

University of London
Imperial College of Science, Technology and Medicine
Department of Computing

Glint:
Growing an Object Oriented language in Ambients
Outsourcing Report

Matthew Sackman
Supervisor: Susan Eisenbach

Contents

1	Introduction	5
2	Ambient Calculi	7
2.1	Mobile Ambients	7
2.1.1	Communication	8
2.1.2	Objective and Subjective moves	8
2.1.3	Objective <code>open</code>	9
2.2	Boxed Ambients	9
2.3	Communication Interference	11
2.4	Typing Ambients	12
2.4.1	Typing Mobile Ambients	12
2.4.1.1	Exchange Types	12
2.4.1.2	Opening Control	13
2.4.1.3	Crossing Control	13
2.4.2	Typing Boxed Ambients	14
2.4.2.1	Typing process communication	14
2.4.2.2	Typing ambient mobility	14
2.5	The Channel Ambient System	15
2.6	Conclusion	16
3	Object Oriented Language Features	17
3.1	Named Parameters	17
3.2	Pattern Matching	17
3.2.1	Extensions of Pattern Matching	18
3.2.2	Predicate Dispatch	18
3.3	Currying	19
3.4	Mixins	19
3.5	Traits	20
3.6	Polymorphism	20
3.6.1	Parameterised Polymorphism	20
3.6.2	Virtual Types Polymorphism	21
3.6.3	<i>MyType</i>	22
3.6.4	Family Polymorphism	22
3.7	Conclusion	23
4	Specification	25
4.1	The Language	25
4.2	Runtime	25
4.3	Tools	26
4.4	Testing	26
4.4.1	Ambient behaviour	26
4.4.2	Language behaviour	27
4.4.3	Runtime and Tools behaviour	27
5	Evaluation	29
	Bibliography	32

A Program listings	33
A.1 JAVA based simple calculator	33
A.2 JAVA Extension of simple calculator	35
A.3 <i>MyType</i> based simple calculator	35
A.4 Extension of <i>MyType</i> virtual type based calculator	37
A.5 Family polymorphism based simple calculator	38
A.6 Extension of family polymorphism based calculator	40

Chapter 1

Introduction

As computers become ever more ubiquitous, new ways of utilising them become possible. In particular, the growth in mobile computers, for example, laptops, PDAs, and mobile phones, has brought to the forefront models concerning the movement of processes from one computer to the next.

Not only is there the potential for these models to be applied to new applications and problems, but there are also many existing problems which have had a number of partial solutions engineered, none of which have been completely successful. The most prominent of these, at least to my mind, is the issue of being able to log into a computer on which you have never previously worked and to be able to access the desktop environment you are used to, instead of being presented with a new, uncustomised and unconfigured desktop.

Microsoft, amongst others, have attempted a couple of notable solutions to this:

- **Roaming Profiles.** This allows for a central server to send to the computer on which you're now working, your documents, desktop and some configuration. The problem comes when you have machines that have different sets of programs installed on them: it can not migrate actual programs from one machine to another, much less running instances of those programs.
- **Remote Desktop.** This allows you to connect to another machine and to be able to manipulate and use that other machine as if you're sitting in front of it. This is known as *thin client* in that the machine in front of which you are sitting is doing no real work at all other than relaying, to the machine to which you are connected, keyboard and mouse inputs and displaying the resulting desktop and programs. The problem with this is that firstly it's very dependant on having a reliable and fast network connection between the machines and secondly, the client machine is not doing any of the work: all the work is sent to the remote machine and this can become a bottleneck. In fact, there is no migration going on whatsoever with this system.

Models of mobile computation allow for a solution to this in that it would be possible, given the right environment and tools, to migrate your processes from a server or other store, to the machine on which you're working. You would receive the running instance of the program: for example, with a web browser you would be able to migrate it and to see the same web pages that you were last looking at. A word processor would migrate with the same documents open, the cursor in the same position; in short, state is maintained. The tools and environment necessary to achieve solutions to these kinds of problems are the focus of this project.

More fine-grained control would also be desirable. Consider being on a train with your laptop. The train doesn't have a network connection, but before you left home, you migrated the word processor and the documents you're working on onto the laptop. During your train journey, you finish writing the document and want to print it, so you press print. Because you don't have a network connection, nothing happens initially. However, when you next get a network connection you re-establish a connection with the ambients still at your home and the print job is sent to your printer at home: when you migrated the word processor you purposefully chose not to migrate its notion of where your printer is, so the processes that deal with printing are still running on your computer at home, whilst the rest of the word processor was migrated onto your laptop.

This example illustrates how fine-grained combinations of mobility and thin-client computing can be combined. However, no programming languages currently exist in which mobility of processes is a prime focus. JAVA, amongst other languages, provides the ability to download and run code from a remote location and it provides the ability to serialise and de-serialise object graphs allowing for the

transfer of state. This, however is a very involved process and there is no framework which comes with the language (or any other language) which allows for the user to objectively move a process from one computer to another.

Two languages which have been designed to deal with distribution and mobility are ALICE¹ and ACUTE². ALICE extends ML with capabilities to allow for better control over distributed and mobile computing but leaves the overhead of dealing with resolving references to non-local data to the programmer, creating almost insurmountable difficulties for the programmer. ACUTE is another extension of ML which has support for explicit marshalling and unmarshalling of data structures and functions. However, in a similar manner to ALICE, no distributed references are supported within the language, meaning that references to local values within data structures or functions that are to be marshalled must be either copied to the destination runtime or are already known to exist at the destination.

Calculi have been designed that seem to provide greater expressivity than existing programming languages for dealing with mobility of processes. The Mobile Ambient calculus, along with variants thereof, model processes which are enclosed in *ambients*. These can be instructed to move into and out of other ambients. This allows for the modelling of ambients (and the processes they enclose) moving between computing nodes. However, these models are only currently in the form of calculi which only address mobility along with some security issues. To date there are no full-scale programming languages in which ambient calculi are incorporated. The aim of this project is the production of an object oriented programming language which incorporates and builds upon the ambient calculi to allow for the evaluation of the utility of these models of mobility in full-scale computer applications.

¹<http://www.ps.uni-sb.de/alice/>

²<http://www.cl.cam.ac.uk/users/pes20/acute/>

Chapter 2

Ambient Calculi

Combining ambient calculi and object oriented language features is akin to combining four wheels and an engine into a car: the details of the fusion of ideas determines the quality of the result. There are several variants of ambient calculi, each successive calculi attempting to address perceived difficulties and weaknesses in the last, sometimes at the cost of expressiveness. What features and what level of expression would make programmers' lives easier?

In this chapter I investigate the various ambient calculi that will influence the design of my language.

2.1 Mobile Ambients

Mobile Ambients is the first of the ambient calculi and was originally proposed in [10]. The calculus is simple but powerful: an ambient is a container for any number of processes and sub-ambients and is named. Restriction rules and scoping issues are as per CCS or π -calculus and the definitions of free and bound names and variables are again similar to CCS and π -calculus. Communication is in the style of π -calculus, is asynchronous on output but is no longer on named channels. Instead a nameless implicit channel is defined within every ambient and only processes directly within the ambient can communicate with one another on this channel.

What Mobile Ambients add, above the definition of the ambients themselves are capabilities. These allow a process to move its containing ambient into a sibling ambient, out of its parent ambient or to open a child ambient. $n[P] | m[in\ n.\ Q] \rightarrow n[P | m[Q]]$ shows how the in capability works: the ambient enclosing the $in\ n.\ Q$ process, m , is transported inside the ambient n . This can only happen when such an ambient as named in the capability exists and is a sibling: if the ambient n didn't exist or was at another location then the $in\ n.\ Q$ process would block until a point at which the capability could execute.

The out capability is equally straight-forward: $n[P | m[out\ n.\ Q]] \rightarrow n[P] | m[Q]$ shows how the out capability moves the process's enclosing ambient (m) out of its own enclosing ambient (n), provided the ambient named in the capability does in fact enclose the process's own ambient. Again, were this not the case, the process would block.

Finally, the open ambient dissolves the named process. $n[open\ m.\ P | m[Q]] \rightarrow n[P | Q]$ shows how the opening of the m ambient has caused all of the processes that resided within m to become processes within n itself and the m ambient no longer exists.

Example 2.1 shows all of these capabilities in action, along with communication between processes.

Example 2.1 Mobile Ambients Example

$$\begin{aligned} & n[in\ m.\ open\ l.\ (x).\ x | Q] | m[l[in\ n.\ \langle out\ m \rangle] | R] \\ \rightarrow & m[n[open\ l.\ (x).\ x | Q] | l[in\ n.\ \langle out\ m \rangle] | R] \\ \rightarrow & m[n[open\ l.\ (x).\ x | Q] | l[\langle out\ m \rangle] | R] \\ \rightarrow & m[n[(x).\ x | Q] | \langle out\ m \rangle] | R] \\ \rightarrow & m[n[out\ m | Q] | R] \\ \rightarrow & n[Q] | m[R] \end{aligned}$$

Because of the manner in which capabilities are defined, it is syntactically valid to construct ambients named with a capability, for example, in $m[P]$. This is clearly nonsensical and it is left to a type system to prevent such ambient names from being constructed.

2.1.1 Communication

In Mobile Ambients, communication takes place on anonymous channels, one per ambient. The content of communication is defined as either names of ambients or capabilities acting on an ambient. It is assumed that if you have received a capability to act on an ambient then from that you can not recover the name of the ambient itself: if this were not the case then upon receiving any capability you could recover the name of the ambient on which it acts and form any new capability to act on that ambient.

Because of the fact that communication can only occur between sibling processes within an ambient, communication with other ambients is slightly more involved. There are two choices: either one of the ambients is opened, or a new ambient is issued from within one ambient that moves inside the other, gets opened by the other and then returns. Whilst clearly more complex, this second form tends to be preferred because it doesn't result in opening either of the ambients: instead, only a small controlled ambient, constructed specifically for such a purpose is opened. Nevertheless, this relies on the receiver of the ambient trusting the contents of the received ambient in some way such that it's happy to open it. Example 2.2 shows both these strategies in use.

Example 2.2 Inter-ambient communication strategies

(a) Opening the ambient with whom to communicate

$$\begin{aligned} & n[\text{open } m.(x).P] \mid m[\text{in } n.\langle j \rangle \mid Q] \\ \rightarrow & n[\text{open } m.(x).P \mid m[\langle j \rangle \mid Q]] \\ \rightarrow & n[(x).P \mid \langle j \rangle \mid Q] \\ \rightarrow & n[P\{j/x\} \mid Q] \end{aligned}$$

(b) Issuing a new ambient to perform the communication

$$\begin{aligned} & n[l_1[\text{out } n.\text{in } m.(y).l_2[\text{out } m.\text{in } n.\langle y \rangle]] \\ & \quad \mid \text{open } l_2.(x).P] \mid m[\text{open } l_1.\langle j \rangle \mid Q] \\ \rightarrow & l_1[\text{in } m.(y).l_2[\text{out } m.\text{in } n.\langle y \rangle]] \\ & \quad \mid n[\text{open } l_2.(x).P] \mid m[\text{open } l_1.\langle j \rangle \mid Q] \\ \rightarrow & n[\text{open } l_2.(x).P] \\ & \quad \mid m[l_1[(y).l_2[\text{out } m.\text{in } n.\langle y \rangle]] \\ & \quad \quad \mid \text{open } l_1.\langle j \rangle \mid Q] \\ \rightarrow & n[\text{open } l_2.(x).P] \\ & \quad \mid m[(y).l_2[\text{out } m.\text{in } n.\langle y \rangle] \mid \langle j \rangle \mid Q] \\ \rightarrow & n[\text{open } l_2.(x).P] \\ & \quad \mid m[l_2[\text{out } m.\text{in } n.\langle j \rangle] \mid Q] \\ \rightarrow & n[\text{open } l_2.(x).P] \mid l_2[\text{in } n.\langle j \rangle] \mid m[Q] \\ \rightarrow & n[l_2[\langle j \rangle] \mid \text{open } l_2.(x).P] \mid m[Q] \\ \rightarrow & n[\langle j \rangle \mid (x).P] \mid m[Q] \\ \rightarrow & n[P\{j/x\}] \mid m[Q] \end{aligned}$$

Obviously, issuing a new ambient leads to more complex code. However, note that in issuing the new ambient, the process Q that originally started in ambient m remains there, and does not migrate into n as is the case with the simpler strategy. Also note that in example 2.2b, when m receives the ambient l_1 it implicitly trusts it when it opens it: there is nothing to stop l_1 from containing processes that monitor and interfere with the proper operation of m .

2.1.2 Objective and Subjective moves

The capabilities in m and $\text{out } m$ are *subjective* rules in that they cause the ambient enclosing the process to move rather than causing an external ambient to move. Such movement would be called *objective* and there is some discussion as to how to best encode this. [10] makes the point that it should be considered who has the authority to move whom. If a malicious party were to compromise a system, what damage could they wreak? With subjective moves, they can move their own ambient anywhere, but they can not

open or move any pre-existing ambients. With objective moves however, they could. As such, objective moves are considered unsafe if they existed as primitives within the calculus.

However, a safer version of objective moves can be encoded via subjective moves. It is treated as safer as it depends on a naming convention, in fact, the very same convention-structure that was used in example 2.2b. There, it was required to move a process out of the ambient n and into the ambient m . As discussed above, this was achieved through issuing a new ambient which was moved into the target ambient and then the target ambient opening the newly arrived ambient. What's more, through restriction, this process can be made completely private as example 2.3 shows: the use of restriction makes private the name of the *messenger* ambient l meaning that no hostile external ambient can enter m and be opened by it as it won't have the correct name.

Example 2.3 A safe encoding of objective moves using restriction

$$\begin{aligned}
& (\nu v)(n[v \text{ out } n . \text{in } m . P] \mid Q] \mid m[\text{open } v]) \\
\rightarrow & (\nu v)(n[Q] \mid v[\text{in } m . P] \mid m[\text{open } v]) \\
\rightarrow & (\nu v)(n[Q] \mid m[\text{open } v \mid v[P]]) \\
\rightarrow & (\nu v)(n[Q] \mid m[P])
\end{aligned}$$

2.1.3 Objective open

The open capability is obviously essential for any level of expression within the calculus: without it, the hierarchy of ambients could move and grow, but could never shrink and worse, no communication could ever take place between processes that didn't start off as siblings within the same ambient.

Given the arguments made for the use of subjective moves, it may appear surprising that open is objective. Again, the reason lies in security considerations: if open was subjective, for example: $n[P \mid m[\text{open} . Q]] \rightarrow n[P \mid Q]$, then it would be possible for any ambient to move itself inside any other ambient and then open itself, gaining access to the processes in the parent ambient. Making open objective prevents this and gives the parent ambient control over what it is and isn't prepared to open, allowing the parent ambient to perform, for example, typing checks prior to opening any ambient.

The slightly complex behaviour of the open capability leads to the need for some care when building type systems around mobile ambients. This is discussed in section 2.4.1 and is one of the chief motivations for dropping the open capability.

2.2 Boxed Ambients

In [7], Boxed Ambients are presented. The semantics of the in and out capabilities, the structure of ambients, the rules for free and bound names and processes are all as for Mobile Ambients. However, the open capability has been dropped completely and in its place are four new synchronous communication primitives that can allow communication across ambient boundaries. These allow for input from a named child ambient: $(x)^n$, input from the parent ambient: $(x)^l$, output to a named child ambient: $\langle M \rangle^n$ and output to the parent ambient: $\langle M \rangle^l$. These communication primitives provide functionality that is sufficient to replace the open primitive, but their behaviour is simpler making subsequent typing systems easier to reason about and implement.

The communication primitives are not required to match in any kind of action/co-action manner: indeed for communication across ambient boundaries, only one of the sending and receiving processes need declare the cross-ambient nature of the communication. This makes it possible to send or receive to or from any process or child ambient. The reduction rules are defined as in figure 2.1.

Also addressed is the issue of named communication channels: it is shown how to encode channels in the style of the π -calculus trivially by using replicated ambients as channels and then translating π -calculus input and output operations into corresponding operations on named ambients as shown in example 2.4. Note however that this is not suitable for a general encoding as the channel ambient c must be visible to both the receiver and sender, in particular, must be a child ambient of the ambients containing the sending and receiving processes. Thus some movement must be achieved in more complex situations.

Figure 2.1 Communication reduction rules for Boxed Ambients

(local)	$(x). P \mid \langle M \rangle . Q \rightarrow P\{M/x\} \mid Q$
(input n)	$(x)^n . P \mid n[\langle M \rangle . Q \mid R] \rightarrow P\{M/x\} \mid n[Q \mid R]$
(input \uparrow)	$\langle M \rangle . P \mid n[(x)^\uparrow . Q \mid R] \rightarrow P \mid n[Q\{M/x\} \mid R]$
(output n)	$\langle M \rangle^n . P \mid n[(x) . Q \mid R] \rightarrow P \mid n[Q\{M/x\} \mid R]$
(output \uparrow)	$(x) . P \mid n[\langle M \rangle^\uparrow . Q \mid R] \rightarrow P\{M/x\} \mid n[Q \mid R]$

Example 2.4 Encoding of π -calculus communication primitives in Boxed Ambients

$$\begin{aligned}
\langle\langle \bar{c}\langle x \rangle \mid c(y) . Q \rangle\rangle &\Rightarrow !c[!(z) . \langle z \rangle \mid \langle x \rangle^c \mid (y)^c . Q \\
&\rightarrow !c[!(z) . \langle z \rangle \mid c[!(z) . \langle z \rangle \mid \langle x \rangle^c \mid (y)^c . Q \\
&\rightarrow !c[!(z) . \langle z \rangle \mid c[!(z) . \langle z \rangle \mid \langle x \rangle \mid (y)^c . Q \\
&\rightarrow !c[!(z) . \langle z \rangle \mid c[!(z) . \langle z \rangle \mid Q\{x/y\} \\
&\rightarrow !c[!(z) . \langle z \rangle \mid Q\{x/y\} \qquad \text{(By structural congruence)}
\end{aligned}$$

The Boxed Ambient is in spirit quite similar to the Seal calculus [33] in which the notation $c^n\langle M \rangle$ indicates a request to write M to channel c in ambient (or seal) n . The seal calculus can be encoded in Boxed Ambients quite simply.

Throughout [7], the example of a process running on a host h downloading an applet a is used as motivation against opening. In particular: $a[\text{in } h . Q] \mid h[P]$. This clearly reduces to $h[a[Q] \mid P]$ which shows the result of downloading the applet a . For communication between P and Q to occur, the authors argue that it is necessary for either the ambient a to be dissolved, thus exposing h to any process that resided in a , or for P (or a clone of P) to enter the applet ambient a , again exposing h to potentially harmful processes as the process P could become compromised.

What isn't considered is a third possibility which most closely resembles the host providing APIs to the applet: the host is responsible for issuing ambients that themselves move inside the applet and are opened by the applet. As shown in example 2.3, it is entirely possible for the ambients issued by the host to share a secret with the host in the form of a restricted name which will serve to guarantee that any communication with returning ambients is private. This method is shown in example 2.5. It is obviously desirable to design the *api* ambient in such a way that no level of interaction with the applet can reveal to the applet the shared secret, the name r .

Example 2.5 Providing APIs to a downloaded applet in a safe manner using just Mobile Ambients

$$\begin{aligned}
&h[a[\text{open } \text{api} . \langle \text{myRequest} \rangle \mid Q] \mid (\nu r)(\text{open } r . (x) . R \mid \text{api}[\text{in } a . (\text{request}) . r[\text{out } a . \langle \text{request} \rangle]]] \\
&\rightarrow h[(\nu r)(a[\text{open } \text{api} . \langle \text{myRequest} \rangle \mid Q \mid \text{api}[(\text{request}) . r[\text{out } a . \langle \text{request} \rangle]]] \mid \text{open } r . (x) . R)] \\
&\rightarrow h[(\nu r)(a[\langle \text{myRequest} \rangle \mid Q \mid (\text{request}) . r[\text{out } a . \langle \text{request} \rangle]] \mid \text{open } r . (x) . R)] \\
&\rightarrow h[(\nu r)(a[Q \mid r[\text{out } a . \langle \text{myRequest} \rangle]] \mid \text{open } r . (x) . R)] \\
&\rightarrow h[a[Q] \mid (\nu r)(\text{open } r . (x) . R \mid r[\langle \text{myRequest} \rangle])] \\
&\rightarrow h[a[Q] \mid (\nu r)((x) . R \mid \langle \text{myRequest} \rangle)] \\
&\rightarrow h[a[Q] \mid (\nu r)(R\{\text{myRequest}/x\})]
\end{aligned}$$

This method seems the most intuitive manner in which to interact with the downloaded applet: providing resources to the applet via APIs which the applet accesses from within its own ambient. The lack of consideration of this approach is unfortunate as it provides the level of protection the authors seek without needing to replace the open capability. Nevertheless, this method is more complex and is

harder to type correctly. Replacing the open capability with more advanced communication capabilities certainly makes understanding and typing communication with the downloaded applet far simpler.

2.3 Communication Interference

In [21], the point is raised that whilst non-determinism can be created in Mobile Ambients purposefully (mimicking CCS's + operator), it can also arise unintentionally. Indeed, given

$$k[n[\text{in } m . P \mid \text{out } k . R] \mid m[Q]]$$

the nondeterminism is almost certainly unwanted and indeed the authors treat it as a programming error. The authors classify a set of these such non-deterministic situations they call grave interferences. They go on to develop a variant of Mobile Ambients called Safe Ambients in which the capabilities can only succeed if there exists a co-action. So $n[\text{in } m . P] \mid m[Q]$ can no longer progress, but $n[\text{in } m . P] \mid m[\bar{\text{in}} m \mid Q]$ can. Note that the name in the co-action refers to the ambient which is being crossed rather than the ambient that is moving. A type system is then used to enforce a notion of *single-threadedness* in which at any time, ambients can only engage in a maximum of one activity across boundaries.

Not only are these situations also possible in Boxed Ambients, but they are in fact more complex because an input can potentially receive from an output in the current ambient, in the parent ambient or in a child ambient, and dually for an output. Hence a similar grave interference can be constructed for Boxed Ambients:

$$m[(x)^n . P \mid n[\langle M \rangle \mid (x) . Q \mid k[(x)^\dagger . R]]]$$

In [8] these issues are addressed for boxed ambients: input and output now must occur with co-actions as figure 2.2 shows. It is then necessary to be able to learn the name of any ambient that enters an ambient so that the parent ambient is able to communicate with the new child. This is achieved through renaming the in capability to enter and only allowing enter to reduce when a co-action is present. The subsequent adjustment is that the ambient wishing to enter has its name provided and bound in the co-action by the runtime system. The same alteration is used for out, creating an exit capability in which the ambient leaving must now unregister itself with its parent before departing. These rules are then further modified by the addition of a key parameter which must match in the action and co-action and provides the function of a password in a similar manner to the SAP calculus [23].

Figure 2.2 Reduction rules for communication and mobility in the NBA calculus

(local)	$(x) . P \mid \langle M \rangle . Q \rightarrow P\{M/x\} \mid Q$
(input)	$(x)^n . P \mid n[\langle M \rangle^\wedge . Q \mid R] \rightarrow P\{M/x\} \mid n[Q \mid R]$
(output)	$\langle M \rangle^n . P \mid n[(x)^\wedge . Q \mid R] \rightarrow P \mid n[Q\{M/x\} \mid R]$
(enter)	$n[\text{enter}\langle m, k \rangle . P_1 \mid P_2] \mid m[\overline{\text{enter}}(x, k) . Q_1 \mid Q_2] \rightarrow m[n[P_1 \mid P_2] \mid Q_1\{n/x\} \mid Q_2]$
(exit)	$n[m[\text{exit}\langle n, k \rangle . P_1 \mid P_2] \mid Q] \mid \overline{\text{exit}}(x, k) . R \rightarrow m[P_1 \mid P_2] \mid n[Q] \mid R\{m/x\}$

These rules, when combined with the presented type system, eliminate communication interference in Boxed Ambients without limiting expressiveness.¹ It is also interesting to note that the refinement of Boxed Ambients into NBA leads to a simpler type system than presented for Boxed Ambients in [7]. This is due to the fact that because now all communication actions happen with co-actions, within ambients, only one side of communication need be tracked and in the paper, this is the upward communication to the parent ambient.

While communication interferences are effectively eliminated, capability based non-determinism is still very much present. It is entirely possible to write in NBA:

$$k[n[\text{enter}(m, a) . P \mid \text{exit}(k, a) . R] \mid m[\overline{\text{enter}}(n, a) . Q]] \mid \overline{\text{exit}}(n, a)$$

Of course, in this form, the presence of the co-capabilities makes it rather more likely that this was intentional and without either of the co-capabilities, the non-determinism is avoided.

¹Other than loss of expression relating directly to the elimination of grave interferences in communication. Indeed, the paper [8] demonstrates how augmenting NBA with a guarded form of non-determinism in the same manner as CCS (+) specifically applied to communication results in a safe recovery of the loss of expression.

2.4 Typing Ambients

There are several aspects of ambients that can be typed and for which type systems have been proposed. The most common type systems are based around source code annotations giving types to messages. These types are called exchange types. There are other type systems for ambients including the ability to type mobility. This allows the type system to verify that when ambients move, they move to locations in which they are allowed and is more powerful than simply typing message exchanges.

All of these type systems are based on the same set of ideas. Firstly a type is simply a group or set which has elements. These are, for example the names of ambients (note that it is the *names* of the ambients rather than the ambients themselves that are members of the groups). The obvious analogy with object oriented languages is that the group is a type and the elements of the group are instances of that type. Furthermore, these groups tend to be parameterised with other types in a manner similar to parameterised polymorphism. The parameters represent for example the types of messages that may be exchanged within ambients named by the names in the group: a group exists, collecting together a set of names. Each name names ambients which all exchange the same type of messages thus the group is parameterised with that type.

There is always an exchange type called *Shh* which represents the absence of exchanges. So for example, the names of ambients in which no exchanges occur are members of a group G , parameterised by the exchange type *Shh*, or $G[\textit{Shh}]$.

The typing of message exchanges often allows for a very simple subtype relation in which the silent exchange type, *Shh*, is a subtype of all other exchange types. This means that an ambient that makes no exchanges is allowed anywhere on the grounds that the processes it contains will never interact with any other processes.

2.4.1 Typing Mobile Ambients

In [9] a series of type systems are proposed, covering typing of messages and ambient mobility. The type systems are presented in stages, initially giving types to messages, processes (including capabilities) and ambients and controlling message exchanges. This is then extended to regulate the opening of ambients and finally crossing control is presented in which the type system is further extended to allow for ambients to regulate which ambients they may cross: this controls the in and out capabilities.

There are a couple of minor modifications made to the Mobile Ambient calculus presented in [10] in this paper. Firstly input and output are extended to polyadic communication and secondly a simple objective move definition becomes a primitive operation. This however can be safely encoded in the basic calculus:

$$\langle\langle \textit{go } N . M [P] \rangle\rangle \Rightarrow (\nu k)(k [N . M [\textit{out } k . P]])$$

2.4.1.1 Exchange Types

There are some subtleties to exchange types. Whilst the typing of parallel processes, restriction, input and output follow in a straight-forward manner, the null (0) process can be given any type. This also applies to any child ambient, the justification for this is that the child ambient can not partake in exchanges in the current ambient as it is a child, so it can be given any type. Similarly, the capabilities in and out can be given any type because they have no bearing on exchanges. This however, contrasts with the typing of ambient *names* which must match the types of the processes enclosed by the ambients named by the name in question.

The open capability unleashes the processes within a child ambient and so must be given special attention. Firstly, the type of the open n capability must match the type of the ambient named in the capability that it is opening and secondly, the type of any continuation of the open n process must be prepared to participate in any exchanges that the unleashed processes may perform, thus must also match the type of the ambient opened. This explains why the ambient names have more restrictive types than the ambients named by those names: the names are used in capabilities and so have a bearing on the type of the current ambient whereas a child ambient itself has no such bearing.

In example 2.6, a possible typing derivation of the process $n [(x : W) . P \mid \textit{open } m . 0] \mid m [Q \mid \textit{in } n . 0]$ is shown. Note that the two ambient names (n and m) turn out to have the same type. This is predictable because the processes P and Q will eventually become part of the same ambient and so will interact with each other. As such, they should have the same type.

Example 2.6 A typing derivation of the process $n[(x : W). P \mid \text{open } m . 0] \mid m[Q \mid \text{in } n . 0]$

$$\begin{array}{l}
n[(x : W). P \mid \text{open } m . 0] \mid m[Q \mid \text{in } n . 0] \\
\quad Q : T \\
\quad \text{in } n : \text{Cap}[T] \\
\quad \text{in } n . 0 : T \\
\quad \quad m : G[T] \\
\quad m[Q \mid \text{in } n . 0] : S \\
\quad \quad \text{open } m : \text{Cap}[T] \\
\quad \quad \text{open } m . 0 : T \\
\quad \quad \quad P : T \\
\quad \quad \quad (x : W). P : T (T \equiv W) \\
\quad \quad \quad n : G[T] \\
\quad n[(x : W). P \mid \text{open } m . 0] : S \\
n[(x : W). P \mid \text{open } m . 0] \mid m[Q \mid \text{in } n . 0] : S
\end{array}$$

2.4.1.2 Opening Control

Opening control adds another parameter to the types of ambients so ambient types are now represented by $G[\circ\{G_1, \dots, G_k\}, T]$. A member of this group G is a name which names ambients within which processes may exchange messages of type T as before and now may only open ambients named within the set $\bigcup_{i=1}^k G_i$. This modification to ambient types similarly affects both capability types and process types which both gain the extra parameter. Thus exercising a capability of type $\text{Cap}[\circ\{G_1, \dots, G_k\}, T]$ may unleash exchanges of type T as before and may now result in the subsequent opening of ambients of a name within $\bigcup_{i=1}^k G_i$. The modification to the types of processes can be interpreted similarly.

Due to complexities in subject reduction with opening control there is an additional requirement on the typing rule for the open capability: the ambient name named in the capability must belong to a group which itself is a member of the set of groups that parameterise the ambient type, representing the names of ambients that may be opened by it. If this were not the case then subject reduction fails without additional subtyping rules, complicating the type system. The upshot of this additional requirement is that if the name of an ambient is a member of the group $G[\circ\mathbf{H}, T]$ and is also a member of the set \mathbf{H} then it can mean two things. Firstly, as already explained, it can mean that opening an ambient of this name may subsequently open ambients named with names within \mathbf{H} or, it may simply mean that the ambient name names ambients that are openable.

This side effect is unnecessary and is only included in order to avoid greater complexity elsewhere in the type system.

2.4.1.3 Crossing Control

Crossing of an ambient m occurs when another ambient enters it via an $\text{in } m$ capability or exits it via an $\text{out } m$ capability. The type system for crossing control treats objective and subjective moves explicitly and this has a bearing on how types are further modified. The type of an ambient name is represented by $G \wedge G'[\wedge G, \circ \mathbf{H}, T]$. An ambient that has a name which is in group G has within it message exchanges of type T and may open ambients named within the groups within \mathbf{H} as before. It now can only cross objectively ambients within groups G' and cross subjectively ambients within groups G . As before, these changes in notation propagate to the process and capability typings.

These changes result in changes to the typing rules for the in and out capabilities, requiring that the name of the ambient which is named in the capability is a member of the groups G representing the names of ambients that can be subjectively crossed in the type of the capability. The open capability is unchanged other than for the obvious syntactic alterations.

One of the results of this typing system is that correct typing derivations allow for the inspection of which ambients are mobile and which are immobile. This is a conservative result in that some ambients may be indicated as mobile when in fact they are not. One of the main motivations for including

objective moves as primitives in the calculus is that it allows for separate typing rules for these moves rather than decomposing the moves into their subjective move encoding. The upshot is that the separate typing rules provide more accurate indication of mobile and immobile moves by the type system when objective moves are used.

2.4.2 Typing Boxed Ambients

2.4.2.1 Typing process communication

One of the chief motivations for Boxed Ambients is that by dropping the open capability it becomes easier to type the calculus. In [7] a communication typing is presented. The basic structure is similar to the typing of Mobile Ambients, though there are important differences. Because of the different communication primitives from Mobile Ambients, communication needs to be tracked differently. A process is now typed by $\text{Pro}[E, F]$ where E and F represent the types of local and upward exchanges respectively.

Ambients are represented in similar fashion, whilst capabilities are typed as $\text{Cap}[E]$. The meaning of this is dependant on the capability: the type of n is parameterised by the type of the local exchanges of the ambient n . Moving the surrounding ambient of the capability, m into the ambient n must not cause exchange typing failures, so the type of the upward exchanges of the m must match the type of the local exchanges of ambient n . The dual explains the typing of out n : the type of this is parameterised by the upward exchanges of ambient n . This is because if, say ambient m is moving outside ambient n then the upward exchanges of m must be compatible with the upward exchanges of n as the two ambients are to become siblings and thus may take part in exchanges with their shared parent.

Because of the ability for a process in an ambient to specifically target child ambients for communication, it becomes possible for the typing to allow for different types of communication between the parent and the child ambients. This contrasts sharply with the dual in Mobile Ambients where, due to the need to open a child ambient to be able to communicate with its enclosed processes, it becomes necessary for all child ambients, if they are to be opened, to have the same type. This is motivated by the fact that any output must be received by a process in the same ambient as the output and can be received by any such process. Thus all the processes that can do a receive must be of the same type as the output and, considering that the receiving process could have arrived as the result of opening a child ambient, it is clear that the typing restriction also applies to child ambients. This is obviously very much more restrictive than the situation with Boxed Ambients.

2.4.2.2 Typing ambient mobility

In [24], a novel type system is presented along with the Boxed Safe Ambient calculus. This type system not only types messages but also types mobility of ambients. Co-capabilities are used in a very similar manner to the NBA calculus (albeit without the password parameter) as described in section 2.3. One difference with the co-capabilities from [8] is that it is allowed for a co-capability to exist without naming the ambient it is to pair with. Such a co-capability expresses a general level of access, unrestricted by ambient names.

The type system is based on the usual structure where names of ambients belong to groups which are parameterised in various ways. Here, an ambient group has a parameter which expresses the set of groups which contain names of ambients in which the ambient in question may reside. Processes and capabilities are also further annotated to express the set of groups of ambients that the process or capability may move the enclosing ambient to.

Subtyping mobility is also covered in this paper. The idea is to allow a process that may move ambients to ambients within the group D wherever a process than moves ambients to ambients within the group D' is expected and $D \subseteq D'$.

The addition of the co-capabilities are also accounted for within the mobility typing: the expression of the set of groups in which an ambient may reside or a process or capability may execute is extended to express a second set of groups which are only populated by the (co-)capabilities and record the groups of ambients that co-capabilities allow to enter or exit the system. This allows the type system to reject processes that try to enter ambients for which there is no suitable co-capability.

2.5 The Channel Ambient System

In [28] a further ambient calculus variant is presented along with an ambient machine, runtime and simple programming language.

The in and out capabilities exist without additional parameters but must be paired with co-capabilities $\overline{\text{in}}$ and $\overline{\text{out}}$ to reduce. This is very similar to Boxed Safe Ambients presented in [24]. In common with Boxed Ambients, there is no open capability.

Communication primitives are unique amongst the ambient calculi, allowing specific targeting of sibling ambients for output. The communication primitives are shown in figure 2.3. Firstly, communication happens on named channel as per the π -calculus. Secondly, there are no primitives for local communication: all communication primitives are cross-boundary. Despite this, there is also no primitive for output to a child ambient on a named channel. For both of these cases, an encoding is provided, as shown in figure 2.4. This choice of communication primitives seems a little odd given how predominant the encoded “missing” primitives are in the rest of the paper. Little explanation is given as to the reasoning behind these choices.

Given the presence of named channels, it seems slightly odd to then further complicate that well understood and researched model with directionality constraints on actions on the channel. What is particularly odd is that the addition of channels removes the programmer’s need to have ambients move to be able to communicate, thus reducing the necessary awareness of the location of ambients. However, adding the directionality constraints seems to unnecessarily complicate the model, requiring that the programmer maintain the same awareness of location, reducing the benefit of having named channels that can cross boundaries.

Furthermore, given the directionality constraints on the channels, it is impossible to use a channel if the ambient in question has moved away from the other ambients that share the channel as the other ambients must not only know where the targets of communication are, but those targets must also be no further away than a parent or sibling otherwise a messenger ambient must be issued as with the other ambient calculi.

Figure 2.3 Communication primitives in the Channel Ambient Calculus: definition and reduction rules

(Sibling Output) $a \cdot x \langle v \rangle$

(Parent Output) $x^\dagger \langle v \rangle$

(Internal Input) $x(u)$

(External Input) $x^\dagger(u)$

$$a[b \cdot x \langle v \rangle . P \mid P'] \mid b[x^\dagger(u) . Q \mid Q'] \rightarrow a[P \mid P'] \mid b[Q\{v/u\} \mid Q']$$

$$a[x^\dagger \langle v \rangle . P \mid P'] \mid x(u) . Q \rightarrow a[P \mid P'] \mid Q\{v/u\}$$

Figure 2.4 Encoding of local output and child output in the Channel Ambient Calculus

(Local Output) $\langle\langle x \langle v \rangle . P \rangle\rangle \Rightarrow (\nu z)(z[x^\dagger \langle v \rangle . z^\dagger \langle \rangle] \mid z^\dagger \langle \rangle) . P$

(Child Output) $\langle\langle a/x \langle v \rangle . P \rangle\rangle \Rightarrow (\nu z)(z[a \cdot x \langle v \rangle . z^\dagger \langle \rangle] \mid z^\dagger \langle \rangle) . P$

The calculus is firmly built around computer networks and differentiates ambients between sites (physical computers) and agents. It is impossible, without the aid of a large hammer, to nest physical computers and that constraint is mirrored in the calculus. Similarly, an agent needs a computer on which to execute so the hierarchy of ambients must place sites at the roots of the trees. Other than these constraints, agent ambients can be nested as normal.

The author goes on to define the *Channel Ambient Machine*, an abstract machine for the direct execution of the calculus. The correctness of the abstract machine is verified with respect to the underlying calculus and a runtime is then defined as a direct mapping from the abstract machine to functional code.

Finally, the *Channel Ambient Language* is defined which is a simple programming language for the calculus. The language is imperative, has no structuring constructs whatsoever and also has only a skeleton library API. That is not to take anything away from it though: it was not designed to be a fully fledged programming language, more as a prototyping and modelling language. At this, it succeeds well.

2.6 Conclusion

The various mobile ambients discussed here differ primarily in the steps needed to achieve communication between processes in different ambients. As a result of these variations, the ability to statically analyse various properties of programs expressed in these calculi varies in complexity.

The apparent simplicity of the Mobile Ambient calculus is offset by the complexity of the typing system whereas the more complex communication primitives of Safe Ambients, Boxed Ambients, NBA and Safe Boxed Ambients all provide easier typings at the expense of more complex and explicit programs.

In terms of utility for programming, my opinion is that the basic Mobile Ambient calculus is unsuitable as the open primitive is so unfamiliar to the vast majority of programmers: there is no similar construct in any language as far as I'm aware. As such, I believe it would form an obstacle to the use of the language and really should be avoided. The Boxed Ambient calculus and the safer variants thereof present more rigid structures that are more instantly familiar to the majority of programmers and conceptually easier to deal with.

Chapter 3

Object Oriented Language Features

Object Oriented languages have seen substantial development over the last thirty years. Much of that development has to do with type systems and the static analysis of properties of programs. Other development is focused on adding new constructs to languages to better promote code reuse or to solve problems which in the past have forced programmers to resort to type-casting or other unsavoury practices.

In this section I review research literature for a number of features that are not commonly found in Object Oriented languages but which are suitable for inclusion in the next generation of programming languages.

3.1 Named Parameters

The majority of programming languages use positional parameters in method invocation: the parameters are assigned to the parameter variables in the method body in the same order as they are listed by the method caller.

The main disadvantage of this is that at the call site, it is often difficult to determine which arguments play which roles for the method call. Given the emphasis within software engineering practises for the use of meaningful variable names, it seems unfortunate that the names of the parameter variables in the method cannot be seen by the caller.

Named parameters (also called *keyword parameters*) have been seen in several languages including SMALLTALK, ADA and more recently PYTHON and RUBY. When using named parameters, the caller must explicitly assign in the method invocation which values are assigned to which parameter names. In the method body, these parameter names are the parameter variables as normal.

Not only can the parameters be defined in any order by the caller, but the callee can also define default parameter values that are used if the caller does not define a value for that parameter.

In [16] positional parameters and named parameters are compared and a tentative endorsement of named parameters made. One of the chief criticisms is that if the name of the parameter is changed in the method body then that will force all calls to that method to need to be modified. Given the power and utility of the programming tools and environments we have today, this is no longer a valid criticism as, with sufficient support, the subsequent changes to the call sites could be entirely automated.

As a mechanism to increase readability of source code, named parameters are a good idea and it seems surprising that they have not seen greater support in language design. Good tool support is required for the additional typing and refactoring overhead to become negligible.

3.2 Pattern Matching

Pattern matching appeared in CLOS and has featured predominantly in a large number of mainly functional languages, including ERLANG and HASKELL. There are two main ways in which it is applied. Firstly, it is common to use it to match a supplied value against a known constant. Secondly, it is used to decompose a supplied value into constituent parts and to assign those parts to variables. Listing 3.1 shows both uses.

Pattern matching is a powerful and useful mechanism for adding both predicates to method bodies and performing decomposition of compound values, commonly lists and tuples. Applying pattern

Listing 3.1 Pattern matching in the `length` function in HASKELL

```

0: length :: [a] -> int
1: length [] = 0
2: length (x:xs) = 1 + length xs

```

matching to object oriented languages is covered in [15] where it is added successfully to JAVA. In general, if pattern matching is being used in the role of determining equality with a supplied value then it is straight-forward to indicate an object instance that the value must match against. Decomposition can generally be avoided as the object itself should provide the ability to decompose itself suitably.

Pattern matching is part of the SCALA language [25] where with some additional syntax, pattern matching over class hierarchies can be performed. However, this is a somewhat limited form of pattern matching and forces the decomposition of the constructor arguments of the object.

3.2.1 Extensions of Pattern Matching

ERLANG extends pattern matching by allowing matching in which variables which have a defined value are used as the reference value to match against. This is primarily as a result of ERLANG's single assignment of variables. An example is shown in listing 3.2 where because the value of `Number` is defined in line 0, that value is matched against the first argument of the tuple returned by the function call in line 2 rather than simply assigning the result of the function call as would be the case with simple pattern matching.

Listing 3.2 Extended pattern matching in action in ERLANG

```

0: test (Number) ->
1:     Even = Number * 2,
2:     {Number, 0} = math:divideByWithRemainder (Even, 2) .

```

3.2.2 Predicate Dispatch

In [13], pattern matching and a number of other method specialisation techniques are unified into a single theory. It provides not only compile time checks that there no missing cases or ambiguous cases in any case set but also demonstrates a level of overriding where a body m_1 overrides body m_2 if m_1 's predicate logically implies m_2 's predicate.

This notion of overriding creates an ordering of the cases, always choosing the most specific, that is, the case of all the cases that pass, having a predicate which is not implied by any other case predicate. This removes the top down ordering of cases as in HASKELL or ML and the lexicographic ordering of LISP, both of which serve to simplify the implementation of the language rather than add utility to the programmer.

Because of this ordering, the ordering of the predicate tests may not occur in the same order as the programmer might expect and may occur more than once. As a result, it is important that these tests do not alter the program state whatsoever. ERLANG limits the functions that can be used in these predicate tests to functions that are defined within the core language and are known to not alter program state. This is somewhat limiting and unfortunate. It would be preferable for a type system to deduce whether a method alters state and reject such methods from use in predicate tests at compile time.

In [13], predicates can be abstracted and reused and there is support for classifiers [17] in which mutually exclusive classifications of objects are defined. These can then be used as normal tests.

Finally, [11] shows how such a unified predicate system can be efficiently implemented using directed abstract graphs and generating binary decision trees subsequently used for the runtime method invocation.

Listing 3.3 Predicate dispatch in action in ERLANG

```

0: positive (Number) when abs (Number) == Number ->
1:     true;
2: positive (Number) ->
3:     false.

```

3.3 Currying

Currying is named after Haskell B. Curry, though he denied inventing the idea and it is commonly credited to Schönfinkel in [31]. With currying, a function is supplied with fewer parameters than it needs to complete. Instead of returning an error, it returns a function that has captured the supplied parameters and awaits only the remaining parameters. When those parameters are supplied, the function will execute as if all parameters had been supplied simultaneously.

Currying has found its way into several functional languages, which is not surprising given that Curry himself used this idea in the λ -calculus. However, one of the chief complaints of currying is that it's far too restrictive: because the parameters have to be supplied in the order dictated by the function, the function author is responsible for defining the order of parameters in an order which is of most use for currying. This is not always possible.

To solve this restriction, I propose that by using named parameters throughout, any set of parameters can be passed as they will be named and are therefore unambiguous. This then gives the caller complete freedom to curry any method with any subset of the parameters that they wish.

Currying is another feature that has made it into the SCALA language [25], though again, not without problems. For a method in SCALA to be curry-able, it must be defined with multiple parameter sections explicitly. Sadly this means that none of the JAVA API libraries can be curried without explicit wrappers. Listing 3.4 shows currying in SCALA.

Listing 3.4 Currying in SCALA

```

0: def sum(x: Int)(y: Int): Int = x + y;
1:
2: def test: Unit = {
3:   var plus4Func = sum(4);
4:   var nine = plus4Func(5);
5: }
```

3.4 Mixins

Mixins, [2], are a mechanism to facilitate code reuse by allowing methods to be abstracted from the class hierarchy into *mixins*. These can then be inserted into the class hierarchy repeatedly, at multiple locations.

A mixin itself has an undefined superclass which allows it to be inserted at many points in the hierarchy. In practise, a mixin will often require its superclass to satisfy a particular type or interface, though it is left to the language as to whether this is supported. Use of the mixin is achieved by specifying that a mixin and (super)class are to be combined, creating a subclass of the superclass.

Mixins themselves are simply normal classes with a parameterised superclass. As such, they can have state in terms of instance fields along with methods as expected. Mixins can be composed and can implement interfaces: again, the precise details depend on the language.

In [1], mixins are added to STRONGTALK and a type checking system is given which checks that mixin applications are valid. In general, there are some limitations to mixins, the most prominent being that it can be the case that two mixins both want to require that their superclass implements the other mixin. Because the application of mixins specifies the linear inheritance explicitly, this circular dependency can never be achieved. In short, mixins abuse inheritance in order to facilitate code reuse. Because inheritance can not be cyclic, limitations are found. There are several other significant issues with mixins which are explained in some detail in [30].

Mixins also appear in SCALA [25], again, with several limitations. In SCALA, mixing a class *C* into another class *D* is legal only if *D*'s superclass is a subclass of *C*'s superclass. This restriction is necessary for type safety reasons: only the delta of the mixin is copied (ie none of its superclasses) into the mixed-in class. As a result, mixed in methods could refer to inherited members that are not present in the new context. The cause is really that in SCALA, mixins are in fact normal classes and so have a specified superclass. As such, they can inherit from that superclass and must still be able to do so in their mixed-in state.

3.5 Traits

Traits were first proposed in [30]. Traits succeed where mixins do not because traits enjoy the *flattening* property: the behaviour of a class which uses several traits is the same as if the contents of the traits had been copied directly into the class definition. It achieves this through several means: traits have no state of themselves and can not directly access state from the classes in which they are used, instead all access must be through methods. These methods form requirements which must be satisfied by the class in some manner. If multiple traits are used and they provide the same method then that conflict must be handled explicitly by the class: there is no implied ordering favouring the method definition from one trait over another.

These design decisions serve to remove traits from the inheritance hierarchy in that using traits is no longer akin to providing a superclass to the trait as is the case with mixins. Much finer-grained control is provided with no ordering between traits. Three additional functions are defined: when using a trait, particular methods can be deliberately excluded; aliases can be defined, renaming methods in traits within the current class; and conflicts must be explicitly resolved by providing an overriding definition in the current class.

The authors implemented traits in the SMALLTALK machine *Squeak* and then refactored the collection hierarchy. The results were 10% fewer methods than in the original implementation and the methods that were refactored into traits were themselves 12% smaller. Although some classes made use of up to 22 traits, the flattening property means that this need not impede comprehension: tool support would simply allow for the display of the methods within those 22 traits within the class itself, potentially obeying exclusion, aliasing and conflict resolution annotations.

Because the initial implementation was performed on SMALLTALK which is dynamically typed, no static type system was given in [30]. In [14], such a static type is given in which the core of traits is expressed as a calculus and type soundness is shown.

3.6 Polymorphism

Polymorphism refers to the ability to use a single part of a program with different types in different contexts. Functional languages have had polymorphism for some time: ML and HASKELL to name just two. Object Oriented languages have had polymorphism in C++ and most recently JAVA has gained polymorphism in the form of *generics* in JAVA 5 though there have been polymorphic extensions to JAVA for some time, [27, 3, 18].

In [27], a generics extension is proposed for JAVA, however the parameterised and non-parameterised versions of the same type are not considered to be the same type and so dealing with legacy code becomes difficult. [3] solves this by introducing a notion of a raw type which represents the unparameterised version of a type and a retrofitting mechanism which allows type parameters to be imposed upon legacy, unparameterised code. Finally, [18] modelled the generics extensions formally in a highly simplified, functional version of JAVA and in doing so uncovered and lead to the repair of one bug in the generics compiler.

Within this section, frequent reference is made to a running example, that of a calculator which needs to be evaluated in a number of ways, thus requiring the *Visitor-pattern* and also needing to be extended safely so as to be able to represent more complex expressions. For the sake of brevity, the code listings can be found in Appendix A.

3.6.1 Parameterised Polymorphism

Parameterised Polymorphism is the most common form of polymorphism. In functional languages, its use is very natural: consider the definition of the `map` function shown in listing 3.5. In object oriented languages, there is slightly more baggage as evident in listing 3.6, though this is largely due to functional languages having first-class support for collections and functions.

It is also worth noting how in the JAVA version, the lack of first-class support for functions causes semantic ambiguity: the type `Transformer<SourceElemType, TargetElemType>` is used to indicate that there is a type `Transformer` which has two parameterised types which are assigned into `SourceElemType` and `TargetElemType`. This does not in any way mean that there is a function which converts objects from the former to the latter: this can only be defined by the `Transformer` class which isn't shown. The HASKELL version contrasts with this as `(a -> b)` does indeed mean a function which takes a single parameter of type `a` and returns a result of type `b`.

Listing 3.5 HASKELL definition of the map function

```

0: map :: (a -> b) -> [a] -> [b]
1: map f [] = []
2: map f (x:xs) = (f x) : (map f xs)

```

Listing 3.6 JAVA definition of the map function

```

0: <SourceElemType, TargetElemType>
1:   Iterator<TargetElemType>
2:     map (Transformer<SourceElemType, TargetElemType> transformer,
3:         Iterator<SourceElemType> source)
4:     {
5:         List<TargetElemType> targetList
6:         = new ArrayList<TargetElemType> ();
7:         while (source.hasNext ())
8:         {
9:             targetList.add(transformer.transform(source.next ()));
10:        }
11:        return targetList.iterator ();
12:    }

```

Map, defined in this way is polymorphic because it can be reused with different types: lists of numbers, strings, colours etc can all be manipulated by the function provided the correlation between the types of the arguments is correct. The HASKELL version requires a function where the JAVA version requires an object instance of the **Transformer** class, but both have the same requirements as to how the parameterised types of the parameters of the map function should be correlated without specifying any concrete class.

Listing A.1 shows how generics can be used within the *Visitor-pattern* in order to allow the visiting of each node to return a value, the type of which varies with the visitor. This is more elegant and comprehensible than capturing the result within the visitor itself. The calculator shown in that listing can only have numbers and subtractions. It is needed to be able to extend the definition such that addition can also be represented. Altering the original definition and adding an extra case to the definition of `Visitor` would require that all the existing implementations be altered. This needs to be avoided. Instead, it is required that by subclassing and extending, this be achieved in such a way that the existing visitors can continue to work on the basic expressions but new visitors can be defined on the more complex expressions.

Basic inheritance can not achieve this, not even with help from parameterised polymorphism as section A.2 explains. What is needed is a more advanced form of polymorphism.

3.6.2 Virtual Types Polymorphism

Virtual types differ subtly from parameterised polymorphism: instead of providing type parameters to the class when instantiating the class, the class is overridden, declaring the types of type-fields in the subclass. Virtual types first appeared in BETA [22] and have been proposed for JAVA in [32].

Virtual types and parameterised types are largely complimentary: parameterised types work well in collections whereas virtual types work particularly well with families of classes: a property that is exploited in family polymorphism (section 3.6.4). The main problem with virtual types is that providing more refined type fields demands a subclass. This makes it far harder to achieve certain subtype relationships than with parameterised polymorphism. For example, consider a `List` of `Strings`. It would seem natural for such an object to be a subtype of a `Collection` of `Strings`. In parameterised polymorphism this is easy as `List` is a subtype of `Collection` and so `List<String>` is a subtype of `Collection<String>`. With virtual types this is much harder: a `StringCollection` class can only be constructed by subclassing `Collection` and similarly `StringList` subclasses `List`. But that leaves no subtype relationship between `StringList` and `StringCollection`.

In [5], the strengths and weaknesses of virtual types are expounded, including a demonstration of the ability of virtual types to deal with mutually recursive types far more effectively than parameterised polymorphism can. The example is too long and complex to reproduce here, so I refer you to that paper for details. The paper continues, using the complimentary strengths of parameterised polymorphism

and virtual types as motivation for the author's *MyType* construct.

3.6.3 *MyType*

In [6, 5, 4], the *MyType* system is proposed. This is an extension of parameterised polymorphism in which additional constructs are presented, providing some unification between virtual types and parameterised polymorphism.

MyType introduces the `ThisClass` keyword which can be thought of as the public type of `this`: it represents a class that subclasses the current class, inheriting all the public members of the current class. It varies in the same way as the meaning of `this` does from class to class. Additionally `ThisType` is another keyword which can be used within interfaces. This is necessary due to subtleties between the type of a class instance and the interfaces the class implements.

The final component is the introduction of exact types. Exact types are represented by prefixing the type name with a '@' and indicates that only a value of the type indicated can be provided and not a subtype. Together, these components can be used to provide a very similar level of expression as provided by family polymorphism (section 3.6.4): the encoding of *MyType* into a family polymorphic system is in fact extremely straightforward.

Appendix A.3 and A.4 show how the calculator can be written and successfully extended using *MyType*. Sadly it is not possible to create a structure of terms which can be visited by visitors returning different types: requiring different return types would require specifying the return type at the time of constructing each term, thus restricting terms to visitors that all return the same type. This is a shame and is caused by the use of parameterised polymorphism in order to achieve families of terms, thus adding the return type further specifies the family.

To walk through the various combinations of visitors and terms explains how the *MyType* system achieves the extension safely. The obvious case is that terms parameterised with `Visitor` can be used with implementations of `Visitor` and terms parameterised with `ExtendedVisitor` can be used with implementations of `ExtendedVisitor`. This is easy to check: the terms can only accept the exact type of their type parameter which extends `Visitor` (or `ExtendedVisitor` for `PlusTerm`). Similarly, the visitors, by use of `ThisType` ensure that the value they receive has a type parameter which matches the type of the visitor.

Next, consider wanting to use an implementation of `ExtendedVisitor` on terms that have been parameterised with `Visitor`. Because of the use of the exact type on the `accept` method, such a visitor could not be used. Finally, consider using an implementation of `Visitor` on terms that have been parameterised with `ExtendedVisitor`. This obviously fails, firstly again due to the use of the exact type and secondly because the definition of `PlusTerm` dictates that the visitor must be a subtype of `ExtendedVisitor`.

3.6.4 Family Polymorphism

Family polymorphism can be viewed as an extension of virtual types where not only are types defined by overriding, but entire inner-class definitions are overridden. This overriding is known as further binding and behaves in a similar but notably different way to basic inheritance, extending the definition of the superclass. Because the classes are grouped together and vary with one another, families are created with dependent types.

Dependent type systems for object systems have been proposed in [26, 19, 20, 12] and feature in SCALA, [25]. They can broadly be grouped into two categories: those that support class-families and those that support object-families.

The purpose is to allow the specification of families of types which interact only within themselves. Subclassing the entire family does not necessarily create subtypes of the classes within the family. For example, if `C.D` refers to a class `D` within family `C` and `C'` is a subtype of `C`, created by explicit subclassing of `C`, then whilst `C'.D'` exists and has the same definition (or even is further bound by overriding), it is not a subtype of `C.D`.

If this were not the case then there would be the potential for disaster: consider the calculator examples as explained in section A.5 and section A.6. If `ExtendedCalculator.Term` were a subtype of `Calculator.Term` then it would mean that you could have, as children of `Calculator.MinusTerm`, instances of `ExtendedCalculator.PlusTerm`. Then, when you send the basic visitor in, you would have Message not understood errors as the `PlusTerm` tried to call the `plusCase` method in the visitor which does not exist.

It is precisely because subclassing of the family and further binding of inherited classes does not make automatic subtypes of inherited classes that such behaviour is prevented.

If you consider the various possible subtype relations of the two `Visitor` interfaces then you soon discover that any subtype relation doesn't make sense. The first and simplest idea is that instances of `Calculator.Visitor` can only work with instances of `Calculator.Term` and instances of `ExtendedCalculator.Visitor` can only work with instances of `ExtendedCalculator.Term`: that is, no subtype relationship between the two visitors; this can be typed successfully. The next idea is that you would like instances of `ExtendedCalculator.Visitor` to be able to work with `Calculator.Term`. This looks fine until you try to type the `plusCase` method of the visitor in the context of the basic calculator: you can't type it as there is no such type for the parameter of type `PlusTerm`. Finally, there is the idea that instances of `Calculator.Visitor` can be accepted by instances of `ExtendedCalculator.Term`. This clearly is bogus as already discussed because instances of `PlusTerm` can appear in the expression and there is no corresponding method in the visitor.

This is typically as far as class-family based family polymorphism systems go: families are defined by nested classes and object instances from different class family definitions are not allowed to mix. Some systems, for example *Concord* [20], allow for the explicit subtyping of inner classes to controllably relax the constraints. It is also possible to combine dependent types with parameterised polymorphism in order to construct methods or classes which work with any subtype of a given family. This has been demonstrated in *.FJ*, [19].

There are many good motivating examples for dependent types: the *Observer-pattern* is one such example in which it becomes possible to construct a family representing the observer and notifier. Only instances of the observers and notifiers from within the same family can interact with one another and subclassing the family allows for specialisation without code duplication.

Object-family systems such as *Tribe* or *SCALA*, [12, 25], extend the restrictions of non-interaction to instances of the family class. The `static` modifier gets dropped from the inner class definitions and use of the inner classes requires the instantiation of the containing class or family first and then the subsequent prefixing of the inner classes with the family instance name. This leads to path types where final instance fields can be used to indicate types. Furthermore, now not only can instances of classes from different class-families not be mixed, but instances of classes from different instances of the same class-family can also not be mixed.

In some respects this seems too restrictive: it becomes very much harder, for example, to create instances of the `Visitor` interface as the `accept` methods of the `Term` interface require that the instance of the `Visitor` be created by the same family instance as the `Terms` themselves. There are, of course instances where this restriction is very useful, and there are applications of such dependent type systems in ownership types. One of the current outstanding issues with object-family systems is decidability of the subtype relation: this is in general undecidable for pure dependent type systems, [29], although certain restrictions have been shown to be decidable.

3.7 Conclusion

Since *JAVA*, there have been many new developments in object oriented languages, most of which have seen exposure amongst the academic community and some of which have been implemented as extensions to existing languages. What has not happened however, is the combination of many of these ideas into a language which has a real appeal to the developer community to be used.

Traits are clearly such a powerful and successful idea that they should be seriously considered for future languages. Currying in combination with named parameters appears to overcome some of the issues relating to currying and will therefore make the feature more powerful and more useful. Pattern matching has already seen widespread exposure and is a familiar language feature to many which certainly endorses its usefulness.

Polymorphism has perhaps only recently come to the attention of a lot of programmers who have not been exposed to *C++* or functional languages and some of the extensions seem to add complexities that may restrict how widespread such features become. Nevertheless, such extensions do provide the ability to work around some of the issues that frequently occur with type systems that support only parameterised polymorphism and basic inheritance. *MyType* is slightly more difficult to work with because of the use of additional constructs and, at least in my example, does not solve the problem of extending the calculator as well as using family polymorphism. In addition, family polymorphism seems more natural and intuitive to work with and as such may be a more worthwhile addition to *GLINT*.

Chapter 4

Specification

Any large engineering project can be viewed as a series of tradeoffs: at what level to provide support for various features is crucial to determining the ease of use of the product and the difficulty of the implementation. Language design is no exception: how tightly coupled should the syntax of the language and any object class system be: should there be syntactic support for lists, sets etc? Should polymorphism be supported and if so, in what manner? Mixins, traits, path types, self-types etc all need consideration and then there is the matter at the core of this project: in what way should ambients be integrated into the language? What type of ambients should be supported: the original calculus with the problematic open capability which complicates typing, the Boxed Ambient calculus with its richer communication primitives or some combination of the two?

A programming language and suite of tools is required which can be used by programmers to construct sophisticated applications and by the users to move running processes repeatably from one machine to another. A written report should detail the decisions made and reasons behind those decisions in the process of developing a solution to these problems.

4.1 The Language

- The language should be a pure object oriented language. If at all possible there should be no notion of *primitives* within the language, that is, everything is an object as opposed to, say, JAVA where certain data types are primitive and non-objects.
- At this point, performance is not a concern so an interpreter rather than a compiler is acceptable.
- Ambients should be embedded within the language at a very fundamental level: ideally, the language should be built on top of ambients themselves.
- An API should be provided to allow controlled and safe access to the underlying ambient calculus. In may be possible to view this as a form of bytecode reflection.
- The language should have some form of polymorphism.
- The language should support traits.
- The language should be statically checked by a number of type systems in order to guarantee certain properties about the programs it passes. These properties should be extensively documented and justified within the report.
- If at all possible, the compiler should be platform agnostic.

4.2 Runtime

- The runtime must be able to correctly execute any program that can be expressed within the language.
- The runtime should be concerned with correctness and reliability as opposed to speed.

- The runtime should be able to cope with the distributed nature of ambients and not fail or cause fatal errors if other runtimes with which it communicates fail to respond.
- If at all possible, the runtime should be platform agnostic. Ideally it should be possible to have multiple runtimes executing on multiple platforms communicating with one another.
- The behaviour of the runtime should be based on a formal semantics for which subject reduction has been proved.
- The runtime should offer debugging facilities in which the state of the ambients and processes can be inspected.

4.3 Tools

- There should be a suite of command-line tools that allow for the inspection and objective movement of ambients.
- It should be possible, after authentication with a runtime, to see a complete listing of ambients.
- It should be possible, after authentication with a runtime, to select a number of ambients and have them move to a different ambient.
- The tools should ideally have a GUI, though this is not essential.
- Programming tools, such as language support from within an IDE such as *Eclipse* would be beneficial but is not essential.
- If at all possible, the tools should be platform agnostic.

4.4 Testing

A large suite of tests should be developed which serve to ensure the various components work correctly. The following itemises the behaviour that should be tested.

4.4.1 Ambient behaviour

- Parallel processes truly are parallel where hardware allows and are not simply sequentially concurrent.
- Communication works correctly in monadic and polyadic forms where reduction can occur.
- Communication blocks correctly and safely where no reduction can take place.
- Capabilities work correctly where reduction can take place and block safely where they can not.
- A process blocked either on communication or capability correctly becomes unblocked as soon as it is able to reduce.
- Processes that successfully reduce to the stop process, 0, are garbage collected appropriately.
- Replication expands correctly and non-eagerly in order to prevent the unnecessary consumption of resources: expansion is only required when the previous expansion has been able to reduce in some manner.
- Restricted naming works in such a way that given known behaviour of all runtime nodes the probability of two restrictions creating the same name is either zero or negligible.

4.4.2 Language behaviour

Because the language features and syntax are yet to be defined, the unit tests here can not be as precise as in the previous section.

- For every language feature, unit tests exist to test the correct behaviour of the feature.
- Every language feature must be able to function correctly in a distributed manner given the ability for the user to objectively manipulate and restructure the distribution of the running instance of the program.

4.4.3 Runtime and Tools behaviour

- The runtime can successfully interact with the various tools concurrently and safely.
- The tools respond appropriately to changes in the state of the runtime as the runtime informs the tools.
- No misuse or other interaction with the runtime via the interfaces and APIs for the tools can cause the runtime to fault or cause any other attached tool to fault.
- No program that the runtime accepts can behave in such a way that it can not be terminated by the runtime nor cause the runtime to fault.

It should be kept in mind that the aim of this project is to create a language and suite of tools in which the power and utility of ambients can be exploited. As such, language and tool features should be chosen so as to better achieve this aim.

Chapter 5

Evaluation

Evaluation of a project such as this can be made to be quantitative but ultimately will be qualitative. The quality of the implementation of the language, runtime and tools can, to a certain degree, be verified through the use of unit tests and other test suits.

Comparison to other languages can be made through metrics as trite as the number of characters needed to express certain constructs, the speed of executing programs or the memory footprint of the runtime, but it will only be through a significant level of experience with this language and other competing languages that any qualitative evaluation can be made.

This is not to take anything away from unit tests and test driven development: a large body of tests is required so as to verify the accuracy and success of the type systems, the runtime systems and the tools. Some of the aspects of the systems produced that require testing are detailed in section 4.4. These must pass and fail as expected by each test. A large number of examples is required to showcase what the language does well and what it perhaps does not do so well.

Clearly, the problems explained in chapter 1 can only be addressed with a large framework and infrastructure which is almost certainly beyond the scope of this project. Instead, a series of smaller problems should be addressed and effective solutions presented.

- **Web Browser.** Many good HTML rendering engines exist today. It should be possible to make use of one and to demonstrate the ability to move a running instance of such a browser from one machine to another. Further more, this is an ideal situation to demonstrate the ability of being able to choose which components of the process to move: by only moving the rendering and GUI components and leaving the fetching component on the initial machine, access to the local file-system will reflect the file-system of the initial machine.
- **Media player.** In a similar manner, some sort of media player, be it audio or video would be an excellent demonstration application: the rendering front end could be moved from one machine to another but the actual file and the file reading components remain on the initial machine. Ideally, the player never misses a beat!
- **Dining distributed philosophers.** The well known dining philosophers problem could be encoded in the language: the meal would obviously remain stationary and initially the philosophers would too, but extensions could be envisaged where there are multiple meals and the philosophers can move between meals with the goal of eating as much as possible as quickly as possible.
- **Distributable web-server.** A good demonstration of the use to the software industry, this web-server would be able to have the CGI and other dynamic content engines easily distributed to additional machines in order to provide load balancing and better scalability.

Ultimately, the popularity of a language is not governed by its features, its design or even its speed. It is governed by a large number of other factors, most of which are outside the control of language designers and some of which are contrary to the evolution of programming: observe that some 40 years after C was created it is still the primary language of development for some of the most important programs currently in development. This spectacular level of resistance to change is extremely worrying and little has been done to address its consequences. As a result, it is extremely unlikely that the language presented by this project will go on to widespread use. What is more important is the justification of the design decisions and the subsequent documentation thereof such that future designs can understand and build upon the work done here.

Bibliography

- [1] L. Bak, G. Bracha, S. Grarup, R. Griesemer, D. Griswold, and U. Hölzle. Mixins in Strongtalk. In *European Conference on Object-Oriented Programming*, 2002. Workshop on Inheritance. 19
- [2] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings of the European Conference on Object-Oriented Programming on Object-oriented programming systems, languages, and applications*, pages 303–311. ACM Press, 1990. 19
- [3] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: adding genericity to the Java programming language. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 183–200, Vancouver, British Columbia, Canada, 1998. ACM Press, New York, NY, USA. 20
- [4] K. B. Bruce and J. N. Foster. LOOJ: Weaving LOOM into Java. In *ECCOP '04: Proceedings of the 18th European Conference on Object-Oriented Programming*, volume 3086 of Lecture Notes in Computer Science, pages 390–414. Springer-Verlag, Jan 2004. 22
- [5] K. B. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, volume 1445 of Lecture Notes In Computer Science, pages 523–549. Springer-Verlag, Jan 1998. 21, 22
- [6] K. B. Bruce and J. C. Vanderwaart. Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. *Electronic notes in Theoretical Computer Science*, 20, 1999. 22
- [7] M. Bugliesi, G. Castgna, and S. Crafa. Boxed ambients. In *Proceedings of the 4th International Symposium on Theoretical Aspects of Computer Software*, volume 2215 of Lecture Notes In Computer Science, pages 38–63. Springer-Verlag, 2001. 9, 10, 11, 14
- [8] M. Bugliesi, S. Crafa, M. Merro, and V. Sassone. Communication interference in mobile boxed ambients. In *Proceedings of the 22nd Conference Kanpur on Foundations of Software Technology and Theoretical Computer Science*, volume 2556 of Lecture Notes In Computer Science, pages 71–84. Springer-Verlag, 2002. 11, 14
- [9] L. Cardelli, G. Ghelli, and A. D. Gordon. Types for the ambient calculus. *Journal of Information and Computation*, 177(3):160–194, 2002. 12
- [10] L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science (TCS)*, 240:177–213, 2000. 7, 8, 12
- [11] C. Chambers and W. Chen. Efficient multiple and predicate dispatching. In *Proceedings of the 1999 ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA '99)*, volume 34 of ACM SIGPLAN Notices, pages 238–255. ACM Press, November 1999. 18
- [12] D. Clarke, S. Drossopoulou, J. Noble, and T. Wrigstad. Tribe: More types for virtual classes. December 2005. 22, 23
- [13] M. D. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *ECOOP '98: 12th European Conference on Object-Oriented Programming*, volume 1445, pages 186–211, Brussels, Belgium, 1998. Springer-Verlag. 18
- [14] K. Fisher and J. Reppy. Statically typed traits. Technical Report TR-2003-13, University of Chicago, Department of Computer Science, December 2003. 20

- [15] R. Forax and G. Roussel. Recursive types and pattern-matching in Java. In *GCSE '99: Proceedings of the First International Symposium on Generative and Component-Based Software Engineering*, pages 147–164. Springer-Verlag, 2000. 18
- [16] W. T. Hardgrave. Positional versus keyword parameter communication in programming languages. *j-SIGPLAN*, 11(5):52–58, May 1976. 17
- [17] J. G. Hosking, J. Hamer, and W. B. Mugridge. Integrating functional and object-oriented programming. *Technology of Object-Oriented Languages and Systems (TOOLS)*, 3:345–355, 1990. 18
- [18] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001. 20
- [19] A. Igarashi, C. Saito, and M. Viroli. Lightweight family polymorphism. In *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS'05)*, volume 3780 of Lecture Notes In Computer Science, pages 161–177. Springer-Verlag, November 2005. 22, 23
- [20] P. Jolly, S. Drossopoulou, C. Anderson, and K. Ostermann. Simple dependent types: Concord. In *Programmings of 6th ECOOP Workshop on Formal Techniques for Java-like Programs (FTfJP2004)*, June 2004. 22, 23
- [21] F. Levi and D. Sangiorgi. Controlling interference in ambients. In *Proceedings of POPL'00*, pages 352–364. ACM Press, 2000. 11
- [22] O. L. Madsen and B. Møller-Pedersen. Virtual classes: A powerful mechanism for object-oriented programming. In *Object-Oriented Programming: Systems, Languages and Applications*. ACM, 1989. 21
- [23] M. Merro and M. Hennessy. Bisimulation congruences in safe ambients. In *Proceedings of POPL'02*, pages 71–80. ACM Press, 2002. 11
- [24] M. Merro and V. Sassone. Typing and subtyping mobility in boxed ambients. In *Proceedings of the 13th International Conference on Concurrency Theory*, volume 2421 of Lecture Notes In Computer Science, pages 304–320. Springer-Verlag, 2002. 14, 15
- [25] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the scala programming language. Technical report, EPFL, July 2004. 18, 19, 22, 23
- [26] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743 of Lecture Notes In Computer Science, pages 201–224. Springer-Verlag, July 2003. 22
- [27] M. Odersky and P. Wadler. Pizza into Java: translating theory into practice. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 146–159, Paris, France, 1997. ACM Press, New York, NY, USA. 20
- [28] A. N. J. B. Phillips. *Specifying and Implementing Secure Mobile Applications in the Channel Ambient System*. PhD thesis, Imperial College, London, 2005. 15
- [29] B. C. Pierce, editor. *Advanced Topics in Types and Programming Languages*, chapter 2. MIT Press, 2005. 23
- [30] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. In *ECOOP 2003: 17th European Conference on Object-Oriented Programming*, volume 2743 of Lecture Notes in Computer Science, pages 248–274, January 2003. 19, 20
- [31] M. Schönfinkel. Über die bausteine der mathematischen logik. *Mathematische Annalen*, 92:305–316, 1924. 19
- [32] K. K. Thorup. Genericity in Java with virtual types. In *European Conference on Object-Oriented Programming*, volume 1241 of Lecture Notes In Computer Science, pages 444–471. Springer-Verlag, 1997. 21
- [33] J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, volume 1686 of Lecture Notes In Computer Science. Springer-Verlag, 1999. 10

Appendix A

Program listings

A.1 JAVA based simple calculator

Listing A.1 shows the initial calculator which is sought to be extended. This calculator can have only simple numbers and negation of pairs of numbers. The expressions are represented in a standard tree form and a visitor pattern is provided which is implemented by a basic evaluator.

Note that the implementations of terms allow access to their contents via public fields. This has been done only in the interests of code length: a more realistic example would most likely use *getter* methods.

Also note that the use of generics allows the variance of the return type of the `accept` method which means that it is simple to define a visitor that, for example performs pretty-printing of the expression. Of course, this could have been achieved by having no return type and maintaining state within the visitor itself but doing so is rather less elegant and is less comprehensible.

Listing A.1 JAVA definition of the simple calculator with a visitor-pattern based evaluator

```

0: public interface Term {
1:     public <T> T accept (Visitor<T> visitor);
2: }
3:
4: public class MinusTerm implements Term {
5:     public final Term left;
6:     public final Term right;
7:
8:     public MinusTerm (final Term l, final Term r) {
9:         left = l;
10:        right = r;
11:    }
12:
13:    public <T> T accept (Visitor<T> visitor) {
14:        return visitor.minusCase (this);
15:    }
16: }
17:
18: public class NumberTerm implements Term {
19:     public final int number;
20:
21:     public NumberTerm (int n) {
22:         number = n;
23:     }
24:
25:     public <T> T accept (Visitor<T> visitor) {
26:         return visitor.numberCase (this);
27:     }
28: }
29:
30: public interface Visitor<ReturnType> {
31:     public ReturnType minusCase (MinusTerm term);
32:     public ReturnType numberCase (NumberTerm term);
33: }
34:
35: public class Evaluator implements Visitor<Integer> {
36:     public Integer minusCase (MinusTerm term) {
37:         Integer lhs = term.left.accept (this);
38:         Integer rhs = term.right.accept (this);
39:         return lhs - rhs;
40:     }
41:
42:     public Integer numberCase (NumberTerm term) {
43:         return term.number;
44:     }
45:
46:     public static void main (String[] args) {
47:         NumberTerm five = new NumberTerm (5);
48:         NumberTerm eight = new NumberTerm (8);
49:         MinusTerm minus = new MinusTerm (eight, five);
50:         Visitor<Integer> visitor = new Evaluator ();
51:         int result = minus.accept (visitor);
52:         System.out.println (result);
53:     }
54:
55: }

```

A.2 JAVA Extension of simple calculator

Listing A.2 shows a failed attempt to extend the calculator. This fails because the `Visitor` type does not include a method that the `PlusTerm` can call. Adding such a method would require all implementations to be modified.

To avoid that modification, an `ExtendedVisitor` interface could be defined which does include a suitable case. However, the definition of `Term` only allows for the basic `Visitor` to be passed to the terms. So to combat that, an `ExtendedTerm` interface could be defined, modifying the `accept` method, allowing the `ExtendedVisitor` to be accepted by the terms. However problems arise at every turn: if the `ExtendedTerm` does not extend `Term` then all the existing implementations of `Term` must be altered which we want to avoid. If `ExtendedTerm` does extend `Term` then the implementation of `PlusTerm` implements `Term` and `ExtendedTerm` which still requires that there must be a case in `Visitor` for `PlusTerm` to call and so we are back where we started.

If, in this last case, we simply return `null` and don't call any method in `Visitor`, thus solving the problem then we still have the worrying consequence that we can use implementations of the basic `Visitor` to visit expressions that contain instances of `PlusTerm` which we would also like to avoid.

Listing A.2 Faulty attempt to extend the JAVA calculator with Plus-terms

```

0: public class PlusTerm implements Term {
1:     public final Term left;
2:     public final Term right;
3:
4:     public PlusTerm (final Term l, final Term r) {
5:         left = l;
6:         right = r;
7:     }
8:
9:     public <T> T accept (Visitor<T> visitor) {
10:        return visitor.plusCase (this); // no such method
11:    }
12: }
```

A.3 MyType based simple calculator

Listing A.3 shows the basic calculator implemented in a Java-like language using *MyType* constructs. The points to notice here are the use of the exact type on the parameter of the `accept` methods, the use of parameterised polymorphism in the `Terms` and the use of `ThisType` in the `Visitor`.

By using the exact type on the parameter to the `accept` methods, only the subtype of `Visitor` with which the `Term` was parameterised with can be passed to the `accept` method and not any subtype. As a result, there is an exact correspondence which can not be violated between `Terms` and `Visitors`. This prevents the mixing of terms which require different visitors and prevents subtypes of visitors from being used on terms that would otherwise create typing violations for the cases within the visitor.

Note that the terms are hard coded to return `ints` as a result of visiting them. Abstracting this as before is impossible due to the fact that the return type would be specified as a type parameter to the `Visitor` interface and therefore would be captured by the terms. This would result in terms that can only be used with visitors that all return the same type thus the goal of term instances that can be visited by visitors returning any type of value in a type safe way can not be achieved.

Listing A.3 Basic calculator implemented using *MyType*

```

0: interface Term <V extends Visitor> {
1:     public int accept (V visitor);
2: }
3:
4: class NumberTerm <V extends Visitor> implements @Term<V> {
5:     public final int number;
6:
7:     public NumberTerm (int n) {
8:         number = n;
9:     }
10:
11:     public int accept (@V visitor) {
12:         return visitor.numberCase (this);
13:     }
14: }
15:
16: class MinusTerm <V extends Visitor> implements @Term<V> {
17:     public final Term<V> left;
18:     public final Term<V> right;
19:
20:     public MinusTerm (Term<V> l, Term<V> r) {
21:         left = l; right = r;
22:     }
23:
24:     public int accept (@V visitor) {
25:         return visitor.minusCase (this);
26:     }
27: }
28:
29: interface Visitor {
30:     public int numberCase (NumberTerm<ThisType> term);
31:     public int minusCase (MinusTerm<ThisType> term);
32: }
33:
34: class Evaluator implements Visitor {
35:
36:     public int numberCase (NumberTerm<ThisType> term) {
37:         return term.number;
38:     }
39:
40:     public int minusCase (MinusTerm<ThisType> term) {
41:         int lhs = term.left.accept (this);
42:         int rhs = term.right.accept (this);
43:         return lhs - rhs;
44:     }
45:
46:     public static void main (String[] args) {
47:         Term<Visitor> five = new NumberTerm<Visitor> (5);
48:         Term<Visitor> three = new NumberTerm<Visitor> (3);
49:         Term<Visitor> minus = new MinusTerm<Visitor> (five, three);
50:
51:         Visitor v = new Evaluator ();
52:         int result = minus.accept (v);
53:     }
54: }

```

A.4 Extension of *MyType* virtual type based calculator

Listing A.4 shows the extension of the basic calculator to successfully cope with `PlusTerms` without unnecessary code duplication. This is achieved through the parameterisation of the terms with the type of the visitor, the acceptance only of the exact type of the visitor and the use of `ThisType` to guarantee that the terms being passed into the visitor are parameterised themselves with the visitor.

As noted above, observe that the return type is fixed as an `int`.

Listing A.4 Basic calculator implemented using *MyType*

```

0: class PlusTerm <V extends ExtendedVisitor> implements @Term<V> {
1:     public final Term<V> left;
2:     public final Term<V> right;
3:
4:     public PlusTerm (Term<V> l, Term<V> r) {
5:         left = l; right = r;
6:     }
7:
8:     public int accept (@V visitor) {
9:         return visitor.plusCase (this);
10:    }
11: }
12:
13: interface ExtendedVisitor extends Visitor {
14:     public int plusCase (PlusTerm<ThisType> term);
15: }
16:
17: class ExtendedEvaluator
18:     extends Evaluator implements ExtendedVisitor {
19:
20:     public int plusCase (PlusTerm<ThisType> term) {
21:         int lhs = term.left.accept (this);
22:         int rhs = term.right.accept (this);
23:         return lhs + rhs;
24:     }
25:
26:     public static void main (String[] args) {
27:         Term<ExtendedVisitor> five
28:             = new NumberTerm<ExtendedVisitor> (5);
29:         Term<ExtendedVisitor> three
30:             = new NumberTerm<ExtendedVisitor> (3);
31:         Term<ExtendedVisitor> minus
32:             = new MinusTerm<ExtendedVisitor> (five, three);
33:         Term<ExtendedVisitor> eight
34:             = new NumberTerm<ExtendedVisitor> (8);
35:         Term<ExtendedVisitor> plus
36:             = new PlusTerm<ExtendedVisitor> (eight, minus);
37:
38:         ExtendedVisitor v = new ExtendedEvaluator ();
39:         int result = plus.accept (v);
40:     }
41: }

```

A.5 Family polymorphism based simple calculator

Listing A.5 shows the basic calculator reimplemented in a JAVA-like language with family polymorphism. The family polymorphism groups the various components of the calculator together and there are changes to the way in which terms are created and used. In general though, there are only minor changes between this version and the JAVA version.

Listing A.5 Simple calculator defined using family polymorphism

```

0: public class Calculator {
1:     public static interface Visitor<ReturnType> {
2:         public ReturnType minusCase (out.MinusTerm term);
3:         public ReturnType numberCase (out.NumberTerm term);
4:     }
5:
6:     public static interface Term {
7:         public <RT> RT accept (out.Visitor<RT> visitor);
8:     }
9:
10:    public static class MinusTerm implements out.Term {
11:        public final out.Term left;
12:        public final out.Term right;
13:
14:        public MinusTerm (out.Term l, out.Term r) {
15:            left = l; right = r;
16:        }
17:
18:        public <RT> RT accept (out.Visitor<RT> visitor) {
19:            return visitor.minusCase (this);
20:        }
21:    }
22:
23:    public static class NumberTerm implements out.Term {
24:        public final int number;
25:
26:        public NumberTerm (int n) {
27:            number = n;
28:        }
29:
30:        public <RT> RT accept (out.Visitor<RT> visitor) {
31:            return visitor.numberCase (this);
32:        }
33:    }
34: }
35:
36: public class Evaluator implements Calculator.Visitor<Integer> {
37:     public Integer minusCase (Calculator.MinusTerm term) {
38:         int lhs = term.left.accept (this);
39:         int rhs = term.right.accept (this);
40:         return lhs - rhs;
41:     }
42:
43:     public Integer numberCase (Calculator.NumberTerm term) {
44:         return term.number;
45:     }
46:
47:     public static void main (String [] args) {
48:         Calculator.NumberTerm five
49:             = new Calculator.NumberTerm (5);
50:         Calculator.NumberTerm eight
51:             = new Calculator.NumberTerm (8);
52:         Calculator.MinusTerm minus
53:             = new Calculator.MinusTerm (eight, five);
54:         Calculator.Visitor<Integer> visitor
55:             = new Evaluator ();
56:
57:         int result = minus.accept (visitor);
58:     }
59: }

```

A.6 Extension of family polymorphism based calculator

Listing A.6 shows the successful extension of the simple calculator so that it is able to deal with addition as well as subtraction. The key here is that further-binding has been used, redefining the `Visitor` interface. This further binding affects the relevant types in the `Number` and `Minus` terms when they are inherited into the `ExtendedCalculator`.

Note the cleanliness of this solution: it is not necessary to deal explicitly with exact types or *myType* as with other solutions and safety is achieved: terms from the two calculators can not be mixed.

Listing A.6 Successful extension of the family polymorphism based calculator with Plus-terms

```

0: public class ExtendedCalculator extends Calculator {
1:     public static interface Visitor {
2:         public Return Type plusCase (out.PlusTerm term);
3:     }
4:
5:     public static class PlusTerm implements out.Term {
6:         public final out.Term left;
7:         public final out.Term right;
8:
9:         public PlusTerm (out.Term l, out.Term r) {
10:             left = l; right = r;
11:         }
12:
13:         public <RT> RT accept (out.Visitor<RT> visitor) {
14:             return visitor.plusCase (this);
15:         }
16:     }
17: }
18:
19: public class ExtendedEvaluator
20:     implements ExtendedCalculator.Visitor<Integer> {
21:     public Integer plusCase (ExtendedCalculator.PlusTerm term) {
22:         int lhs = term.left.accept (this);
23:         int rhs = term.right.accept (this);
24:         return lhs + rhs;
25:     }
26:
27:     public Integer minusCase (ExtendedCalculator.MinusTerm term) {
28:         int lhs = term.left.accept (this);
29:         int rhs = term.right.accept (this);
30:         return lhs - rhs;
31:     }
32:
33:     public Integer numberCase (ExtendedCalculator.NumberTerm term) {
34:         return term.number;
35:     }
36:
37:     public static void main (String [] args) {
38:         ExtendedCalculator.NumberTerm five
39:             = new ExtendedCalculator.NumberTerm (5);
40:         ExtendedCalculator.NumberTerm eight
41:             = new ExtendedCalculator.NumberTerm (8);
42:         ExtendedCalculator.MinusTerm minus
43:             = new ExtendedCalculator.MinusTerm (eight, five);
44:         ExtendedCalculator.NumberTerm two
45:             = new ExtendedCalculator.NumberTerm (2);
46:         ExtendedCalculator.PlusTerm plus
47:             = new ExtendedCalculator.PlusTerm (minus, two);
48:         ExtendedCalculator.Visitor<Integer> visitor
49:             = new ExtendedEvaluator ();
50:
51:         int result = plus.accept (visitor);
52:     }
53: }

```
