked and compiled successfully:

```
APF (\ s -> removeDirectory  "." >> return ((),s)) :: APF ()
```

This code chunk removes the current directory of the mobile agent process. Obviously, this is not wanted by the agent platform administrator. When using this code chunk with `runAgent`, type-checking will exit issuing the following error message:

```
Cannot unify the type-signature variable ‘m’ with the type ‘APF’
   Expected type: [CodeFragment] -> Suitcase -> MetaData -> m ()
   Inferred type: [CodeFragment] -> Suitcase -> MetaData -> APF ()
```

Although, the `APF` monad will be used in the definition of `runAgent`, the mobile agent code cannot contain the above code of type `APF ()` to be successfully type-checked, because of the rank-2 polymorphic type of the first argument of `runAgent`. The expected type is more general than the inferred one.

To sum up, with the rank-2 polymorphic type of the first argument of `runAgent` it is possible to use only monadic functions defined in the class `Agent`. These functions are, besides `migrate`, `getPFID`, `getPPFInfo`, `hasPPFs`, and `replaceCF`, the possibly-provided functions available on the particular platform. Other monadic functions, most notably functions defined in the `IO` monad, are not usable in the mobile agent code.

The Glasgow Haskell Compiler, which is used for compiling the mobile agent code, provides a back door into the `IO` monad for allowing I/O computation to be performed in a non-monadic context, using the function `unsafePerformIO`. In the context of the Haskell Mobile Agent Platform this is not a problem at all, since the function `unsafePerformIO` cannot be used unless a module containing it has been imported. Importing any module is not possible in the mobile agent code, since the mobile agent code is just one function, not a complete module.

By this means, the rank-2 polymorphism and the fact that the mobile agent code is just one function, not a module, ensures that the mobile agent cannot do any I/O, except for the I/O provided within the available possibly-provided functions.

In the implementation of `runAgent`, the mobile agent function `agent` will be applied to the code, the suitcase, and the meta data resulting in a value of type `APF ()`. The application of `unAPF` results in a function of type `APFState -> IO ((),APFState)`, which will be applied to an initial agent platform state.

```
1   runAgent agent code suitcase metadata platform =
2     unAPF (agent code suitcase metadata) (mkAPFState platform)
3      >>= \ (_,s) ->
4       let mobileAgent = mkMobileAgent code suitcase metadata'
5           metadata'   = metadata {logs = (pfID platform, logsForAgent s)
6                                          : logs metadata }
7       in migrateToPF (homePF metadata) mobileAgent
8          >> maybe (return ())
9                   (\proxyPF -> migrateToPF proxyPF mobileAgent)
10                  (proxyPF metadata)
11         >> maybe (return ())
12                  (\ownerEmail -> migrateByMail ownerEmail mobileAgent)
13                  (ownerEmail metadata)
```

As already mentioned, the mobile agent migrates under its own control using the function `migrate`, and `migrate` exits the mobile agent process after a successful migration. In this case the code in line 3 to line 13 will not be executed. If the migration was not successful, the mobile agent including the new logs (line 4 – 6) will be migrated to the home platform (line 7). If this fails and the mobile agent meta data contains a proxy platform, the mobile agent will be migrated to the proxy platform (line 8 – 10). If this also does not work and the mobile agent meta data contains the electronic mail address of the owner of the mobile agent, the mobile agent will be sent by electronic mail to this address (line 11 – 13).

Obviously, this kind of error handling only works if the mobile agent code terminates. Since `runAgent` is executed as a stand-alone process under the control of `processAgent`, it is possible to limit the execution time of the mobile agent process. After a time-out, the mobile agent process will be terminated and the mobile agent will be sent home. The time-out interval can be set depending on the used possibly-provided functions. Furthermore, the mobile agent can request an amount of execution time by including the value in its meta data. If the requested amount of execution time is too large, the agent platform can avoid executing the mobile agent and send it back to the home platform. If the mobile agent meta data also contains a list of agent platforms, the mobile agent wants to migrate to, the agent platform can send the mobile agent to one of these platforms. To avoid the problem of sending the mobile agent from platform to platform in an infinite loop, the current platform identifier will be removed from this list of platforms.

Integrating Security Levels

So far, all mobile agents are able to use the same set of possibly-provided functions on a particular agent platform. With a concept of security levels it is possible to provide different sets of functions for mobile agents depending on their trustworthiness and authentication. Within the approach presented in this thesis it is not necessary that all agent platforms use security levels. If the agent platform does not provide security levels or the authentication of a mobile agent has failed, the mobile agent can still be executed without any security level or in the lowest security level.

To run a mobile agent in a higher security level the following scenarios are conceivable in the context of HaMAP:

1. The mobile agent code is cryptographically signed. Therefore, the cryptographic signature of the mobile agent code is included in the meta data of the mobile agent. The platform needs the public key of the owner to verify the signature. The drawback of this is that the mobile agent must not transform its code.

2. The mobile agent has migrated over a trusted connection or from a trusted agent platform. This introduces the risk that if one of the trusted agent platforms is compromised, the whole network of trusted agent platforms is vulnerable.

3. The `processAgent` program can be started manually with a security level as optional command line argument. This is especially useful when using mobile

agents that require user interaction and need resources to interact with the user, e.g. more I/O functionality or access to the graphical user interface.

4. The agent platform can provide a function for authentication. After a successful authentication the mobile agent can be lifted to a higher security level. This is problematic, because the information needed for an authentication has to be part of the mobile agent and can be spied out by other platforms. Anyway, it is quite useful in a network of trustworthy agent platforms.

For scenario 1 – 3 a verification of the security level has to be included into the transformation function `transformAgentCode`. The security level has to be passed as additional argument to `transformAgentCode` and for each possibly-provided function the availability in the particular security level has to be checked.

For scenario 4 a verification of the security level has to be included in each possibly-provided function, except the possibly-provided functions in the lowest security level. The current security level has to be included in the agent platform state and has to be increased after a successful authentication. For each possibly-provided function in a higher security level, the current security level has to be checked before the platform provided function will be executed. The wrapper function used is called `execPPFIfAllowedInSecLevel`.

```
execPPFIfAllowedInSecLevel name ppf = APF $ \s ->
  if ppfAllowedInSecLevel (platform s) name (securityLevel s)
  then unAPF ppf s
  else return (Nothing
              ,addToLogsForAgent s $ name ++ " not allowed in SecLevel")
```

As long as the mobile agent programmer uses all possibly-provided functions in a `PPF` code fragment this concept is sufficient. Without security levels using possibly-provided functions in `Code` code fragments has no advantage. It only introduces the possibility that the mobile agent code is not type-checkable and compilable, because a used possibly-provided function, which is not available, was not removed in the `transformAgentCode` function. By this means one can assume that all mobile agent programmers use possibly-provided functions only in the `PPF` code fragment and, if they do not, this is no problem for the agent platform, since the mobile agent will not be executed.

With the introduction of security levels this changes. Possibly-provided functions that are included in a `Code` code fragment or are part of the arguments of a possibly-

provided function are neither checked in the `transformAgentCode` function nor are they checked during execution with the `execPPFIfAllowedInSecLevel` function. Consider the following code chunk, which could be part of the mobile agent code.

```
,Code "ppf1 >> "
,PPF  "ppf2" ["arg1 arg2"," >> ppf3 "]
```

The function `ppf2` will be checked, but `ppf1` and `ppf3` will not be checked. So, if `ppf1` or `ppf3` are not allowed in the current security level, nevertheless, they will be executed. This undermines the concept of security levels.

To solve this problem there are mainly three different approaches. The first solution is to check the complete code for occurrences of possibly-provided functions. The second option is to replace all occurrences of `PPF` code chunks with `return Nothing` and try to compile the code without importing any of the possibly-provided functions. If the compilation succeeds, the `Code` code chunks do not contain possibly-provided functions. But the argument list of the `PPF` code chunks have to be checked, too. This can be done by replacing the possibly-provided functions with dummy functions and trying to compile this.

The most elegant way is to use so-called "Security Cookies". A security cookie is a value of type `Cookie` which is passed as an additional argument to each possibly-provided function. The cookie is a random value, which will be included in the mobile agent code during the transformation. During the execution the same cookie is contained in the agent platform state. The evaluation of a possibly-provided function always starts with the comparison of the cookie passed as argument and the cookie contained in the agent platform state. Only if they are equal, the possibly-provided function will be executed.

In the above code chunk `ppf2` will be replaced with `ppf2 cookie`, whereas `ppf1` and `ppf3` will not be replaced. In this case the type-checking of `ppf1` and `ppf3` already fails. If a mobile agent programmer includes a cookie in the code after `ppf1` and `ppf3` it is almost impossible that this cookie is identical with the random cookie, which will be calculated during the `transformAgentCode` function.

Since the cookies will be included only locally on an agent platform, it is possible that some agent platforms use security cookies whereas others do not.

8.2.2 Home Platform

The life-cycle of a mobile agent begins and ends on its home platform. The home platform takes a mobile agent generated by HOPS and pushes it to an agent platform or a proxy platform. If the mobile agent has finished its work, the home platform accepts the mobile agent on a socket, by electronic mail or through the common gateway interface, and presents the value to the user. In case the home platform is not always online, the mobile agent will be stored on the proxy platform and the home platform is able to fetch the mobile agent from the proxy platform once the home platform will get online again.

The mobile agent generated with the code output facility of HOPS consists of the mobile agent code, the suitcase and a function to extract the value the user wants to see from the suitcase. The home platform has to split off the function to extract the value and the home platform has to insert the mobile agent meta data. The home platform generates an identifier for the mobile agent which will be included in the meta data, besides the platform identifier of the home platform, the electronic mail address of the owner of the mobile agent and other optional information. If needed, the home platform also calculates the cryptographic signature of the mobile agent code and puts it in the corresponding meta data field. With the mobile agent identifier it is possible to associate the function to extract the value and the mobile agent, once the mobile agent returns. The extracting function will be stored on the home platform, and the mobile agent will be sent either to an agent platform or to a proxy platform.

8.2.3 Proxy Platform

The proxy platform is used to relay mobile agents from a home platform to an agent platform and vice versa. The home platform sends a mobile agent to the proxy platform. The proxy platform forwards the mobile agent either to one of the agent platforms from the list of platform identifiers included in the mobile agent meta data or, if the list is empty or none of the platforms can be reached, to another platform known by the proxy platform. Furthermore, a proxy platform can implement an analysis of the mobile agent code with respect to the needed possibly-provided functions and can send the mobile agent to an agent platform providing the needed set of functions.

If a mobile agent has migrated from an agent platform to a proxy platform, the proxy platform tries to send the mobile agent at regular intervals to the home platform. Furthermore, the proxy platform runs a server program which allows a home platform to fetch the mobile agent.

8.3 Migration and Inter-Platform Communication

Migration of mobile agents and communication between agent platforms can be done in different ways. Section 8.3.1 introduces the Haskell Mobile Agent Platform Protocol, a network protocol based on TCP/IP which has been completely implemented in the prototype. Section 8.3.2 describes a simple migration mechanism using electronic mail and Section 8.3.3 discusses the migration over the common gateway interface. Both of them, the electronic mail and the common gateway interface migration, are only partially integrated in the prototypical implementation yet.

8.3.1 Haskell Mobile Agent Platform Protocol

The Haskell Mobile Agent Platform Protocol is a network protocol based on TCP/IP. It makes use of the socket mechanism provided by the Glasgow Haskell Compiler GHC. Processes on different machines or on the same machine can exchange messages through data streams using the concept of internet sockets. In order to use these messages for migration and communication of mobile agent platforms a protocol has to be defined. The Haskell Mobile Agent Platform Protocol is based on the `APPRequest` and the `APPResponse` data types.

```
data APPRequest = TREQ
                | DATA MobileAgent
                | FETCH PFID MAID
                | SENDPPFINFO PFID
                | EAPPRequest

data APPResponse = TACK | TREJ
                 | ACK PFID MAID
                 | NAVAIL PFID MAID
                 | PPFS [(PFID,(EpochTime,[PPFInfo]))]
                 | SHUTDOWN PFID
                 | EAPPResponse
```

A value of type `APPRequest` is called an agent platform protocol request, and a value of type `APPResponse` is called an agent platform protocol response. In order to send a message to a socket, a process has to listen on this socket. This process is called `agentd`, the agent daemon. The term "daemon" is used in the context of UNIX and the Internet for a process that runs silently and permanent as a background process. All kind of server processes, e.g. a web server or an ftp server are daemon processes on the machine they are running on.

To connect to a daemon process, the communication partner has to know the machine identifier, i.e. the Internet Protocol number or the host-name, and the port number. In the context of HaMAP this information is contained in the platform identifier.

The Haskell Mobile Agent Platform Protocol can be divided into a transfer and a communication part. With the transfer part it is possible to migrate and fetch mobile agents. Messages used in the transfer part of the protocol are: `TREQ`, `TACK`, `TREJ`, `DATA ma`, and `ACK pfid maid`. The messages `FETCH pfid maid`, `NAVAIL pfid maid`, `PPFS ppfs`, `SENDPPFINFO pfid`, and `SHUTDOWN pfid` are used for the inter-platform communication. The `EAPPRequest` and the `EAPPResponse` constructors represent error messages, which can be used to reset the connection. The error messages are also used for the handling of communication errors for example if an incoming message cannot be parsed.

Migrating a Mobile Agent

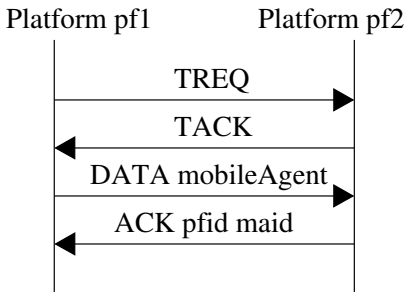


Figure 8.3: A migration of a mobile agent

When a mobile agent wants to migrate from platform `pf1` to platform `pf2`, `pf1` tries to establish a connection to `pf2`. If the connection is established, `pf1` sends a transfer request message `TREQ` to `pf2`. If `pf2` accepts the transfer, it will answer with the transfer acknowledge message `TACK` (see Figure 8.3). Otherwise, `pf2` sends the transfer reject message `TREJ` and closes the connection (see Figure 8.4). If `pf1` received the transfer

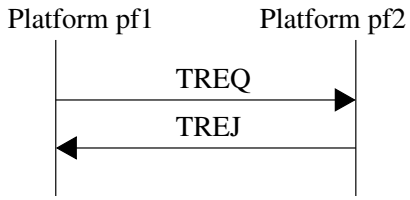


Figure 8.4: A migration request, which has not been acknowledged

acknowledge message, it sends the mobile agent in a data message `DATA mobileAgent`. `pf2` then answers with the acknowledge `ACK pfid maid`, where `pfid` is the identifier of the home platform and `maid` is the identifier of the mobile agent.

Fetching a Mobile Agent

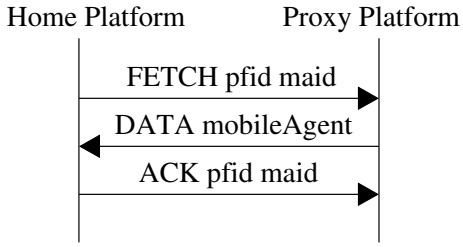


Figure 8.5: Fetching of a mobile agent

A mobile agent can be fetched to the home platform from a proxy platform. The home platform sends the fetch request `FETCH pfid maid` to the proxy platform. As mentioned before, the combination of the platform identifier `pfid` and the mobile agent identifier `maid` is a unique identifier of a mobile agent. If the mobile agent specified by `pfid` and `maid` is available on the proxy platform, the proxy platform will send this mobile agent as a data message. The home platform acknowledges with `ACK pfid maid` (see Figure 8.5). If the mobile agent is not available, the proxy platform sends

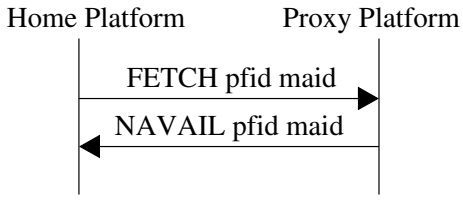


Figure 8.6: A fetch request of a mobile agent, which is not available

the non-available message `NAVAIL pfid maid` and closes the connection (see Figure 8.6).

Inter-Platform Communication

Each agent platform and each proxy platform maintains a list of platform provided functions on other platforms. This list is especially useful for a mobile agent that wants to migrate to an agent platform where a particular set of possibly-provided functions is available. The mobile agent can check this with the `getPPFInfo` and the `hasPPFs` function introduced in Section 8.1.2.

Obviously, it is important that the list of possibly-provided functions on other agent platforms is up-to-date. An agent platform can change its set of possibly-provided functions or it can be shut down. Moreover, new agent platforms can be made available.

To keep the information up to date, inter-platform communication will be used. This could be done also with mobile agents, but this kind of computation is more efficient with direct communication of mobile agent platforms, especially in this approach, where a mobile agent is transformed, compiled and executed, and not only interpreted.

An element `(pfid,(time,ppfs))` of the list of possibly-provided functions has the type `(PFID,(EpochTime,[PPFInfo]))`. The value `ppfs` is the list of possibly-provided functions available on the agent platform `pfid`. The value `time` is the timestamp of the information. With this timestamp it is possible to distinguish between newer and older information about an agent platform, and it is possible to define a time interval after which the information expires.

To avoid that the information on other agent platform expires, an agent platform has to broadcast its list in regular intervals to all known agent platforms. This is done by using the possibly-provided functions message `PPFS list`. An agent platform receiving this message merges this information with its own information in the following way:

- Information which is older than the specified interval is removed from the list.
- Information about agent platforms that is not yet part of the own information is inserted.
- Information about agent platforms that is already part of the own information is used to replace the old information about this agent platform, unless the timestamp of the old information is newer.

It is known to the author, that sending always the complete list does not scale very well. Anyway, it is used in this first approach to provide a simple mechanism to

distribute information about a new platform automatically. By this means a new agent platform should sent a list containing information about itself to an existing agent platform. Furthermore, a new platform can request the list from another agent platform by sending the message `SENDPPFINFO pfid` to the other platform. The value `pfid` has to be the platform identifier of the new agent platform. The other agent platform answers with the `PPFS list` message.

An agent platform that receives information about a new agent platform can broadcast this information immediately. To avoid high traffic it is recommended that only an agent platform which receives a singleton list[3] broadcasts this information immediately.

Even though information about an agent platform that is no longer available expires after a specified time interval, it is recommended that an agent platform sends the shutdown message `SHUTDOWN pfid` to all known platforms. Again, `pfid` is the identifier of the sending agent platform. The receiving agent platforms can remove the information about this agent platform immediately and does not need to wait for the expiration.

As mentioned above, proxy platforms also maintain a list of possibly-provided functions available on other platforms. Since proxy platforms are never receiving a list from an agent platform automatically, proxy platforms have to use the `SENDPPFINFO pfid` message in a regular interval to request the list from an agent platform in order to keep their list up to date. Home platforms do not communicate with other platforms, they are used only for sending, receiving and fetching agents.

8.3.2 Electronic Mail

Electronic mail (Crocker, 1982) can be used to migrate mobile agents. Migration of mobile agents by electronic mail can be done automatically or manually on the sending and on the receiving communication side. On the sending side the file containing the mobile agent has to be attached to an electronic mail message and has to be sent either to an electronic mail address of an agent platform or a proxy platform, or to an electronic mail address of a human user. A platform receiving a mobile agent in an electronic mail message splits off the attached file containing the mobile agent. After that, the platform can process the mobile agent as usual. A mobile agent that wants to migrate by electronic mail to a mobile agent platform has to pass the address `email` as value `Email_PF email` of type `PFID` to the `migrate` function.

[3] A singleton list is a list containing exactly one element.

8.3.3 Common Gateway Interface

The Common Gateway Interface (CGI) (Robinson and Coar, 2003) is a standard for interfacing external applications with web servers. In the context of HaMAP this is used for passing a file containing a mobile agent to an agent platform or a proxy platform. Furthermore, it is also possible to download a mobile agent that has been stored on a proxy platform.

9 Transforming I-DSL to HaMAP-Code

The last steps of the development of a mobile agent in HOPS are the transformation of an I-DSL mobile agent to a HaMAP mobile agent and the generation of Haskell code using the simple code output facility of HOPS. In Figure 9.1 the HOPS strategy for the transformation from I-DSL to a HaMAP DAG is shown. This strategy references six strategies, namely the `ReplaceAgentCombinators` strategy, the `UseMetaData` strategy, the `FlattenBind` strategy, the `CodeTransformation` strategy, the `HaMAP_Sharing` strategy, and the `DistinguishPPFName` strategy.

The `ReplaceAgentCombinators` strategy is used to replace the agent combinators with bricks representing appropriate Haskell code. It is introduced in Section 9.1. The `UseMetaData` strategy, which replaces meta data values with the appropriate access functions, is explained in Section 9.2. The monad laws are applied to the mobile agent DAG by the `FlattenBind` strategy (see Section 9.3). The functions to transform the mobile agent code during the execution on an agent platform are inserted by the `CodeTransformation` strategy (see Section 9.4). The `HaMAP_Sharing` strategy transforms shared sub-DAGs and is introduced in Section 9.5. The result of the transformation is a DAG which can be used to generate code using the simple code-output mechanism of HOPS. The simple code-output rules and the `DistinguishPPFName` strategy are introduced in Section 9.6. The `DistinguishPPFName` strategy is needed to distinguish `PPFName` bricks which are successors of meta agent combinators from `PPFName` bricks which are successors of the `hasPPFs` brick.

9.1 Replace Agent Combinators

The `ReplaceAgentCombinators` strategy is used to replace the internal agent and the basic agent combinators with bricks representing the appropriate Haskell code.

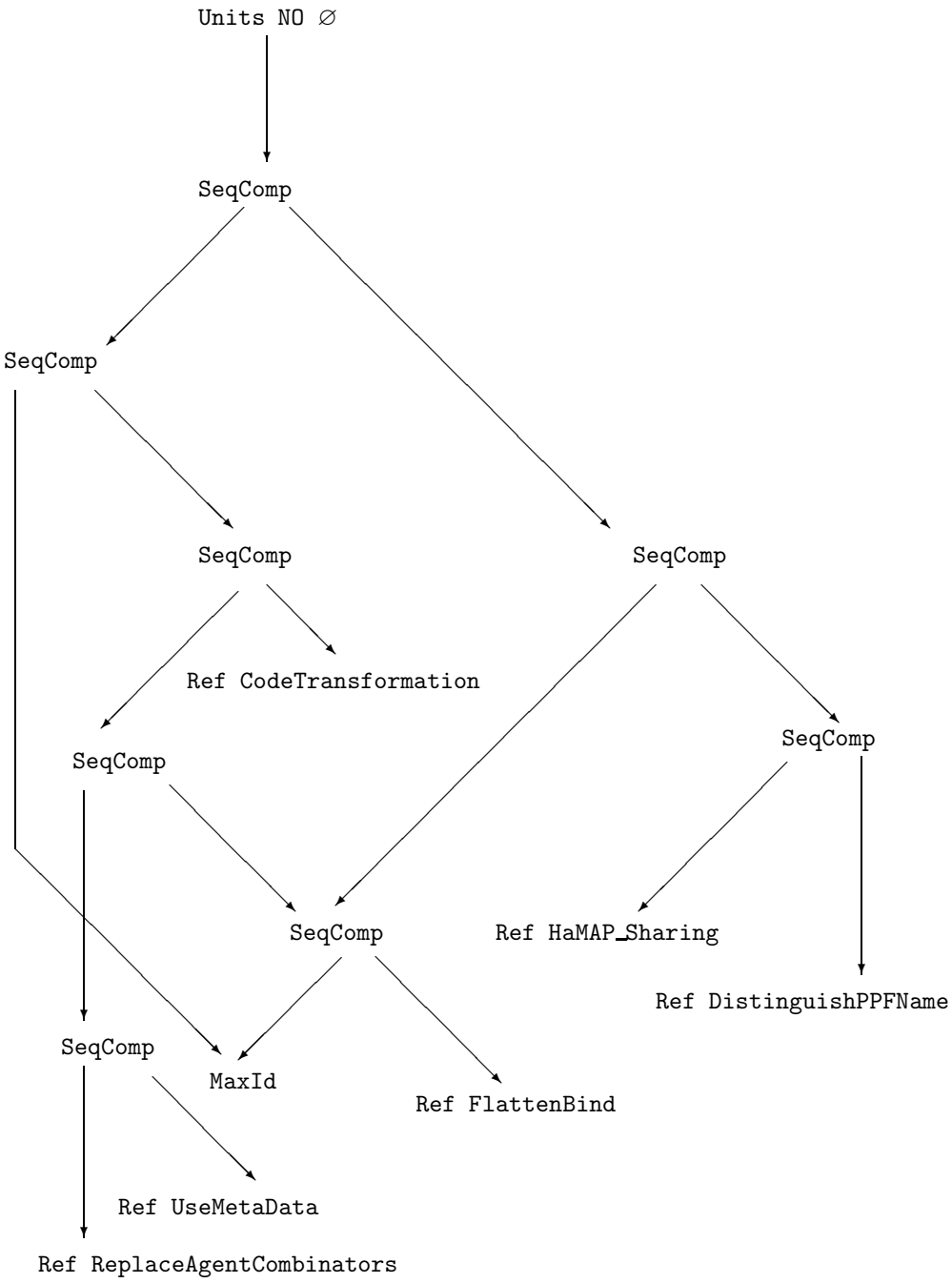


Figure 9.1: Strategy to transform an I-DSL mobile agent into a HaMAP mobile agent

Furthermore, the internal mobile agent will be replaced with the HaMAP mobile agent. Since mobile agent code is encapsulated in the agent monad (see Section 8.1.2) the basic bricks for this purpose are `bind` and `return`. The HOPS declaration of both is shown in Figure 9.2.

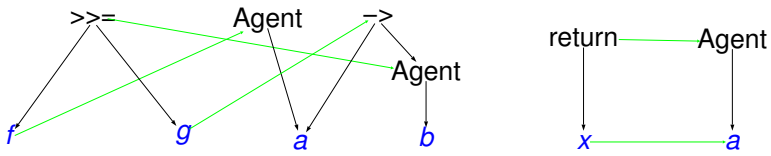


Figure 9.2: HOPS declaration of `bind` and `return`

While navigating top-down the `ReplaceAgentCombinators` strategy applies rules from a set of rules with the label `AgentCombinatorsToHaskell`. The strategy is shown in Figure 9.3.

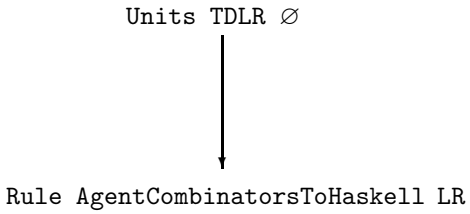


Figure 9.3: Strategy to transform primitive agents, basic agent combinators, and the internal mobile agent into Haskell

The transformation of an internal agent to the appropriate Haskell version is straightforward. The corresponding HOPS rule is shown in Figure 9.4.

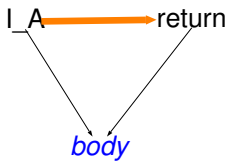


Figure 9.4: Rule to transform an internal agent into Haskell

The $\lambda_{\texttt{Agent}}$ combinator can be replaced with a λ abstraction. When replacing the $@_{\texttt{Agent}}$ combinator with `bind` the sequence of the successors has to be changed. The HOPS rules for the $\lambda_{\texttt{Agent}}$ combinator and the $@_{\texttt{Agent}}$ combinator are shown in Figure 9.5.

The transformation rules for the $(,)_{\texttt{Agent}}$ combinator and for the $\pi_{\texttt{Agent}}$ combinator are shown in Figure 9.6. The transformation rule for the $\rho_{\texttt{Agent}}$ combinator is similar to the rule for the $\pi_{\texttt{Agent}}$ combinator except for using ρ instead of π.

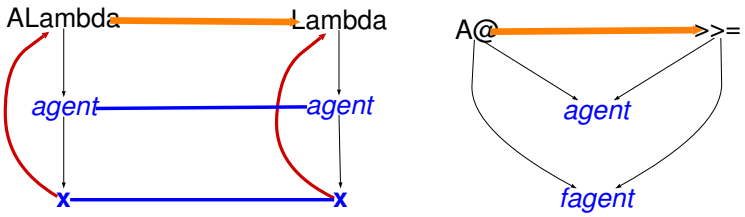


Figure 9.5: Rules to transform the $\lambda_{\texttt{Agent}}$ combinator and the $@_{\texttt{Agent}}$ combinator into Haskell

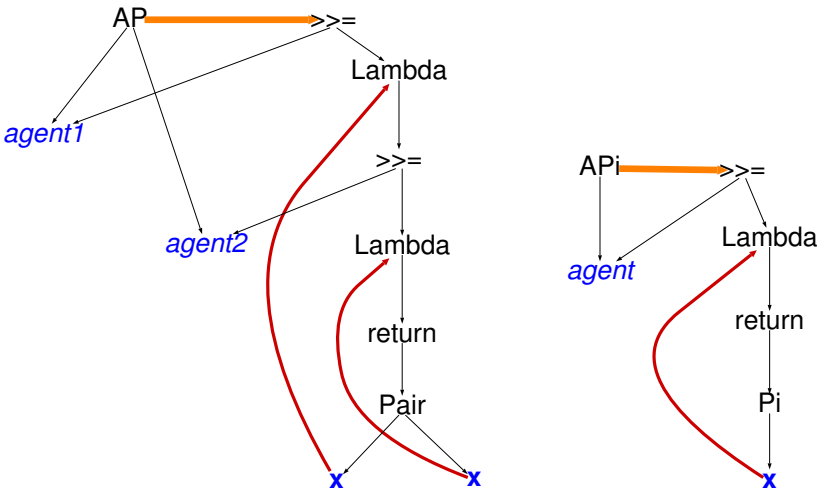


Figure 9.6: Rules to transform the $(,)_{\texttt{Agent}}$ combinator and the $\pi_{\texttt{Agent}}$ combinator into Haskell

The HaMAP mobile agent consists of the `code`, the `initial suitcase`, and the `value from suitcase` function. The `code` is a monadic function with three arguments returning the value `()`. The three arguments are the list of code fragments representing the mobile agent code, the suitcase in its string representation and the meta data suitcase. The `initial suitcase` is the initial value of the global suitcase of the mobile agent, and the `value from suitcase` function is the function to extract the value from the suitcase. The HOPS declaration of the HaMAP mobile agent is shown in Figure 9.7.

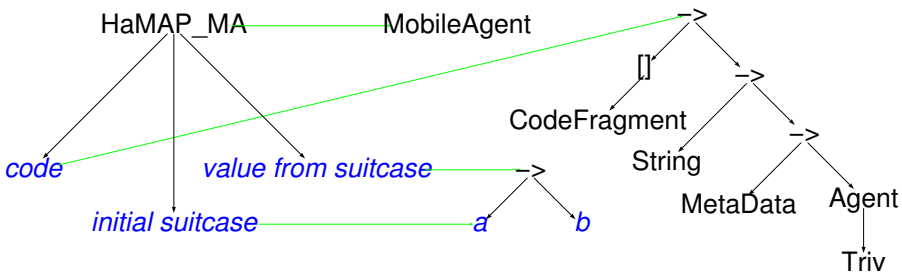


Figure 9.7: HOPS declaration of the HaMAP mobile agent

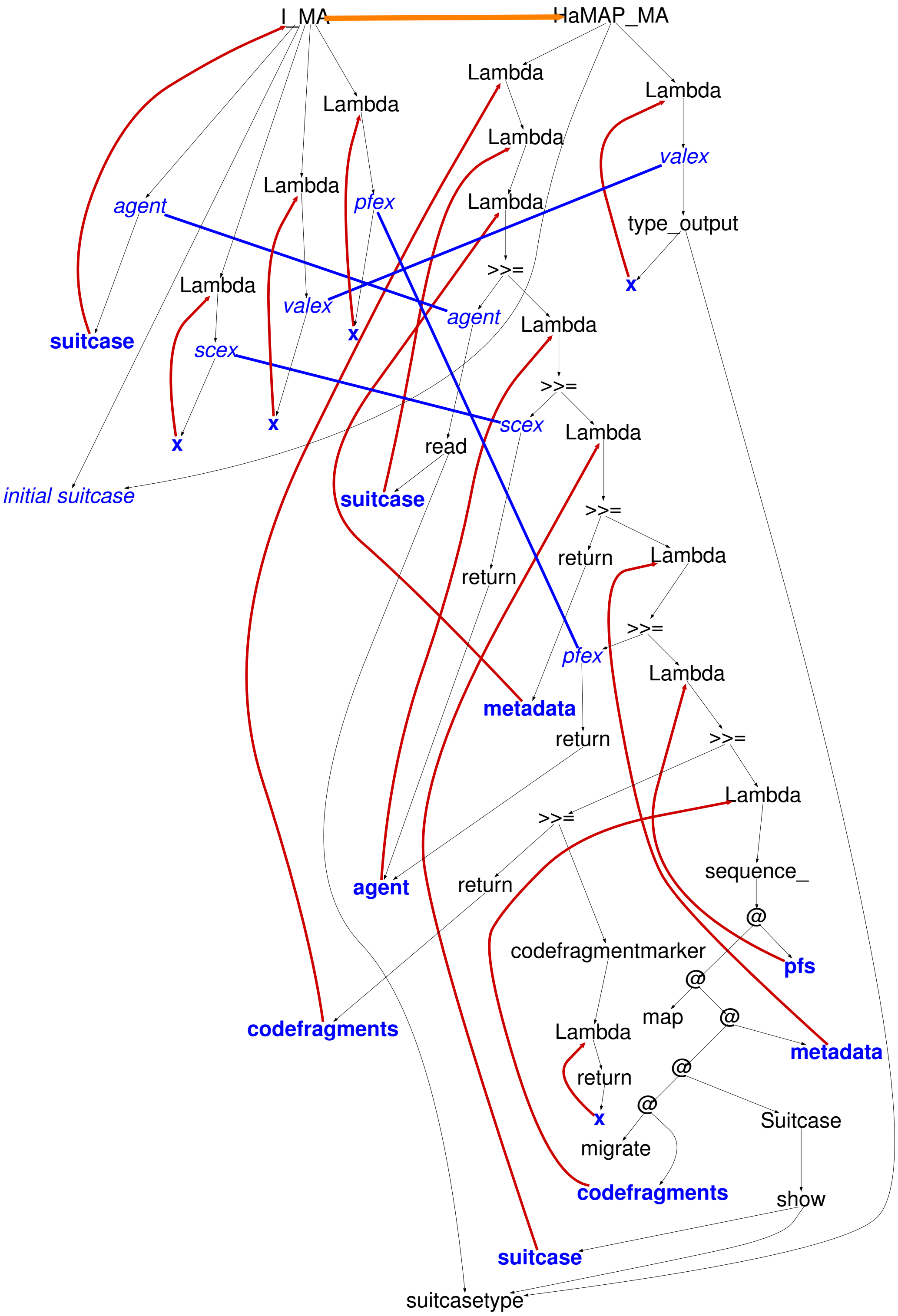


Figure 9.8: Rule to transform an I-DSL mobile agent to a HaMAP mobile agent

The rule to transform an I-DSL mobile agent brick to a HaMAP mobile agent brick is shown in Figure 9.8. The first successor of the HaMAP mobile agent is the monadic

function which represents the mobile agent code. The `agent` is identical to the `agent` in the internal mobile agent. Since the `suitcase` variable of the HaMAP mobile agent represents the string representation of the actual suitcase value the `read` function is inserted between the `agent` and the `suitcase`.

Normally, the λ occurrences in the left-hand side of a rule like the one shown in Figure 9.8 make the rule harder to fit into transformation sequences. However, since the I-DSL mobile agent has been automatically generated by the transformation sequences introduced in Chapter 7 the λ's are guaranteed to be available at this stage of transformation.

In Haskell it is necessary that the type of a value which should be converted using the read function and the show function is instance of the `Read` class respectively of the `Show` class. The `suitcasetype` brick as second successor of `read`, `show`, and `type_output` is used to make sure the type of the suitcase does not contain any type-variables in the generated code.

The `scex` and the `pfex` are utilised to extract the suitcase respectively the list of platforms from the result of `agent`. The `metadata` is simply returned in this phase of the transformation. This gives the possibility to change the meta data within the `UseMetaData` strategy (see Section 9.2).

The code fragments representing the mobile agent code are returned by the function marked with `codefragmentmarker`. Using this marker the code transforming functions can be inserted with the `CodeTransformation` strategy which is described in Section 9.4. The sub-DAG induced by the `sequence_` brick is responsible for the migration to one of the agent platforms in the list of agent platforms calculated by the agent.

9.2 Meta Data

The aim of the `UseMetaData` strategy is to replace all UI-DSL bricks representing an element of the meta data with a function returning this element from the meta data suitcase of the mobile agent. The `UseMetaData` strategy is shown in Figure 9.9.

In Figure 9.10 the rule for the `homePF` — the home platform which is element of the meta data — is shown as an example for rules used within the `UseMetaData` strategy.

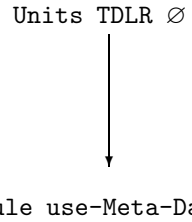


Figure 9.9: `UseMetaData` strategy

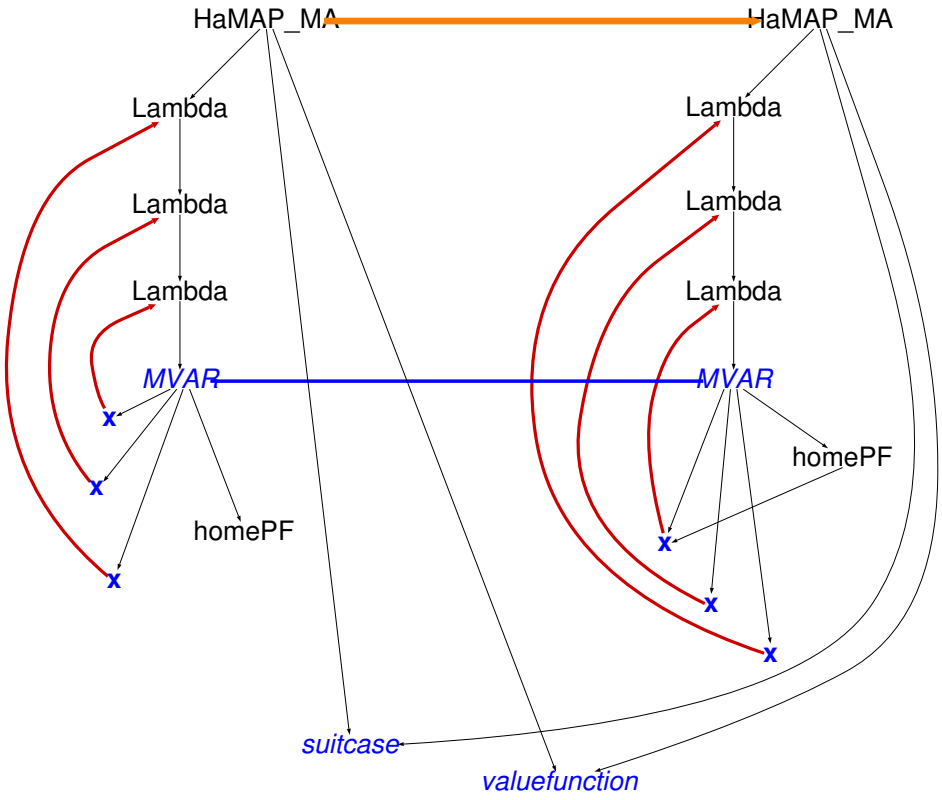


Figure 9.10: Rule to transform the UI-DSL `homePF` brick into the HaMAP `homePF` function

9.3 Monad Laws

As already mentioned in Section 8.0, a monad is a mathematical structure for which the following three laws will hold:

```
(return x) >>= f == f x
m >>= return     == m
(f >>= g) >>= h  == f >>= (\x -> g x >>= h)
```

The HOPS rules representing those monad laws are shown in Figure 9.11.

Within the `FlattenBind` strategy the monad rules shown in Figure 9.11 are applied to remove unnecessary parts using the first two laws and to obtain a “flat” structure

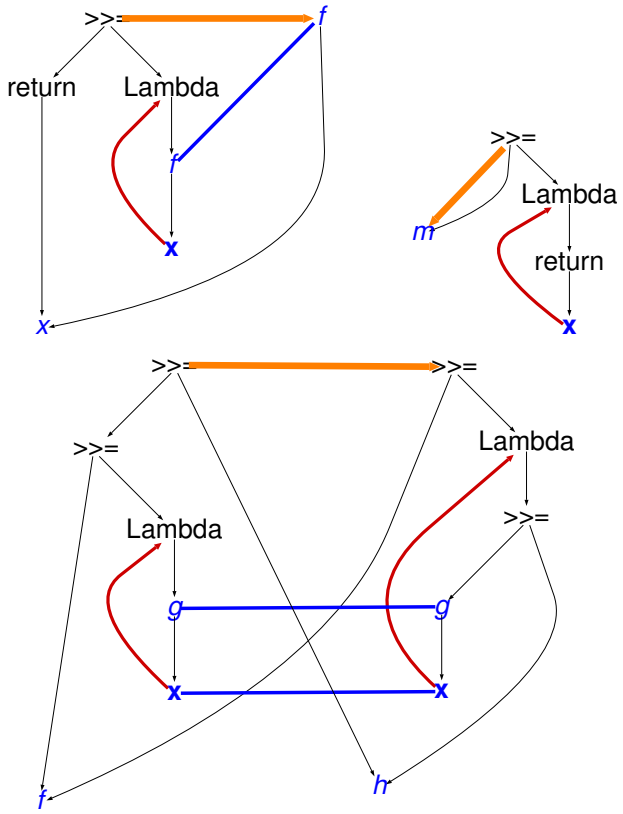


Figure 9.11: HOPS declaration of the monad laws

with respect to `>>=` using the third law. Flat structure with respect to `>>=` means that the DAG induced by the first successor of `>>=` should not contain any `>>=` brick. In Figure 9.12 the `FlattenBind` strategy is shown.

The intention of the `FlattenBind` strategy is to reorder the monadic computations in a way such that the value of a monadic computation is available in all following monadic computations. In the code chunk `(f >>= g) >>= h` , for instance, the value of `f` is available in `g` but not in `h` unless `g` returns the value of `f`.

After applying the `FlattenBind` strategy it is possible to optimise the code using rules like the one shown in Figure 9.13 which removes a duplicate of a monadic computation.

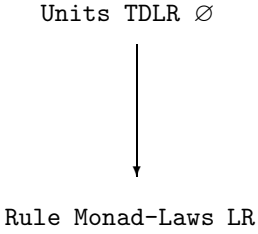


Figure 9.12: `FlattenBind` strategy

The rule shown in Figure 9.13 is part of the `HaMAPSharing` strategy which is described in Section 9.5.

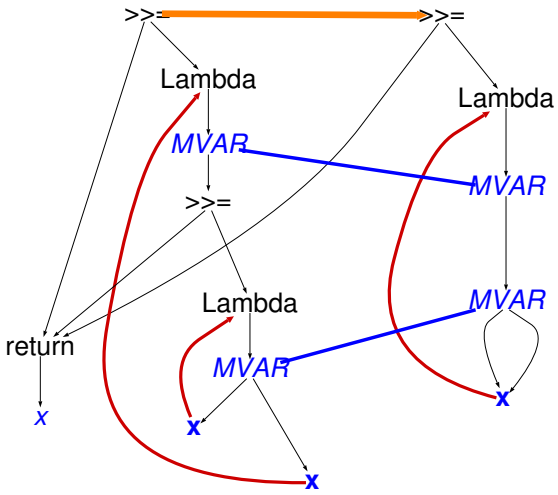


Figure 9.13: Rule to remove a duplicate of a monadic computation

In Figure 9.14 an example is shown to illustrate the mode of operation of the `Flatten-Bind` strategy and the subsequent optimisation. The DAG on the left hand side is the starting point. The DAG in the middle of Figure 9.14 is the result after applying the `FlattenBind` strategy to the DAG on the left hand side. The DAG on the right hand side is the result of the application of the rule shown in Figure 9.13 to the DAG in the middle of Figure 9.14.

9.4 Code Transformation

The `CodeTransformation` strategy is used to insert the functions to transform the mobile agent code during the execution on an agent platform. Obviously, this strategy should be applied only to the `code` of the mobile agent. Therefore, the term-graph pattern which is shown in Figure 9.15 is used to encapsulate the `CodeTransformation` strategy.

The term graph pattern, which matches on the top-node of the DAG, indicates with the integer `1` near the `code` node that the first action should be applied to the `code` node. The first action is the application of the `CodeTransformation` strategy which is described below. Since there are no further integers in the term graph pattern, this action is the only action which takes place.

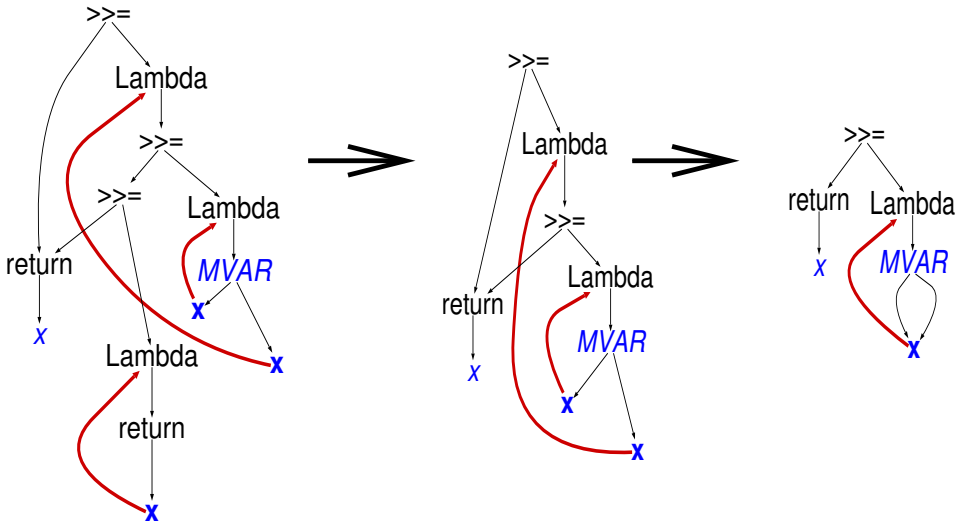


Figure 9.14: Example DAG, same Example DAG after applying the `FlattenBind` strategy, and same Example DAG after applying the `FlattenBind` strategy and the rule shown in Figure 9.13

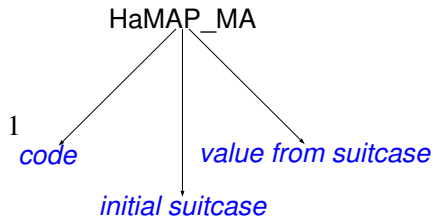


Figure 9.15: Term-graph pattern for the `CodeTransformation` strategy

While navigating top-down with local restart the `CodeTransformation` strategy replaces the `Replace` combinator, the `ValueMarker` and the `OldValueMarker`. The `CodeTransformation` strategy is shown on Figure 9.16. The rules which are element of the set of rules with the label `CodeTransformation` are described in the remainder of this section.

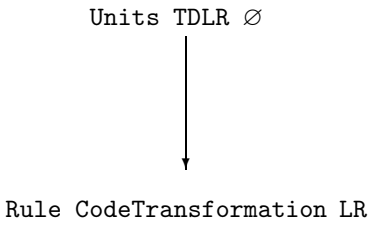


Figure 9.16: `CodeTransformation` strategy

In order to explain the rules to insert the code transforming functions, two new bricks have to be introduced first. The `Replaceable` brick is used to mark the code chunk

`code` which will be replaced. The `Replaceable` brick corresponds to the HaMAP `CodeFragment` constructor `Replaceable` (see Section 8.1.1). The `int` is used to ensure that the replaceable code and the replacement for this code are of the same type `a`. Therefore, `int` has to be a meta variable which is used as first successor of the `Replaceable` brick and as first successor of the `replaceCF` brick which will be introduced in the following paragraph. Using a meta variable instead of a constant brick or a DAG has the advantage that the meta variable will not be identified with another meta variable and, thus, can be distinguished from each other. By this means, each `int` meta variable is a global identifier for exactly one replaceable code chunk and its replacement. The name `int` has been chosen, because an integer is used in the Haskell code to represent this identifier. The `Replaceable` brick is shown in Figure 9.17.

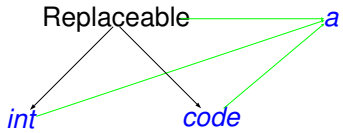


Figure 9.17: `Replaceable` brick

The other new brick is the `replaceCF` brick which is shown in Figure 9.18. The `replaceCF` brick corresponds to the HaMAP function `replaceCF` (see Section 8.1.2). There has to be always a corresponding `Replaceable` brick for the `replaceCF` brick which uses the same `int`. The replaceable code chunk in `old` will be replaced with the replacement `cf`.

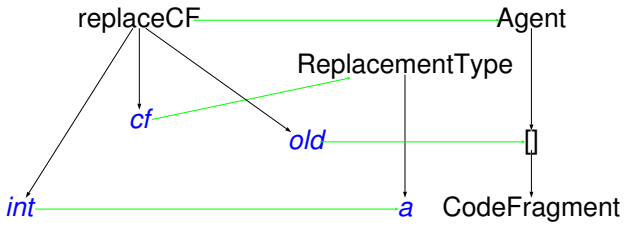


Figure 9.18: `replaceCF` brick

In Figure 9.19 the rule to replace the `Replace` combinator is shown. The second successor of the `Replace` combinator, namely `old`, will become successor of a `Replaceable` brick. The `cond` function is applied to the value calculated by `old` in order to calculate the condition for the replacement. Only if the calculated value is true, `old` will be replaced. In case `old` shall be replaced the replacement has to be calculated first by the application of the function `new` to the value calculated by `old`. The sub-DAG to replace `old` will become also successor of a `Replaceable` brick, because once `old`

has been replaced the replacement function is not needed anymore. After replacing `old` the replacement function will be replaced with `return c` using the corresponding `replaceCF` sub-DAG.

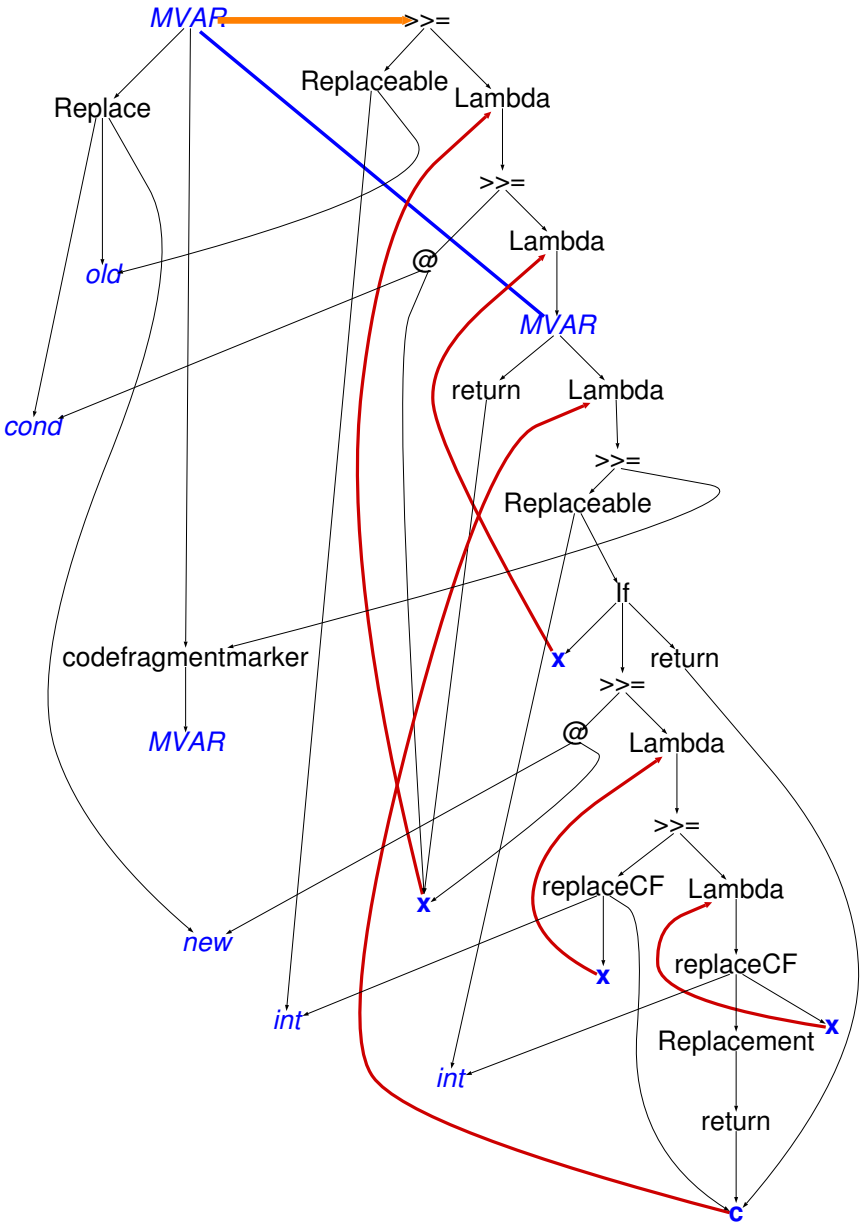


Figure 9.19: Rule to replace the `Replace` combinator with code transforming functions

In Figure 9.20 the rule to replace the `ValueMarker` and the `OldValueMarke` with code transforming functions is shown. The main difference between this rule and the rule shown in Figure 9.19 is the fact that this rule does not include the code transforming functions as successor of a `Replaceable` brick, because the replaceable code fragment has to be replaced on each platform, not only once.

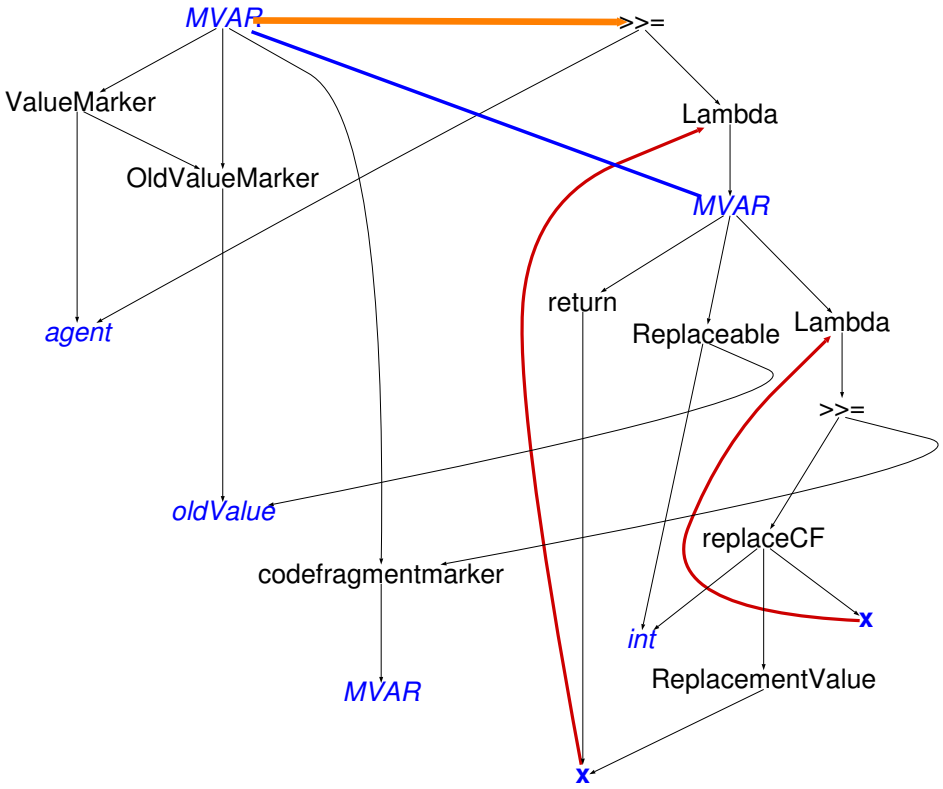


Figure 9.20: Rule to replace the `ValueMarker` and the `OldValueMarke` with code transforming functions

9.5 HaMAP Sharing

The `HaMAP_Sharing` strategy consists of two strategies, namely the `SharingPattern` strategy and the `ReverseSharing` strategy, which are applied sequentially to the HAMAP mobile agent DAG. The `HaMAP_Sharing` strategy is shown in Figure 9.21.

The `SharingPattern` strategy uses a term-graph pattern similar to the term-graph pattern shown in Figure 9.15 to ensure that the rules are only applied to the `code` part of the HaMAP mobile agent DAG. The `SharingPattern` strategy uses a rule which is quite similar to the rule used in the `Sharing` strategy (see Section 7.2). In Figure 9.22 the `SharingPattern` strategy is shown. The rule is shown in Figure 9.23. Moreover, rules like the one shown in Figure 9.13 are used.

The `ReverseSharing` strategy is used to reverse the application of the rule shown in Figure 9.23 in two special cases: In the `PPFName` parts of the meta agent combinators and if the second successor of a `@` brick is a bound variable. For the latter case the rule shown in Figure 9.24 is applied. In Figure 9.25 the `ReverseSharing` strategy is shown.

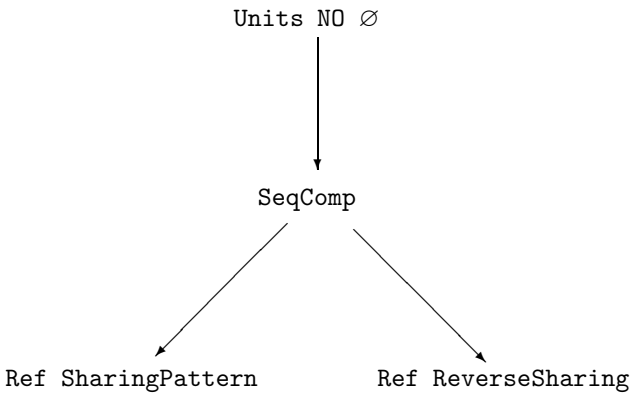


Figure 9.21: `HaMAP_Sharing` strategy

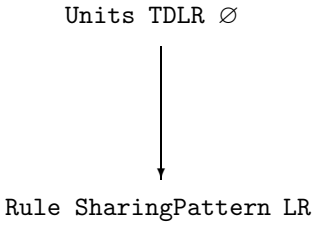


Figure 9.22: `SharingPattern` strategy

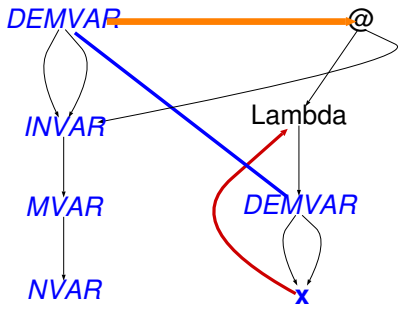


Figure 9.23: HaMAP sharing rule

Reversing in the `PPFName` parts of the meta agent combinators is necessary, because these parts are already needed for the preprocessor of the Haskell Mobile Agent Platform (see Section 8.2.1) which transforms the mobile agent code before its compilation and execution. In Figure 9.26 two rules are shown exemplary for the rules used in the `ReverseSharing` strategy.

9.6 Code Output

The final step to create Haskell code from a mobile agent DAG is code output using the simple code output facility of HOPS (see Chapter 2). With this code output facility it is

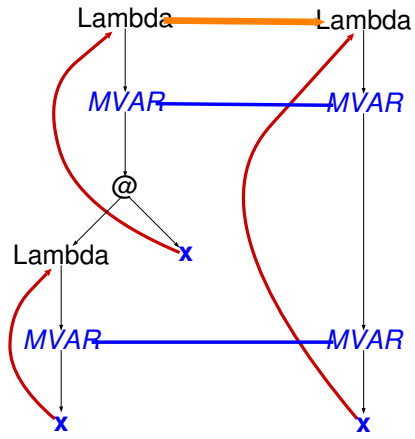


Figure 9.24: Beta reduction if the argument is a bound variable

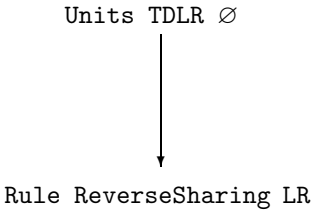


Figure 9.25: `HaMAP_Sharing` strategy

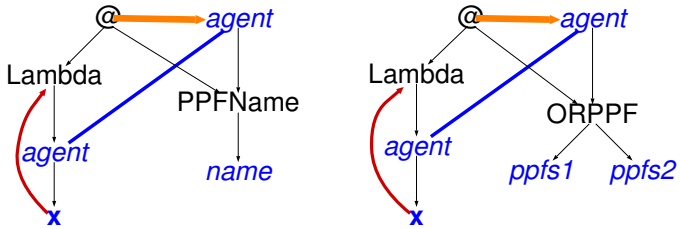


Figure 9.26: Two rules to reverse the sharing of the `PPFName` part of a meta agent combinator

not possible to define different output strings for one brick in dependence on its context, i.e. the successors or predecessors of this brick. But the code generated by the `PPFName` bricks should be different depending on whether they occur as successor of a meta agent combinator or as successor of the `hasPPFs` brick, because the former are needed for the HaMAP preprocessor whereas the latter are used during the execution of the mobile agent. Therefore, the `DistinguishPPFName` strategy is applied to the mobile agent DAG before generating Haskell code. In the `DistinguishPPFName` strategy all bricks of `PPFName` sub-DAGs of $\texttt{I\_ALL}_{\texttt{PPF}}$ combinators and $\texttt{I\_OR}_{\texttt{PPF}}$ combinators are replaced with another brick of the same semantics. In order to replace only these bricks, and not also the `PPFName` bricks which are successor of the `hasPPFs` brick, two term-graph patterns are used. The term-graph patterns are shown in Figure 9.27.

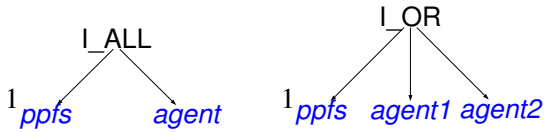


Figure 9.27: Two term-graph patterns for the `DistinguishPPFName` strategy

The term graph patterns match on the $\texttt{I\_ALL}_{\texttt{PPF}}$ respectively the $\texttt{I\_OR}_{\texttt{PPF}}$ combinator and apply the `DistinguishPPFName` strategy to the first successor, which is the `PPFName` part of the particular meta agent combinator.

In Figure 9.28 the `DistinguishPPFName` strategy is shown. Each rule in the set of rules labelled with `DistinguishPPFName` replaces exactly one brick with the corresponding brick.

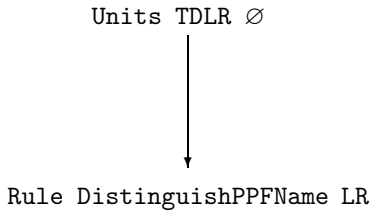


Figure 9.28: `DistinguishPPFName` strategy

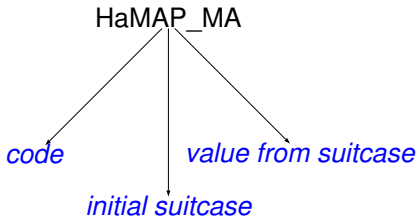
After applying the `DistinguishPPFName` strategy the Haskell code can be generated. The generated code has to be a value of type `MobileAgent_HOPS`.

```
data MobileAgent_HOPS = MobileAgent_HOPS
                        { code_HOPS     :: [CodeFragment]
                        , suitcase_HOPS :: [CodeFragment]
                        , value_HOPS    :: [CodeFragment]
                        }
```

This value has to be transformed into a value of type `MobileAgent` on the home platform. The `value_HOPS` function will be stored on the home platform and the meta data, which is generated by the home platform, will be added. In the remainder of this section output strings for various bricks are presented.

The code generation starts with the HaMAP mobile agent brick. In Figure 9.29 the HaMAP mobile agent brick and the generated code are shown. The term `$1$` denotes the code generated for the first successor, `$2$` and `$3$` denote the code generated for the second respectively the third successor.

The code chunk `Code "x$DLa$$La$"` is generated for a bound variable. As mentioned in Section 2.6, `$DLa$` generates an integer which is unique with respect to the current



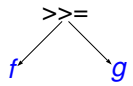

```
MobileAgent_HOPS {code_HOPS     = [$1$]
                 ,suitcase_HOPS = [$2$]
                 ,value_HOPS    = [$3$]
                 }
```

Figure 9.29: HaMAP mobile agent brick and generated code

code output process and which is associated with an instance of the brick, i.e. if an integer has been already generated for a node, `$DLa$` does not generate a new one. With the term `$La$` this integer can be used for the code output. By this means, all occurrences of a bound variable in the generated code are named the same whereas all occurrences of an other bound variable are named different.

The code chunk `Code "$DLa$$La$"` is used for a meta variable. As mentioned in Section 9.4, meta variables are only used for associating replaceable code and the corresponding replacement. In HaMAP this association is denoted by a unique integer.

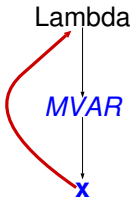
The code which is generated for the `bind` brick is shown in Figure 9.30. The code generated for bricks like `return`, `@`, or `++` is quite similar to the code shown in Figure 9.30.




```
Code "(", $1$, Code ") >>= (", $2$, Code ")"
```

Figure 9.30: `bind` brick and generated code

In Figure 9.31 the code generated for the λ brick is shown. The \ is escaped since it is element of a string. The term `$B1$` denotes the code generated for the bound variable.




```
Code "\\", $B1$, Code "->", $1$
```

Figure 9.31: λ brick and generated code

The generated code for the `read` brick, which corresponds to the Haskell function `read`[1], contains the generated code for the type of the result, which is denoted with the term `$T$`.

[1] `read :: Read a => String -> a`

```
Code "read (",$1$,Code ") :: (",$T$,Code ")"
```

In Table 9.1 the generated code for the `PPFName` bricks, depending on whether they are successor of the $\texttt{I\_ALL}_{\texttt{PPF}}$ or the $\texttt{I\_OR}_{\texttt{PPF}}$ meta agent combinator, or of the `hasPPFs` brick, is shown.

Name	Code for brick which is successor of $\texttt{I\_ALL}_{\texttt{PPF}}$ or $\texttt{I\_OR}_{\texttt{PPF}}$	Code for brick which is successor of `hasPPFs`
`emptyList`	`EmptyPPF`	`Code "[]"`
`Cons`	`$1$,$2$`	`Code "(",$1$,Code ") : (", $2$,Code ")"`
`PPFName`	`PPFName "$1$"`	`Code "PPFName \"$1$\""`
`ALLPPF`	`ALLPPF [$1$]`	`Code "ALLPPF (",$1$,Code ")"`
`ORPPF`	`ORPPF [$1$] [$2$]`	`Code "ORPPF (",$1$,Code ")(", $2$,Code ")"`
`ReplaceablePPF`	`ReplaceablePPF $1$ [$2$]`	`Code "ReplaceablePPF $1$ (", $2$,Code ")"`

Table 9.1: Code for the `PPFName` bricks in dependence on their context

The following code is generated for the meta agent combinators $\texttt{I\_ALL}_{\texttt{PPF}}$ and $\texttt{I\_OR}_{\texttt{PPF}}$.

```
ALL ($1$) [$2$,Code ">>= return . Just"]
OR  ($1$) [$2$] [$3$]
```

In Table 9.2 the generated code for the replacement bricks is shown. The replacement code for the `Replacement` brick is left untouched by the HaMAP preprocessor whereas the replacement code for the `ReplacementValue` brick is calculated during the execution of the mobile agent. The `Replacement@` brick combines both approaches: The code for the first successor is left untouched whereas the code for the second successor is calculated during the execution.

Brick	Generated code
`Replacement`	`Replacement [$1$]`
`ReplacementValue`	`Code "[Code (show (",$1$,Code "))]"`
`Replacement@`	`CFList [Replacement [Code "(",$1$,Code ")"], Code " ++ [Code \"(\",Code (show (",$2$, Code ")),Code \")\"]"]`

Table 9.2: Code for the replacement bricks

10 Example Agents

The following UI-DSL representations of mobile agents are presented in order to illustrate the usage of the approach of an mobile agent environment based on HOPS and Haskell. For the sake of clearness the mobile agents are split into several DAGs, each of them representing an agent that is responsible for a particular concern. The completely expanded UI-DSL representation consisting of primitive agents and agent combinators, the I-DSL representation, the HaMAP DAG and the generated Haskell code can be found in Appendix A.

10.1 `GetListOfPossiblyProvidedFunctions` Agent

The first example of a mobile agent is a very simple mobile agent which does not use any possibly-provided function. The mobile agent visits all agent platforms and calculates a list of possibly-provided functions the platforms provide. In Figure 10.1 the `GetListOfPossiblyProvidedFunctions` agent is shown. The `visitAll` agent, which is used as platform agent, is introduced below. The value agent adds the information about the agent platform the agent is currently running on to its suitcase. The information about the current platform is always the first element of the list returned by the platform function `getPPFInfo`.

The implementation of the platform agent is shown as HOPS rule in Figure 10.2. By defining a HOPS rule it is possible to use the `visitAll` brick in the definition of the mobile agent which makes the mobile agent DAG much more concise. Obviously, this rule has to be applied before any other transformation.

The `visitAll` agent can be divided basically into three parts, two stateful agents and one stateless agent. One of the stateful agents is used to calculate the list of visited platforms including the platform the agent is currently running on. The second stateful agent calculates a list of platforms to visit by adding all platforms known by the current platform to the list in the suitcase and filtering out the platforms calculated

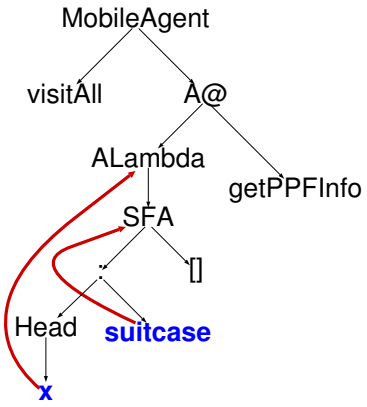


Figure 10.1: `GetListOfPossiblyProvidedFunctions` agent

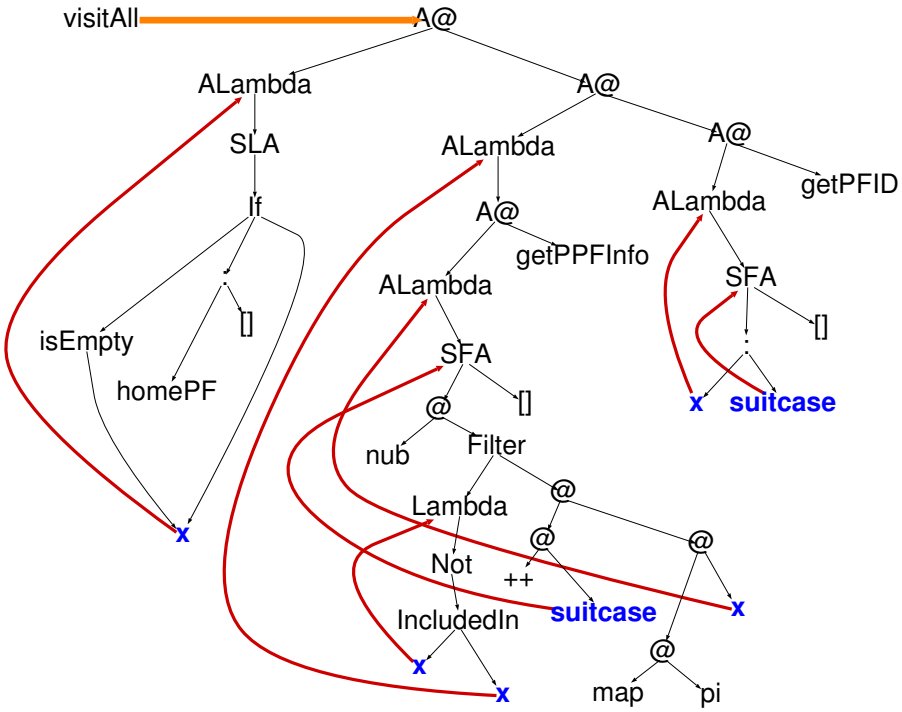


Figure 10.2: Rule for `visitAll`

by the first stateful agent. Although, even if the list contains duplicates no platform is visited more than once, the function `nub` is used to remove duplicates which makes the suitcase "smaller". The stateless agent ensures the "home-coming" of the mobile agent if the list of platforms to visit is empty.

10.2 GetFlight Agent

The GetFlight agent, which is shown in Figure 10.3, visits all platforms providing the possibly-provided function GetFlight and calculates the cheapest flight from Munich to Cologne on 31.12.03.

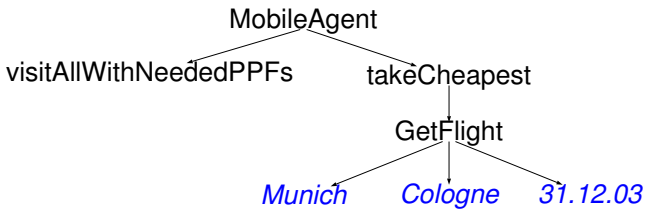


Figure 10.3: GetFlight agent

In Figure 10.4 the implementation of the visitAllWithNeededPPFs is shown. This agent is based on the visitAll2 agent which is almost identical to the visitAll agent which is shown in Figure 10.2. The only difference is that visitAll2 does not return the home platform when the list of platforms is empty. The list of needed possibly-provided functions is denoted by the brick neededPPFs and will be generated automatically while transforming the mobile agent into an internal mobile agent.

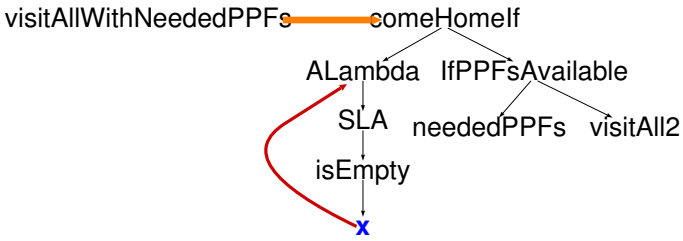


Figure 10.4: Rule for visitAllWithNeededPPFs

The IfPPFsAvailable agent is shown in Figure 10.5. This agent uses the platform function HasPPFs to filter all platforms providing the possibly-provided functions ppfs from the list of pfs.

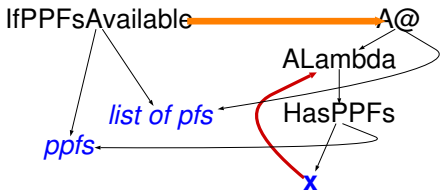


Figure 10.5: Rule for IfPPFsAvailable

The `comeHomeIf` agent is shown in Figure 10.6. This agent returns the singleton list containing the home platform identifier if the condition is true. Otherwise it returns the list `listOfPlatforms`.

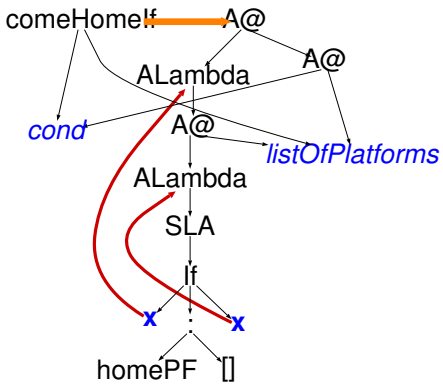


Figure 10.6: Rule for `comeHomeIf`

The `takeCheapest` agent is shown in Figure 10.7. This agent compares the first component of a pair, which is an integer, with the first component of the pair in the suitcase. The pair which has the smaller first component is stored in the suitcase. Since both pairs are encapsulated in a `Maybe` type the `takeCheapest` also deals with possible values of `Nothing`.

10.3 Travel-Searching Agent 1

In Figure 10.8 the Travel-Searching Agent 1 is shown. This agent calculates the cheapest flight from Munich to New York on 24.12.03, the cheapest flight from New York to Munich on 06.01.04, and the cheapest hotel in New York from 24.12.03 to 06.01.04. The agent uses the `visitAllWithNeededPPFs` agent (see Figure 10.4) as platform agent. `GetHotel` and `GetFlight` are possibly-provided functions returning a value of type `Maybe (Int,String)`, where the integer component is the price and the string component is an description. The `takeCheapest` agent is shown in Figure 10.7.

The Travel-Searching Agent 1 calculates the cheapest hotel and the cheapest outward and return flights independently, i.e., the hotel and the flights may be found on different platforms.

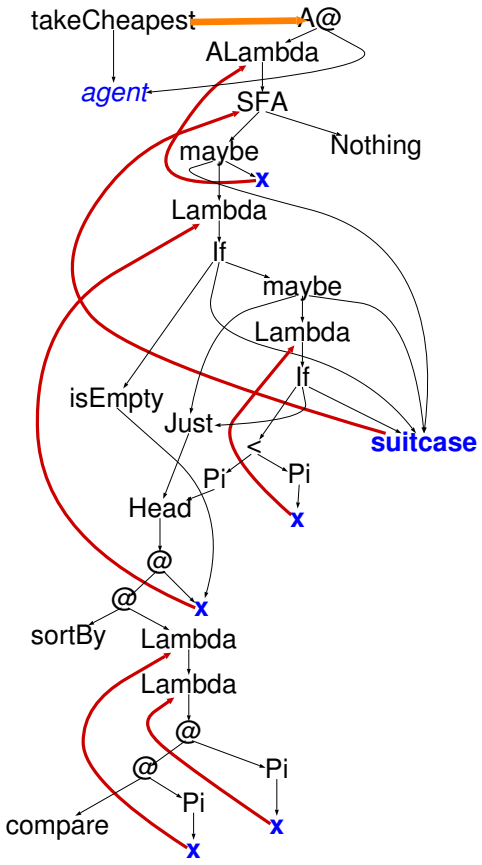


Figure 10.7: Rule for `takeCheapest`

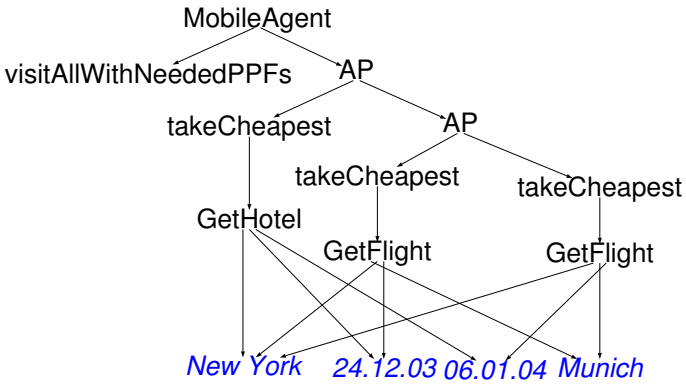


Figure 10.8: Travel-Searching Agent 1

10.4 Travel-Searching Agent 2

In Figure 10.9 the Travel-Searching Agent 2 is shown. This agent calculates the cheapest bundle consisting of a flight from Munich to New York on 24.12.03, a flight from New York to Munich on 06.01.04, and a hotel in New York from 24.12.03 to 06.01.04.

Bundle means both flights and the hotel has to be provided by the same platform. The agent uses the `visitAllWithNeededPPFs` agent (see Figure 10.4) as platform agent.

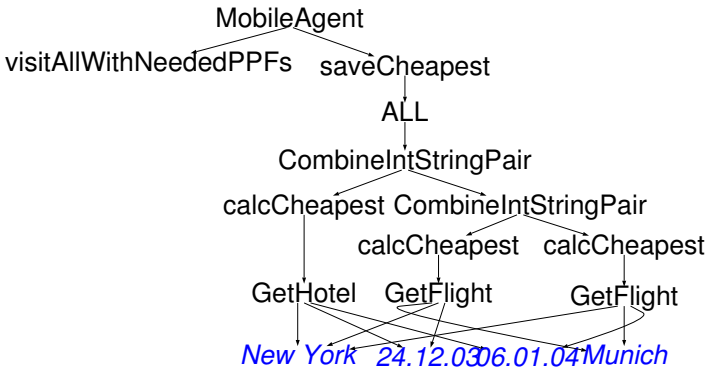


Figure 10.9: Travel-Searching Agent 2

The utilisation of the `visitAllWithNeededPPFs` agent together with the $\texttt{ALL}_{\texttt{PPF}}$ combinator ensure that the mobile agent migrates only to platforms providing both possibly-provided functions, namely `GetFlight` and `GetHotel`.

The `takeCheapest` agent (see Figure 10.7) is split into the `calcCheapest` and the `saveCheapest` agent. The pairs, consisting of an integer and a string, returned by the possibly-provided functions `GetFlight` and `GetHotel` are combined using the `CombineIntStringPair` agent.

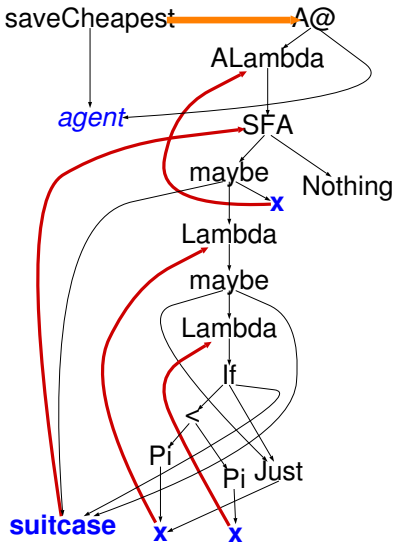


Figure 10.10: Rule for `saveCheapest`

The `calcCheapest` agent calculates the cheapest pair from a list of pairs returned by `ppf`. The `saveCheapest` agent compares a pair with the pair in its suitcase, and

stores the cheaper one in its suitcase. Cheaper in this context means that the first component of the pair, which is an integer, is smaller. The `saveCheapest` agent is shown in Figure 10.10. The `calcCheapest` agent is shown in Figure 10.11.

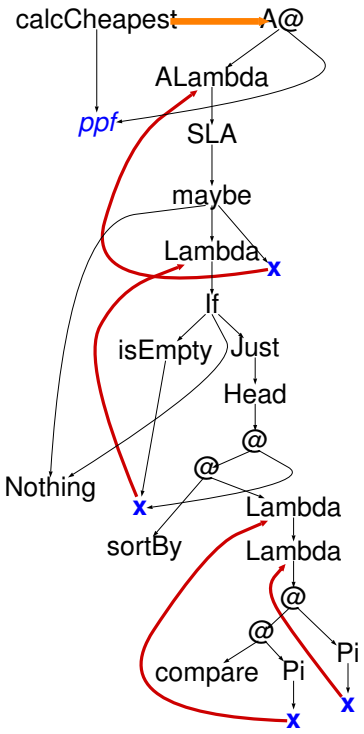


Figure 10.11: Rule for `calcCheapest`

The `CombineIntStringPair` agent is shown in Figure 10.12. If one of the values of `agent1` and `agent2` is `Nothing`, the `CombineIntStringPair` agent returns `Nothing`. Otherwise the first components of the pairs encapsulated in the `Maybe` type are added and the second components are concatenated.

10.5 Travel-Searching Agent 3

The difference between the Travel-Searching Agent 2, which has been introduced in the previous section, and the Travel-Searching Agent 3 is that the latter uses the $\texttt{OR}_{\texttt{PPF}}$ combinator and a third possibly-provided function, namely `GetAllExpenseTour`. This agent searches for an all expense tour. Only if the `GetAllExpenseTour` function is not provided on the current platform, the mobile agent uses `GetHotel` and `GetFlight` to calculate a bundle consisting of two flights and the hotel. The `visitAllWithNeededPPFs` agent in combination with the $\texttt{OR}_{\texttt{PPF}}$ and the $\texttt{ALL}_{\texttt{PPF}}$ combinator calculates a list

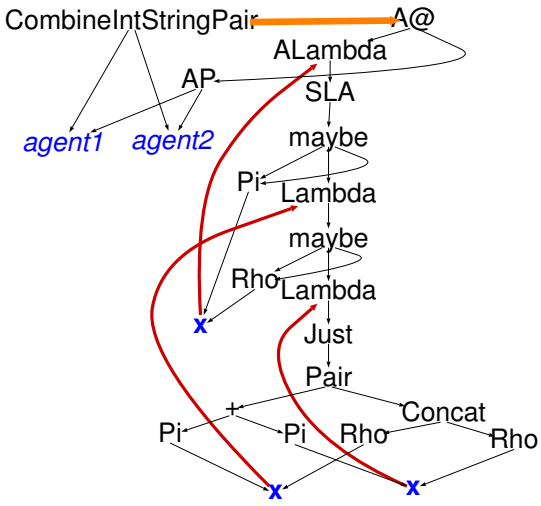


Figure 10.12: Rule for `CombineIntStringPair`

of platforms providing either `GetAllExpenseTour` or `GetHotel` and `GetFlight`.

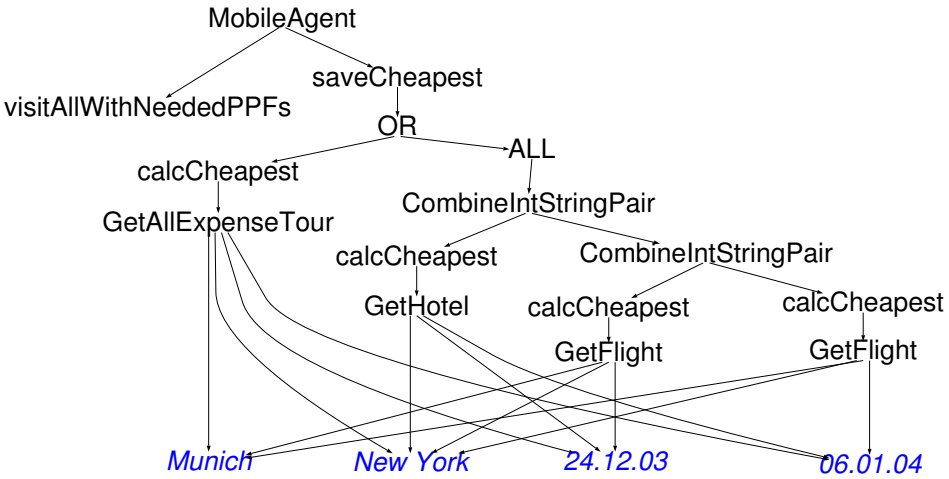


Figure 10.13: Travel-Searching Agent 3

10.6 Travel-Searching and Booking Agent

In Figure 10.14 the Travel-Searching and Booking Agent is shown. This agent searches the cheapest journey to New York from 24.12.03 to 06.01.04 starting in Munich using the possibly-provided function `GetAllExpenseTour`. After visiting all platforms providing this possibly-provided function the mobile agent returns to the platform where the cheapest journey was found and books that journey. The `IfPPFsAvailable` agent, the `visitAll2` agent, the `comeHomeIf` agent, and the `calcCheapest` agent

have been introduced already in the previous sections. The difference between the `saveCheapestWithPFID` agent and the `saveCheapest` agent is that the former saves the pair consisting of the cheapest journey and the platform identifier providing this journey, whereas the latter saves only the cheapest journey.

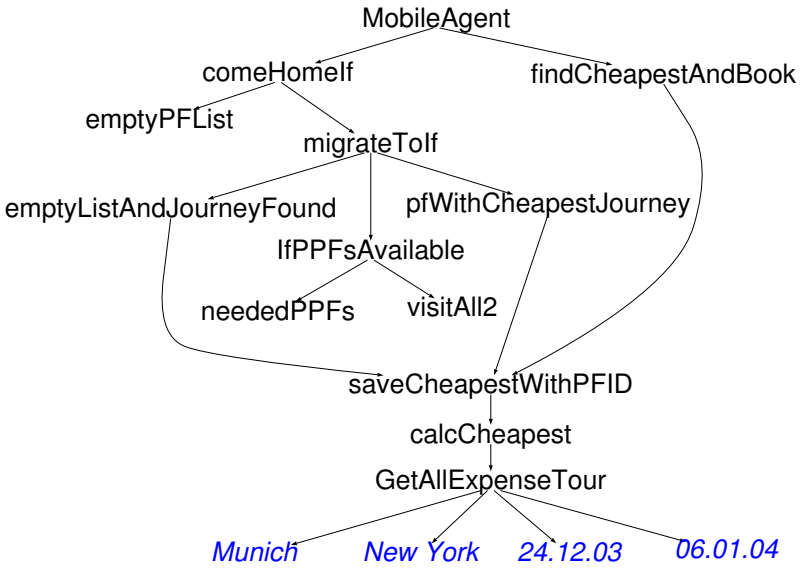


Figure 10.14: Travel-Searching Agent 4

In Figure 10.15 the `emptyPFList` agent, which tests whether a list is empty or not, is shown.

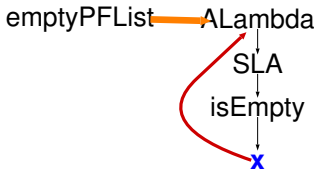


Figure 10.15: Rule for `emptyPFList`

In Figure 10.16 the `emptyListAndJourneyFound` agent is shown. This agent checks whether the list of platforms, which is the bindable variable bound to the root of the right-hand rule side, is empty and whether the current platform is the platform providing the cheapest journey.

The `pfWithCheapestJourney` agent is shown in Figure 10.17. This agent returns a singleton list containing the platform identifier of the platform providing the cheapest journey or an empty list if no journey was found.

In Figure 10.18 the `migrateToIf` agent is shown. This agent returns `new` if the condition `cond` applied to `listOfPlatforms` gives true. Otherwise `listOfPlatforms` is

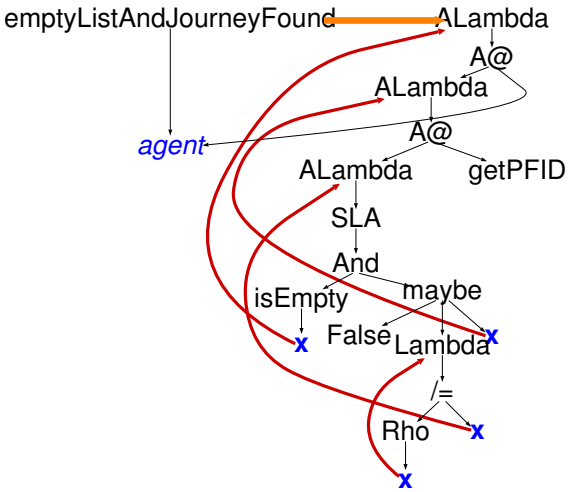


Figure 10.16: Rule for `emptyListAndJourneyFound`

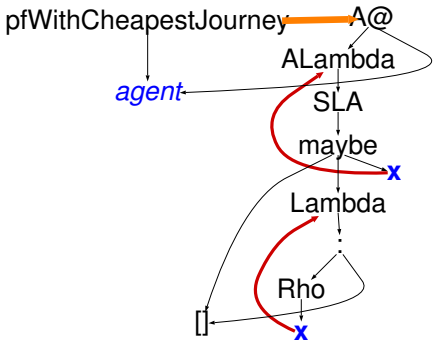


Figure 10.17: Rule for `pfWithCheapestJourney`

returned. By this means, the occurrence of `migrateToIf` in the mobile agent shown in Figure 10.14 returns the list of platforms providing the `GetAllExpenseTour` function unless this list is empty and a journey has been found already. In that case, the singleton list containing the platform providing the cheapest journey is returned.

In Figure 10.19 the `findCheapestAndBook` agent is shown. This agent calculates a pair consisting of a boolean which indicates whether a journey has been booked and the cheapest journey. For this purpose two pairs are constructed using $(,)_{\texttt{Agent}}$ combinators. One of those pairs consists of the constant value `Just False` and the value returned from the `bookJourney` function, which is also of type `Maybe Bool`. The other pair consists of the journey calculated by `agent` and the value calculated by `agent` on the previous platform. The latter value is inserted by the combination of the `ValueMaker` and the `OldValueMarker` combinator. If the platform identifier of the current platform is identical to the platform identifier of the platform on which

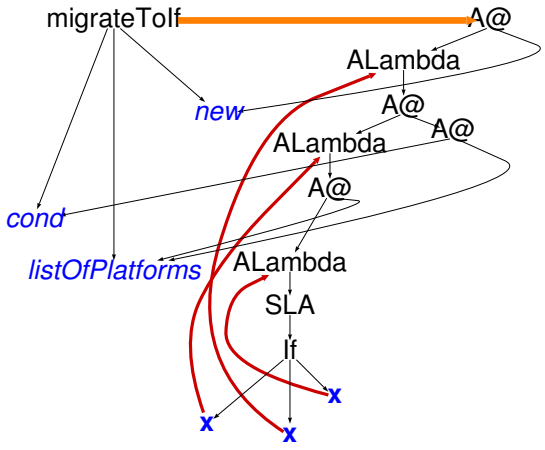


Figure 10.18: Rule for `migrateToIf`

the cheapest journey has been found, the second components of the two pairs are used to build the return value of the `findCheapestAndBook` agent. Otherwise the first components are used. By this means, this agent searches for the cheapest journey on different platforms and books the cheapest journey on the appropriate platform. Due to lazy evaluation `bookJourney` is only executed if this component is used.

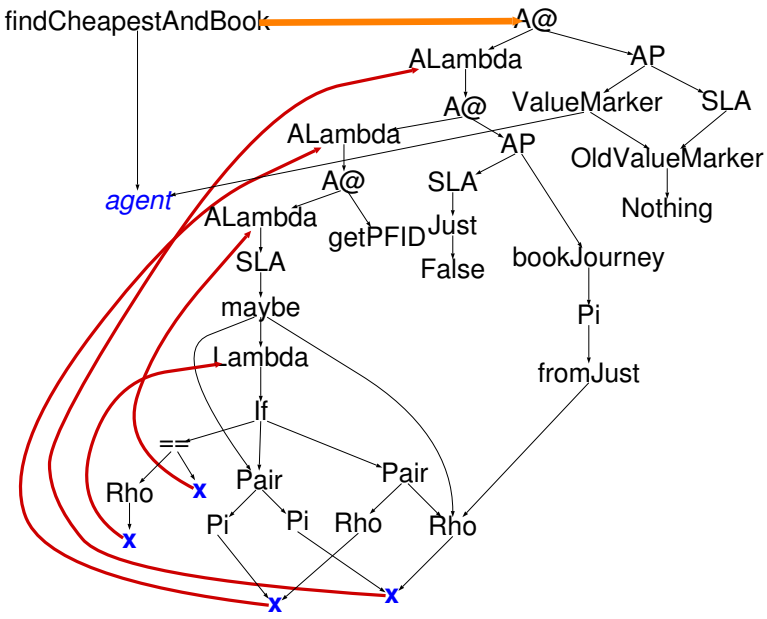


Figure 10.19: Rule for `findCheapestAndBook`

10.7 `GetWeatherAndTrafficJam` Agent

In Figure 10.20 the `GetWeatherAndTrafficJam` Agent is shown. This agent visits platforms providing `getWeatherIn` or `getTrafficJamMessagesBetween`. If one of those functions returns a value `Just x`, this function will be replaced by this value. If both values are calculated, indicated by a value of `Just x` for appropriate `x`, the mobile agent returns to its home platform.

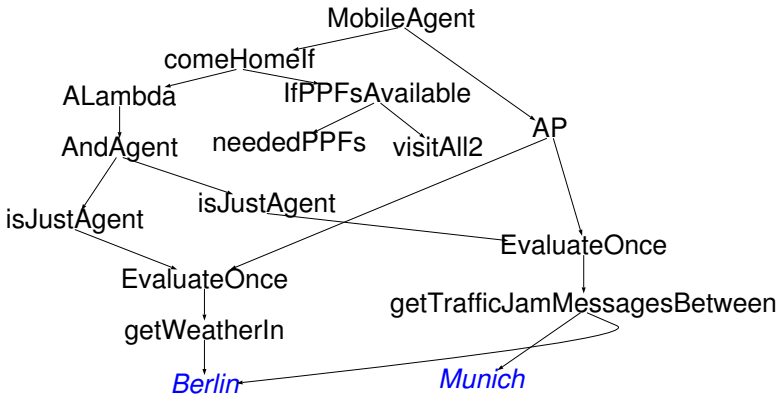


Figure 10.20: `GetWeatherAndTrafficJam` Agent

In Figure 10.21 the `EvaluateOnce` agent is shown. This agent is based on the `Replace` combinator which can be used to replace a code fragment with another code fragment. The condition agent, the first successor of `Replace`, verifies whether the value calculated by `agent` is `Just x`. In that case `agent` will be replaced by the replacement agent, the third successor of `Replace`.

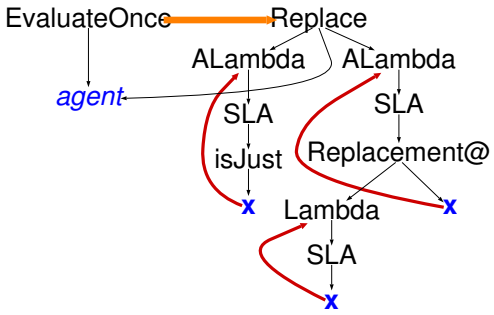


Figure 10.21: Rule for `EvaluateOnce`

In Figure 10.22 the `AndAgent` and the `isJustAgent` are shown. The `AndAgent` combines the values of `agent1` and `agent2` with the logical *and*-operator. The `isJustAgent` returns false if and only if `agent` returns the value `Nothing`.

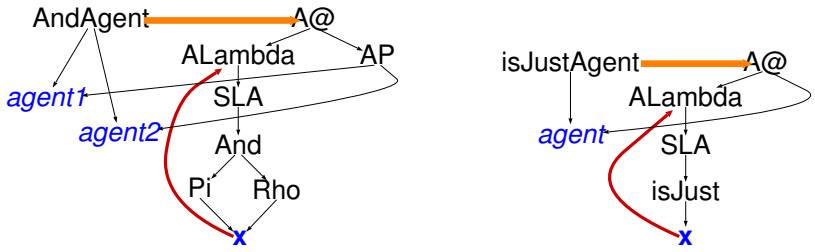


Figure 10.22: Rule for `AndAgent` and `isJustAgent`

11 Conclusions and Future Work

We have examined the design and implementation of a mobile agent programming and execution environment based on term graph transformation and functional programming. The mobile agent execution environment uses the well-understood type system of Haskell to guarantee that a mobile agent cannot do arbitrary I/O on an agent platform and therefore the execution of a mobile agent cannot be a security risk for the machine on which the agent platform is running.

An agent platform provides only a small, manageable set of possibly-provided functions, a set of five platform functions, and the functions defined in the Haskell 98 report (Peyton Jones et al., 2002) except for I/O functions. The set of possibly-provided functions can be different on each platform, i.e., a particular possibly-provided function need not be available on all platforms. We use a preprocessor and a code fragment data type in order to replace non-available possibly-provided functions temporarily on a platform. Therefore we demand that all possibly-provided functions return a value of type `Maybe a` for arbitrary `a`. Furthermore, it is possible to replace arbitrary parts of the mobile agent temporarily depending on the availability of possibly-provided functions using so-called meta agent combinators.

The mobile agent is able to transform its own code permanently, which enables, e.g., the simulation of partial evaluation. The mobile agent can replace a code fragment with its value once the value has been calculated on a platform. The concept of transforming the mobile agent code depending on the availability of possibly-provided functions and the capability of the mobile agent to transform its own code cannot be found in any other existing mobile agent environment.

We have developed a powerful, easy-to-use and extensible domain-specific language for the definition of a mobile agent in HOPS. This term graph language is called the User Interface Domain-Specific Language (UI-DSL). UI-DSL is a combinator language consisting of primitive agents and agent combinators. By using a combinator language we are able to separate the concerns and reuse parts of a mobile agent in an elegant way. In order to use the mobile agent it has to be transformed into a DAG representing the

appropriate Haskell code. In Figure 11.1 the transformation process from the UI-DSL mobile agent to Haskell code is shown.

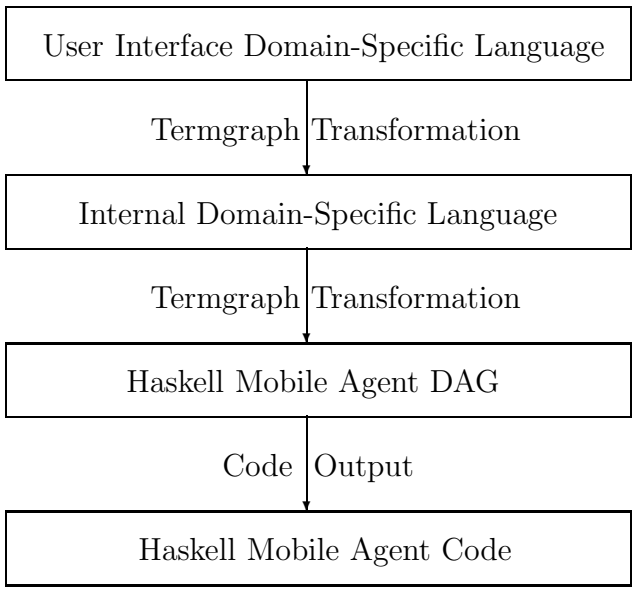


Figure 11.1: Steps from UI-DSL to Haskell Code

After defining a mobile agent in UI-DSL it is transformed into an internal mobile agent. I-DSL is the Internal Domain-Specific Language. It is used to represent a mobile agent with an automatically generated suitcase. The suitcase is the data part of the mobile agent. In UI-DSL only local suitcases of primitive agents have to be defined. Those local suitcases are the data part for one concern or even a part of a concern. Furthermore, it is ensured by the transformation from UI-DSL to I-DSL that the value of the value agent is part of the global suitcase. A function for extracting this value is also generated automatically.

We have developed the Haskell Mobile Agent Platform (HaMAP) from scratch. HaMAP is a mobile agent execution environment that satisfies the requirements of our mobile agents, e.g., the possibility to transform the mobile agent code. HaMAP is a first working prototype of a fully-fledged mobile agent environment. Scalability, efficiency and interoperability with existing environments have not been taken into consideration yet, since this goes beyond the scope of this thesis.

Before Haskell code can be generated, the mobile agent DAG has to be transformed into a monadic form. With the `FlattenBind` strategy, for instance, we have shown how to optimise the code size by applying mathematical laws. The code output encapsulates the mobile agent in a list of code fragments. Examples of mobile agents, as UI-DSL

mobile agent, as internal mobile agent in I-DSL, as Haskell DAG, and as Haskell code, were presented in order to illustrate our approach.

Our transformation rules and strategies are used to convert a mobile agent represented in terms of a domain-specific language into a standard programming language. With this work we have shown how term graph transformation can aid the process of software development. Summing up, our system based on HOPS is a very powerful approach of a mobile agent programming environment, especially for those who are already familiar with functional programming and term graph transformation.

Although our system was a first approach with a prototypical implementation intended to be a proof of concept, the implementation is ready-to-use. We have tested the implementation with approximately 20 agent platforms and 100 concurrently running mobile agents which was satisfactory with respect to efficiency. The main scalability issue is the replication of the information about possibly-provided functions within the Haskell Mobile Agent Platform Protocol. We have only implemented a very simple protocol because this thesis focusses on the development of mobile agents and not networking protocols. In a future version of HaMAP also dissemination of information may be done with mobile agents.

The mobile agents developed with HOPS use weak mobility, since it appears in the functional programming context to be the more natural way to express mobility. Bettini and Nicola (2001) have introduced a purely syntactic translation from strong mobility to weak mobility. This translation can be integrated into future versions of our HOPS strategies.

Furthermore, it is possible to adapt the UI-DSL to the need of the particular mobile agent programmer. Our first approach uses one value agent and one platform agent, since we wanted to develop mobile agents visiting a list of platforms and calculating a value using the same function on each platform. Another scenario can be, for instance, a mobile agent which uses different functions on different types of platforms. The user can simply add new agent combinators and the appropriate rules to transform them into the necessary layout.

Adding interoperability with existing mobile agent environments is counterproductive at the moment, because no other environment supports our concept of possibly-provided functions and transformable agent code.

Further enhancements of HaMAP can be, e.g., support for agent communication or cloning of mobile agents. The version of HOPS we used for our approach is an academic

prototype. Although, it fulfils our requirements for providing an academic prototype of our system, it has not yet been optimised with respect to efficiency. We have slightly adapted HOPS to our needs. We have added, for example, three new hard-coded bricks in order to be able to formulate rules like the one shown in Figure 9.23 on page 112.

A Transformed Example Agents

For the sake of clarity, the example agents in Chapter 10 have been split into several DAGs, each of them representing an agent that is responsible for a particular concern. Therefore, the completely expanded UI-DSL representations consisting of primitive agents and agent combinators, the I-DSL representations, the HaMAP DAGs and the generated Haskell code are presented in the following pages.

The DAGs are generated with the Smalltalk version of the Higher Object Programming System HOPS. The layout has been calculated automatically. All nodes are represented as rectangles with the particular node label. The edges have no arrowhead. Thin edges represent the successor relation. They are directed top-down. The thick edges represent the binding relation. They are directed bottom-up. Other edges cannot be found in the DAGs presented in this Appendix.

The Haskell code is generated by the code output facility of HOPS and is prettified using a pretty-printer written in Haskell.

Contents

A.1 `GetListOfPossiblyProvidedFunctions` Agent

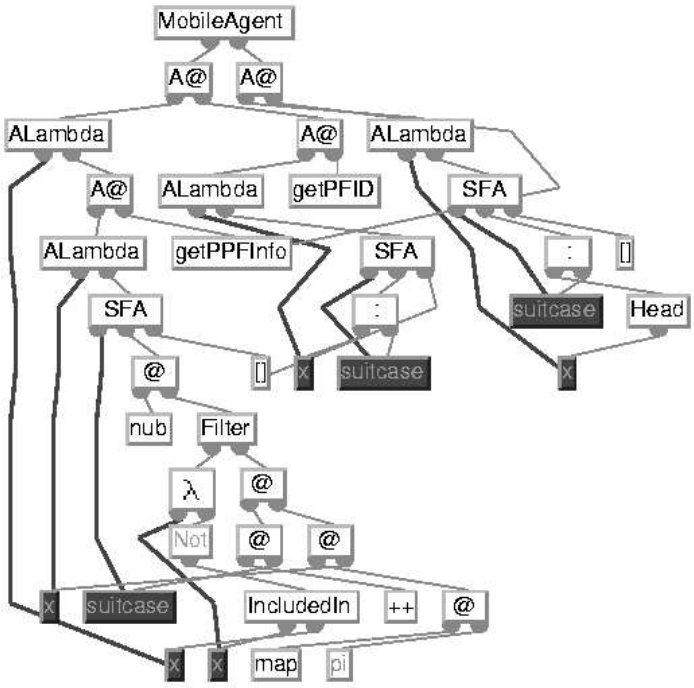


Figure A.1: `GetListOfPossiblyProvidedFunctions` Agent in UI-DSL

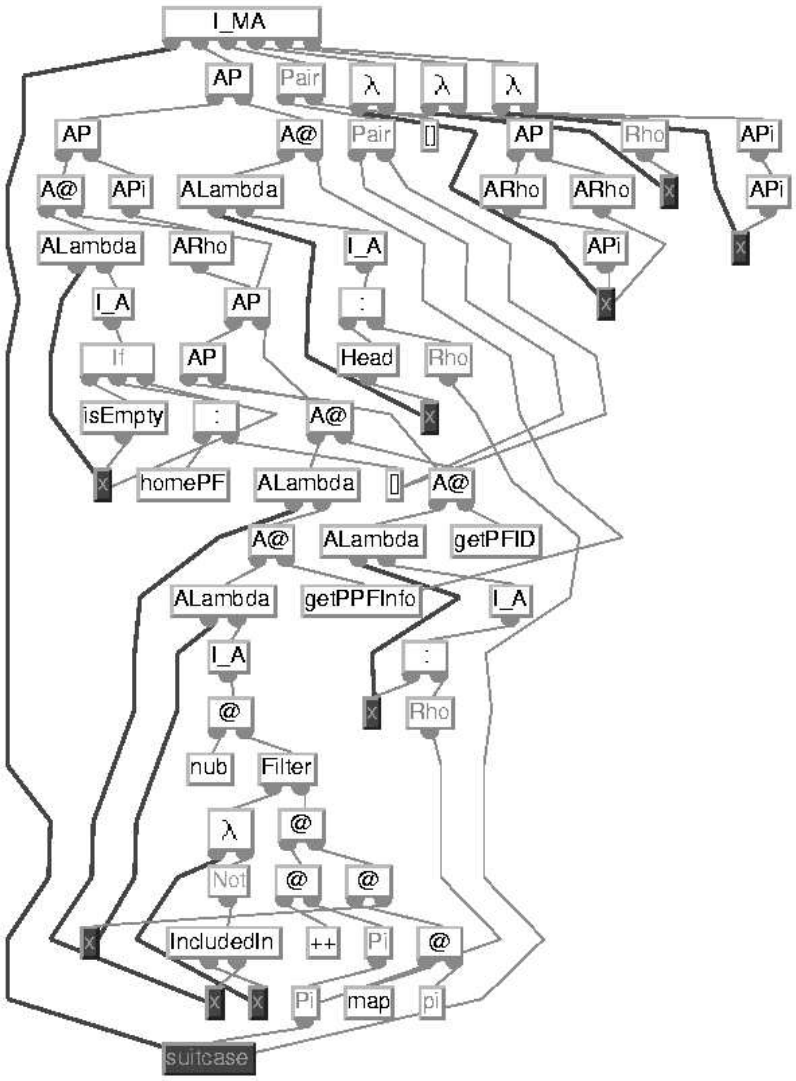


Figure A.2: GetListOfPossiblyProvidedFunctions Agent in I-DSL

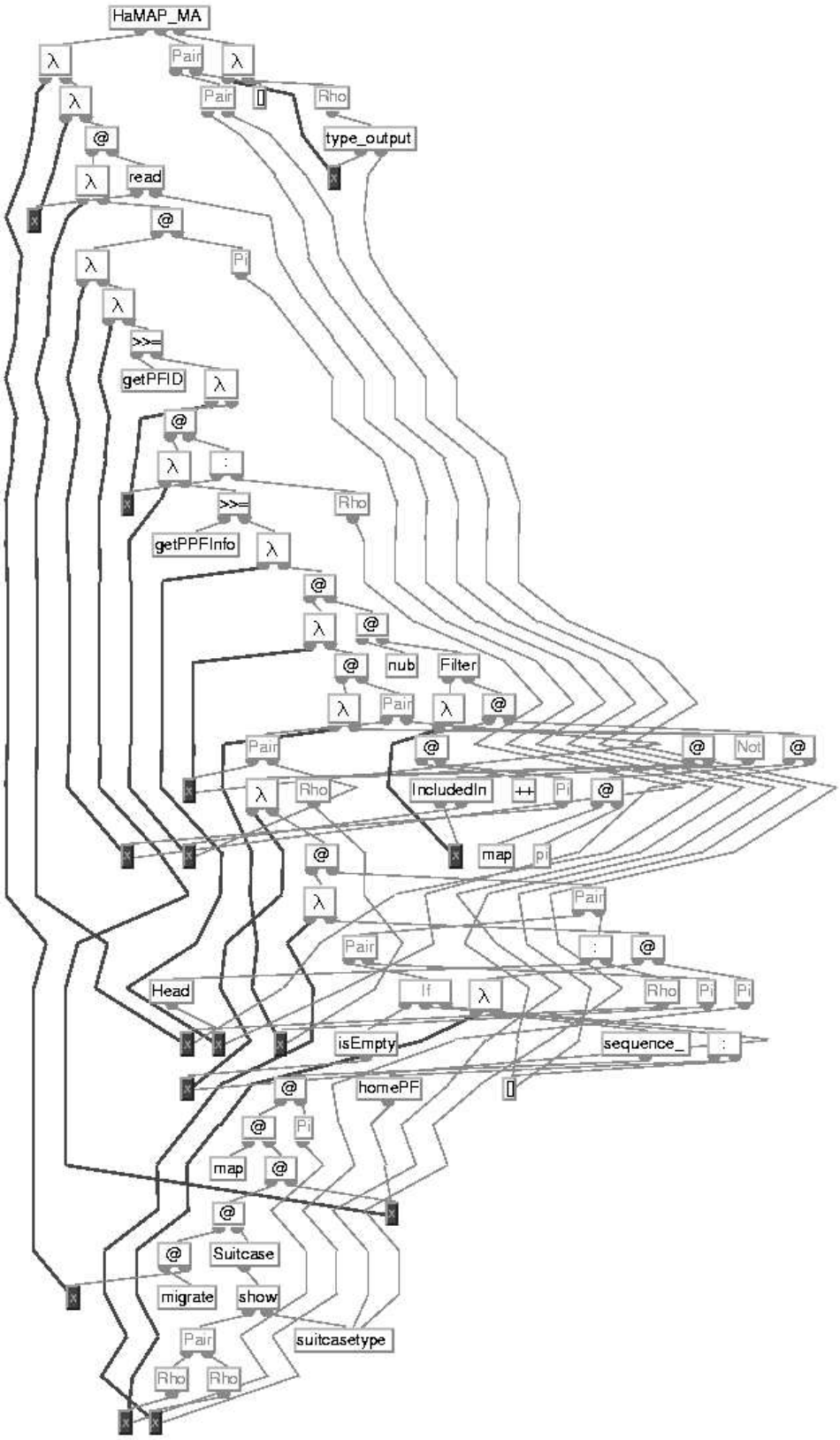


Figure A.3: `GetListOfPossiblyProvidedFunctions` Agent as HaMAP mobile agent DAG

```
MobileAgent_HOPS
 {code_HOPS =
  [Code "\\x0 -> \\x13 -> (\\x7 -> (\\x11 -> \\x3 -> getPFID >>= (\\x12 ->"
  ,Code "(\\ x9 -> getPPFInfo >>= (\\x6 -> (\\x8 -> (\\x5 -> (\\x4 ->"
  ,Code "(\\x2 -> (\\x1 -> sequence_ (( map (( migrate x0 ( Suitcase"
  ,Code "(show ((snd x1,snd x2) :: (([PFID],[PFID]),[(PFID,[PPFInfo])])))))"
  ,Code "x3))(fst x1)))(fst x2))(((if null x4 then ((homePF x3) : []) else"
  ,Code "x4,fst x5),(head x6) : (snd x7))))(snd x5))(((x8,x9),x8)))(nub"
  ,Code "(filter (\\x10 -> not (elem x10 x9)) (((++)(fst x11))((map (\\x ->"
  ,Code "fst x)) x6))))))(x12 : (snd x11))))(fst x7))(read x13 :: ((([PFID]"
  ,Code ",[PFID]), [(PFID,[PPFInfo])])))"]
 ,suitcase_HOPS = [Code "(([],[]),[])"]
 ,value_HOPS =
  [Code "\\x14 -> snd (x14 :: ((([PFID],[PFID]),[(PFID,[PPFInfo])])))"]
 }
```

Figure A.4: Generated Haskell code for the `GetListOfPossiblyProvidedFunctions` Agent

A.2 `GetFlight` Agent

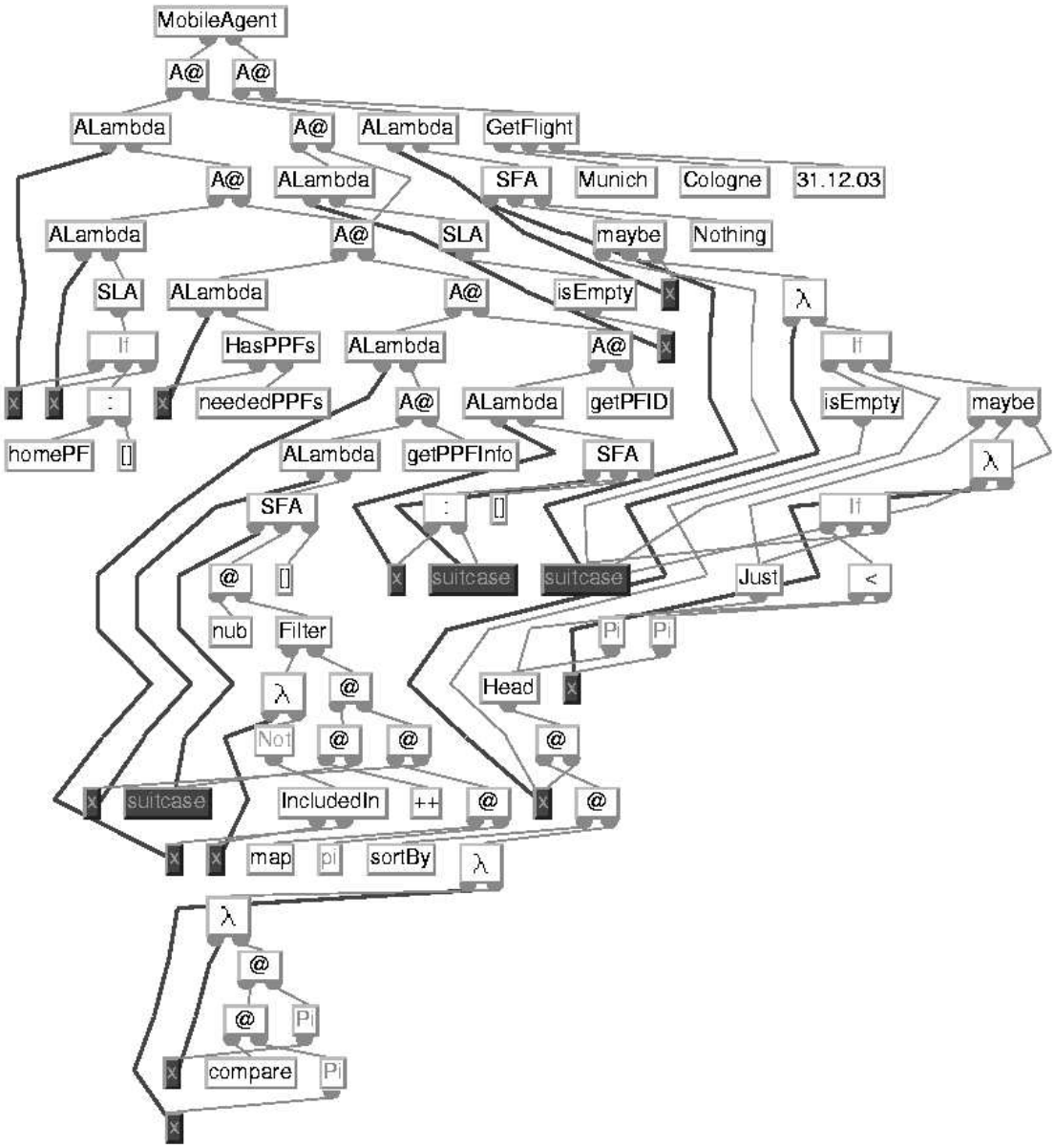


Figure A.5: `GetFlight` Agent in UI-DSL

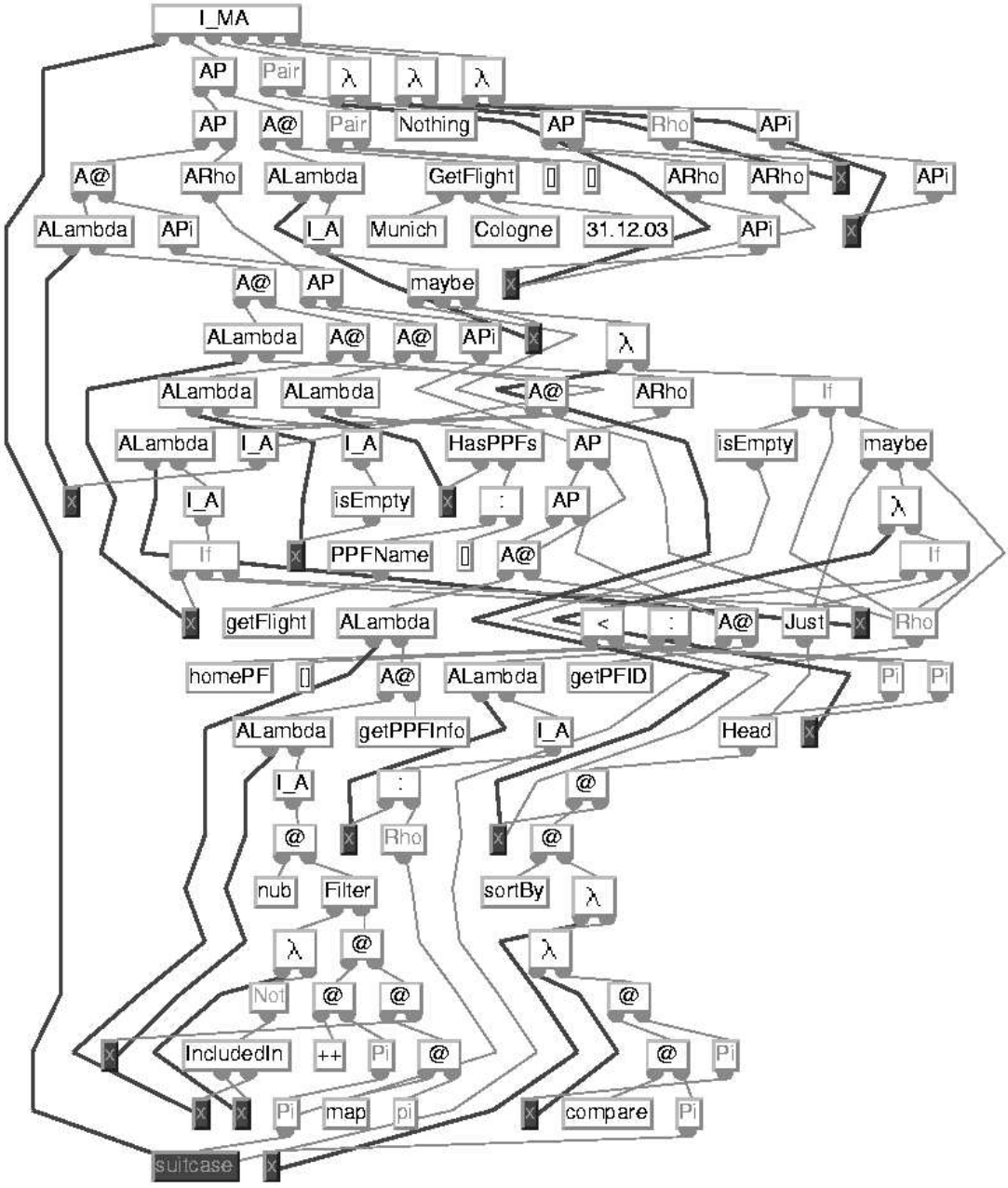


Figure A.6: `GetFlight` Agent in I-DSL

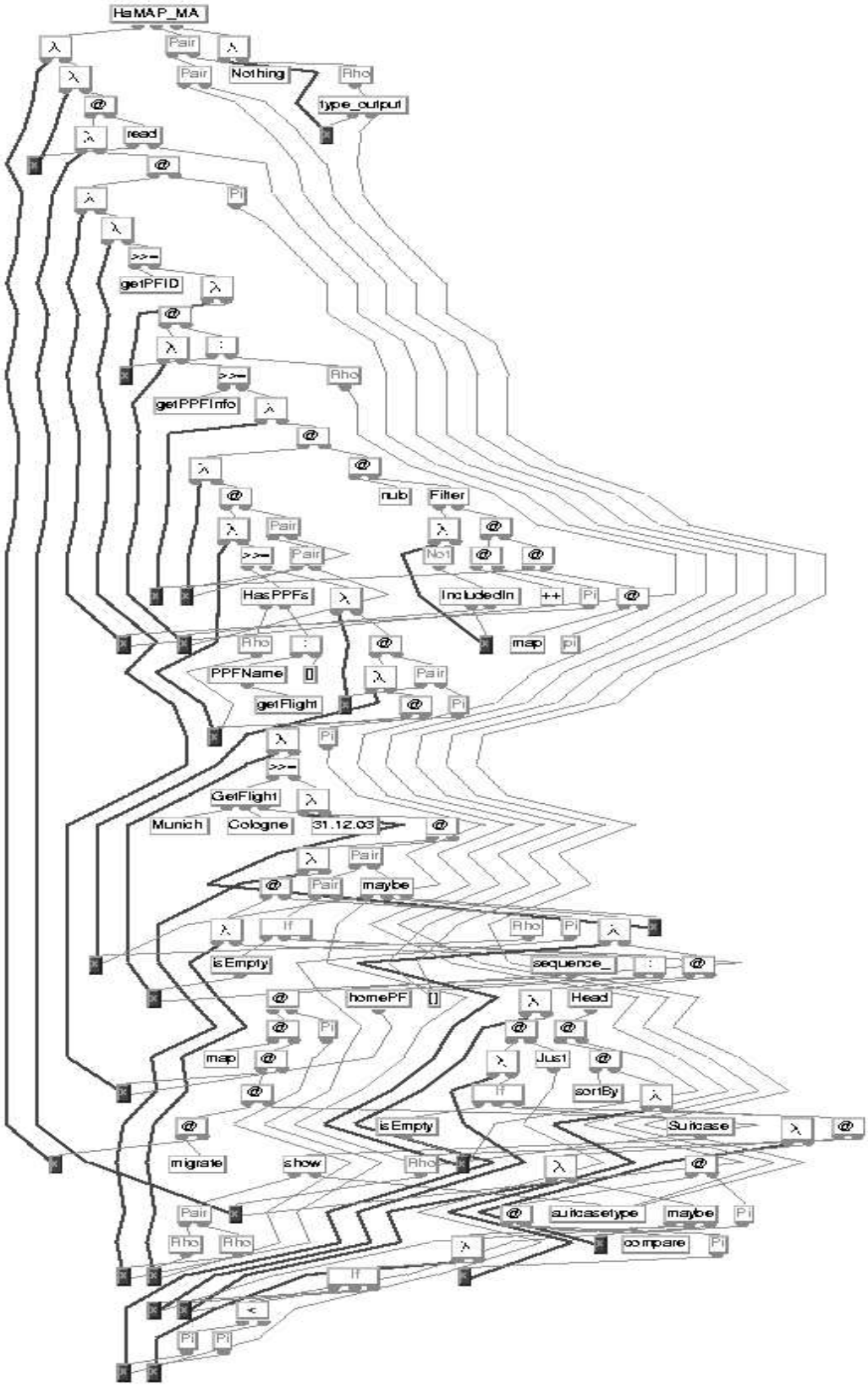


Figure A.7: `GetFlight` Agent as HaMAP mobile agent DAG

```
MobileAgent_HOPS
 {code_HOPS =
  [Code "\\x1 -> \\x23 -> (\\x7 -> (\\x20 -> \\x4 -> getPFID >>= (\\x22 ->"
  ,Code "(\\x18 -> getPPFInfo >>= (\\x21 -> (\\x17 -> (\\x0 -> (hasPPFs (snd"
  ,Code "x0) ((PPFName \"getFlight\"):[])) >>= (\\x16 -> (\\x6 -> (\\x5 -> ("
  ,PPF (PPFName "getFlight") [Code "\"Munich\" \"Cologne\" \"31.12.03\""]
  ,Code ") >>= (\\x15 -> (\\x3 -> (\\x2 -> sequence_ ((map (((migrate x1)("
  ,Code "Suitcase (show (((snd x2,snd x3))::((([PFID],[PFID]),Maybe((Integer"
  ,Code ",String)))))))) x4))(fst x2)))(fst x3))(((if (null x5) then (("
  ,Code "homePF x4):[]) else x5, snd x6),maybe (snd x7)(\\x8 -> (\\x10 -> ("
  ,Code "\\x9 -> if (null x8) then (snd x7) else ((\\x12 -> maybe x9 (\\x11"
  ,Code "-> if ((fst x10) < (fst x11)) then x9 else x12) x12)(snd x7)))(Just"
  ,Code "x10))(head ((sortBy (\\x13 -> \\x14 -> ((compare :: ((Integer -> ("
  ,Code "Integer -> Ordering))))(fst x13))(fst x14))) x8))) x15))))(fst x6))"
  ,Code "((x16,fst x0)))))(((x17,x18),x17)))(nub (filter (\\x19 -> not (elem"
  ,Code "x19 x18))(((++)(fst x20))((map (\\x -> fst x)) x21))))))(x22:(snd"
  ,Code "x20))))(fst x7))(read x23 :: ((([PFID],[PFID]),Maybe ((Integer,"
  ,Code "String)))))"]
 ,suitcase_HOPS = [Code "(([],[]),Nothing)"]
 ,value_HOPS =
  [Code "\\x24 -> snd (x24 :: ((([PFID],[PFID]),Maybe ((Integer,String)))))"]
 }
```

Figure A.8: Generated Haskell code for the GetFlight Agent

A.3 Travel-Searching Agent 1

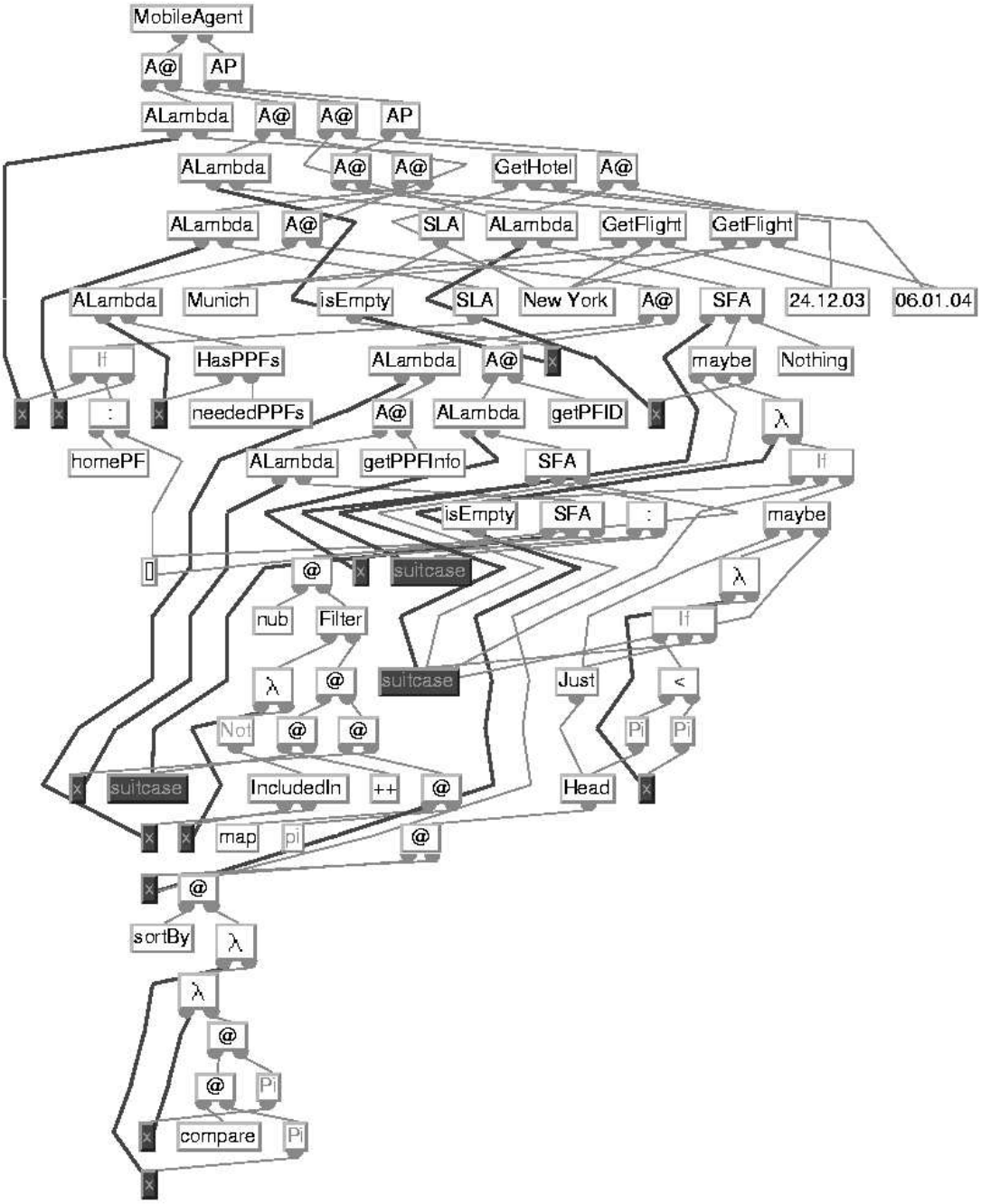


Figure A.9: Travel-Searching Agent 1 in UI-DSL

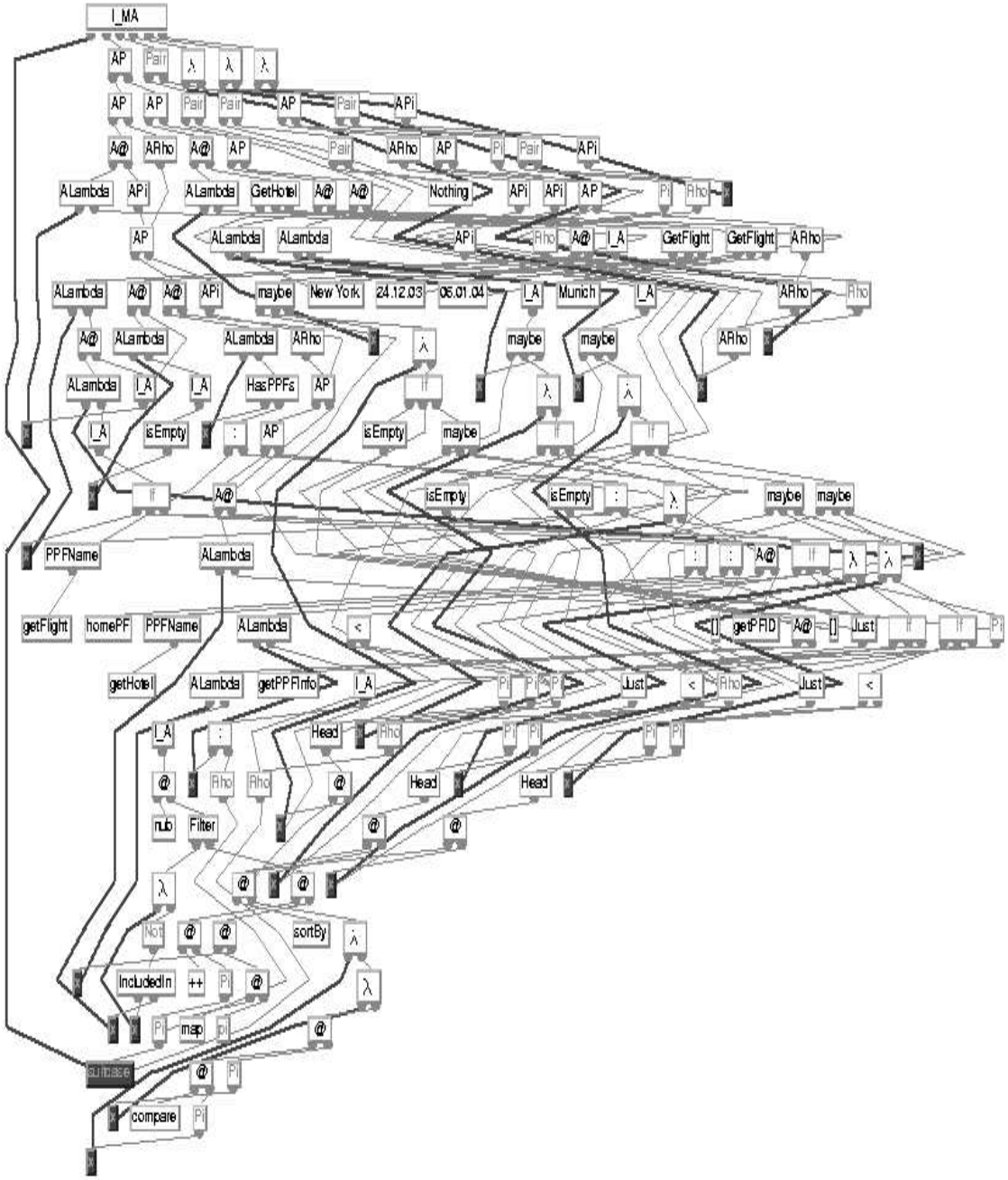


Figure A.10: Travel-Searching Agent 1 in I-DSL

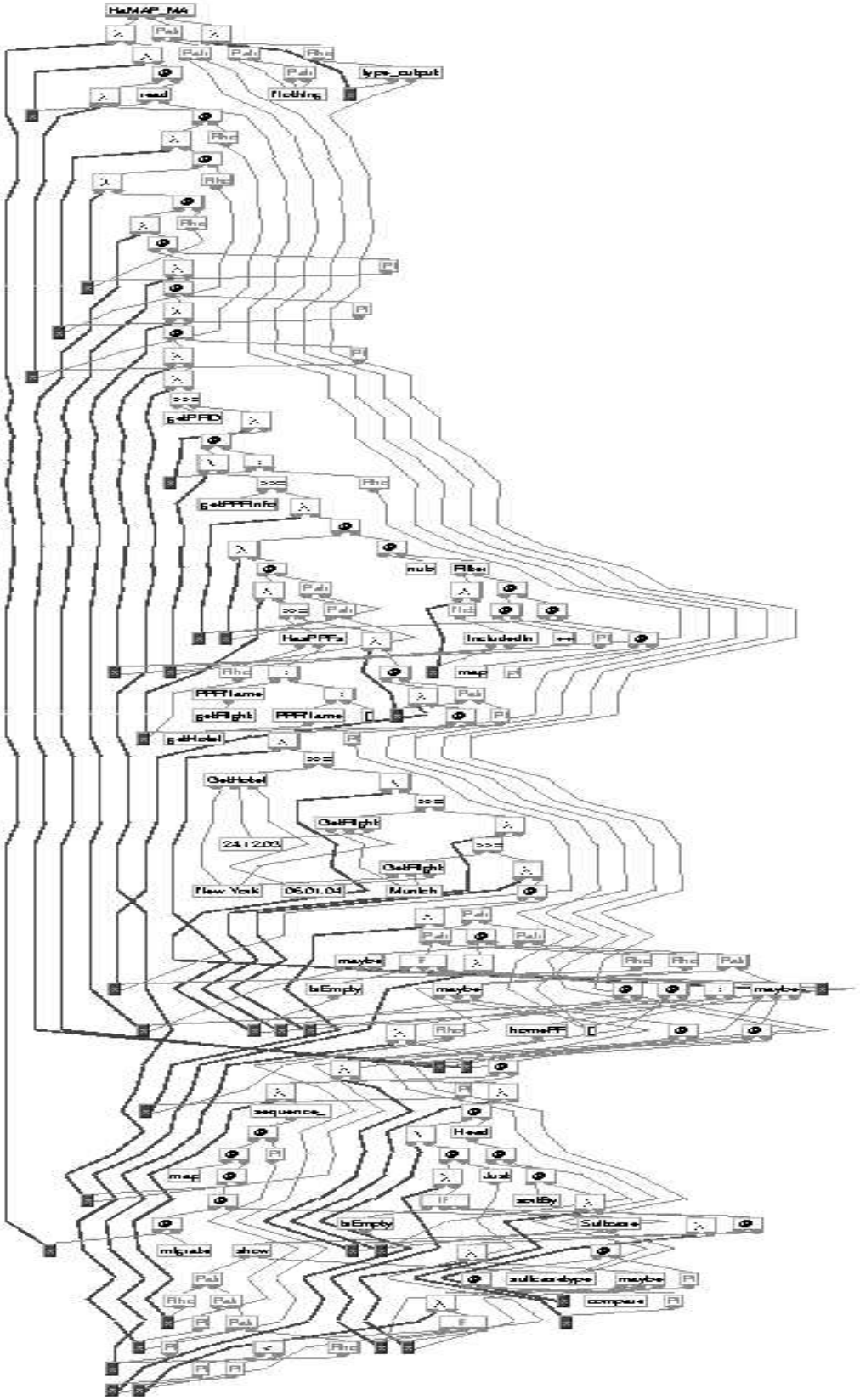


Figure A.11: Travel-Searching Agent 1 as HaMAP mobile agent DAG

```
MobileAgent_HOPS
 {code_HOPS =
  [Code "\\ x1 -> \\ x33 -> ( \\ x30 -> ( \\ x31 -> ( \\ x32 -> ( \\ x21 -> ( \\"
  ,Code "x19 -> ( \\ x9 -> ( \\ x27 -> \\ x5 -> ( getPFID ) >>= ( \\ x29 -> ("
  ,Code "\\ x25 -> ( getPPFInfo ) >>= ( \\ x28 -> ( \\ x24 -> ( \\ x0 -> ("
  ,Code "hasPPFs ( snd ( x0 ) ) ( ( PPFName \"getFlight\" ) : ( ("
  ,Code "PPFName \"getHotel\" ) : ( [] ) ) ) ) >>= ( \\ x23 -> ( \\ x8 -> ( \\"
  ,Code "x7 -> ("
  ,PPF (PPFName "getHotel") [Code "\"New York\" \"24.12.03\" \"06.01.04\""]
  ,Code ") >>= ( \\ x18 -> ("
  ,PPF (PPFName "getFlight") [Code "\"Munich\" \"New York\" \"24.12.03\""]
  ,Code ") >>= ( \\ x20 -> ("
  ,PPF (PPFName "getFlight") [Code "\"New York\" \"Munich\" \"06.01.04\""]
  ,Code ") >>= ( \\ x22 -> ( \\ x6 -> ( \\ x3 -> ( \\ x4 -> ( \\ x2 ->"
  ,Code "sequence_ ( ( ( map )( ( ( ( migrate )( x1 ) )( Suitcase ("
  ,Code "show (( ( snd ( x2 ) , ( fst ( x3 ) , ( fst ( x4 ) , snd ( x4 )"
  ,Code ") ) ) ) :: ( ( ( [ PFID ] , [ PFID ] ) , ( Maybe ( ( Integer ,"
  ,Code "String ) ) , ( Maybe ( ( Integer , String ) ) , Maybe ( ("
  ,Code "Integer , String ) ) ) ) ) )) ) ) )( x5 ) ) )( fst ( x2 ) ) ) )("
  ,Code "fst ( x6 ) ) )( snd ( x3 ) ) )( snd ( x6 ) ) )( ( ( if ( null ("
  ,Code "x7 ) ) then ( ( homePF ( x5 ) ) : ( [] ) ) else ( x7 ) , snd ("
  ,Code "x8 ) ) , ( maybe ( x9 ) ( ( \\ x11 -> \\ x10 -> ( \\ x13 -> ( \\ x12"
  ,Code "-> if ( null ( x10 ) ) then ( x11 ) else ( ( \\ x15 -> maybe ("
  ,Code "x12 ) ( \\ x14 -> if ( ( fst ( x13 ) ) < ( fst ( x14 ) ) ) then ("
  ,Code "x12 ) else ( x15 ) ) ( x15 ) )( x11 ) ) )( Just ( x13 ) ) )("
  ,Code "head ( ( ( sortBy )( \\ x16 -> \\ x17 -> ( ( compare :: ( ("
  ,Code "Integer -> ( Integer -> Ordering ) ) ) )( fst ( x16 ) ) )( fst ("
  ,Code "x17 ) ) ) )( x10 ) ) ) )( x9 ) ) ( x18 ) , ( maybe ( x19 ) ( ( \\"
  ,Code "x11 -> \\ x10 -> ( \\ x13 -> ( \\ x12 -> if ( null ( x10 ) ) then ("
  ,Code "x11 ) else ( ( \\ x15 -> maybe ( x12 ) ( \\ x14 -> if ( ( fst ("
  ,Code "x13 ) ) < ( fst ( x14 ) ) ) then ( x12 ) else ( x15 ) ) ( x15 )"
  ,Code ")( x11 ) ) )( Just ( x13 ) ) )( head ( ( ( sortBy )( \\ x16 -> \\"
  ,Code "x17 -> ( ( compare :: ( ( Integer -> ( Integer -> Ordering ) ) )"
  ,Code ")( fst ( x16 ) ) )( fst ( x17 ) ) ) )( x10 ) ) ) )( x19 ) ) ("
  ,Code "x20 ) , maybe ( x21 ) ( ( \\ x11 -> \\ x10 -> ( \\ x13 -> ( \\ x12"
  ,Code "-> if ( null ( x10 ) ) then ( x11 ) else ( ( \\ x15 -> maybe ("
  ,Code "x12 ) ( \\ x14 -> if ( ( fst ( x13 ) ) < ( fst ( x14 ) ) ) then ("
  ,Code "x12 ) else ( x15 ) ) ( x15 ) )( x11 ) ) )( Just ( x13 ) ) )("
  ,Code "head ( ( ( sortBy )( \\ x16 -> \\ x17 -> ( ( compare :: ( ("
  ,Code "Integer -> ( Integer -> Ordering ) ) ) )( fst ( x16 ) ) )( fst ("
  ,Code "x17 ) ) ) )( x10 ) ) ) )( x21 ) ) ( x22 ) ) ) ) ) ) ) )( fst ("
  ,Code "x8 ) ) )( ( x23 , fst ( x0 ) ) ) ) )( ( ( x24 , x25 ) , x24 ) )"
  ,Code ")( ( nub )( filter ( \\ x26 -> not ( elem ( x26 ) ( x25 ) ) ) ( ("
  ,Code "( (++) )( fst ( x27 ) ) )( ( ( map )( \\x -> fst x ) )( x28 ) ) )"
  ,Code ") ) ) )( ( x29 ) : ( snd ( x27 ) ) ) ) )( fst ( x30 ) ) )( fst ("
  ,Code "x31 ) ) )( fst ( x32 ) ) )( snd ( x32 ) ) )( snd ( x31 ) ) )("
  ,Code "snd ( x30 ) ) )( read ( x33 ) :: ( ( ( [ PFID ] , [ PFID ] ) , ("
  ,Code "Maybe ( ( Integer , String ) ) , ( Maybe ( ( Integer , String )"
  ,Code ") , Maybe ( ( Integer , String ) ) ) ) ) ) )"]
 ,suitcase_HOPS = [Code "( ( [] , [] ) , ( Nothing , ( Nothing , Nothing ) ) )"]
 ,value_HOPS =
  [Code "\\ x34 -> ( fst ( snd ( x34 :: ( ( ( [ PFID ] , [ PFID ] ) , ("
  ,Code "Maybe ( ( Integer , String ) ) , ( Maybe ( ( Integer , String )"
  ,Code ") , Maybe ( ( Integer , String ) ) ) ) ) ) ) , ( fst ( snd ("
  ,Code "snd ( x34 :: ( ( ( [ PFID ] , [ PFID ] ) , ( Maybe ( ( Integer ,"
  ,Code "String ) ) , ( Maybe ( ( Integer , String ) ) , Maybe ( ("
  ,Code "Integer , String ) ) ) ) ) ) ) , snd ( snd ( snd ( x34 :: ("
  ,Code "( ( [ PFID ] , [ PFID ] ) , ( Maybe ( ( Integer , String ) ) , ("
  ,Code "Maybe ( ( Integer , String ) ) , Maybe ( ( Integer , String ) )"
  ,Code ") ) ) ) ) ) ) )"]
 }
```

Figure A.12: Generated Haskell code for the Travel-Searching Agent 1

A.4 Travel-Searching Agent 2

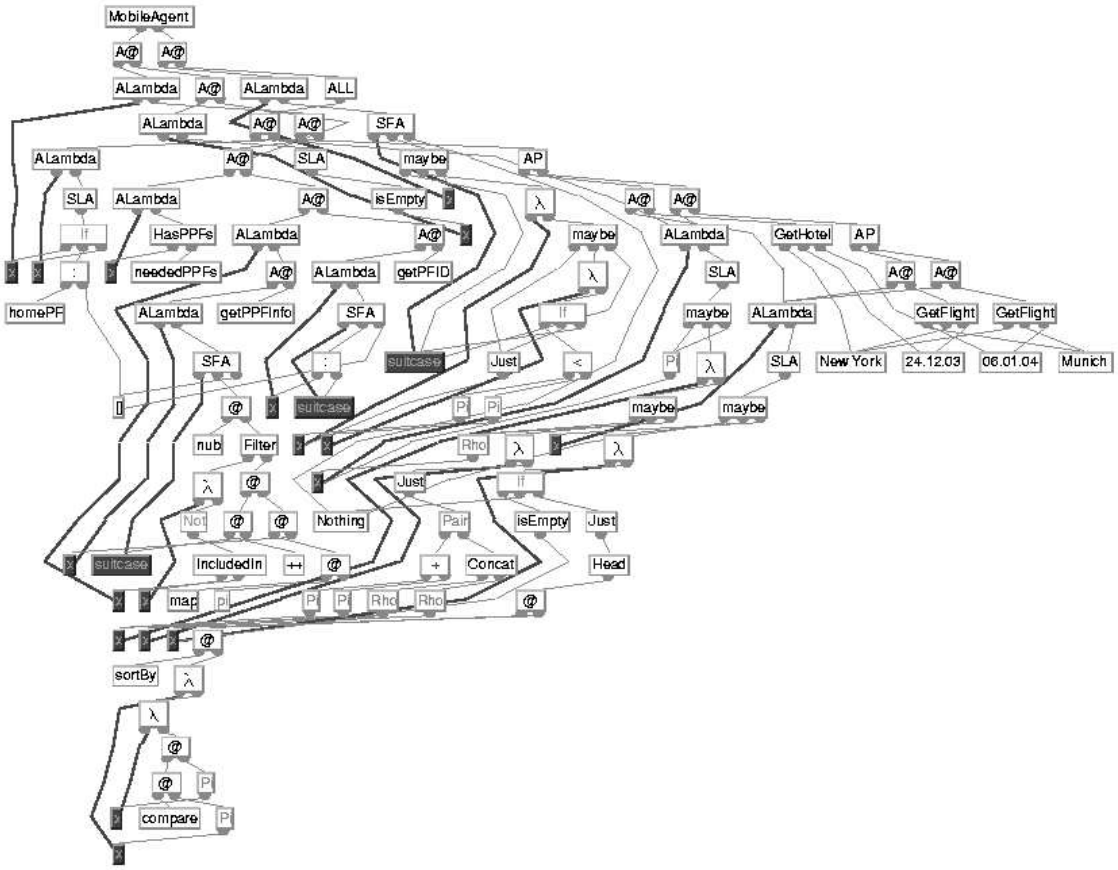


Figure A.13: Travel-Searching Agent 2 in UI-DSL

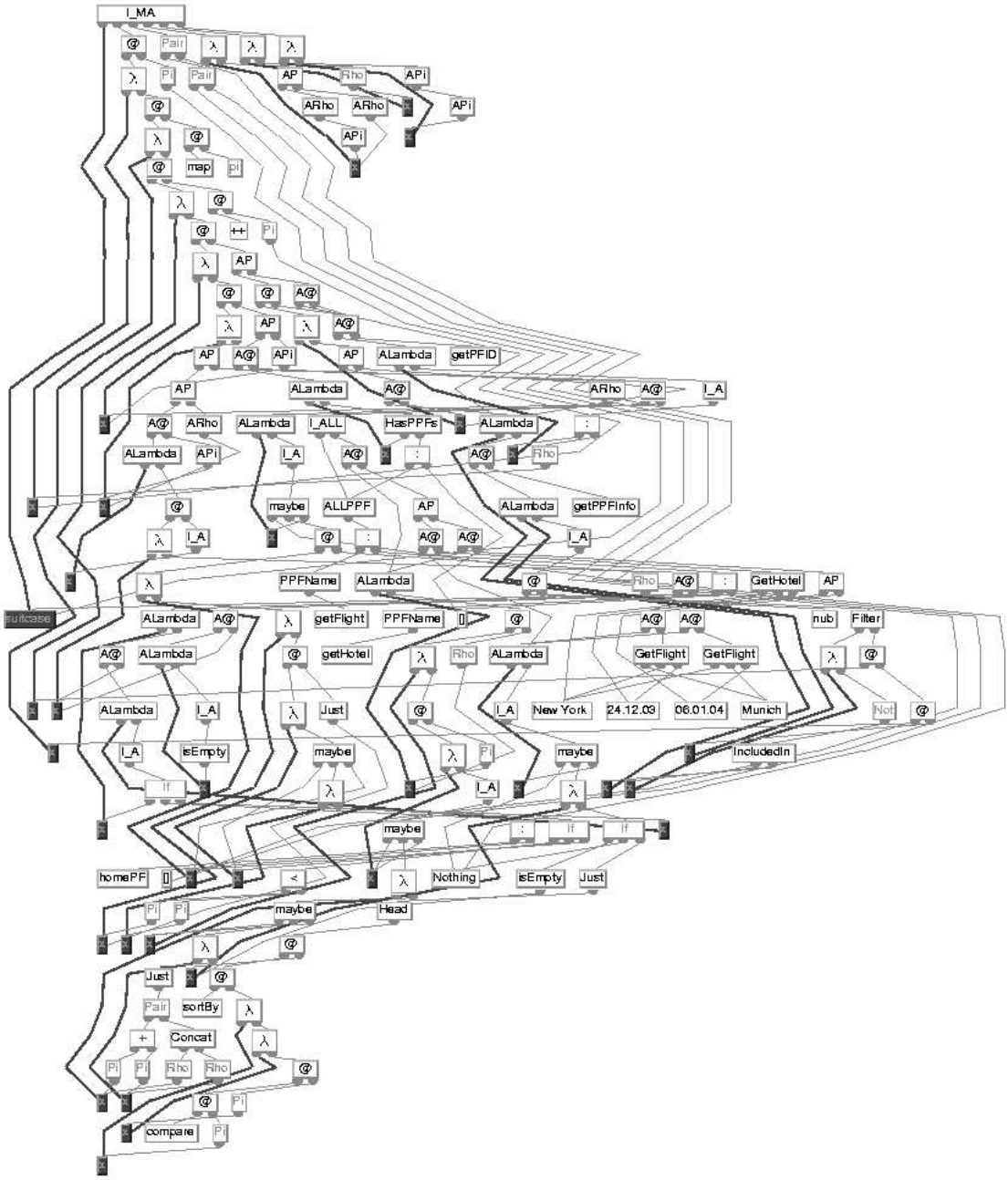


Figure A.14: Travel-Searching Agent 2 in I-DSL

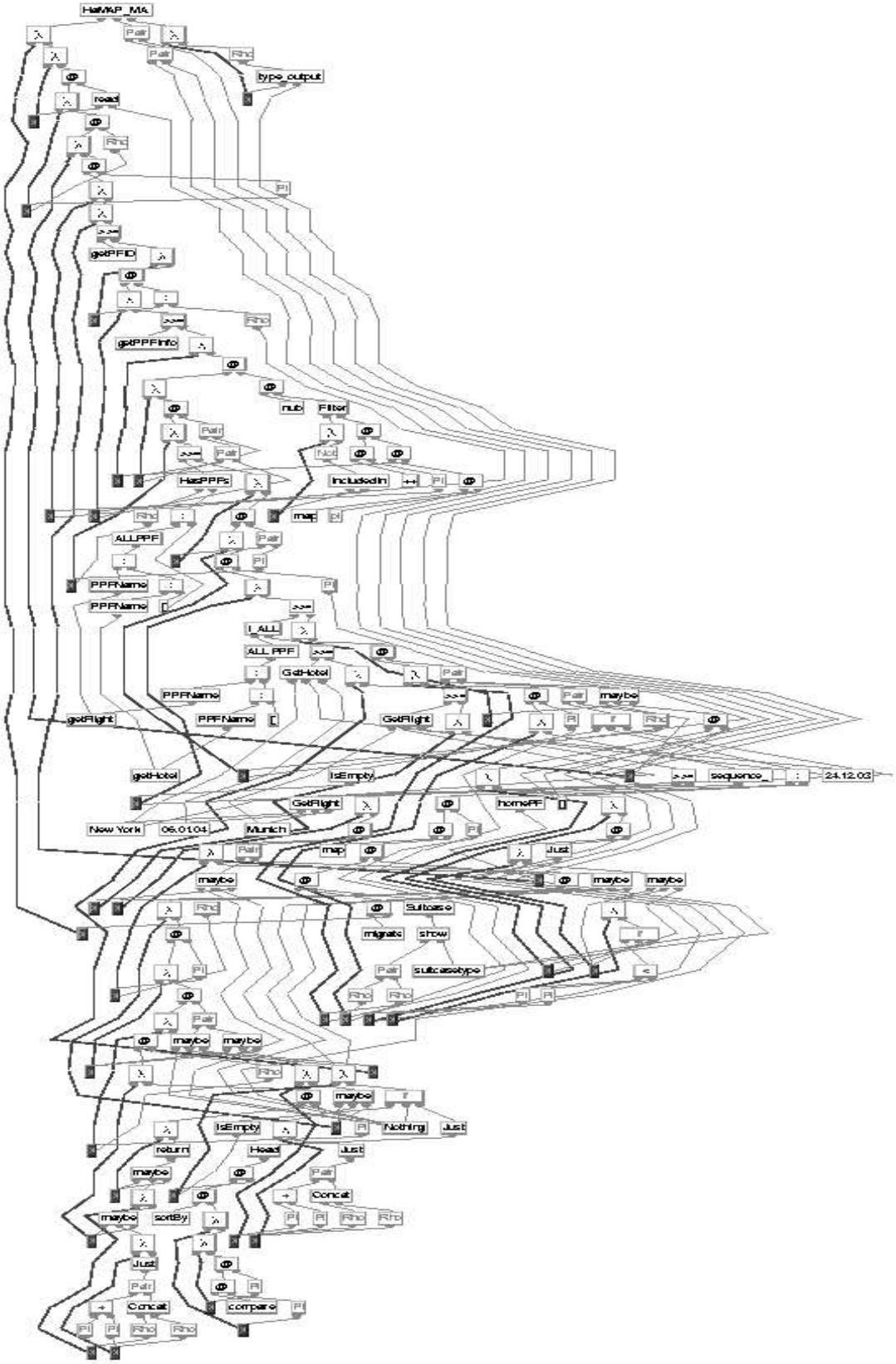


Figure A.15: Travel-Searching Agent 2 as HaMAP mobile agent DAG

```
MobileAgent_HOPS
 {code_HOPS =
  [Code "\\ x17 -> \\ x37 -> ( \\ x36 -> ( \\ x23 -> ( \\ x33 -> \\ x20 -> ("
  ,Code "getPFID ) >>= ( \\ x35 -> ( \\ x31 -> ( getPPFInfo ) >>= ( \\ x34"
  ,Code "-> ( \\ x30 -> ( \\ x0 -> ( hasPPFs ( snd ( x0 ) ) ( ( ALLPPF ( ("
  ,Code "PPFName \"getFlight\" ) : ( ( PPFName \"getHotel\" ) : ( [] ) ) )"
  ,Code ") : ( [] ) ) ) >>= ( \\ x29 -> ( \\ x22 -> ( \\ x21 -> ("
  ,ALL (ALLPPF [PPFName "getFlight",PPFName "getHotel",EmptyPPF])
       [Code "("
       ,PPF (PPFName "getHotel") [Code "\"New York\" \"24.12.03\" \"06.01.04\""]
       ,Code ") >>= ( \\ x9 -> ("
       ,PPF (PPFName "getFlight") [Code "\"Munich\" \"New York\" \"24.12.03\""]
       ,Code ") >>= ( \\ x15 -> ("
       ,PPF (PPFName "getFlight") [Code "\"New York\" \"Munich\" \"06.01.04\""]
       ,Code ") >>= ( \\ x16 -> ( \\ x14 -> ( \\ x11 -> ( \\ x10 -> ( \\ x5 -> (\\"
       ,Code "x2 -> ( \\ x1 -> return ( maybe ( x1 ) ( \\ x3 -> maybe ( x2 ) ( \\"
       ,Code "x4 -> Just ( ( ( fst ( x3 ) ) + ( fst ( x4 ) ) , ( snd ( x3 )"
       ,Code ") ++ ( snd ( x4 ) ) ) ) ) ( x2 ) ) ( x1 ) ) )( fst ( x5 ) ) )("
       ,Code "snd ( x5 ) ) )( ( maybe ( Nothing ) ( \\ x6 -> if ( null ( x6 )"
       ,Code ") then ( Nothing ) else ( Just ( head ( ( ( sortBy )( \\ x7 -> \\"
       ,Code "x8 -> ( ( compare :: ( ( Integer -> ( Integer -> Ordering ) ) )"
       ,Code ")( fst ( x7 ) ) )( fst ( x8 ) ) ) )( x6 ) ) ) ) ) ( x9 ) ,"
       ,Code "maybe ( x10 ) ( \\ x12 -> maybe ( x11 ) ( \\ x13 -> Just ( ( ("
       ,Code "fst ( x12 ) ) + ( fst ( x13 ) ) , ( snd ( x12 ) ) ++ ( snd ( x13"
       ,Code ") ) ) ) ) ( x11 ) ) ( x10 ) ) ) )( fst ( x14 ) ) )( snd ( x14 )"
       ,Code ") )( ( maybe ( Nothing ) ( \\ x6 -> if ( null ( x6 ) ) then ("
       ,Code "Nothing ) else ( Just ( head ( ( ( sortBy )( \\ x7 -> \\ x8 -> ( ("
       ,Code "compare :: ( ( Integer -> ( Integer -> Ordering ) ) ) )( fst ("
       ,Code "x7 ) ) )( fst ( x8 ) ) ) )( x6 ) ) ) ) ) ( x15 ) , maybe ("
       ,Code "Nothing ) ( \\ x6 -> if ( null ( x6 ) ) then ( Nothing ) else ("
       ,Code "Just ( head ( ( ( sortBy )( \\ x7 -> \\ x8 -> ( ( compare :: ( ("
       ,Code "Integer -> ( Integer -> Ordering ) ) ) )( fst ( x7 ) ) )( fst ("
       ,Code "x8 ) ) ) )( x6 ) ) ) ) ) ( x16 ) ) ) ) ) )"]
  ,Code ") >>= ( \\ x28 -> ( \\ x19 -> ( \\ x18 -> sequence_ ( ( ( map )( ("
  ,Code "( ( migrate )( x17 ) )( Suitcase ( show (( ( snd ( x18 ) , snd ("
  ,Code "x19 ) ) ) :: ( ( ( [ PFID ] , [ PFID ] ) , Maybe ( ( Integer ,"
  ,Code "String ) ) ) ) ) ) )( x20 ) ) )( fst ( x18 ) ) ) )( fst ( x19 )"
  ,Code ") )( ( ( if ( null ( x21 ) ) then ( ( homePF ( x20 ) ) : ( [] )"
  ,Code ") else ( x21 ) , snd ( x22 ) ) , maybe ( x23 ) ( ( \\ x27 -> \\"
  ,Code "x25 -> ( \\ x24 -> maybe ( x24 ) ( \\ x26 -> if ( ( fst ( x25 )"
  ,Code ") < ( fst ( x26 ) ) ) then ( x24 ) else ( x27 ) ) ( x27 ) )("
  ,Code "Just ( x25 ) ) )( x23 ) ) ( x28 ) ) ) ) )( fst ( x22 ) ) )( ("
  ,Code "x29 , fst ( x0 ) ) ) ) ) )( ( ( x30 , x31 ) , x30 ) ) )( ( nub )("
  ,Code "filter ( \\ x32 -> not ( elem ( x32 ) ( x31 ) ) ) ( ( ( (++) )("
  ,Code "fst ( x33 ) ) )( ( ( map )( \\x -> fst x ) )( x34 ) ) ) ) ) ) )("
  ,Code "( x35 ) : ( snd ( x33 ) ) ) ) )( fst ( x36 ) ) )( snd ( x36 ) )"
  ,Code ")( read ( x37 ) :: ( ( ( [ PFID ] , [ PFID ] ) , Maybe ( ("
  ,Code "Integer , String ) ) ) ) )"]
 ,suitcase_HOPS = [Code "( ( [] , [] ) , Nothing )"]
 ,value_HOPS = [Code "\\ x38 -> snd ( x38 :: ( ( ( [ PFID ] , [ PFID ] ) ,"
              ,Code " Maybe ( ( Integer , String ) ) ) ) )"]}
```

Figure A.16: Generated Haskell code for the Travel-Searching Agent 2

A.5 Travel-Searching Agent 3

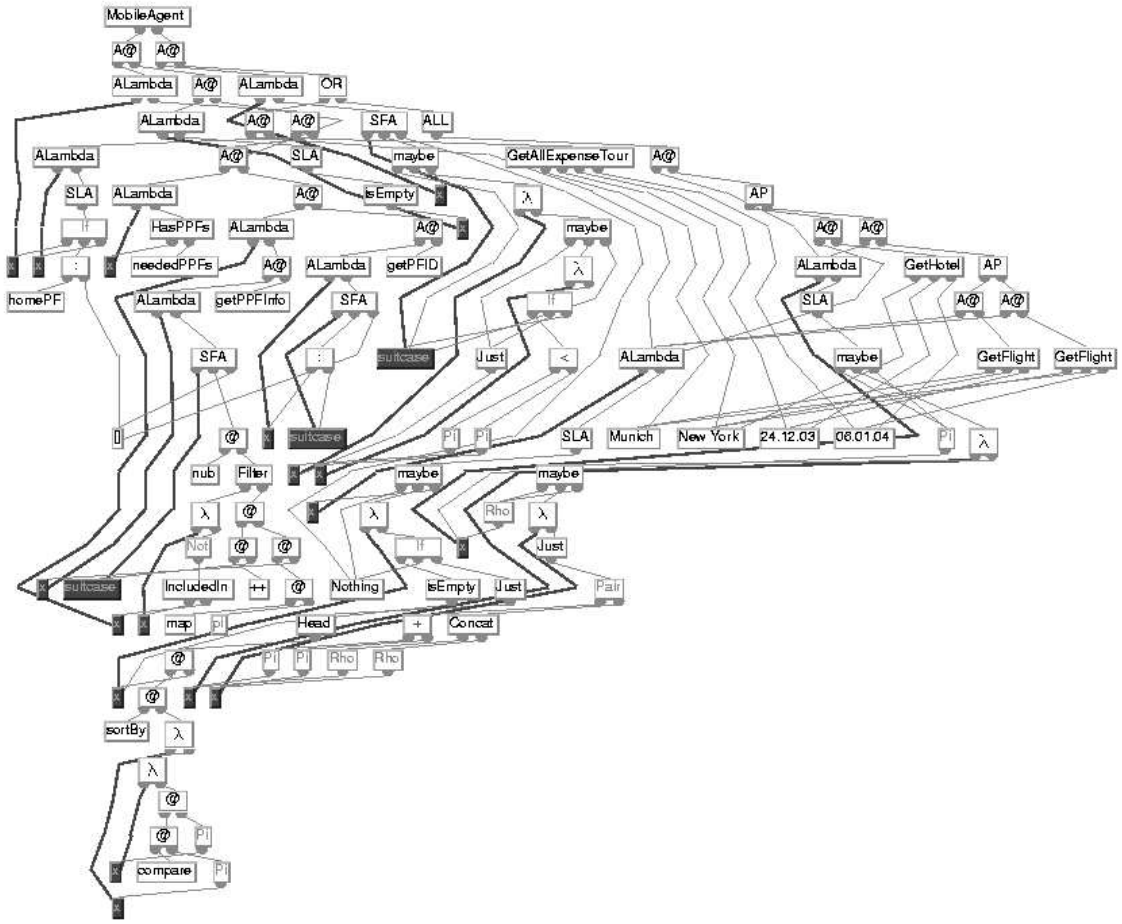


Figure A.17: Travel-Searching Agent 3 in UI-DSL

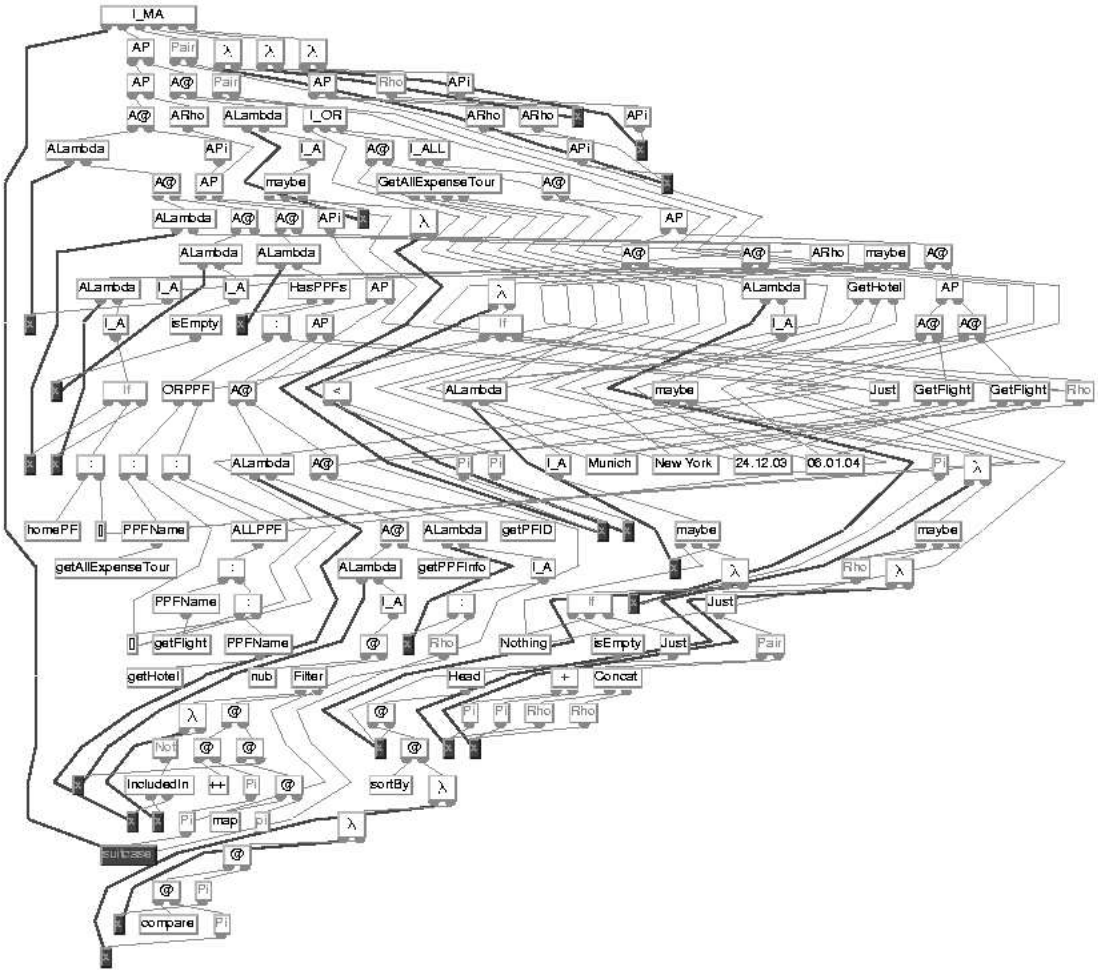


Figure A.18: Travel-Searching Agent 3 in I-DSL

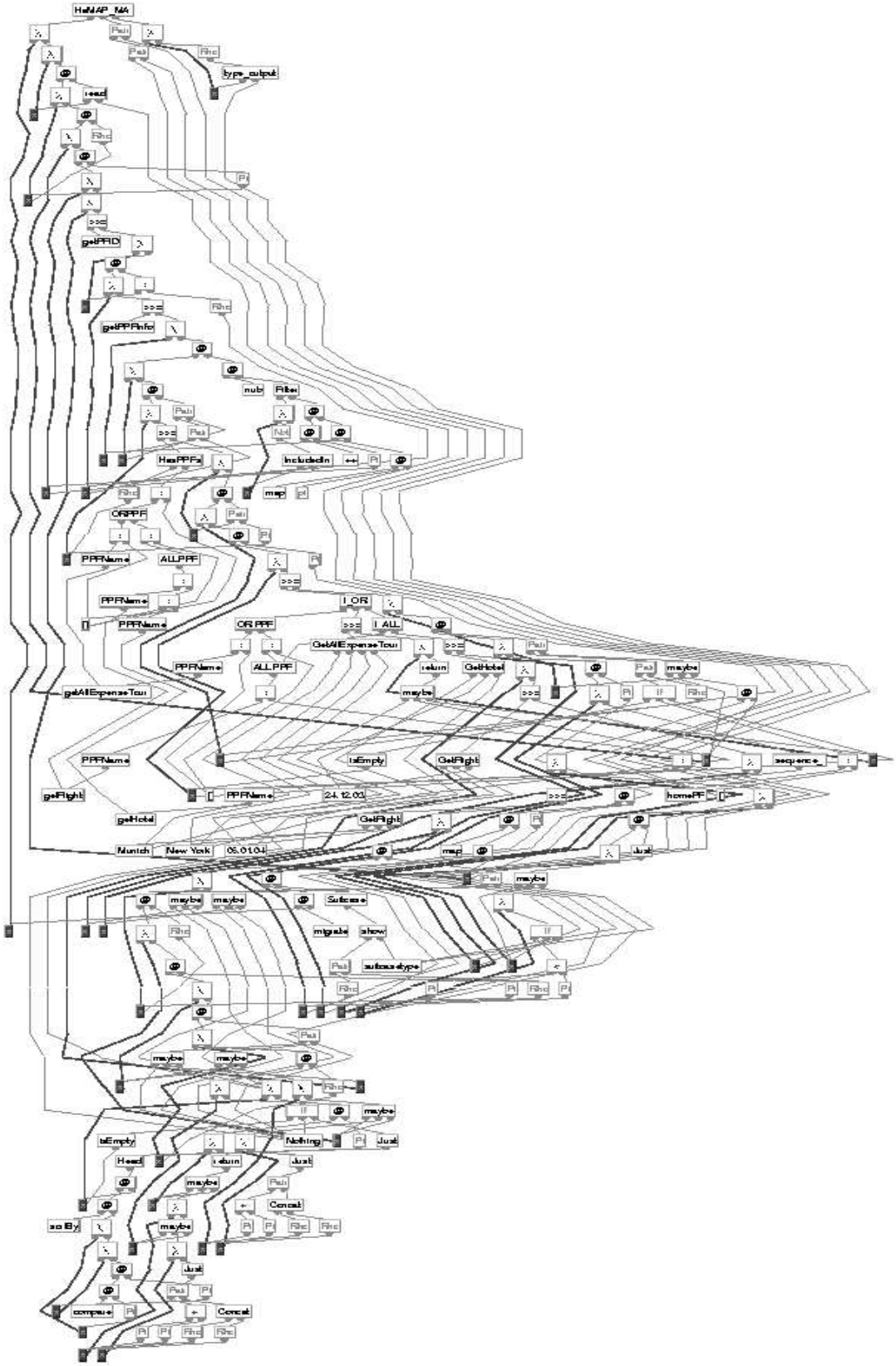


Figure A.19: Travel-Searching Agent 3 as HaMAP mobile agent DAG

```
MobileAgent_HOPS
 {code_HOPS =
  [Code "\\ x18 -> \\ x38 -> ( \\ x37 -> ( \\ x24 -> ( \\ x34 -> \\ x21 -> ("
  ,Code "getPFID ) >>= ( \\ x36 -> ( \\ x32 -> ( getPPFInfo ) >>= ( \\ x35"
  ,Code "-> ( \\ x31 -> ( \\ x0 -> ( hasPPFs ( snd ( x0 ) ) ( ( ORPPF ( ("
  ,Code "PPFName \"getAllExpenseTour\" ) : ( [] ) ) ( ( ALLPPF ( ("
  ,Code "PPFName \"getFlight\" ) : ( ( PPFName \"getHotel\" ) : ( [] ) ) )"
  ,Code ") : ( [] ) ) ) : ( [] ) ) ) >>= ( \\ x30 -> ( \\ x23 -> ( \\ x22 ->"
  ,Code "("
  ,OR (ORPPF [PPFName "getAllExpenseTour",EmptyPPF]
             [ALLPPF [PPFName "getFlight",PPFName "getHotel",EmptyPPF],EmptyPPF])
      [Code "("
      ,PPF (PPFName "getAllExpenseTour")
           [Code "\"Munich\" \"New York\" \"24.12.03\" \"06.01.04\""]
      ,Code ") >>= ( \\ x4 -> return ( maybe ( Nothing ) ( \\ x1 -> if ( null ("
      ,Code "x1 ) ) then ( Nothing ) else ( Just ( head ( ( ( sortBy )( \\ x2"
      ,Code "-> \\ x3 -> ( ( compare :: ( ( Integer -> ( Integer -> Ordering )"
      ,Code ") ) )( fst ( x2 ) ) )( fst ( x3 ) ) ) )( x1 ) ) ) ) ) ( x4 ) ) )"]
      [ALL (ALLPPF [PPFName "getFlight",PPFName "getHotel",EmptyPPF])
           [Code "("
           ,PPF (PPFName "getHotel") [Code "\"New York\" \"24.12.03\" \"06.01.04\""]
           ,Code ") >>= ( \\ x10 -> ("
           ,PPF (PPFName "getFlight") [Code "\"Munich\" \"New York\" \"24.12.03\""]
           ,Code ") >>= ( \\ x16 -> ("
           ,PPF (PPFName "getFlight") [Code "\"New York\" \"Munich\" \"06.01.04\""]
           ,Code ") >>= ( \\ x17 -> ( \\ x15 -> ( \\ x12 -> ( \\ x11 -> ( \\ x9 -> ( \\"
           ,Code "x6 -> ( \\ x5 -> return ( maybe ( x5 ) ( \\ x7 -> maybe ( x6 ) ( \\"
           ,Code "x8 -> Just ( ( ( fst ( x7 ) ) + ( fst ( x8 ) ) , ( snd ( x7 )"
           ,Code ") ++ ( snd ( x8 ) ) ) ) ) ( x6 ) ) ( x5 ) ) )( fst ( x9 ) ) )("
           ,Code "snd ( x9 ) ) )( ( maybe ( Nothing ) ( \\ x1 -> if ( null ( x1 )"
           ,Code ") then ( Nothing ) else ( Just ( head ( ( ( sortBy )( \\ x2 -> \\"
           ,Code "x3 -> ( ( compare :: ( ( Integer -> ( Integer -> Ordering ) ) )"
           ,Code ")( fst ( x2 ) ) )( fst ( x3 ) ) ) )( x1 ) ) ) ) ) ( x10 ) ,"
           ,Code "maybe ( x11 ) ( \\ x13 -> maybe ( x12 ) ( \\ x14 -> Just ( ( ("
           ,Code "fst ( x13 ) ) + ( fst ( x14 ) ) , ( snd ( x13 ) ) ++ ( snd ( x14"
           ,Code ") ) ) ) ) ( x12 ) ) ( x11 ) ) ) )( fst ( x15 ) ) )( snd ( x15 )"
           ,Code ") )( ( maybe ( Nothing ) ( \\ x1 -> if ( null ( x1 ) ) then ("
           ,Code "Nothing ) else ( Just ( head ( ( ( sortBy )( \\ x2 -> \\ x3 -> ( ("
           ,Code "compare :: ( ( Integer -> ( Integer -> Ordering ) ) ) )( fst ("
           ,Code "x2 ) ) )( fst ( x3 ) ) ) )( x1 ) ) ) ) ) ( x16 ) , maybe ("
           ,Code "Nothing ) ( \\ x1 -> if ( null ( x1 ) ) then ( Nothing ) else ("
           ,Code "Just ( head ( ( ( sortBy )( \\ x2 -> \\ x3 -> ( ( compare :: ( ("
           ,Code "Integer -> ( Integer -> Ordering ) ) ) )( fst ( x2 ) ) )( fst ("
           ,Code "x3 ) ) ) )( x1 ) ) ) ) ) ( x17 ) ) ) ) ) )"]]
  ,Code ") >>= ( \\ x29 -> ( \\ x20 -> ( \\ x19 -> sequence_ ( ( ( map )( ("
  ,Code "( ( migrate )( x18 ) )( Suitcase ( show (( ( snd ( x19 ) , snd ("
  ,Code "x20 ) ) ) :: ( ( ( [ PFID ] , [ PFID ] ) , Maybe ( ( Integer ,"
  ,Code "String ) ) ) )) ) ) )( x21 ) ) )( fst ( x19 ) ) ) )( fst ( x20 )"
  ,Code ") )( ( ( if ( null ( x22 ) ) then ( ( homePF ( x21 ) ) : ( [] )"
  ,Code ") else ( x22 ) , snd ( x23 ) ) , maybe ( x24 ) ( ( \\ x28 -> \\"
  ,Code "x26 -> ( \\ x25 -> maybe ( x25 ) ( \\ x27 -> if ( ( fst ( x26 )"
  ,Code ") < ( fst ( x27 ) ) ) then ( x25 ) else ( x28 ) ) ( x28 ) )("
  ,Code "Just ( x26 ) ) )( x24 ) ) ( x29 ) ) ) ) )( fst ( x23 ) ) )( ("
  ,Code "x30 , fst ( x0 ) ) ) ) )( ( ( x31 , x32 ) , x31 ) ) )( ( nub )("
  ,Code "filter ( \\ x33 -> not ( elem ( x33 ) ( x32 ) ) ) ( ( ( (++) )("
  ,Code "fst ( x34 ) ) )( ( ( map )( \\x -> fst x ) )( x35 ) ) ) ) ) ) )("
  ,Code "( x36 ) : ( snd ( x34 ) ) ) ) )( fst ( x37 ) ) )( snd ( x37 ) )"
  ,Code ")( read ( x38 ) :: ( ( ( [ PFID ] , [ PFID ] ) , Maybe ( ("
  ,Code "Integer , String ) ) ) ) )"]
 ,suitcase_HOPS = [Code "( ( [] , [] ) , Nothing )"]
 ,value_HOPS = [Code "\\ x39 -> snd ( x39 :: ( ( ( [ PFID ] , [ PFID ] ) , Maybe ( ("
               ,Code "Integer , String ) ) ) ) )"]}
```

Figure A.20: Generated Haskell code for the Travel-Searching Agent 3

A.6 Travel-Searching and Booking Agent

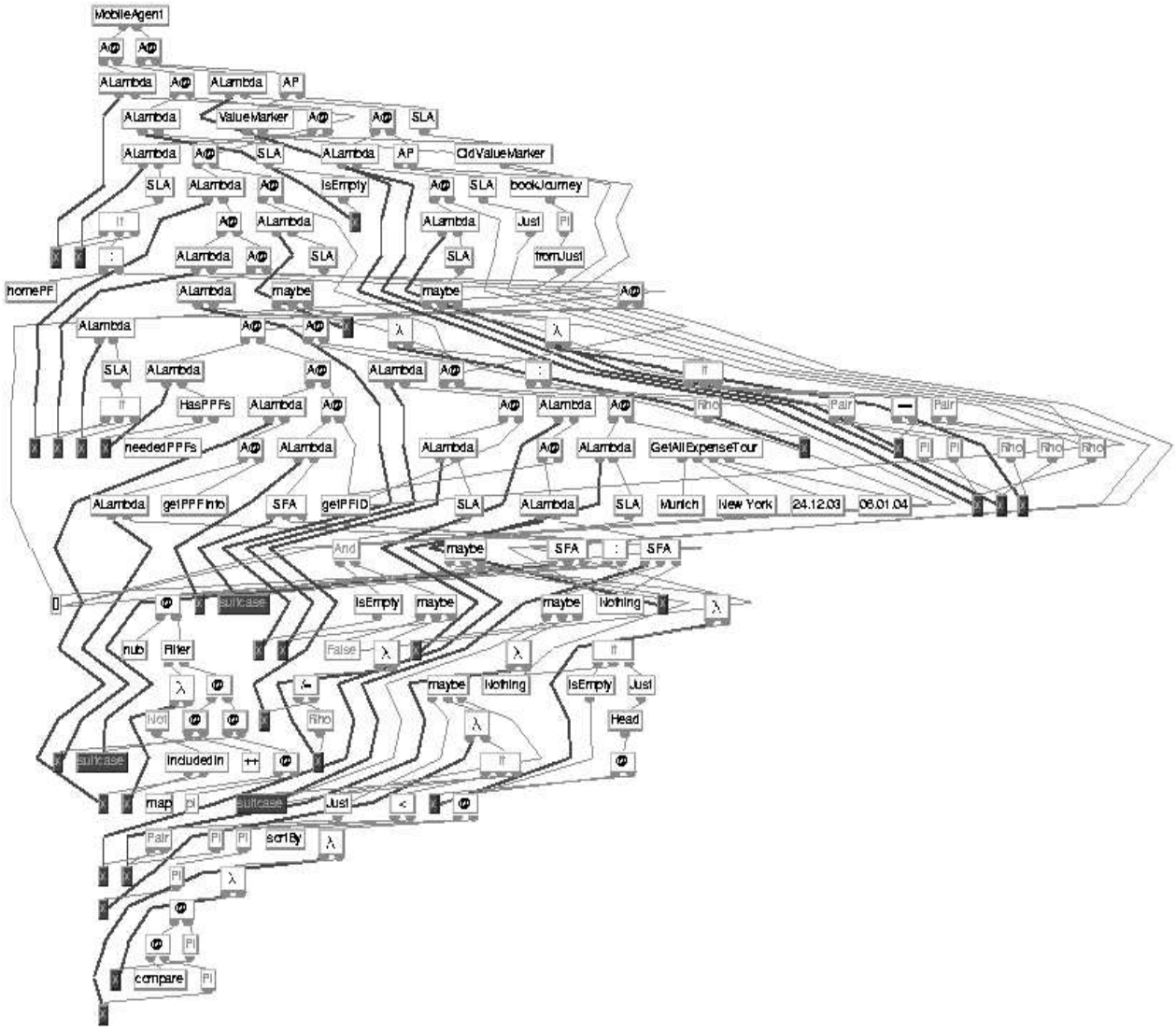


Figure A.21: Travel-Searching and Booking Agent in UI-DSL

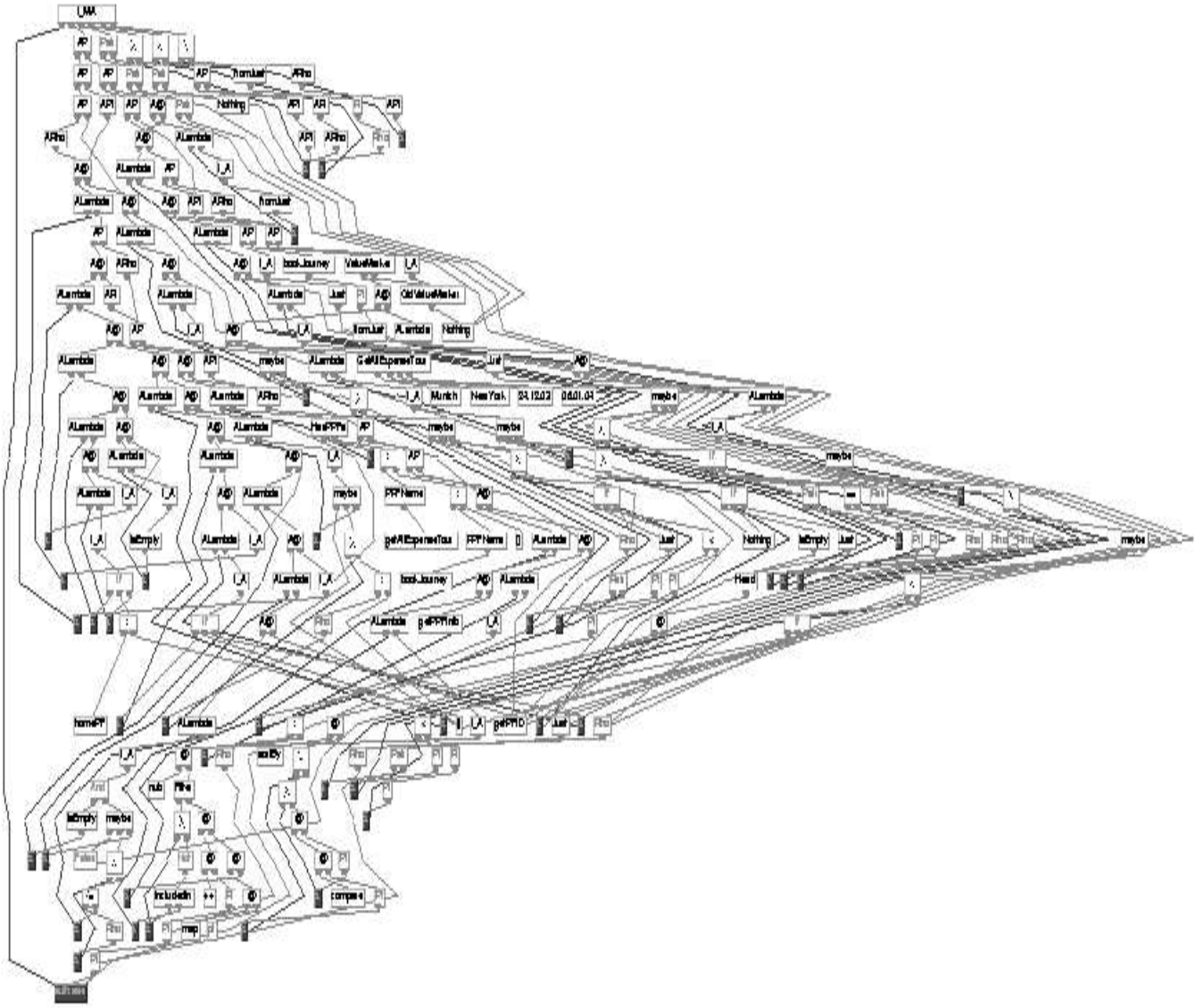


Figure A.22: Travel-Searching and Booking Agent in I-DSL

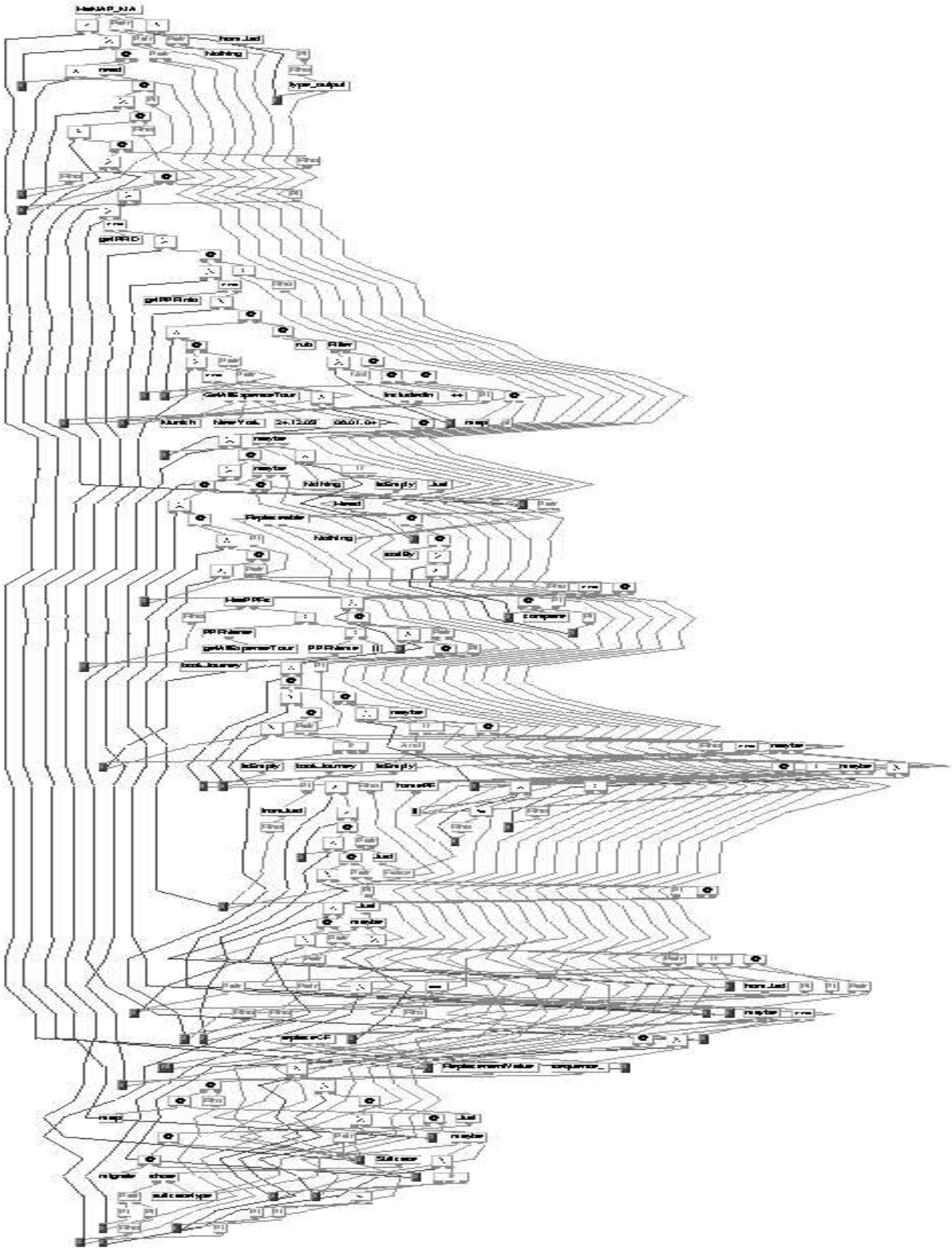


Figure A.23: Travel-Searching and Booking Agent as HaMAP mobile agent DAG

```
MobileAgent_HOPS
 {code_HOPS =
  [Code "\\ x4 -> \\ x44 -> ( \\ x43 -> ( \\ x42 -> ( \\ x10 -> ( \\ x32 -> ( \\"
  ,Code "x40 -> \\ x8 -> ( getPFID ) >>= ( \\ x15 -> ( \\ x38 -> ("
  ,Code "getPPFInfo ) >>= ( \\ x41 -> ( \\ x37 -> ( \\ x0 -> ("
  ,PPF (PPFName "getAllExpenseTour")
       [Code "\"Munich\" \"New York\" \"24.12.03\" \"06.01.04\""]
  ,Code ") >>= ( \\ x36 -> ( \\ x16 -> ( \\ x3 -> ( \\ x31 -> ( \\ x18 -> ( \\"
  ,Code "x1 -> ( hasPPFs ( snd ( x0 ) ) ( ( PPFName \"getAllExpenseTour\""
  ,Code ") : ( ( PPFName \"bookJourney\" ) : ( [] ) ) ) ) >>= ( \\ x30 -> ("
  ,Code "\\ x25 -> ( \\ x26 -> ( \\ x24 -> ( \\ x9 -> ("
  ,PPF (PPFName "bookJourney") [Code "( fst ( fromJust ( snd ( x1 ) ) ) )"]
  ,Code ") >>= ( ( \\ x22 -> \\ x23 -> ( \\ x21 -> ( \\ x19 -> ( \\ x17 -> ( \\"
  ,Code "x7 -> ( \\ x6 -> ( replaceCF 2 ( [Code (show ( x3 ))] ) ( x4 )"
  ,Code ") >>= ( \\ x5 -> sequence_ ( ( ( map )( ( ( migrate )( x5 ) )("
  ,Code "Suitcase ( show (( ( fst ( x6 ) , fst ( snd ( x7 ) ) ) ) :: ( ("
  ,Code "( ( [ PFID ] , [ PFID ] ) , Maybe ( ( ( Integer , String ) ,"
  ,Code "PFID ) ) ) , ( Maybe ( ( Maybe ( Bool ) , Maybe ( ( ( Integer ,"
  ,Code "String ) , PFID ) ) ) ) , Maybe ( ( ( Integer , String ) , PFID"
  ,Code ") ) ) ) )) ) ) )( x8 ) ) )( snd ( x6 ) ) ) )( fst ( x7 ) ) )("
  ,Code "( ( ( snd ( x9 ) , maybe ( x10 ) ( ( \\ x14 -> \\ x12 -> ( \\ x11"
  ,Code "-> maybe ( x11 ) ( \\ x13 -> if ( ( fst ( x12 ) ) < ( fst ( fst ("
  ,Code "x13 ) ) ) ) then ( x11 ) else ( x14 ) ) ( x14 ) )( Just ( ( x12"
  ,Code ", x15 ) ) ) )( x10 ) ) ( x16 ) ) , fst ( x9 ) ) , ( ( x17 , x18"
  ,Code ") , fromJust ( x17 ) ) ) ) )( Just ( maybe ( x19 ) ( \\ x20 ->"
  ,Code "if ( ( snd ( x20 ) ) == ( x15 ) ) then ( x19 ) else ( ( snd ("
  ,Code "x21 ) , x22 ) ) ( x22 ) ) ) )( ( fst ( x21 ) , fst ( x1 ) ) )"
  ,Code ")( ( Just ( False ) , x23 ) ) )( snd ( x1 ) ) ) )( ( if ( null ("
  ,Code "x24 ) ) then ( ( homePF ( x8 ) ) : ( [] ) ) else ( x24 ) , snd ("
  ,Code "x25 ) ) ) )( ( \\ x28 -> if ( ( null ( x26 ) ) && ( maybe ( False"
  ,Code ") ( \\ x27 -> ( snd ( x27 ) ) /= ( x15 ) ) ( x28 ) ) ) then ("
  ,Code "maybe ( [] ) ( \\ x29 -> ( snd ( x29 ) ) : ( [] ) ) ( x28 )"
  ,Code ") else ( x26 ) )( maybe ( x10 ) ( ( \\ x14 -> \\ x12 -> ( \\ x11 ->"
  ,Code "maybe ( x11 ) ( \\ x13 -> if ( ( fst ( x12 ) ) < ( fst ( fst ("
  ,Code "x13 ) ) ) ) then ( x11 ) else ( x14 ) ) ( x14 ) )( Just ( ( x12"
  ,Code ", x15 ) ) ) )( x10 ) ) ( x16 ) ) ) )( fst ( x25 ) ) )( ( x30 ,"
  ,Code "fst ( x0 ) ) ) ) )( ( x18 , snd ( x31 ) ) ) )( fst ( x31 ) ) )("
  ,Code "( x3 ,"
  ,Replaceable 2 [Code "Nothing"]
  ,Code ") ) )( maybe ( x32 ) ( ( \\ x14 -> \\ x12 -> ( \\ x11 -> maybe ("
  ,Code "x11 ) ( \\ x13 -> if ( ( fst ( x12 ) ) < ( fst ( fst ( x13 ) ) )"
  ,Code ") then ( x11 ) else ( x14 ) ) ( x14 ) )( Just ( ( x12 , x15 ) )"
  ,Code ") )( x32 ) ) ( x16 ) ) )( maybe ( Nothing ) ( \\ x33 -> if ("
  ,Code "null ( x33 ) ) then ( Nothing ) else ( Just ( head ( ( ( sortBy"
  ,Code ")( \\ x34 -> \\ x35 -> ( ( compare :: ( ( Integer -> ( Integer ->"
  ,Code "Ordering ) ) ) )( fst ( x34 ) ) )( fst ( x35 ) ) ) )( x33 ) ) )"
  ,Code ") ) ( x36 ) ) ) )( ( ( x37 , x38 ) , x37 ) ) )( ( nub )("
  ,Code "filter ( \\ x39 -> not ( elem ( x39 ) ( x38 ) ) ) ( ( ( (++) )("
  ,Code "fst ( x40 ) ) )( ( ( map )( \\x -> fst x ) )( x41 ) ) ) ) ) )("
  ,Code "( x15 ) : ( snd ( x40 ) ) ) ) )( fst ( x42 ) ) )( snd ( snd ("
  ,Code "x43 ) ) ) )( snd ( x42 ) ) )( fst ( x43 ) ) )( read ( x44 ) :: ("
  ,Code "( ( ( [ PFID ] , [ PFID ] ) , Maybe ( ( ( Integer , String ) ,"
  ,Code "PFID ) ) ) , ( Maybe ( ( Maybe ( Bool ) , Maybe ( ( ( Integer ,"
  ,Code "String ) , PFID ) ) ) ) , Maybe ( ( ( Integer , String ) , PFID"
  ,Code ") ) ) ) ) )"]
 ,suitcase_HOPS =
  [Code "( ( ( [] , [] ) , Nothing ) , ( Just (Nothing, Nothing) , Nothing ) )"]
 ,value_HOPS =
  [Code "\\ x45 -> fromJust ( fst ( snd ( x45 :: ( ( ( ( [ PFID ] , [ PFID"
  ,Code "] ) , Maybe ( ( ( Integer , String ) , PFID ) ) ) , ( Maybe ( ("
  ,Code "Maybe ( Bool ) , Maybe ( ( ( Integer , String ) , PFID ) ) ) ) ,"
  ,Code "Maybe ( ( ( Integer , String ) , PFID ) ) ) ) ) ) ) )"]}
```

Figure A.24: Generated Haskell code for the Travel-Searching and Booking Agent

A.7 `GetWeatherAndTrafficJam` Agent

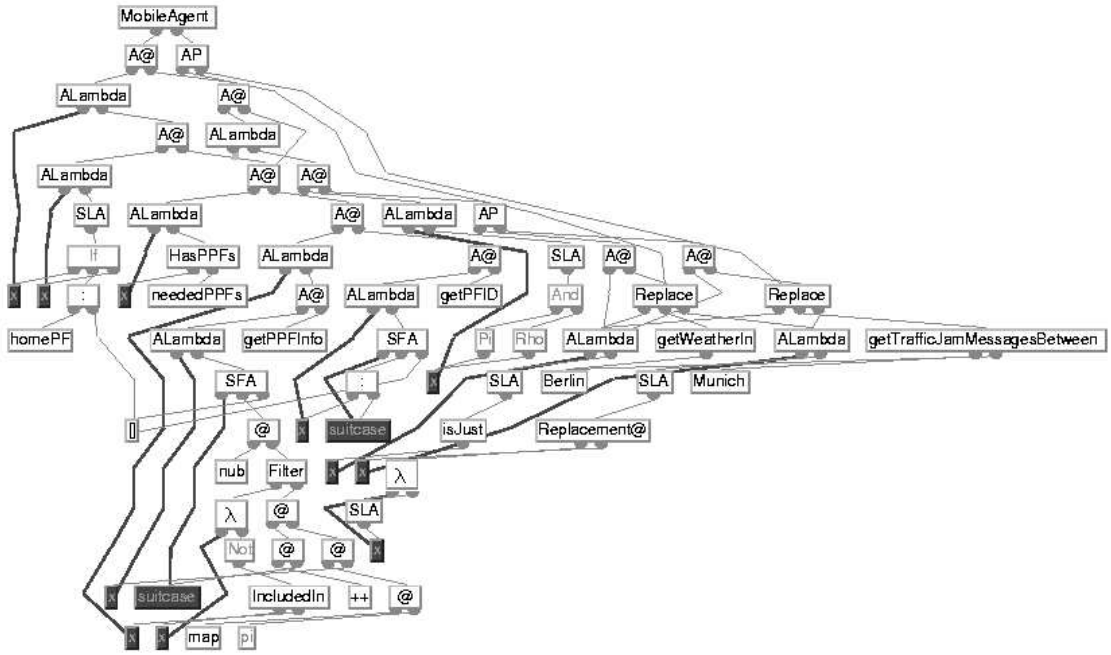


Figure A.25: `GetWeatherAndTrafficJam` Agent in UI-DSL

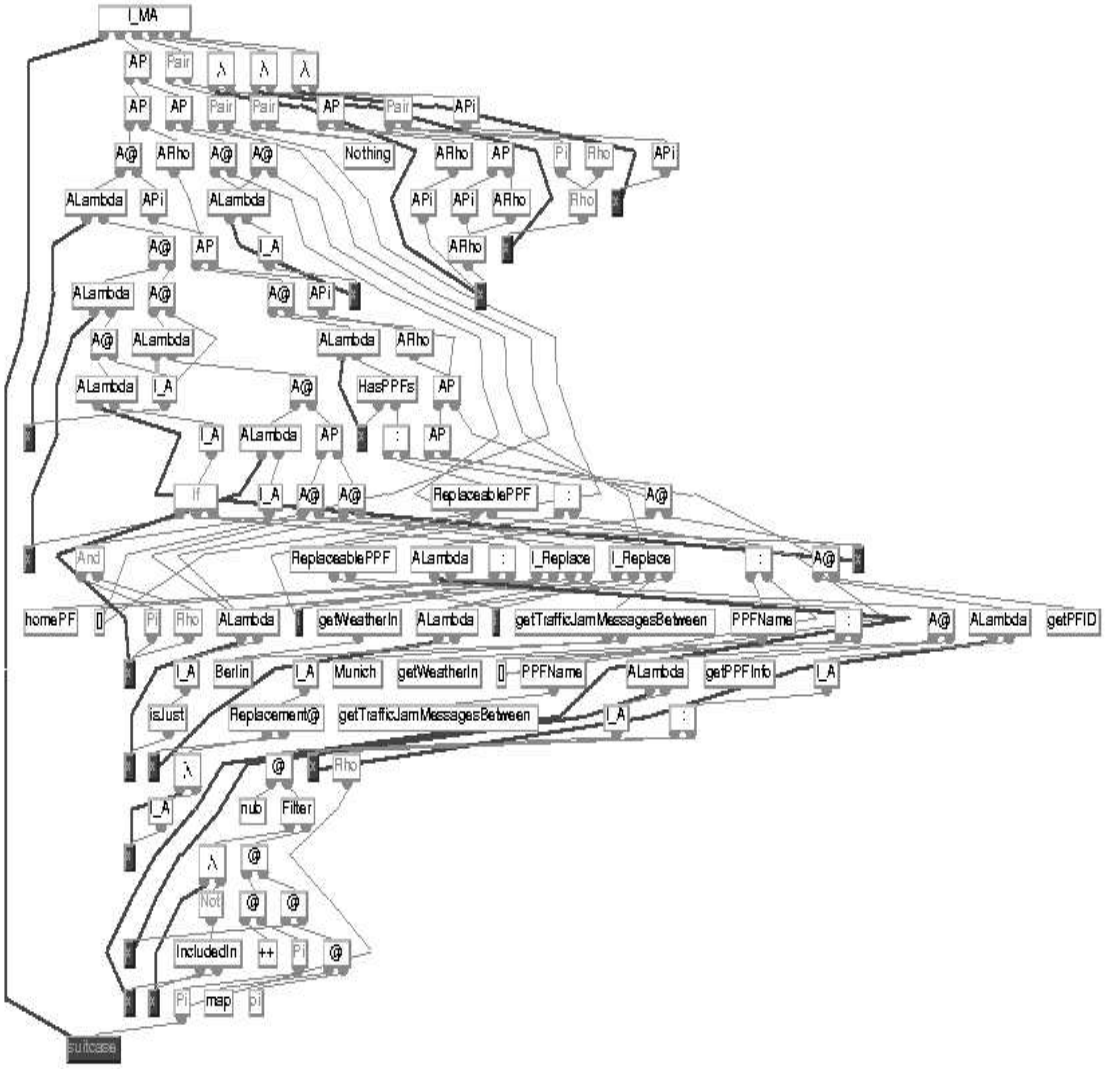


Figure A.26: `GetWeatherAndTrafficJam` Agent in I-DSL

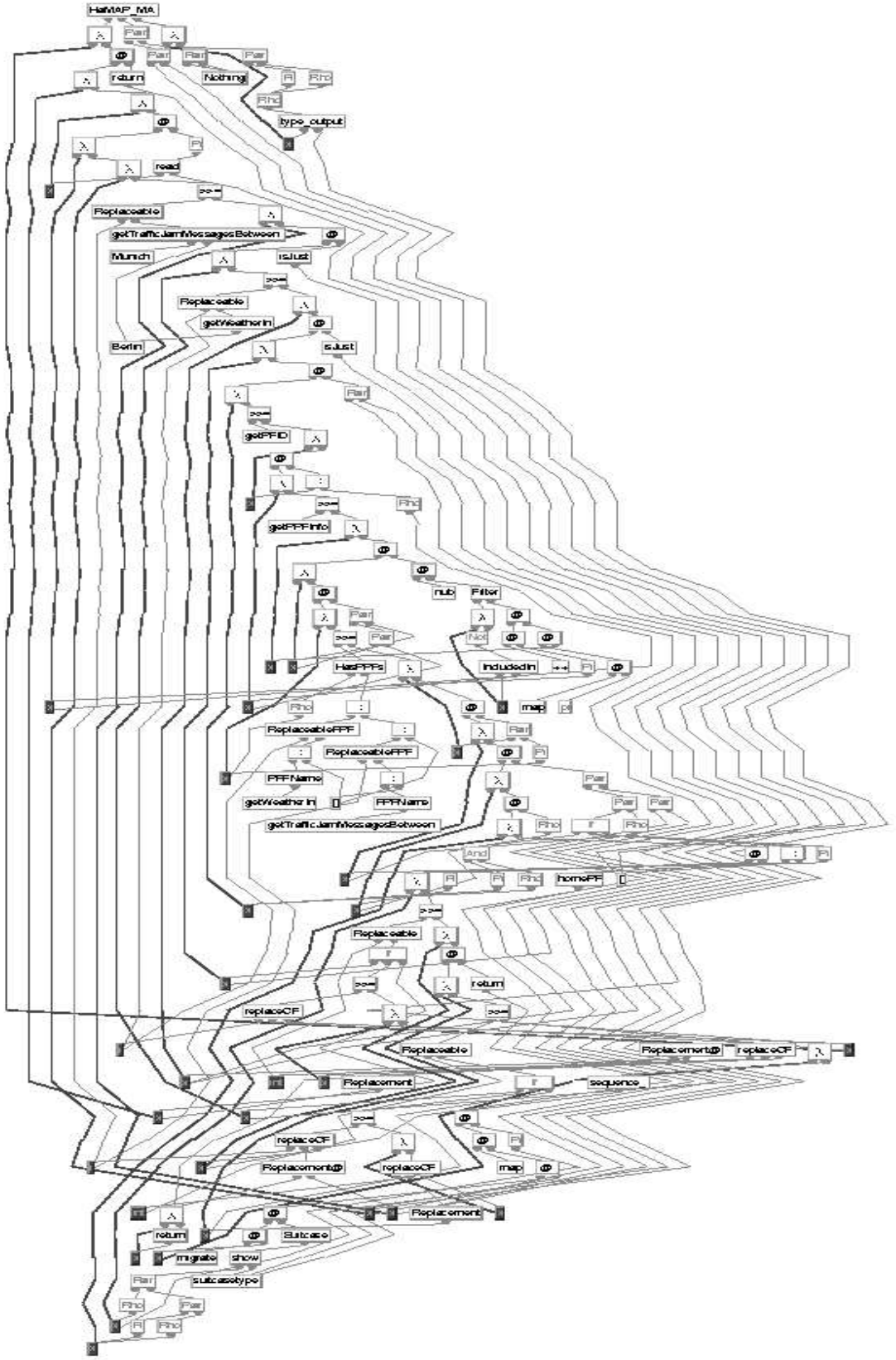


Figure A.27: `GetWeatherAndTrafficJam` Agent as HaMAP mobile agent DAG

```
MobileAgent_HOPS
 {code_HOPS =
  [Code "\\ x7 -> ( \\ x8 -> \\ x30 -> ( \\ x27 -> \\ x19 -> ("
  ,Replaceable 0 [PPF (PPFName "getTrafficJamMessagesBetween")
                      [Code "\"Munich\" \"Berlin\""]]
  ,Code ") >>= ( \\ x12 -> ( \\ x11 -> ("
  ,Replaceable 1 [PPF (PPFName "getWeatherIn") [Code "\"Berlin\""]]
  ,Code ") >>= ( \\ x6 -> ( \\ x4 -> ( \\ x21 -> ( getPFID ) >>= ( \\ x29 ->"
  ,Code "( \\ x25 -> ( getPPFInfo ) >>= ( \\ x28 -> ( \\ x24 -> ( \\ x2 -> ("
  ,Code "hasPPFs ( snd ( x2 ) ) ( ( ReplaceablePPF 1 ( ("
  ,Code "PPFName \"getWeatherIn\" ) : ( [] ) ) ) : ( ( ReplaceablePPF 0 ( ("
  ,Code "PPFName \"getTrafficJamMessagesBetween\" ) : ( [] ) ) ) : ( [] ) )"
  ,Code ") ) >>= ( \\ x23 -> ( \\ x22 -> ( \\ x20 -> ( \\ x18 -> ( \\ x17 -> ("
  ,Replaceable 3 [Code "if ( x4 ) then ( ( replaceCF 1 ("
                 ,CFList
                   [Replacement [Code "( \\ x5 -> return ( x5 ) )"]
                   ,Code " ++ [Code \"(\" ,Code (show ( x6 )) ,Code \")\"]"]
                 ,Code ") ( x7 ) ) >>= ( \\ x9 -> replaceCF 3 ("
                 ,Replacement [Code "x8"]
                 ,Code ") ( x9 ) ) ) else ( x8 )"]
  ,Code ") >>= ( \\ x13 -> ( \\ x14 -> ("
  ,Replaceable 10 [Code "if ( x11 ) then ( ( replaceCF 0 ("
                  ,CFList
                    [Replacement [Code "( \\ x5 -> return ( x5 ) )"]
                    ,Code " ++ [Code \"(\" ,Code (show ( x12 )) ,Code \")\"]"]
                  ,Code ") ( x13 ) ) >>= ( \\ x15 -> replaceCF 10 ("
                  ,Replacement [Code "x14"]
                  ,Code ") ( x15 ) ) ) else ( x14 )"]
  ,Code ") >>= ( \\ x16 -> sequence_ ( ( ( map )( ( ( ( migrate )( x16 )"
  ,Code ")( Suitcase ( show (( ( snd ( x17 ) , ( fst ( x18 ) , snd ( x18"
  ,Code ") ) ) ) :: ( ( ( [ PFID ] , [ PFID ] ) , ( Maybe ( String ) ,"
  ,Code "Maybe ( String ) ) ) )) ) ) )( x19 ) ) )( fst ( x17 ) ) ) ) )("
  ,Code "return ( x13 ) ) ) )( fst ( x20 ) ) )( snd ( x20 ) ) )( ( ( if ("
  ,Code "( fst ( x21 ) ) && ( snd ( x21 ) ) ) then ( ( homePF ( x19 )"
  ,Code ") : ( [] ) ) else ( fst ( x22 ) ) , snd ( x22 ) ) , ( x6 , x12 )"
  ,Code ") ) )( ( x23 , fst ( x2 ) ) ) ) )( ( ( x24 , x25 ) , x24 ) ) )("
  ,Code "( nub )( filter ( \\ x26 -> not ( elem ( x26 ) ( x25 ) ) ) ( ( ("
  ,Code "(++) )( fst ( x27 ) ) )( ( ( map )( \\x -> fst x ) )( x28 ) ) ) )"
  ,Code ") ) )( ( x29 ) : ( snd ( x27 ) ) ) ) )( ( x4 , x11 ) ) )("
  ,Code "isJust ( x6 ) ) ) )( isJust ( x12 ) ) ) )( fst ( read ( x30"
  ,Code ") :: ( ( ( [ PFID ] , [ PFID ] ) , ( Maybe ( String ) , Maybe ("
  ,Code "String ) ) ) ) ) ) )( return ( x7 ) )"]
 ,suitcase_HOPS = [Code "( ( [] , [] ) , ( Nothing , Nothing ) )"]
 ,value_HOPS =
  [Code "\\ x31 -> ( fst ( snd ( x31 :: ( ( ( [ PFID ] , [ PFID ] ) , ("
  ,Code "Maybe ( String ) , Maybe ( String ) ) ) ) ) ) , snd ( snd ( x31"
  ,Code ":: ( ( ( [ PFID ] , [ PFID ] ) , ( Maybe ( String ) , Maybe ("
  ,Code "String ) ) ) ) ) ) )"]}
```

Figure A.28: Generated Haskell code for the GetWeatherAndTrafficJam Agent

B Z-Notation

In the formalisation presented in Chapter 2, parts of the Z-notation (Spivey, 1992) is used. The descriptive notion of a set in Z is called set comprehension and uses the pattern "$\{signature \mid predicate \bullet term\}$", for example $\{n : \mathbb{N} \mid n < 4 \bullet n^2\} = \{0, 1, 4, 9\}$. For a constantly true predicate it is also possible to write "$\{signature \bullet term\}$" as in $\{x : \mathbb{B} \bullet (x, x)\} = \{(\texttt{True}, \texttt{True}), (\texttt{False}, \texttt{False})\}$. If the *term* is just a variable or a tuple of the variables introduced by *signature* it is also possible to write "$\{signature \mid predicate\}$", e.g. $\{x, y : \mathbb{B} \mid x \neq y\} = \{(\texttt{True}, \texttt{False}), (\texttt{False}, \texttt{True})\}$. Quantification uses the same pattern, for example $\forall\, x : \mathbb{N} \bullet x > 3$. The powerset of a set A is written $\mathbb{P}\, A$. The set of relations between two sets A and B is written $A \leftrightarrow B$. Partial functions from A to B are written $A \nrightarrow B$; total functions are written $A \rightarrow B$. Application of a function $f : A \nrightarrow B$ to an argument $x : A$ is written $f.x$. The domain of a relation $R : A \leftrightarrow B$ is $\mathrm{dom}\, .R := \{(x, y) : R \bullet x\}$, the range is $\mathrm{ran}\, .R := \{(x, y) : R \bullet y\}$. The set of finite sequences of elements of a set A is written A^*. The sequences are considered to be partial functions of type $\mathbb{N} \nrightarrow A$ with contiguous domain starting with zero. The length of a sequence can be calculated by the function function $\mathrm{len} : A^* \rightarrow \mathbb{N}$. The $(i + 1)$-th element of a sequence l is denoted by $l.i$.

The identity relation $I_A : A \leftrightarrow A$ on a set A is usually written I. For two sets A and B the universal relation is $\top_{A,B} := A \times B$ and the empty relation is $\bot_{A,B} := \varnothing$, which are also usually written $\top$ and $\bot$. For two relations $R, S : A \leftrightarrow B$, their union is $R \cup S$ and their intersection is $R \cap S$. The inclusion is written $R \subseteq S$. The complement of R is $\overline{R}$. The converse of R is $R^\smile : B \leftrightarrow A$, defined by $R^\smile := \{(x, y) : R \bullet (y, x)\}$. The composition of two relations $R : A \leftrightarrow B$ and $S : B \leftrightarrow C$ is denoted by $R \fatsemi S : A \leftrightarrow C$ and defined by $R \fatsemi S := \{(x, y) : R;\ (u, z) : S \mid y = u \bullet (x, z)\}$. For a homogeneous relation $R : A \leftrightarrow A$ the transitive closure is R^+ and the reflexive transitive closure is R^*. For a relation $R : A \leftrightarrow B$ the domain restriction on a set $C \subseteq A$ is defined by $C \lhd R := \{(a, b) : R \mid a \in C\}$, the range restriction on a set $D \subseteq B$ is defined by $R \rhd D := \{(a, b) : R \mid b \in D\}$. When a relation $R : A \leftrightarrow A$ is considered as a graph, a node $y : A$ is reachable from another node $x : A$ if and only if $(x, y) \in R^*$. If

$R^+ \subseteq \overline{I}$ the relation is called acyclic. A node b dominates a node a if for every node x and every path from x to a either b lies on that path or x is reachable from b. If a is dominated by b, a is reachable from b. A node r is a source if $r \notin \mathrm{ran}\,.R$. If r the only source r is called root.

Bibliography

Abadi, M. and Gordon, A. D. (1998). A Calculus for Cryptographic Protocols: The Spy Calculus. Research Report 149, Digital Systems Research Center, Palo Alto, CA, USA.

Asperti, A. and Longo, G. (1991). *Categories, types, and structures: An introduction to category theory for the working computer scientist.* MIT Press.

Barendregt, H. P. (1992). Lambda calculi with types. In Abramsky, S., Gabbay, D. M., and Maibaum, T., editors, *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Oxford Univertity Press.

Barrett, D. J. and Silverman, R. (2001). *SSH, The Secure Shell: The Definitive Guide.* O'Reilly.

Baumann, J. (2000). *Mobile Agents: Control Algorithms*, volume 1658 of *Lect. Notes in Comp. Sci.* Springer-Verlag.

Bayer, A., Grobauer, B., Kahl, W., Kempf, P., Schmalhofer, F., Schmidt, G., and Winter, M. (1996). The Higher Object Programming System HOPS. Technical report, Inst. für Informatik der Univ. der Bundeswehr München. Internal Report. 206 p.

Bettini, L. and Nicola, R. D. (2001). Translating Strong Mobility into Weak Mobility. In *Mobile Agents*, pages 182–197.

Birrell, A. D. and Nelson, B. J. (1983). Implementing remote procedure calls. In *Proceedings of the ACM Symposium on Operating System Principles*, page 3, Bretton Woods, NH. Association for Computing Machinery.

Boquist, U. (1999). *Code Optimisation Techniques for Lazy Functional Languages.* PhD thesis, Chalmers University of Technology, Gothenburg.

Brandt, R. and Reiser, H. (2001). Dynamic Adaption of Mobile Agents in Heterogenous Environments. In (Picco, 2001), pages 70–87. 5th International Conference, MA 2001 Atlanta, GA, USA.

Chess, D., Harrison, C., and Kershenbaum, A. (1994). Mobile Agents: Are They a Good Idea? Technical Report RC 19887 (December 21, 1994 - Declassified March 16, 1995), Yorktown Heights, New York.

Ciancarini, P. and Wooldridge, M. J., editors (2000). *Agent-Oriented Software Engineering*, volume 1957 of *Lect. Notes in Comp. Sci.* Springer-Verlag. First International Workshop, AOSE 2000, Limerick, Ireland.

Collberg, C., Thomborson, C., and Low, D. (1997). A Taxonomy of Obfuscating Transformations. Technical Report 148.

Crocker, D. (1982). *RFC 822: Standard for ARPA Internet Text Messages.*

Dam, M., editor (1996). *Analysis and Verification of Multiple-Agent Languages*, volume 1192 of *Lect. Notes in Comp. Sci.* Springer-Verlag. 5th LOMAPS Workshop, Selected Papers, Stockholm, Sweden.

Derichsweiler, F. (2002). *Strategiegesteuerte Transformation von Termgraphen.* Der Andere Verlag, Osnabrück. ISBN 3-89959-026-0; also doctoral dissertation at Fakultät für Informatik, Universität der Bundeswehr München.

Ehrig, H., Engels, G., Kreowski, H.-J., and Rozenberg, G., editors (1999). *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools.* World Scientific, Singapore.

Faxén, K.-F. (1997). *Analysing, Transforming and Compiling Lazy Functional Programs.* Phd thesis, Royal Institute of Technology, Department of Teleinformatics, Stockholm.

Feng, X., Cao, J., Lü, J., and Chan, H. (2001). An Efficient Mailbox-Based Algorithm for Message Delivery in Mobile Agent Systems. In (Picco, 2001), pages 135–151. 5th International Conference, MA 2001 Atlanta, GA, USA.

Fritzinger, J. S. and Mueller, M. (1996). *Java Security.*

Gaspari, M. and Zavattaro, G. (1999). An algebra of actors. In *Proc. 3nd IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 3–18. Kluwer Academic Publishers.

Gordon, A. D. (2000). Notes on Nominal Calculi for Security and Mobility. In Focardi, R. and Gorrieri, R., editors, *Foundations of Security Analysis and Design*, volume 2171 of *Lect. Notes in Comp. Sci.* Springer-Verlag. FOSAD 2000, Bertinoro, Italy.

Gray, R. (1997). *Agent Tcl: A flexible and secure mobile-agent system.* PhD thesis, Dept. of Computer Science, Dartmouth College. Available as Dartmouth Computer Science Technical Report TR98-327.

Gray, R. S., Kotz, D., Cybenko, G., and Rus, D. (1998). D'Agents: Security in a multiple-language, mobile-agent system. *Lect. Notes in Comp. Sci.*, 1419:154–187.

Gray, R. S., Kotz, D., Cybenko, G., and Rus, D. (2000). Mobile Agents: Motivations and State-of-the-Art Systems. Technical Report TR2000-365, Dartmouth College, Computer Science, Hanover, NH.

Gray, R. S., Kotz, D., Peterson, R. A., Gerken, P., Hofmann, M., Chacon, D., Hill, G., and Suri, N. (2001). Mobile-Agent versus Client/Server Performance: Scalability in an Information-Retrieval Task. Technical Report TR2001-386, Dartmouth College, Computer Science, Hanover, NH.

Haller, K. and Schuldt, H. (2001). Using Predicates for Specifying Targets of Migration and Messages in a Peer-to-Peer Mobile Agent environment. In (Picco, 2001), pages 152–168. 5th International Conference, MA 2001 Atlanta, GA, USA.

Hayzelden, A. L. and Bigham, J., editors (1999). *Software Agents for Future Communication Systems.* Springer-Verlag.

Horlait, E., editor (2000). *Mobile Agents for Telecommunication Applications*, volume 1931 of *Lect. Notes in Comp. Sci.* Springer-Verlag. Second International Workshop, MATA 2000, Paris, France.

Huch, F. and Norbisrath, U. (2000). Distributed Programming in Haskell with Ports. *Lect. Notes in Comp. Sci.*, 2011:107–121.

Jeffrey, A. (1999). A distributed object calculus. Technical report, DePaul University, Chicago, USA.

Johansen, D., van Renesse, R., and Schneider, F. B. (1996). Supporting Broad Internet Access to TACOMA. In *Proceedings of the 7th SIGOPS European Workshop*, pages 55–58, Connemara, Ireland.

Kahl, W. (1996). *Algebraische Termgraphersetzung mit gebundenen Variablen*. Reihe Informatik. Herbert Utz Verlag Wissenschaft, München. ISBN 3-931327-60-4; also doctoral dissertation at Fakultät für Informatik, Universität der Bundeswehr München.

Kahl, W. (1998). Internally typed second-order term graphs. In Hromkovič, J. and Sýkora, O., editors, *Graph-Theoretic Concepts in Computer Science, 24th International Workshop, WG '98, Smolenice Castle, Slovak Republic, June 1998, Proceedings*, volume 1517 of *Lect. Notes in Comp. Sci.*, pages 149–163. Springer-Verlag.

Kahl, W. (1999). The term graph programming system HOPS. In Berghammer, R. and Lakhnech, Y., editors, *Tool Support for System Specification, Development and Verification*, Advances in Computing Science, pages 136–149, Wien. Springer-Verlag. ISBN: 3-211-83282-3.

Kahl, W. and Derichsweiler, F. (2001). Declarative Term Graph Attribution for Program Generation. *J. UCS*, 7(1):54–70.

Karjoth, G., Lange, D., and Oshima, M. (1997). A security model for aglets. *IEEE Internet Computing*, 1(4):68–77.

Kelsey, R. and Hudak, P. (1989). Realistic compilation by program transformation — detailed summary. Pages 281–292.

Klop, J. W. (1980). Combinatory reduction systems. Mathematical Centre Tracts 127, Centre for Mathematics and Computer Science, Amsterdam. PhD thesis.

Klusch, M., editor (1999). *Intelligent Information Agents: Agent-Based Information Discovery and Management on the Internet*. Springer-Verlag.

Knabe, F. C. (1995). *Language Support for Mobile Agents*. Phd thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh.

Kotz, D. and Gray, R. S. (1999). Mobile agents and the future of the internet. *Operating Systems Review*, 33(3):7–13.

Kotz, D., Gray, R. S., and Rus, D. (2002). Future Directions for Mobile-Agent Research. Technical Report TR2002-415, Dartmouth College, Computer Science, Hanover, NH.

Kotz, D. and Mattern, F., editors (2000). *Agent Systems, Mobile Agents, and Applications*, volume 1882 of *Lect. Notes in Comp. Sci.* Springer-Verlag. Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agants, ASA/MA 2000, Zurich, Switzerland.

Lange, D. B. and Oshima, M. (1999). Seven good reasons for mobile agents. *Commun. ACM*, 42(3):88–89.

Launchbury, J. and Jones, S. L. P. (1994). Lazy functional state threads. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–35.

Leijen, D. (2000). Parsec, a fast combinator parser.

Lucco, S., Sharp, O., and Wahbe, R. (1995). Omniware: A universal substrate for web programming. In *Fourth International World Wide Web Conference*.

Milner, R., Parrow, J., and Walker, D. (1992). A calculus of mobile processes, part I/II. *J. Inform. Comput.*, 100:1–77.

Milojicic, D. S., Guday, S., and Wheeler, R. (1997). Old Wine in New Bottles, Applying OS Process Migration Technology to Mobile Agents. In *Proceedings of the 3rd Workshop on Mobile Object Systems, 11th European Conference on Object-Oriented Programming*, Jyväskylä, Finland.

Morbach, H.-P. (2001). *Sichere Agenten und Agentenplattform in Haskell*. Studienarbeit, Fakultät für Informatik, Universität der Bundeswehr München.

Necula, G. C. and Lee, P. (1997). Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris.

Partsch, H. A. (1990). *Specification and Transformation of Programs. A Formal Approach to Software Development*. Springer-Verlag.

Peine, H. and Stolpmann, T. (1997). The architecture of the Ara platform for mobile agents. In Popescu-Zeletin, R. and Rothermel, K., editors, *First International Workshop on Mobile Agents MA'97*, volume 1219 of *Lecture Notes in Computer Science*, pages 50–61, Berlin, Germany. Springer Verlag.

Peyton Jones, S. L. (1993). The Glasgow Haskell compiler: A technical overview.

Peyton Jones, S. L. et al. (2002). Haskell 98 language and libraries. See also `http://haskell.org/`.

Picco, G. P., editor (2001). *Mobile Agents*, volume 2240 of *Lect. Notes in Comp. Sci.* Springer-Verlag. 5th International Conference, MA 2001 Atlanta, GA, USA.

Pierre, S. and Glitho, R., editors (2001). *Mobile Agents for Telecommunication Applications*, volume 2164 of *Lect. Notes in Comp. Sci.* Springer-Verlag. Third International Workshop, MATA 2001, Montreal, Canada.

Posegga, J. and Karjoth, G. (2000). Mobile Agents and Telcos Nightmares. In *Annales des Telecommunications*, volume 55, pages 388–400.

Poslad, S., Buckle, P., and Hadingham, R. (2000). The FIPA-OS Agent Platform: Open Source for Open Standards. See also `http://fipa-os.sourceforge.net/`.

Postel, J. B. (1982). *RFC 821: Simple Mail Transfer Protocol*. Network Working Group.

Robinson, D. and Coar, K. (2003). *Internet Draft: The Common Gateway Interface (CGI) Version 1.1*.

Rozenberg, G., editor (1997). *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*. World Scientific, Singapore.

Ryan, P. Y. A. (2000). Mathematical Models of Computer Security. In Focardi, R. and Gorrieri, R., editors, *Foundations of Security Analysis and Design*, volume 2171 of *Lect. Notes in Comp. Sci.* Springer-Verlag. FOSAD 2000, Bertinoro, Italy.

Sangiorgi, D. (1992). *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis CST–99–93, Department of Computer Science, University of Edinburgh.

Sanneck, H., Berger, M., and Bauer, B. (2002). Application of agent technology to next generation wireless/mobile networks.

Serugendo, G. D. M., Muhugusa, M., and Tschudin, C. (1998). A survey of theories for mobile agents. *World Wide Web Journal, special issue on Distributed World Wide Web Processing: Applications and Techniques of Web Agents*.

Sheard, T. and Peyton Jones, S. (2002). Template metaprogramming for Haskell. In Chakravarty, M. M. T., editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press.

Spivey, J. M. (1992). *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition. (1st edition, 1989).

Swierstra, S. D. and Duponcheel, L. (1996). Deterministic, error-correcting combinator parsers. In Launchbury, J., Meijer, E., and Sheard, T., editors, *Second International Summer School on Advanced Functional Programming*, volume 1126 of *Lect. Notes in Comp. Sci.*, pages 184–207.

Syverson, P. and Cervesato, I. (2000). The Logic of Authentication Protocols. In Focardi, R. and Gorrieri, R., editors, *Foundations of Security Analysis and Design*, volume 2171 of *Lect. Notes in Comp. Sci.* Springer-Verlag. FOSAD 2000, Bertinoro, Italy.

Taha, W., editor (2001). *Semantics, Applications, and Implementation of Program Generation*, volume 2196 of *Lect. Notes in Comp. Sci.* Springer-Verlag. Second International Workshop, SAIG 2001, Florence, Italy.

Thati, P., Chang, P.-H., and Agha, G. (2001). Crawlets: Agents for High Performance Web Search Engines. In (Picco, 2001), pages 119–134. 5th International Conference, MA 2001 Atlanta, GA, USA.

Thomsen, B., Leth, L., Prasad, S., Kuo, T.-M., Kramer, A., Knabe, F., and Giacalone, A. (1993). Facile Antigua release programming guide. Technical Report ECRC-93-20, European Computer-Industry Research Centre, Munich, Germany.

Thomsen, L. L. and Thomsen, B. (1997). Mobile agents – the new paradigm in computing. *ICL Technical Journal*, (12):14–40.

Tripathi, A., Karnik, N. M., Vora, M., Singh, R. D., Ahmed, T., Eberhard, J., and Prakash, A. (1999). Development of Mobile Agent Applications with Ajanta. Technical report, Department of Computer Science, University of Minnesota. Available at http://www.cs.umn.edu/Ajanta.

Tripathi, A. R., Karnik, N. M., Ahmed, T., Singh, R. D., Prakash, A., Kakani, V., Vora, M. K., and Pathak, M. (2002). Design of the Ajanta System for Mobile Agent Programming. *Journal of Systems and Software*, 62(2):123–140.

Venners, B. (1997). Solve real problems with aglets, a type of mobile agent.

Wallace, M. (1995). *Functional Programming and Embedded Systems*. PhD thesis, Dept. Of Computer Science, University of York, UK.

Wallace, M. and Runciman, C. (1994). Type-checked message-passing between functional processes. In Hammond, T. and Sansom, editors, *GLA*, pages 245–254. Springer-Verlag.

White, J. (1994). Mobile agents white paper.

Wooldridge, M. and Jennings, N. R. (1994). Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2).

Zhong, Q. and Edwards, N. (1998). Security in the large: Is java's sandbox scalable. Technical report, HP Laboratories Bristol.

Zierer, H., Schmidt, G., and Berghammer, R. (1986). An interactive graphical manipulation system for higher objects based on relational algebra. In Tinhofer, G. and Schmidt, G., editors, *WG '86*, volume 246 of *Lect. Notes in Comp. Sci.*, pages 68–81, Bernried, Starnberger See. Springer.

Glossary

BVARo	Bindable Variable on Object Layer
DAG	Directed Acyclic Graph
DSL	Domain-Specific Language
HaMAP	Haskell Mobile Agent Platform
HOPS	Higher Object Programming System
I-DSL	Internal Domain-Specific Language
MVARo	Metavariable on Object Layer
MVARt	Metavariable on Type Layer
PPF	Possibly-Provided Function
SFA	Stateful Agent
SLA	Stateless Agent
UI-DSL	User Interface Domain-Specific Language

Index