TUM

INSTITUT FÜR INFORMATIK

A Formal Foundation for Concurrent Object Oriented Programming

Radu Grosu

TUM-I9444
Januar 1995

TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM-INFO-01-1995-I9444-350/1.-FI

Typescript: ---

Druck: Mathematisches Institut und
Institut für Informatik der
Technischen Universität München

Institut für Informatik
der Technischen Universität München

A Formal Foundation for Concurrent Object Oriented Programming

Radu Grosu

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. J. Schlichter
Prüfer der Dissertation:		
	1.	Univ.-Prof. Dr. M. Broy
	2.	Univ.-Prof. T. Nipkow, Ph.D.

Die Dissertation wurde am 12. Juli 1994 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 15. November 1994 angenommen.

Abstract

In this thesis we develop a novel, implicitly typed λ–calculus for objects, by viewing these as extendible case–functions rather than as extendible records.

This novel view, allows to unify the concepts of function, object and process into one concept, that of a functional entity which is self contained and provided with a uniform communication protocol. We use this view to give a formal foundation for both sequential and concurrent object oriented languages. In the later case, we view objects as case–functions communicating asynchronously over unbounded channels.

Our calculus is a conservative extension of the polymorphic type system of Aiken and Wimmers, to include case–function extension and lazy data types. Its soundness is proven with respect to a semantical model based on ideals. Subtyping and case–function extension play a central role in our modeling of generalization/specialization and inheritance. To model self and self–class, our calculus includes recursive types. These are also necessary to model streams and provide the theoretical background for passing streams themselves as messages. We use higher order streams to express mobile systems. For the study of mobile systems we additionally devise a network calculus on the top of the lambda calculus.

The implicitly typed λ–calculus is accompanied with a decidable type inference algorithm, which always delivers the least type of a term (if this exists). An implementation of this algorithm was written in Common Lisp.

Acknowledgements

One small page is not enough to do justice to all the people that helped me directly or indirectly, with the production of this thesis.

My special thanks go to Manfred Broy. He gave me the opportunity to work in a highly qualified research group and guided my activity in the past four years. His own work provided much of the motivation of this thesis. Special thanks go also to Tobias Nipkow. His pertinent advice played an important role in the development of the object model presented in this thesis.

Particular thanks to Alexander Aiken, for our fruitful dialogue about his type system and his support in my own extensions, to Cliff Jones for showing me how objects can be modeled in π–calculus and to Robin Milner for introducing me in π–calculus and mobility.

Thanks go also to all of my colleagues from our department, especially to Ketil Stølen, Gheorghe Stefănescu, Cornel Klein, Max Fuchs, Franz Regensburger and Bernhard Reus for their comments and interest in my work.

On a personal note, I would like to thank my parents for their love and support. A final and special thank–you to my wife Anca, for reminding me of the better things in life and for believing in me when I didn't.

Contents

Chapter 1

Introduction

1.1 The Object Oriented Paradigm

The continuing development of programming languages and software technology allows the construction of increasingly complex systems. In order to master their complexity, methodologies are elaborated, which guide the analysis, design and implementation phases of these systems [96, 95, 30, 40]. Inevitably, both methodologies and programming languages are organized around a conceptual model or software paradigm.

In the last years a great popularity has gained the object oriented (OO) paradigm. Nowadays there is a great diversity of both OO methods and OO languages. Among the OO methods, probably the best known are the Object Modeling Technique of Rumbaugh [86], the Object Oriented Analysis and Design of Booch [13, 14], the Fusion method of Coleman [33], the Object Oriented Analysis and Design of Coad and Yourdan [32, 31], the Object Oriented Software Engineering of Jackobson [48] and the Object Oriented Analysis and Design of Martin and Odell [58]. Programming languages range form untyped [43, 54] to typed [88, 62] from class based [88, 62] to delegation based [29] from functional [54, 87] to procedural [88, 62] and from sequential [43, 88, 62, 29, 54, 87] to concurrent [10, 50, 4].

Given this diversity it is naturally to wonder what makes the OO paradigm so popular. For this purpose it is very useful to consider three different views or aspects of every reasonable complex system: the static or structural view, the functional or transformational view and the dynamic or temporal view.

1.1.1 Views of a Software System

The functional Model

The functional model concentrates on the data transformation aspect of a system, completely ignoring structural and temporal considerations. It captures *what* a system does without regarding when.

Systems are regarded as functions and complexity is managed by decomposing functions in sub-functions each performing a well defined task. This can be characterized by the following equation:

$$\begin{aligned}\mathsf{Functional\ Decomposition} = {} & \mathsf{Functions} \\ & + \mathsf{Sub\text{-}Functions} \\ & + \mathsf{Functional\ Interfaces}\end{aligned}$$

The above decomposition and the well behaving of the whole system is guaranteed by the following principle for managing complexity:

Definition 1.1 Functional abstraction

> The principle that any operation that achieves a well defined effect can be treated by its users as a single entity, despite the fact that the operations may actually be achieved by some sequence of lower level operations. □

In other words a function is completely characterized by its input/output behavior. This is also called referential transparency, and beside assuring a simple semantics, it allows the application of powerful verification techniques.

However, functional decomposition is guided by the transformation done on data and not by the objects which actually occur in the task to be modeled. This has two immediate consequences:

- The data structuring aspect, which normally is determined by the objects occuring in the system, is in this model very poor.

- The model itself, is extremely volatile to future changes, since later versions of a system may consider more or fewer characteristics of these objects.

Further, by ignoring the time sensitive behavior of the modeled objects, the dynamic aspect of the system is lost.

The Information Model

The information model represents the static (or structural or data) aspect of a system. Modeling the world by data, helps to capture the problem domain content. In this model, a system is decomposed by finding objects in the real world, and describing them with attributes. These objects are then classified according to the similarity of their attributes and relationships to other objects. Briefly, this decomposition process is given by the following equation:

$$\begin{aligned}\mathsf{Information\ Modeling} = {} & \mathsf{Objects} \\ & + \mathsf{Attributes} \\ & + \mathsf{Relationships}\end{aligned}$$

Among the relationships between object classes, an important role plays the containment or *part-of* relation and the generalization/specialization or *is-a* relation.

Structural decomposition is very general. As a principle for managing complexity, it is employed in most human activities.

Definition 1.2 Pervading methods of organization [16]

In apprehending the real world, people constantly employ three methods of organization, which pervade all of their thinking:

- The differentiation of experience into particular *objects* and their *attributes* – e.g. when they distinguish between a tree and its size or spatial relations to other objects.
- The distinction between *whole* objects and their component *parts* – e.g. when they contrast a tree with its component branches.
- The formation of and the distinction between different *classes of objects* – e.g. when they form the class of all stones and distinguish between them.

□

Definition 1.3 Association [39]

An association is a union or connection of ideas. □

By capturing the problem domain content, structural decomposition is very stable to future changes. This is the reason why it plays a central role in all of the OO methodologies. However, this modeling is incomplete, because it ignores the functional and the temporal aspects of a system. It only describes *which* are the entities involved in a system but not what are they doing and when.

The Dynamic Model

Temporal aspects of a system are more difficult to understand. This understanding is significantly simplified by first identifying the static structure of the system i.e. the objects and their interrelations. Then, one can examine the changes to the objects and their relationships over the time.

Those aspects of a system that are concerned with *time* and *changes* build *the dynamic model.* Changing over time is closely related with the functional aspects. Together they form the *behavior* of objects. Characterizing objects according to their behavior is also a very general principle of managing complexity.

Definition 1.4 Behavioral Classification [16]

Objects can be classified

- On the basis of immediate causation,
- On the similarity of evolutionary history (change over the time),
- On the similarity of function.

□

Objects interact by exchanging messages. Receiving a message is for an object an event determining a precise reaction and change of state (i.e. of attributes and relations). As with behavioral classification, communication with messages is a very common human activity. For example a message is defined in [39] as follows.

Definition 1.5 Message

A message is any communication written or oral sent between humans. □

Summarizing, the dynamic view describes the sequencing of operations based on events and states, defining the context of these events without regard for what the operations really do. It describes *when* without regard of what and which. Its understanding is however intimately related with the other views. The corresponding equation for the dynamic view can be given as follows.

$$\begin{aligned}\textsf{Dynamic Model} &= \textsf{Change Over Time}\\ &\quad + \textsf{Communication with Messages}\end{aligned}$$

1.1.2 The Object Oriented Methods

The successful application and acceptance of a method is determined by the scope, number and relative emphasis it places on the various modeling components.

For example, the Structured-Analysis/Structured-Design method supports like the OO methods all three modeling components. However, by putting the main emphasis on the functional aspect, it leads to models which are difficult to comprehend by the customers and extremely volatile to future changes.

On the other hand, the OO methods put the main emphasis on the structural aspect. A system is decomposed by identifying the objects in the real world. However, the static view of objects is augmented with functional and dynamic aspects. An object does not only contain attributes (i.e. a state) but also services (i.e. functions). By encapsulating attributes and functions together, the functional view is not anymore applied to the whole system, but localized in particular objects. This new augmented view of objects allows the application of two further principles of managing complexity:

Definition 1.6 Data Abstraction

> The principle of defining a data type in terms of the operations that apply to objects of the type, with the constraint that the values of such objects can be modified and observed only by the use of the operations. □

Definition 1.7 Encapsulation (Information Hiding)

> Encapsulation is the process of hiding all of the details of an object that do not contribute to its essential characteristics. □

By adding behavior, objects become dynamic. An object does not call anymore a procedure from another object, but it sends a message with the requested task. The receiver object itself "knows" which service has to perform the required task. This permits to augment the *generalization/specialization principle* (i.e. the third pervading method of organization) and derive a new principle of managing complexity[1]:

Definition 1.8 Inheritance

> A mechanism for expressing similarity among classes simplifying definition of classes similar to one(s) previously defined. It portrays generalization/specialization *making common attributes and services explicit* within a class hierarchy or lattice. □

[1] A precise analysis of the relation between inheritance and generalization/specialization is given in the next chapter.

The dynamic aspect like the functional one is distributed among objects. These communicate and synchronize each other with messages.

In summary, the principles for managing complexity used by the OO methodology are given in the following table.

Principle		Purpose
Abstraction	Procedural	Define Objects
	Data	
Encapsulation		Implement Abstraction
Classification	Structural	Define and Compare Classes of Objects
	Behavioral	
Association		Define Interrelations
Communication with Messages		Define Interactions

All these principles are implied by the following equation of the OO approach:

$$\begin{aligned}\textsf{Object Oriented} &= \textsf{Classes and objects}\\ &+ \textsf{Inheritance}\\ &+ \textsf{Communication with messages}\end{aligned}$$

In contrast to Structured-Analysis/Structured-Design the OO–methodology offers a uniform framework for the analysis, design and implementation of a system. The design and implementation phases extend the analysis model with new objects and with design/implementation details. Further, by encapsulating attributes and functions in objects it also assures a good stability to future modifications.

1.2 Current Work in Formal Foundations

1.2.1 The Sequential World

Despite of the terminology of *message passing* most existing object oriented languages are sequential in nature [43, 88, 62, 29, 54, 87]. Their formal foundation was the focus of intensive research activity.

In 1984 Cardelli suggested that basic concepts of OO programming could be understood type-theoretically, using the following rough correspondence [22, 23]:

Object Oriented Languages	Typed Lambda-Calculus
class	record type
object	record
subclass	subtype
method	function
method call	function call

This correspondence was extended or slightly modified during the intervening years [18], [26, 25, 24], [35, 21], [69, 72], [47], [80], [81], [91, 93]. Some of these extensions and modifications are given in the following Table.

Object Oriented Languages	Typed Lambda-Calculus
class	parameterized record definition
class interface	recursive record type
attribute	bound variable
object modification	functional record update
method inheritance	cascaded record construction
self	record recursion variable
selfType	type recursion variable

New type theoretical approaches were also developed for example in [1, 72, 79, 77].

1.2.2 The Concurrent World

Concurrent OO languages are by far less represented as their sequential counterparts. Among them, probably best known are the Actor languages [3, 4], POOL [9, 10, 11], $\pi o\beta\lambda$ [50, 51], Abacus [75, 74] and Maude [60].

There is also less consensus for their formal foundation. This is in particular a consequence of the various synchronization mechanisms which can be used to explain message passing. For example POOL, $\pi o\beta\lambda$ and Abacus use synchronous communication while Actor languages use asynchronous communication.

1.3 Claims and Outline of the Thesis

The purpose of this thesis is to give a formal foundation for the principal concepts of the OO paradigm like objects, classes, message passing, inheritance and polymorphism in a concurrent, asynchronous setting. By interpreting objects as functions communicating asynchronously over unbounded channels we extend the results from the sequential world to the concurrent one. Our major claims are that:

- Objects, functions and processes can be unified by using an appropriate type system and communication model.
- This model gives a better description of the objects' behavior by unifying their functional and the dynamic aspects.
- This model can be effectively used in conjunction with OO methods and leads to a better understanding and design of software systems.

These claims are materialized in our thesis as follows.

In Chapter 2 we critically review the definitions of objects, classes, inheritance, polymorphism and associations as given in some of the best known OO methods and OO programming languages, and extract our own informal definitions. This is necessary, because the great diversity of OO methods and OO programming languages, often not accompanied by a clear formal foundation, causes some confusion about the OO terminology. The same names often denote slightly different things in different OO dialects.

In Chapter 3 we develop the syntax and the semantics of the typed lambda calculus which forms the basis of our object model. The syntax is given as a type system i.e. as a system of rules which simultaneously defines the well formed terms and their associated types.

Starting with Cardelli, most typed lambda calculi for objects identify objects with records, classes with parameterized record definitions and class interfaces with record types. Generic classes are polymorphic class definitions and inheritance is a cascaded record construction in presence of subtyping. Given a record term, the reconstruction of its associated type, is not a trivial task in a type discipline which includes parametric polymorphism and subtyping. To make this problem decidable, most type systems for records require explicit type information for the variables inside the terms [18, 26, 25, 24, 35, 21, 69]. Such systems are therefore known as explicitly typed. However, since polymorphic languages involve quite a bit of type information, the process of declaring the type of variables is usually very tedious. This is the motivation of more restricted languages which make this declaration process optional. Type systems for such languages are known as implicitly typed. They all have a decidable type inference algorithm. The most important example of such a language was obtained by encoding record inclusion with parametric polymorphism [81, 91, 93]. This encoding is however very rough, because the depth of inclusion is limited to the depth of quantification i.e. to one.

In contrast to the record approach, we model objects in a new way, as case functions. This modeling has the advantage that it can be easily extended to describe objects working on message histories. In our model classes are parameterized case–function definitions and class interfaces are particular function types. Moreover, inheritance is a cascaded case–function construction in presence of subtyping and generic classes are polymorphic class definitions.

Our typed lambda calculus is defined by an implicitly typed system. It includes union, intersection and conditional types beside the more usual function and constructor types. As in the record approach, it also includes recursive type definitions, subtyping and parametric polymorphism. A very important characteristic of our calculus is its decidability. In Section 3.2.7 we give a type inference algorithm which infers the least type of a term if this is well formed; otherwise the term is rejected.

The semantics of our language is given in a Curry–style. This means, the interpretation of a term e is some element of an untyped model U given by a semantical function $[\![-]\!]$ which is defined by induction on the structure of terms. Typing is a matter of predication; a typing statement involving a term e is an assertion about $[\![e]\!]$. According to this point of view, types are predicates (i.e. sets) and subtyping is inclusion.

In Chapter 4 we use the lambda calculus to define our object model. This model is purely functional and asynchronous. It is therefore most closely related to Actor languages. However, in contrast to Actor languages, we use a typed formalism. Moreover, the set of messages that are sent but not yet received i.e. the message histories are explicitly modeled by infinite lists of messages, also known as streams [17]. The processes are continuous case–functions operating in a sequential manner on streams. The main advantage of this approach is that it effectively unifies the concepts of object and process into one concept, that of a functional entity which is self contained and provided with a unified communication protocol. Based on this model we give a precise definition for objects, communication with messages, classes, inheritance and parameterization.

Since the major interest for using objects, is to construct systems where each object contributes to the overall behavior by interacting with one another, Chapter 5 is devoted to the study of object configurations. These configurations are classified by methodological criteria in aggregation systems and mobile systems. The first ones are hierarchical systems which can be treated as a whole. The second ones are systems in which the communication partners can change on the basis of computation and interaction. We start by discussing aggregation networks. We give some typical examples and investigate their properties. We then analyze mobile systems. We discuss their properties and show how mobility can be achieved with higher order streams. This is an important result, since it is usually believed that stream processing function configurations are static. Finally for the study of the logical properties necessary to express mobility, we devise a network calculus. This calculus does not view a system as a function, but as a collection of equations with designated input/output channels.

In Chapter 6 we present a system which implements the type inference algorithm from Chapter 3. We give some typical examples together with their types as inferred by the system.

Finally in Chapter 7 we summarize the main results of the thesis and describe additional areas for further study. For each of these areas we outline some preliminary ideas.

Chapter 2

Basic Principles

From the pioneering work in CLU [56] and Smalltalk until the recent explosion of OO programming languages, the OO concepts evolved and diversified. OO languages range today from untyped to typed, from sequential to parallel from class based to delegation based etc. As a consequence, the same names denote slightly different things in different dialects. This was also amplified by the lack of a clear formal foundation. In this section we define the OO concepts which we model later by comparing definitions from several well known OO literature sources. Since our framework is functional we will have a preference for functional dialects.

In subsection 2.1 we define the objects i.e. the basic building blocks for OO systems. In order to understand and manipulate them, objects are classified according to their similarities. Object classes are defined in subsection 2.2. Classification was also heavily used outside the OO community in the theory of (abstract) data types. In subsection 2.3 we point out the difference between the abstraction/encapsulation mechanisms used by abstract data types and classes. In particular we show that classes *are not* abstract data types (ADTs) implementations as it is usually believed in the OO community. To simplify the comparison we use a functional, non–concurrent dialect. Since the comparison is rather technical it can be skipped at a first lecture. In section 2.4 we emphasize the difference between generalization/specialization and inheritance. Generalization hierarchies allow to define polymorphic methods i.e. methods which occur with the same name but with different bodies at different levels of the hierarchy. This kind of polymorphism is known as subtype polymorphism. Another form of polymorphism is the parametric polymorphism. It allows to define generic classes as for example generic lists. Parametric polymorphism is discussed in section 2.5. Finally, in section 2.6 we discuss associations. They allow to describe statically the potential object configurations.

2.1 Objects and Messages

In every OO methodology a system is conceived, by analogy with the real life, as a collection of interacting objects. The objects are abstractions of their real life counterparts. They exhibit only those properties which are meaningful for the system. Let us look to the object definitions given in some of the most popular OO sources:

[86] An object is a concept, abstraction or thing with crisp boundaries and meaning for the problem at hand. All objects have identity and are distinguishable.

[13] An object has state, behavior and identity; the structure and behavior of similar objects are defined in their common class; the terms instance and object are interchangeable.

[9] An object is an integrated unit of data and procedures acting on these data. The data in the objects is stored in variables not accessible to other objects: they are strictly private.

The first definition is deliberately very general and thought for the analysis phase of systems, while the other ones are more closely related to the design and implementation phases. However, all agree that objects are uniquely identified, have some internal data and exhibit a particular behavior. More precisely, we define objects as follows:

Definition 2.1 Objects

An object is a clearly delimited software entity which has:

- a *state* i.e. it contains some private data,
- a *behavior* i.e. it can execute certain procedures,
- a *unique identity*.

□

The private data is stored in variables local to the object also known as *instance variables* or *attributes*. Methods and programming languages differ with respect to the nature of the private data.

In pure OO languages like Smalltalk [43], POOL [9] or Actor Systems [4], variables contain only references to other objects (i.e. object identifiers). Only these are manipulated by objects. As a consequence all data values, even booleans or integers, have to be modeled as objects. The addition 2+3 written as `2!add(3)` is performed by sending to the object referenced by `2` the message `add` with the reference `3` as parameter. As a result of the addition the object returns the reference 7 of the object behaving as the integer 7[1].

Hybrid OO languages like C++ [88], Objective-C [37] or Eiffel [62] have the standard data types like integers, characters or booleans built in. Their elements are values and they are distinguished from (object) references.

Finally, in OO methods like [86], it is suggested that attributes should contain only data values. References should be treated as elements of the (directed or undirected) relations which occur between classes of objects. They are not subordinated anymore to objects but get instead a separate status.

The procedures inside an object are known as the object *methods*. Only these are accessible to the outside world. A graphical representation of objects is given in Figure 2.1.

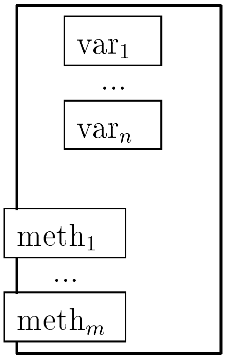


Figure 2.1: Object structure

A particular method is activated by sending the object a message. Again, citing from the literature:

[9] The only way objects can interact is by sending messages to each other. Such a message is in fact a request from the sender for the receiver to execute a procedure. Such procedures, which are executed in response to messages are called *methods*. The receiver decides whether and when it executes such a method and in some cases[2] it even depends on the receiver which method is executed.

[37] An object is requested to perform one of its operations by sending it a message telling the object what to do. The receiver responds to the message by executing this operation and returning control to the caller.

[1] In a pure functional OO language 2 and 3 are the objects themselves instead of references. Hence add would return the object 7.

[2] When using inheritance.

[94] Computation or information processing is represented as a sequence of message passing among objects.

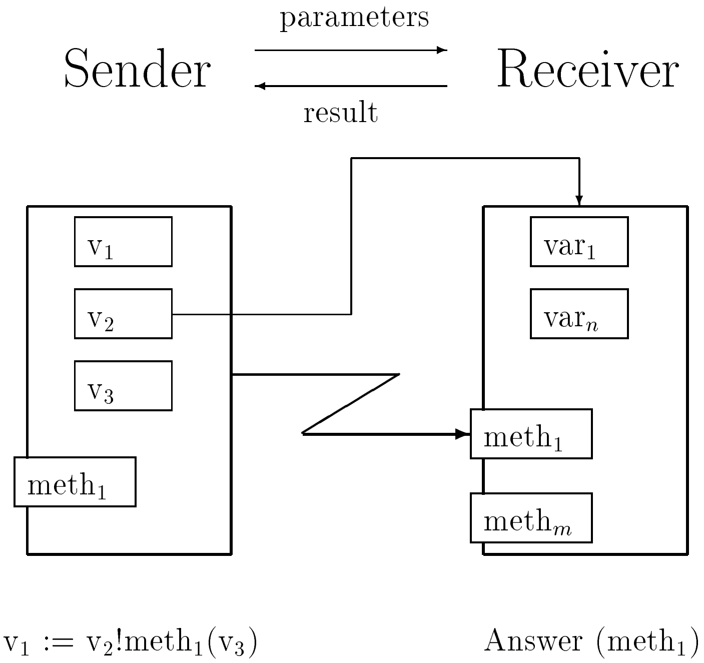


Figure 2.2: Communication between objects

We can conclude that:

Definition 2.2 Communication and messages

Objects can interact by exchanging messages according to a precisely determined message interface. A message consists of a method name and actual parameters to be passed the method. The receiver alone determines when and which method to execute in response to a message. The method can return a result which is passed back to the sender. □

Hence, interaction among objects respects the abstraction/encapsulation principles. The objects alone are responsible to maintain their local data in a consistent state. A graphical representation of communication inspired from [9] is given in Figure 2.2. The object pointed by v_2 receives the message $\text{meth}_1(v_3)$. This is written as $v_2!\text{meth}_1(v_3)$. In response to the message this object activates the corresponding method. The result is returned to the caller which updates v_1.

Objects have a dynamic nature. Citing again:

[61] Object is a run time notion; every object is an instance of a certain class, created at execution time and made of a number of fields.

[9] Objects are entities of dynamic nature. At any point in the execution of a program a new object can be created, so that an arbitrary large number of objects can come into existence. Objects are never destroyed explicitly. However, they can be removed by garbage collection if it is certain that this will not influence the correct execution of the program.

We can conclude that objects do not explicitly occur in a program. More precisely:

Definition 2.3 Dynamic nature of objects

Objects are *semantic entities* which occur only in the meaning of a program. They are dynamically created and destroyed by executing i.e. by interpreting the program. □

2.2 Classes

As we already pointed out objects are semantic entities. On the *syntactic level,* the corresponding notion is that of a class. Citing from the literature:

[32] A class is a description of one or more objects with a uniform set of attributes and services, including a description of how to create new objects in the class.

[86] An (object) class describes a group of objects with similar properties (attributes), common behavior (operations), common relationship to other objects and common semantics.

[9] In order to describe systems with many objects, the objects are grouped in classes. All elements (the instances) of a class have the same name and types for their variables (although each object has its own set of variables) and they all execute the same code for their methods. In this way, a class can serve as blueprint for the creation of its instances.

[61] A class is an ADT implementation[3], not the ADT itself. It is a language construct combining the module and type aspect. Classes are a purely static description of a set of possible objects – the instances of a class. At run time we have only objects, in the program we see classes.

[35] A class is a parameterized object definition. Different instantiations of the parameters permit the creation of different objects.

[3] As we will show in the next section this is not true.

Excepting the last one, all the above definitions exhibit a dual understanding and use of a class: as a *type* and as a *function* (or module).

As a type it is used for classification purposes. In this case:

- A class describes the properties (or attributes) and the behavior (or methods) of its own objects.

Hence a class can be understood as a predicate selecting from all objects with the same interface those behaving as stated in the class definition.

As a function it is used as a generator. In this case:

- A class is used to create new objects, behaving as stated in the class definition.

The class definition however, is actually encoded in the object creation function. Trivially this function generates all objects behaving as described in the class definition. Hence we can state that:

Definition 2.4 Class

> A class is a parameterized object definition. Different instantiations of the parameters permit the creation of different objects. In other words a class is the definition of an object creation function. □

In most OO programming languages part of the definition is implicit in the specification of the attributes and of the methods. Hence when a user defines his own object creation function he only describes how to assign initial values to the attributes.

Although in some languages (like Smalltalk for example) classes are allowed to be dynamically created we will consider them to be *static*.

If we view a class as a function it is natural to ask what is the type of the class. Usually this type is exactly the *syntactic interface* of the objects in this class. Distinguishing between classes and their types Cook, Hill and Canning put an end to the long lasting confusion between inheritance and subtyping in [35]. A more detailed presentation is given in section 2.4.

Most OO methodologies support a graphical notation for classes which is actually nothing more than the type (or syntactic interface) of the objects in this class. For example [86] suggest for classes the notation from Figure 2.3.

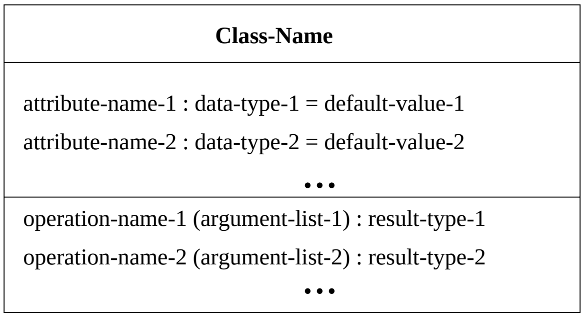

Class-Name
attribute-name-1 : data-type-1 = default-value-1
attribute-name-2 : data-type-2 = default-value-2
•••
operation-name-1 (argument-list-1) : result-type-1
operation-name-2 (argument-list-2) : result-type-2
•••

Figure 2.3: OMT Notation for Classes

A class is represented as a box which may have as many as three regions. The regions contain from top to bottom: the class name, a list of attributes and a list of operations. Each attribute may be followed by optional details such as type and default value. Each operation name may be followed by optional details such as argument list and result type. Attributes and operations may not be shown; it depends on the level of detail.

A class diagram corresponds to an infinite set of instance (i.e object) diagrams. Instance diagrams are useful for documenting test cases (especially scenarios) and discussing examples. Figure 2.4 shows a class (left) and two possible instance diagrams it describes.

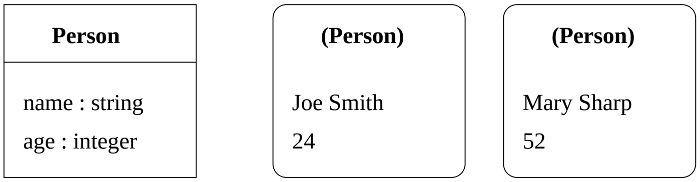


Figure 2.4: The Class Person and Two Instances

2.3 Classes and Abstract Data Types

One may argue, the above definitions are nothing but a formulation of well known concepts in new words: classes are actually abstract data types, objects are abstract data type values and message sending is procedure call. So, what is the difference between abstract data types and classes? In this section we try to answer this question.

Both abstract data types and classes have a type aspect i.e. they both can be understood as a set of objects with uniform behavior. However, abstract data types and classes observe this behavior differently. Abstract data types regard objects as values. Hence, uniform behavior means they have the same set of applicable operations. Classes regard objects as procedures[4]. Hence, uniform behavior means they respond in the same way to the same input values. Both views allow *data abstraction* by creating objects with *abstract constructors* and observing them with *abstract observers*. However, the observers are in the first case functions while in the second case messages.

Strongly related to abstraction is the concept of *encapsulation.* The relation between abstraction and encapsulation is similar to the relation between *specification* and *implementation.* A specification can be regarded as an *incomplete program description.* It defines only the "externally observable" behavior of objects and it is this behavior on which other objects are expected to rely on. An implementation on the other hand, can be regarded as a *complete program specification.* However, by *encapsulating* or hiding the additional details, other objects are forced to use only the properties given by the abstract specification. As a consequence, implementations can be changed without affecting the correctness of the other modules. Because of the strong connection between abstraction and encapsulation, abstract data types and classes use, as expected, different encapsulation principles. In the first case, encapsulation is achieved by hiding the *concrete representation* of the data values. In the second case, encapsulation is achieved by using *a functional (or procedural) interface* (i.e. by λ–abstraction).

In the following subsections we compare the abstraction/encapsulation principles of classes and abstract data types by means of an example: the specification and implementation of lists[5]. For the simplicity of exposition we ignore object identities and take a purely functional view of objects. For specifications we use a first order logic of higher order functions with a relatively powerful type system.

2.3.1 The List Example

The abstract constructors for integer lists are nil, which is the empty list, and cons, which takes a list and an integer and delivers a new list with the integer in front of the list. The observers are null which is true for empty lists and false for cons lists, head which returns the head of the list and tail which returns the rest of the list. Postponing for the moment what observers really are, an abstract specification for lists can be given as a matrix with constructors on one axis and observers on the other one. Each cell of the matrix defines the behavior of a constructor/observer pair, as shown in Figure 2.5.

[4] We will therefore use sometimes the phrase Procedural Data Types (PDT) instead of classes.

[5] This example is inspired from [34].

const obs	nil	cons l x
null	true	false
head	$\perp$	x
tail	$\perp$	l

Figure 2.5: Abstract specification matrix

2.3.2 The ADT Variant

In this case the observers are functions. Their definition corresponds to the partitioning of the matrix into horizontal slices as shown in Figure 2.6.

const obs	nil	cons l x
null	true	false
head	$\perp$	x
tail	$\perp$	l

Figure 2.6: Decomposition into observers

The equivalent axiomatic specification is given below. It consists of two parts: a signature listing the names and types of the available operations and a set of axioms describing the behavior of the operations. The signature and the axioms are also called the *syntactic* and the *semantic interface* respectively.

```
ListADT = ∃t. {

    nil  : t;
    cons : t ⟶ Int ⟶ t;
    head : t ⟶ Int;
    tail  : t ⟶ t;

    null  : t ⟶ Bool;
    eq    : t ⟶ t ⟶ Bool;

    axioms ∀ x : Int, l : t in
          head(nil) = ⊥;
          head(cons l x) = x;

          tail(nil) = ⊥;
          tail(cons l x) = l;

          null(nil) = true;
          null(cons l x) = false;

          eq(nil)(l) = null(l);
          eq(cons l x)(l') = ¬null(l') ∧ x == hd(l') ∧ eq(l)(tail(l'));
}
```

The existential quantifier is written in algebraic specification languages (e.g. OBJ) inside brackets as **sort t**. An implementation for ListADT is considered in this case to be an algebra i.e. a set together with operations on this set.

The existential quantification over types allows to construct and manipulate algebras syntactically. If the above specification is abbreviated as $\exists t.\{\Sigma(t), Ax(t)\}$, then its signature is the existentially quantified record type $\exists t.\{\Sigma(t)\}$. To simplify the notation we will write it as $\exists t.\Sigma(t)$. The elements of this type must contain a representation type τ for t and a record of representation functions $A : \Sigma(\tau)$. In other words they are pairs $\mathcal{A} =< \tau, A >$. Since from $\Sigma(\tau)$ it is usually impossible to recover $\Sigma(t)$ (τ may already occur in $\Sigma(t)$) these pairs are written as $< t = \tau, A : \Sigma(t) >$.

The formulas-as-types analogy [70] immediately provides us the necessary typing rules for introducing/eliminating syntactic algebras. They are the introduction/elimination rules of an existential quantifier[6]:

$$(\exists i)\quad \frac{\Gamma \rhd A[\tau/t] : \Sigma[\tau/t]}{\Gamma \rhd < t = \tau, A : \Sigma > : \exists t.\Sigma(t)}$$

$$(\exists e)\quad \frac{\Gamma \rhd \mathcal{A} : \exists t.\Sigma \quad \Gamma, x : \Sigma \rhd N : \rho}{\Gamma \rhd \textbf{open } \mathcal{A} \textbf{ as } < t, x > \textbf{ in } N : \rho} \quad \left\{ \begin{array}{l} t \text{ not free in } \rho \text{ or } \Gamma(y) \\ \text{where } y \neq x \text{ is free in } N \end{array} \right.$$

[6] An expression of the form $\Gamma \rhd t : \sigma(t)$ with $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ is called a *typing assertion*. It intuitively says that if variables $x_1, \ldots, x_n$ have types $\sigma_1, \ldots, \sigma_n$ then t is a well formed term of type σ. Γ is a *type assignment* or *type context* with no x_i occuring twice. See Section 3.2 or [46, 70] for more details.

The axiomatic semantics of these rules is given by the following law:

$$\textbf{open} \;< t = \tau,\; A : \Sigma(t) > \;\textbf{as}\; < s,\; x : \Sigma(s) > \;\textbf{in}\; N = N[\tau/s][A/x]$$

N can be any expression of the language. The variables s and x are bound to the scope of N. As a consequence, terms built with **open** are unique modulo α -equivalence. This use of s as a bound variable makes it "abstract" and different from every other type[7]. The side conditions in rule $(\exists e)$ play a central role on the observability (or abstractness) of the elements of τ. First, no free identifier $y : \sigma$ is allowed in N if t occurs free in σ. As a consequence the elements of the abstract type t can be accessed only by the explicitly declared operations x. Second, t is not allowed to occur free in the type of N. This prevents values of the abstract data type to be visible outside the scope of the declaration. Let us denote the subset of τ reachable with A by τ_Σ. Then each algebra $\mathcal{A} =< \tau, A : \Sigma(\tau) >$ is an implementation of ListADT only if it additionally satisfies $Ax(\tau_\Sigma)$[8].

As an example, suppose the type List was declared as follows:

```
data List = Nil | Cell(Integer, List)
```

Then it is easy to prove that the algebra $\mathcal{A}$ given below is an implementation of ListADT.

```
A = <
      t = List,

      nil = Nil;
      cons l x = Cell(x,l);

      head(l) = case l of
            Nil       ⇒ ⊥
            Cell(x,l) ⇒ x;
      tail(l) = case l of
            Nil       ⇒ ⊥
            Cell(x,l) ⇒ l;

      null(l) = case l of
            Nil       ⇒ true
            Cell(x,l) ⇒ false;
      eq(l)(m) = case l of
            Nil       ⇒ null(m)
```

[7] The **open** expression is written in [73] as **abstype** t **with** $x : \Sigma(t)$ **is** $\mathcal{A}$ **in** N.

[8] Sometimes it is also necessary to replace in $Ax(\tau_\Sigma)$ the equality symbol on τ with a congruence eq on τ. Then $\mathcal{A}$ has to satisfy $Ax(\tau_\Sigma)[eq/=]$.

```
        Cell(x,l') ⇒ case m of
                          Nil        ⇒ false
                          Cell(y,m') ⇒ x==y ∧ eq(l')(m')
>
```

The representation type List is given by the data type definition as a labeled union with cases Nil and Cell. This automatically introduces a case functional which is used in the definition of the observers head, tail, null and equal. The function $\perp$ is defined to not terminate. The abstract constructors nil and cons are taken to be the same as Nil and Cell. Since each use of $\mathcal{A}$ has to be of the form **open** $\mathcal{A}$ **as** $< t,\ x : \Sigma(t) >$ **in** N, the list representation is hidden. This has the following beneficial consequence: it can be changed in the future without affecting N. Hence, encapsulation is achieved in this case by *type abstraction*.

In algebraic formalisms using initial semantics [41] the **open** functional corresponds to a parameterized specification. In that case N is also an algebraic specification and the actual parameters are initial algebras. As expected, they must contain the necessary operations and satisfy the formal parameter axioms[9].

A restricted form of ADTs is given by type classes [90, 76, 45, 44]. In this case, all algebras from a type class must have a different representation type. This allows, for each context, the automatic inference of the right algebra. As a consequence no explicit **open** operation is necessary. Moreover, N does not need to be a specification.

Since axioms cannot be checked automatically, programming languages use only the signature of an interface. In most of them, similarly to algebraic specifications, signatures do not live in the same universe as the other types. A typical example is ML where the **open** functional is known as a *functor* in order to distinguish it from "normal" functions. However, in [73] Mitchell and Plotkin develop the language SOL in which existential types live in the same universe as the other types. As a consequence, algebras may be passed as arguments to functions or returned as results. A similar approach is taken by Cardelli and Wegner in [26].

2.3.3 The CLASS Variant

In this case the observers are messages and the objects are procedures containing a separate branch for each possible observation[10]. One can also view objects as procedures with multiple entry points[11]. These are modeled in typed functional languages by *records with functional fields*. Their definition corresponds to the

[9] To allow different instantiations, formal parameters are loosely interpreted.

[10] This view will prove to be particularly useful when considering concurrent OO programs.

[11] Both views were already suggested in [97].

vertical slices obtained by decomposing the specification matrix into constructors as shown in Figure 2.7,

obs \ const	nil	cons l x
null	true	false
head	⊥	x
tail	⊥	l

Figure 2.7: Decomposition into constructors

In this case each constructor is converted into a template or *class* and the arguments of the constructors become the *local state* or *instance variables* of the functional objects.

Let us now examine the axiomatic specification.

```
ListPDT = {
    data ListPdt = μt. {
        cons : Int ⟶ t;
        head : Int;
        tail : t;
        null : Bool;
        eq   : t ⟶ Bool;
    };
    Nil  : ListPdt;
    Cell : Int × ListPdt ⟶ ListPdt;
    Cell hidden
    axioms ∀ x : Int, l : ListPdt in
        Nil.head          = ⊥;
        Nil.tail          = ⊥;
        Nil.cons(x)       = Cell(x,Nil);
        Nil.null          = true;
        Nil.eq(l)         = l.null;

        Cell(x,l).head    = x;
        Cell(x,l).tail    = l;
        Cell(x,l).cons(y) = Cell(y,Cell(x,l));
```

```
          Cell(x,l).null      = false;
          Cell(x,l).eq(l')    = ¬l'.null ∧ x == l'.head ∧ l.eq(l'.tail);
}
```

As with the specification ListADT, the signature is obtained by erasing the axioms. However, the signature of ListPDT does not classify algebras but object creation functions for the list objects. The recursive type ListPdt of list objects closely resembles the signature of ListADT. If $f : t \rightarrow \sigma(t)$ occurs in the signature of ListADT then $f : \sigma(t)$ occurs in ListPdt. With one exception. The constant nil : t occurring in ListADT does not occur in ListPdt. Instead, it is made an object creation constant Nil : ListPdt. But there is another important difference. The binding operator for t. In the first case it is an *existential quantifier*. As a consequence, ListADT classifies *algebras* owning a concrete representation type for list elements. The **open** functional hides this representation. In the second case the binding operator is the *recursion operator*. As a consequence, ListPdt does not classify algebras but *recursive records*. These records are the list elements. Although their internal representation is hidden with a functional interface, lists are not anymore abstract objects but multi-procedures. This is the key difference between abstract data types and classes. The *encapsulation principle* is different. In the first case each function inside the abstracted algebra knows the representation of all of its arguments. In the second case, only the representation of the first argument is known. For the other arguments it is only known that they have a corresponding syntactic interface.

It is easy to check that the following function Nil is an implementation of the above specification.

```
Nil = fix λself.{
      null   = true
      head   = ⊥
      tail   = ⊥
      cons   = λy. Cell(y,self)
      eq     = λm. m.null }

where {

Cell(x,l) = fix λself.{
      null   = false
      head   = x
      tail   = l
      cons   = λy. Cell(y,self)
      eq     = λm. ¬ m.null ∧ (x == m.head) ∧ (l.eq(m.tail)) }}
```

Nil is the only visible class constructor. As a consequence, list objects are created, inspected or modified by sending it messages. Encapsulation is achieved by λ–abstraction.

Note that the definition of Cell uses two levels of recursion. The first one is due to the recursive use of self and it is also known as *object recursion*. The second one is due to the recursive use of Cell and it is known as *class recursion*[12]. An adequate theory of procedural data abstraction (for short PDT) for a non–concurrent, functional setting is given by *closures* [80].

2.3.4 Comparison

In the table 2.1 we summarize the most important aspects of both formalisms. We assume that X_i does not contain the abstracted type t and that Y_i is arbitrary. We also abbreviate $\{x_i \mid i \in I\}$ by x_i. This table clearly reveals that abstract data types are different form classes. However, it is natural to wonder if they are isomorphical. In that case we could use only one formalism and automatically translate the implementations into the other formalism if necessary.

Let us first give a translation function pdtadt: ListPDT $\longrightarrow$ ListADT mapping each record Nil satisfying ListPDT into an algebra $\mathcal{A}$ satisfying ListADT.

$$\begin{array}{lll}
\mathsf{pdtadt(Nil)} = < & \mathsf{t} & = \mathsf{ListPdt}, \\
 & \mathsf{nil} & = \mathsf{Nil}; \\
 & \mathsf{cons} & = \lambda \mathsf{l} : \mathsf{ListPdt}.\ \lambda \mathsf{x} : \mathsf{Int}.\ \mathsf{l.cons(x)}; \\
 & \mathsf{head} & = \lambda \mathsf{l} : \mathsf{ListPdt}.\ \mathsf{l.head}; \\
 & \mathsf{tail} & = \lambda \mathsf{l} : \mathsf{ListPdt}.\ \mathsf{l.tail}; \\
 & \mathsf{null} & = \lambda \mathsf{l} : \mathsf{ListPdt}.\ \mathsf{l.null}; \\
 & \mathsf{eq} & = \lambda \mathsf{l} : \mathsf{ListPdt}.\ \lambda \mathsf{m} : \mathsf{ListPdt}.\ \mathsf{l.eq(m)}; \\
 > & &
\end{array}$$

This is indeed an implementation of ListADT if the above functions satisfy the ListADT axioms on all elements reachable with nil, cons and tail. But this is the same as to require that Nil satisfies the ListPDT axioms. Hence $\mathcal{A}$ implements ListADT.

In general, given an object creation function satisfying a class specification we can automatically generate an algebra satisfying the associated abstract data type specification. Two different object creation functions are mapped in two different algebras. So, only loosely speaking, each object creation function is an implementation of an abstract data type specification.

The other way around, let us define adtpdt: ListADT $\longrightarrow$ ListPDT as follows:

[12] We will use a related technique for our concurrent framework in Section 4.2. In that case object recursion is eliminated by the use of streams.

	ADT	PDT
specification	$\exists t. < \Sigma(t), Ax(t) >$ where $\Sigma(t) = \; o_i : t \rightarrow Y_i;$ $c_i : X_i \rightarrow t;$	$< c_i : X_i \rightarrow T, \; Ax(T) >$ where $\Sigma'(t) = o_i : Y_i;$ $T = \mu t.\Sigma'(t)$
implementation	algebra $\mathcal{A} =< \tau, A : \Sigma(\tau) >$	function tuple $c_i : X_i \rightarrow T$
abstraction	type abstraction	procedural abstraction
encapsulation	hidden representation of elements **open** $\mathcal{A}$ **as** $< x, t >$ **in** N	hidden state and function representation $c_i : X_i \rightarrow T$
objects	abstract elements $e : t$	multiprocedures $m : T$
creation	abstract constructors $c_i : X_i \rightarrow t$	object creation functions $c_i : X_i \rightarrow T$
observation	abstract observers $o_i : t \rightarrow Y_i$	messages $m.o_i : Y_i'$
protection level	algebra	object
security	ok	dangereous if $m : T$ can be constructed without using c_i

Table 2.1: Comparison between ADTs and PDTs

```
adtpdt(𝒜) = open 𝒜 as
      <t,
        nil_𝒜  : t;
        cons_𝒜: t ⟶ Int ⟶ t;
        head_𝒜 : t ⟶ Int;
        tail_𝒜 : t ⟶ t;
        null_𝒜 : t ⟶ Bool;
        eq_𝒜   : t ⟶ t ⟶ Bool;
      >
in Nil where {
      Nil    = abs(nil_𝒜);
      abs(l) = { cons = abs ∘ cons_𝒜(l);
                 head = head_𝒜(l);
                 tail  = abs(tail_𝒜(l));
                 null  = null_𝒜(l);
                 eq    = eq_𝒜(l) ∘ rep };
      rep(r) = if r.null then nil_𝒜 else cons_𝒜(rep(r.tail)) (r.head)
}
```

In order to define adtpdt we need two auxiliary functions: $\mathsf{abs} : \mathsf{List}_{\mathcal{A}} \longrightarrow \mathsf{ListPdt}$ and $\mathsf{rep} : \mathsf{ListPdt} \longrightarrow \mathsf{List}_{\mathcal{A}}$ such that $\mathsf{rep} \circ \mathsf{abs} = \mathsf{id}_{List_{\mathcal{A}}}$. We can not expect that $\mathsf{abs} \circ \mathsf{rep} = \mathsf{id}_{ListPdt}$ because ListPdt also contains records which are not valid lists. Then we define the fields of the recursive record as in Figure 2.8:

$$\begin{tikzcd}
\mathrm{Int} \arrow[r, "\mathrm{cons}_{\mathcal{A}}(l)"] \arrow[dr, dashed, "\mathrm{cons}"'] & \mathrm{List}_{\mathcal{A}} \arrow[d, "\mathrm{abs}"] & \mathrm{List}_{\mathcal{A}} \arrow[r, "\mathrm{eq}_{\mathcal{A}}(l)"] & \mathrm{Bool} \\
 & \mathrm{ListPdt} & \mathrm{ListPdt} \arrow[u, "\mathrm{rep}"] \arrow[ur, dashed, "\mathrm{eq}"'] &
\end{tikzcd}$$

Figure 2.8: Definition of the functional fields

It is easy to verify that the record Nil satisfies the axioms from ListPDT.

In general, if all functions in the abstract data type signature contain the abstract sort t only once in an argument position then abs can be defined entirely automatic. If this is not the case (e.g. the function eq) we also need a representation function rep. This function however, cannot be defined automatically. It needs additional information in order to discriminate the objects on their constructor basis. For lists we “knew” that the record field null discriminates Nil from the other records.

In conclusion, we can automatically translate each object creation function into an algebra. The other way arround needs however, discrimination information. Even if this would be the case, $\mathsf{pdtadt}(\mathsf{adtpdt}(\mathcal{A})) \neq \mathcal{A}$ because their sorts are different. The most important difference between abstract data types and classes is their *encapsulation* mechanism. The protection achieved by abstract data types is at type level while the protection achieved with classes is at object level. Hence, procedural data abstraction leads to a more fine grained protection as type abstraction. While for sequential programs, the advantage of a finer protection is not so evident, this proves to be very important in the case of concurrent systems.

2.4 Inheritance and Subtyping

One of the merits of OO methodologies, is their effort to reduce the conceptual gap between different phases of software development, by unifying related concepts occuring in each of their phases. However, the price for this unification is often a loss of precision and also a source of confusion. The best example for such a confusion is inheritance.

From the programmer's (or implementor's) point of view, it is very convenient when defining a new class, to start with all the ingredients (attributes and methods) of an existing class, and to add some more and/or possibly redefine some in order to get the desired new class. The new class is said to *inherit* the attributes and the methods from the old one. This can be repeated several times, and one can even allow a class to inherit from more than one class – multiple inheritance. In this way a complete *inheritance hierarchy* arises. By sharing code among classes in this way, the total amount of code in the system can be significantly reduced and its maintenance simplified.

From the analyst's point of view, an important role in managing complexity play the so called "pervading methods of organization (or classification)". Among them, is also "the formation and distinction between classes of objects". Identifying commonalities between classes with respect to their *observable* structure and behavior, allows the construction of complete *generalization/specialization hierarchies.* A more specialized class (*a subclass*) is also said to inherit the properties of its *superclass.*

In most OO methodologies, these two points of view are unified. For example:

[14] Inheritance is a relationship among classes, wherein one class shares the structure or behavior defined in one (single inheritance) or more (multiple inheritance) other classes. Inheritance defines an *is–a* hierarchy among classes in which a subclass inherits from one or more generalized super–classes; a subclass typically specializes its superclass by augmenting or redefining existing structure and behavior.

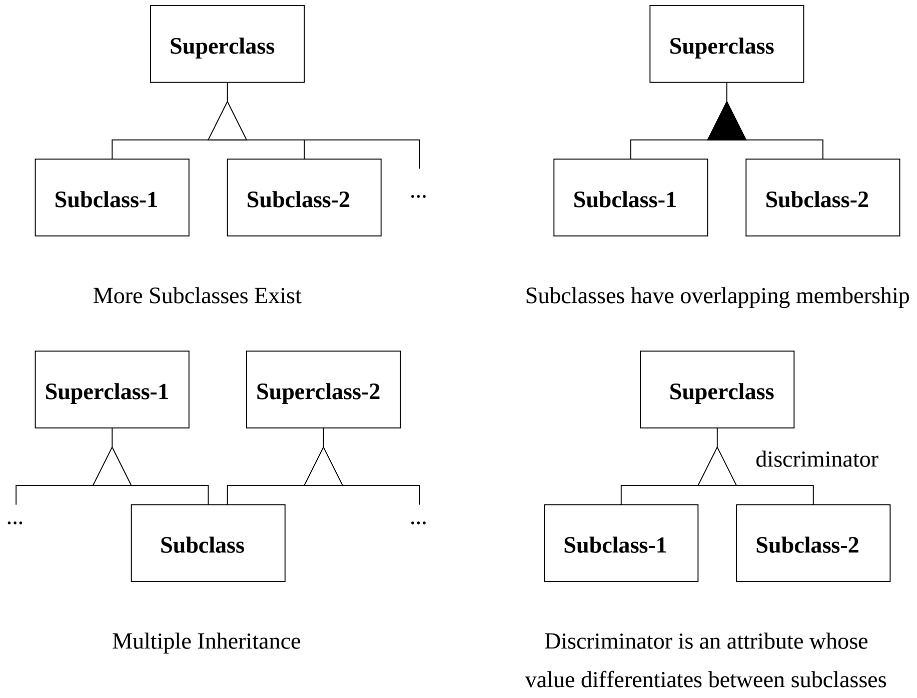
[86] Inheritance is an OO mechanism that permits classes to share attributes and operations based on a relationship, usually generalization.

[33] Inheritance is a relationship between classes where the inheriting class has all the properties of the inherited class and may have some more. Specialization is a relationship between a class and the superclass from which it inherits all the attributes and relationships.

However, generalization/specialization and incremental definition are related, but quite different concepts. In some cases they may be even contradictory. Both concepts are very useful and should be present in every OO method, but they reflect completely different concerns.

2.4.1 Generalization and Specialization

Generalization/specialization, or the *is–a* relationship, is concerned with the *externally* observable behavior of objects i.e. with *what* the objects are expected to do and not with *how* they do it. If objects from a more specialized class have all the observable properties (or behavior) of the objects in a more general class and possibly some more, we can safely use these objects in a context requiring objects from the more general class. The externally observable behavior of objects in a class is given by their *class interface.* The class is said to implement this interface and the interface is a property of the class.



More Subclasses Exist

Subclasses have overlapping membership

Multiple Inheritance

Discriminator is an attribute whose value differentiates between subclasses

Figure 2.9: OMT notation for generalization/specialization

In its simplest form, an interface is *a type.* In this case, generalization/specialization is *subtyping.*

Definition 2.5 Subtyping

> A type σ is a subtype of a type τ written as $\sigma \leq \tau$ if any expression of type σ is allowed in every context requiring an expression of type τ. □

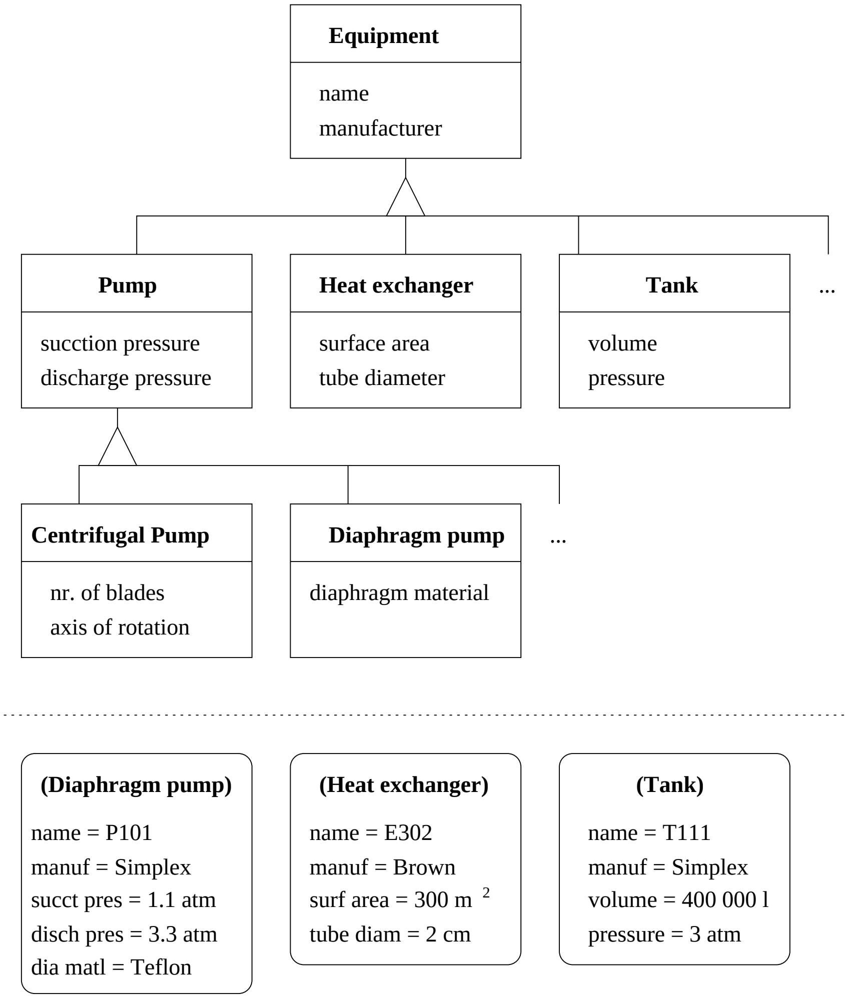


Figure 2.10: A multilevel inheritance hierarchy with instances

In a more sophisticated system (e.g. axiomatic specification languages) the type is only the *syntactic* part of the interface. In this case the properties of the components listed in the syntactic interface can be stated more precisely by a set of *axioms.* They form the *semantic interface.* Hence, an interface is a pair (τ, ϕ). A specification (σ, ψ) is a specialization of (τ, ϕ) if both $\sigma \leq \tau$ and $\psi \Rightarrow \phi$ hold. A class $A : \sigma$ is an implementation of the interface (σ, ϕ) if A satisfies the axioms in ϕ.

Each OO methodology provides a graphical notation for constructing generalization/specialization hierarchies. For example, the OMT notation is given in Figure 2.9.

Using this notation, one can describe for example the class hierarchy and the associated instances for different kinds of equipments as in Figure 2.10.

2.4.2 Inheritance

Inheritance is concerned with the *internal* structure of objects, their attributes and the code they execute for their methods. In other words, inheritance is concerned with *how* the objects do what they do. Objects in a class which inherits methods from another class have the *same code* for the inherited methods as the objects from the superclass. This is the reason why Mitchell defines inheritance in [68] as follows:

> [68] Inheritance is a mechanism for implementing objects of one class by reusing the implementation of another.

In a more general perspective, inheritance can be understood as a general mechanism for incremental code modification. Since objects and classes are recursive or self referencing (the pseudo variables *self* and *self class* are used for this purpose in e.g. Smalltalk) we can formulate the following definition:

Definition 2.6 Inheritance

> Inheritance is a mechanism for incremental extension of recursive structures. □

This definition can be instantiated on the object, class and type level. Since recursion on the object level also implies recursion on the class level which in turn implies recursion on the type level, inheritance on the object level implies inheritance on the class level which in turn implies inheritance on the type level.

2.4.3 Inheritance versus Subtyping

The separation between generalization/specialization and inheritance is analogous to the separation between an interface and its implementations. There are cases in which we want inheritance without specialization or specialization without inheritance. For example, in implementing a class queue it may be convenient to inherit the code from the class array. However we do not want Queue to be a subtype of Array because we do not want that all operations applicable to arrays to be applicable to queues too. Moreover, adding a new method or overriding an old one, may violate some invariant of the superclass and therefore destroy the subtype relation.

inheritance	subtyping
a construction	a property
internal view	external view
reuses class code	relates class interfaces

Table 2.2: Inheritance versus subtyping

The other way around, suppose we specialize the type Queue to LQueue by adding a new method measuring the length of a queue. Then it is possible to have a class implementing Queue and a class implementing LQueue which are not at all related by inheritance.

The first who distinguished between subtyping and inheritance were Cook, Hill and Canning in [35]. This distinction is also done in some recent OO methods. For example Martin/Odell define generalization/specialization as:

- the result (or act) of distinguishing an object type as being more general or inclusive then other,

and class inheritance as:

- an implementation of generalization which permits all the features of an OOPL class to be physically available to, or reusable by, its subclasses – as though they were the features of the subclass.

The table 2.2 summarizes the above discussion.

2.5 Parametric classes

A feature which considerably improves the flexibility of typed OO languages is the possibility to define *parameterized* or *generic* classes.

[13] A generic class is a class that serves as a template for other classes, in which the template may be parameterized by other classes, objects and/or operations. A generic class must be instantiated (its parameters filled in) before objects can be created. Generic classes are typically used as container classes. The terms generic class and parameterized class are interchangeable.

[86] A parameterized class is a template for creating real classes that may differ in well-defined ways as specified by parameters at the time of creation. The parameters are other data types or classes, but may include other attributes, such as the size of a collection (also called generic classes).

A typical generic class is the list class which can be instantiated as a *list of points* or a *list of integers.* The generic methods ft, rt and cons on the list implicitly depend on the type of the element. Local variables within the method have generic type that depends on the instance. An example of a parameterized class list is given in section 4.4. Parameterized classes are available in Eiffel and C++.

2.6 Associations

In the previous sections we analyzed objects in isolation. However, the major interest for using objects is the construction of systems where each object contributes to the overall behavior by interacting with one another.

For the static description of potential object configurations OO analysis uses *links* and *associations.* Both concepts are borrowed from information modeling [30].

Definition 2.7 Link

> A link is a physical or conceptual connection between objects. Mathematically, a link is a tuple of objects. □

Definition 2.8 Association

> An association describes a group of links with common structure and common semantics. All the links in an association connect objects from the same classes. Mathematically an association is a relation among collections of objects. A link is an instance of an association. □

An association describes a set of potential links in the same way a class describes a set of potential objects. Graphically, links and associations are represented e.g. in OMT by a (possibly named) line between objects and classes respectively, as shown in Figures 2.11 and 2.12.

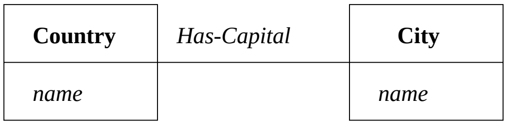


Figure 2.11: Associations – class diagram

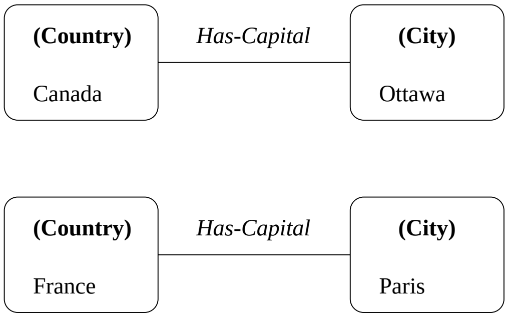


Figure 2.12: Associations – instance diagram

Associations are inherently bidirectional (i.e. symmetric). However, as in the case of relations they can be traversed (or read) either in the *forward* or in the *backward* direction. A particular direction or end of an association may be identified with a *role name.* Each role on a binary association identifies a set of objects associated with an object at the other end as shown in Figure 2.13.

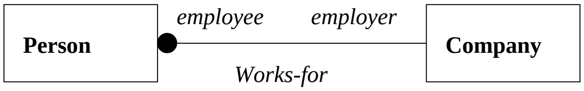


Figure 2.13: An association with roles

Similarly to relations, associations can be binary, ternary or higher order. However, the vast majority are binary. The later ones are usually annotated with multiplicity information which constraints the number of related objects. More precisely, *multiplicity* specifies how many objects of one class may relate to a single object of an associated class. Different forms of multiplicity information are shown in Figure 2.14.

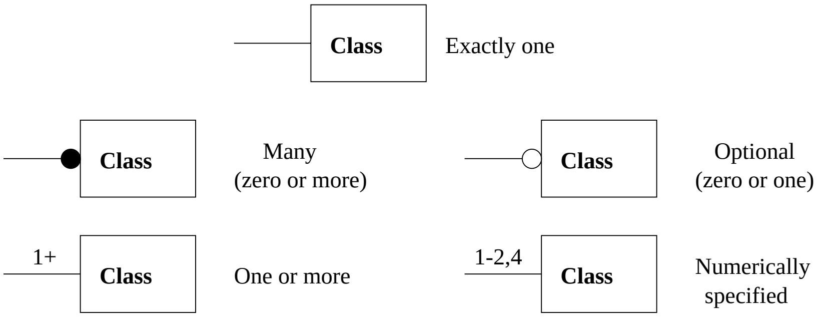


Figure 2.14: Multiplicity information

2.6.1 Aggregation

A very important form of association is the *part-of* association. This kind of association encourages a hierarchical top down design. A system is decomposed in a design step into subsystems (the "parts" of the system) which themselves can be further decomposed until they are simple enough to be directly coded. Such systems are known in the OO methodology as *aggregates.* Citing from [86]:

Definition 2.9 Aggregates

> An aggregate is semantically an extended object that is treated as a unit in many operations, although physically it is made of several lesser objects. Aggregation is a special form of *transitive* and *antisymmetric* association where a group of component objects form a single semantic entity. Operations on an aggregate often propagate to the components. □

Because of their importance, in most OO methodologies, *part-of* associations are marked with a special symbol which distinguishes them from other associations. In [86] they are marked with a diamond. For example, a microcomputer can be described as an aggregation as shown in Figure 2.15.

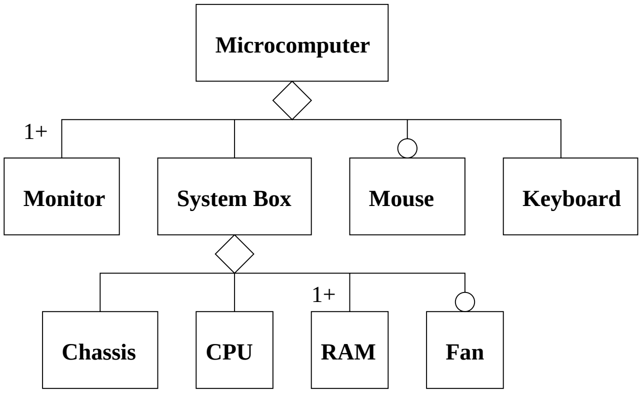


Figure 2.15: A microcomputer as aggregate

An important characteristic of aggregates is that the *role* of each constituent is *fixed* within the whole system. Constituents cannot later leave the original system (i.e. the aggregate) and "associate" to build another system. They are *private* to the original system. For example in Figure 2.16 a company is represented as an aggregation of its divisions which are in turn aggregations of their departments.

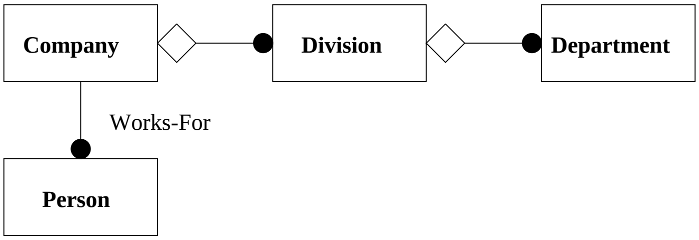


Figure 2.16: Aggregation and association

However, a company is not an aggregation of its employees since company and person are objects with equal status. Employees may work for a company or leave the company. We say that the relation between the company and the employees may vary dynamically. This issue is discussed thoroughly in the next subsection.

The microcomputer example has shown a two level aggregation. According to [86], aggregation can be *fixed, variable* or *recursive.*

- A *fixed aggregate* has a fixed structure. The *number* and *types* of subparts are predefined. A fixed aggregate is shown in Figure 2.17.

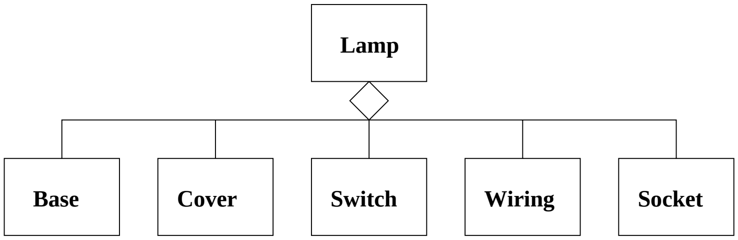


Figure 2.17: A fixed aggregate

- A *variable aggregate* has a finite number of levels, but the *number* of parts may vary. The microcomputer is a two level aggregate with a variable number of monitors, mouses, RAMs and Fans.

- A *recursive aggregate* contains directly or indirectly an instance of the same kind of aggregate; the number of potential levels is unlimited. Figure 2.18 shows a computer program which is an aggregation of blocks with optionally recursive compound statements.

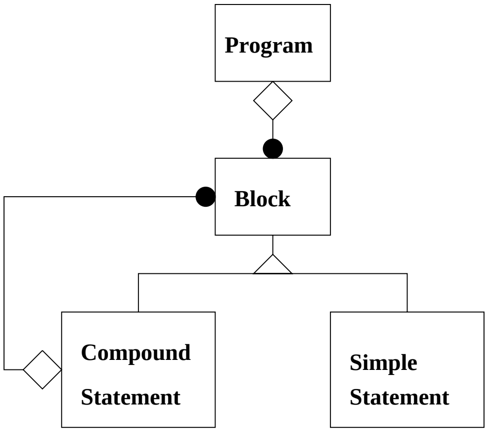


Figure 2.18: A recursive aggregate

Aggregation is not the same thing as generalization. Aggregation relates instances. Two distinct objects are involved; one of them is part of another. Generalization relates classes and is a way of structuring the description of a single object. With generalization an object is simultaneously an instance of the superclass and an instance of the subclass.

2.6.2 Dynamic Associations

The aggregates presented in the previous section impose a hierarchical view of systems. Each component has a precisely stated position in the hierarchy which can not be changed anymore. Although this is adequate in many cases, and in principle each system can be designed in this way, a “flat structure” seems to be preferable for systems which we call *mobile* (or *democratic*). In such systems, the role of components may change as the system evolves. They may associate at a given point of time to build a system and leave this system to build another one later on.

Definition 2.10 Mobile Systems

> Systems in which every object can change its communication partners on the basis of computation and interaction are designated as *mobile.* □

The first systems exhibiting mobility were the Actor systems [3]. However, the lack of a clear mathematical foundation limited their influence until recently when plenty of mobile systems appeared in the literature. Among them is the π-Calculus which not only adds mobility but also simplifies CCS [66, 67, 65], the Chemical Abstract Machine [12] which consecrated the multi-set laws and the Rewriting Logic of Meseguer [59] which is based on parallel rewriting modulo associative-commutative laws.

Chapter 3

The λ–Calculus for Objects

When giving a formal foundation for objects, a very important concern is to devise an appropriate type system. The main reasons for introducing a type system are:

- the detection of erroneous expressions at compile time (e.g. the use of a natural number as if it where a function),
- the enhancement of program efficiency by using compile time information,
- the support for data abstraction and modularity.

The type of an object can be understood as a property or theorem about that object. This property says that it never delivers an answer of the form *message not understood,* as do objects in an untyped language like Smalltalk or Object Scheme. We are therefore concerned in this chapter with the definition of a *typed* language for objects.

The object model which we present in the next chapter, views objects roughly as *case–functions with a hidden state,* where each branch of the function corresponds to a method of the object. This contrasts to the usual modeling of objects in the typed λ–calculus where objects are interpreted as *records.* In section 3.1 we compare these two models and show why the first one is more appropriate for our purpose. We also intuitively introduce a type system for this model by highlighting the problems which occur in a naive typing with variants. The next two sections describe in detail the corresponding typed λ–calculus. In section 3.2 we present its syntax and in section 3.3 we present its semantics.

3.1 Design Decisions

The comparison between the record and the case models uses a small but typical OO–programming example: Cartesian points. Let us first introduce this example as it would be written in a typed OO language which distinguishes inheritance from subtyping as e.g. [36].

3.1.1 The Cartesian Points Example

A Cartesian point is an object with two attributes (or instance variables), the coordinates xc and yc and with four methods, x (), y (), mv (dx, dy) and eq (p). The first two return the coordinates values, the third one increments them with dx and dy respectively and the fourth one tests if the current point is equal with the point p.

The *interface* of the Cartesian points class is defined as follows:

```
interface Point
    x ()             returns Real;
    y ()             returns Real;
    mv (Real, Real)  returns Point;
    eq (Point)       returns Bool;
```

It is a description of the messages understood by the Cartesian point objects where each message is given with the parameters and the result types. As we will discuss later, the interfaces are used to ensure that message sending never generates a run time error.

```
class cart_point (xc : Real, yc : Real)
    implements Point;
    method x () returns Real
        return xc;
    method y () returns Real
        return yc;
    method mv (dx : Real, dy : Real) returns Point
        return new myclass (self.x () + dx, self.y () + dy);
    method eq (p : Point) returns Bool
        return self.x () == p.x () ∧ self.y () == p.y ();
```

The *class* is a pattern that can be used to create objects with common structure. It describes what *attributes* the objects (or instances) will have, the code for each *method* and how to *create* objects. The messages understood by the objects in a class are listed in the class interface. They can be regarded themselves as the interfaces for abstract operations. The methods are particular implementations of these operations. This is the reason why the class cart_point point is declared to be an implementation of the interface Point.

The definition of the class cart_point reveals a common characteristics of objects: they are recursive. First, recursion is necessary to allow methods to refer each other. For example the method eq refers both x() and y()[1]. Like in Smalltalk, we used the pseudo variable *self* to refer to the object itself. When the methods eq or mv are invoked, *self* is bound to the receiver of the message. This level of recursion is known as *object recursion.* Beside *self*, Smalltalk also uses the construct *self class* to create an object of the same class as *self* (we used instead *myclass*). This is actually another level of recursion (or self reference) which is known as *class recursion.* A graphical illustration is given in figure 3.1

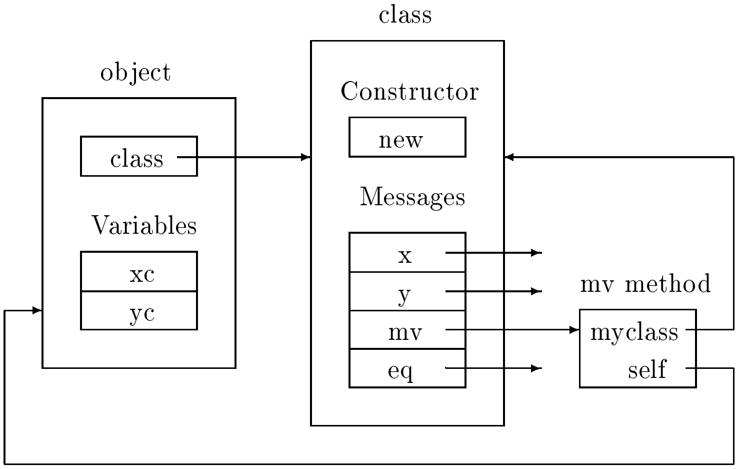


Figure 3.1: Recursive structure of objects

The recursion in the class definition is also reflected in the interface definition. The interfaces of both mv and eq use Point recursively. The separation between classes and interfaces allows multiple implementations respecting the same message protocol. Safety is guaranteed by compile time checking which uses the declaration of instance variables and formal parameters in the form $v : < interface >$ **in class** $< class >$. For example:

p : Point **in class** cart_point

3.1.2 The Record versus The Case Semantical Model

In order to give the objects a denotational semantics and in particular to correctly type inheritance, these are often regarded as *closures* i.e. as functions (or data structures containing functions) with some local bindings to values or storage locations

[1] In this respect the example is not very convincing since instead of x() and y() one could have used xc and yc. However, in general this is not the case.

[80]. In this modeling the instance variables are simply the λ or *let* bound variables. For the explanation of message passing there are two possible alternatives, as already suggested by Zilles in [97]. In the first case, each method is a separate entry point of a multi–procedure. The method name is used to select the appropriate entry point. In the second case, each method is a separate branch of a single procedure. The method name is used to select the appropriate branch. Although very similar, these two different approaches lead to different (data) structures. In the first case objects become *records* with functional fields as first pointed out by Cardelli in [22, 23]. The method name corresponds to a record field so it is *applied* as field selection to the object. In the second case the objects become *case functions* as pointed out by Adams and Rees in [2]. The method name corresponds to a case selection so it is *passed* to the object which chooses the appropriate branch. Let us examine these approaches on the Cartesian point example. To simplify notation we do not explicitly write the type annotations.

The record variant is as follows:

$$
\begin{array}{l}
\mathcal{P} = \lambda myclass.\lambda(xc, yc).\lambda self \\
\qquad \begin{array}{lll}
\{\ \mathsf{x} & = & xc, \\
\ \ \mathsf{y} & = & yc, \\
\ \ \mathsf{mv} & = & \lambda(dx, dy).\ \mathsf{fix}\ myclass(self.\mathsf{x} + dx, self.\mathsf{y} + dy), \\
\ \ \mathsf{eq} & = & \lambda p.\,(self.\mathsf{x} == p.\mathsf{x}) \wedge (self.\mathsf{y} == p.\mathsf{y})\ \}
\end{array}
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{cart\text{-}point} = \mathsf{fix}\ \mathcal{P} & --\textit{class definition} \\
o = \mathsf{fix}\ \mathsf{cart\_point}\,(a,b) & --\textit{object creation} \\
(o.\mathsf{mv})(dx, dy) & --\textit{message passing}
\end{array}
$$

$\mathcal{P}$ can be regarded as the definition of the class cart_point. The class itself is obtained by binding $myclass$ to this definition. The self reference role of $myclass$ is made explicit as a fixed point variable. A Cartesian point object is obtained by instantiating xc and yc with actual coordinates and by binding $self$ to the object. Again, as with $myclass$ the self reference role of $self$ is made explicit. Objects are passed only the method arguments. The method name is used to select the appropriate method.

The object–as–record modeling was extensively analyzed in the literature including type checking systems [23, 26, 25, 21], type inference systems [81, 91, 93, 49] and semantical models [23, 19, 20, 15]. The above double recursion scheme for the modeling of objects first appeared in [35].

The alternative object–as–case–function variant is as follows:

$$
\begin{array}{l}
\mathcal{P} = \lambda myclass.\lambda(xc, yc).\lambda self.\lambda m. \\
\qquad \textbf{case}\ m\ \textbf{of}
\end{array}
$$

$$\begin{array}{ll}
\mathsf{x} & \Rightarrow xc \\
\mathsf{y} & \Rightarrow yc \\
\mathsf{mv}\,(dx, dy) & \Rightarrow \mathsf{fix}\ myclass(self(\mathsf{x}) + dx,\ \ self(\mathsf{y}) + dy) \\
\mathsf{eq}\,(p) & \Rightarrow (self(\mathsf{x}) == p(\mathsf{x})) \wedge (self(\mathsf{y}) == p(\mathsf{y}))
\end{array}$$

$$\begin{array}{ll}
\mathsf{cart\_point} = \mathsf{fix}\ \mathcal{P} & --class\ definition \\
o = \mathsf{fix}\ \mathsf{cart\_point}\,(a,b) & --object\ creation \\
o\ \mathsf{mv}(dx, dx) & --message\ passing
\end{array}$$

The treatment of instance variables and of recursion is similar to the record variant. However, in this case the whole message (including the method name) is passed to the object. This selects the appropriate branch with the message arguments substituted for the pattern variables. The object–as–case–function modeling was persuaded in Scheme. More precisely, since Scheme is an untyped language, Scheme–objects are actually conditional–functions instead of case–functions. For example, a Scheme modeling of the Cartesian points, similar to the one discussed in [2] but in our syntax and without side effects like $!set$, would be:

$$\mathcal{P} = \lambda myclass.\lambda(xc, yc).\lambda self.\lambda m.$$

$$\begin{array}{l}
\textbf{cond} \\
\quad \begin{array}{ll}
m == \mathsf{x} & \Rightarrow \lambda().xc \\
m == \mathsf{y} & \Rightarrow \lambda().yc \\
m == \mathsf{mv} & \Rightarrow \lambda(dx, dy).\ \mathsf{Y}\ myclass(self(\mathsf{x}) + dx,\ \ self(\mathsf{y}) + dy) \\
m == \mathsf{eq} & \Rightarrow \lambda p.(self(\mathsf{x}) == p(\mathsf{x})) \wedge (self(\mathsf{y}) == p(\mathsf{y})) \\
\mathsf{true} & \Rightarrow \mathsf{error}
\end{array}
\end{array}$$

Y is the untyped version of the fixed point operator.

Both the above record and function models make a significant simplification which does not correspond to the OO programming intuition. Asking a point about its x or y coordinate destroys that point. We actually expect the point to return its coordinate and subsequently be able to answer to other messages. As a consequence an object should be a function mapping histories of input messages onto histories of answers. Beside being closer to our intuition this modeling will also allow to move from the sequential to the parallel world.

Note that considering an object as a record of functions working on message histories is not of very much use because field selection would also destroy the object. However, the object as case–function model can be easily extended to histories by replacing the single message m by a history of messages s and modifying each branch to consume the first message and to recursively call the object on the rest of the history. More details are given in section 4.2. This is the first important design decision we take:

Objects are case–functions extended to message histories.

3.1.3 Problems Encountered in a Naive Typing

Although the object–as–case–function modeling is more appropriate for our concurrent setting, an adequate typing proves to be a non trivial task in this case. First, let us show the problems encountered in a naive typing with *variants.*

Suppose we have points only with a coordinate xc and a color col which we loosely define as follows:

$$\begin{aligned} \mathcal{XC} &= \lambda myclass.\lambda(xc, col).\lambda self.\lambda m \\ &\quad \textbf{case } m \textbf{ of} \\ &\qquad \mathsf{x} \Rightarrow\ xc \\ &\qquad \mathsf{c} \Rightarrow\ col \\ &\quad \mathsf{XC} = \mathsf{fix}\ \mathcal{XC} \end{aligned}$$

A point p with the coordinate 0 and the color red is obtained as $\mathsf{fix}(\mathsf{XC}(0,\ \mathsf{red}))$. A tentative typing for p could be the following one:

$$p :\ [\mathsf{x},\ \mathsf{c}] \longrightarrow [\mathsf{x} : \mathsf{Real},\ \mathsf{c} : \mathsf{Color}]$$

where $[\mathsf{l}_1 : \sigma_1, \ldots, \mathsf{l}_n : \sigma_n]$ is a *variant* or a labeled sum. The variant type $[\mathsf{x},\ \mathsf{c}]$ is an abbreviation for $[\mathsf{x} : (),\ \mathsf{c} : ()]$ where $()$ is the *unit type* which contains only one member, the nullary constructor $()$. To correctly reflect this typing we have to modify the above definition of $\mathcal{XC}$ as follows:

$$\begin{aligned} \mathcal{XC}' &= \lambda myclass.\lambda(xc, col).\lambda self.\lambda m \\ &\quad \textbf{case } m \textbf{ of} \\ &\qquad \mathsf{x} \Rightarrow [\mathsf{x} = xc] \\ &\qquad \mathsf{c} \Rightarrow [\mathsf{c} = col] \end{aligned}$$

A first negative consequence of this typing is that we are forced to explicitly label the answers. This is naturally an inconvenience compared with the object–as–record approach. However, there is a much more serious problem: the typing $[\mathsf{x},\ \mathsf{c}] \longrightarrow [\mathsf{x} : \mathsf{Real},\ \mathsf{c} : \mathsf{Color}]$ is not accurate enough. Although, our objects always deliver the coordinate when asked for the coordinate and the color when asked for the color, the above type also contains functions which deliver the coordinate when asked for the color or the color when asked for the coordinate. In order to see why this matters, suppose we extend the above definition by adding the coordinate yc.

$$
\begin{array}{l}
\mathcal{XCY}' = \lambda myclass.\lambda(xc, col, yc).\lambda self.\lambda m \\
\qquad\qquad \textbf{case } m \textbf{ of} \\
\qquad\qquad\quad \mathsf{x} \Rightarrow [\mathsf{x} = xc] \\
\qquad\qquad\quad \mathsf{c} \Rightarrow [\mathsf{c} = col] \\
\qquad\qquad\quad \mathsf{y} \Rightarrow [\mathsf{y} = yc]
\end{array}
$$

Then an object q generated by this class definition has the type:

$$q : [\mathsf{x}, \mathsf{c}, \mathsf{y}] \longrightarrow [\mathsf{x} : \mathsf{Real}, \mathsf{c} : \mathsf{Color}, \mathsf{y} : \mathsf{Real}]$$

This object behaves exactly as the object p for the messages $[\mathsf{x}]$, $[\mathsf{c}]$ and additionally delivers $[\mathsf{y} = yc]$ in response to the message $[\mathsf{y}]$. Hence, we would expect to be able to use q in every context in which p is allowed. However, nothing prevents a function with the same type as q to deliver $[\mathsf{y} = yc]$ as the answer to the messages $[\mathsf{c}]$ or $[\mathsf{x}]$. This could generate a run time error. As a consequence, the type of q is not a subtype of the type of p. More formally, the sub–typing rule for functional types is:

$$(\leq\rightarrow) \quad \frac{\sigma' \leq \sigma \qquad \tau \leq \tau'}{\sigma \rightarrow \tau \;\leq\; \sigma' \rightarrow \tau'}$$

It says that we are allowed to increase the domain of the input values or to decrease the domain of output values. Although $[\mathsf{x}, \mathsf{c}] \leq [\mathsf{x}, \mathsf{c}, \mathsf{y}]$ it is not the case that $[\mathsf{x} : \mathsf{real}, \mathsf{c} : \mathsf{Color}, \mathsf{y} : \mathsf{Real}] \leq [\mathsf{x} : \mathsf{Real}, \mathsf{c} : \mathsf{Color}]$ as the above rule requires, but the other way around. As a consequence

$$[\mathsf{x}, \mathsf{c}, \mathsf{y}] \longrightarrow [\mathsf{x} : \mathsf{Real}, \mathsf{c} : \mathsf{Color}, \mathsf{y} : \mathsf{Real}] \not\leq [\mathsf{x}, \mathsf{c}] \longrightarrow [\mathsf{x} : \mathsf{Real}, \mathsf{c} : \mathsf{Color}]$$

which implies that the extension q of p is not allowed in the contexts in which p is. Since, as we show later, inheritance is nothing but a general mechanism for such function extensions, this also implies that we cannot correctly type inheritance. In conclusion, neither records nor variants are adequate for our functional modeling of objects. However, we can use related ideas when searching for a more appropriate type system.

3.1.4 An Adequate Type System

Our first criticism of variants was the obligation of output labeling. This forced us to replace the definition $\mathcal{XC}$ of colored points by $\mathcal{XC}'$. However $\mathcal{XC}$ is a well typed term in a type discipline allowing *union* types [84, 6]. In this case, the output type of $\mathcal{XC}$ is the union $\mathsf{Real} \cup \mathsf{Color}$[2]. Variants and their subtyping relation are also easily

[2] Typically Real will be disjoint from Color.

recovered as follows. Each element constructor is considered to determine a unique sort constructor which has the same name and arity. Moreover, it is distinct from any other sort constructor, including the function space constructor. For example x is the only member of the sort x and $\mathsf{mv}(dx,dy)$ is an element of the sort $\mathsf{mv}(\mathsf{Real},\mathsf{Real})$. Then $[\mathsf{x}, \mathsf{c}]$ is given by $\mathsf{x} \cup \mathsf{c}$ and $[\mathsf{x} : \mathsf{Real}, \mathsf{c} : \mathsf{Color}]$ is given by $\mathsf{x}(\mathsf{Real}) \cup \mathsf{c}(\mathsf{Color})$. Adding the axioms $\alpha \leq \alpha \cup \beta$ and $\beta \leq \alpha \cup \beta$ (where $\leq$ is understood as inclusion) we also get the previous subtype relations:

$\mathsf{x} \cup \mathsf{c} \leq \mathsf{x} \cup \mathsf{c} \cup \mathsf{y}$
$\mathsf{x}(\mathsf{Real}) \cup \mathsf{c}(\mathsf{Color}) \leq \mathsf{x}(\mathsf{Real}) \cup \mathsf{c}(\mathsf{Color}) \cup \mathsf{y}(\mathsf{Real})$

In the presence of unions we can consider that:

$p : \mathsf{x} \cup \mathsf{c} \longrightarrow \mathsf{Real} \cup \mathsf{Color}$

Although it is also the case that

$\mathsf{x} \cup \mathsf{c} \cup \mathsf{y} \longrightarrow \mathsf{Real} \cup \mathsf{Color} \leq \mathsf{x} \cup \mathsf{c} \longrightarrow \mathsf{Real} \cup \mathsf{Color}$

unions are in general not enough to solve the second problem. For example, if we first considered points only with a x and a y coordinate and subsequently added a color, we would have obtained that:

$\mathsf{x} \cup \mathsf{y} \cup \mathsf{c} \longrightarrow \mathsf{Real} \cup \mathsf{Color} \not\leq \mathsf{x} \cup \mathsf{y} \longrightarrow \mathsf{Real}$

In order to solve this problem, let us first discuss a very desirable property of modern strongly typed languages (e.g. ML), namely *parametric polymorphism.* For example, the function:

$\mathsf{cons} : \forall\alpha.\ \alpha \times \mathsf{List}\ \alpha \longrightarrow \mathsf{List}\ \alpha$

is said to be polymorphic. Given a universe U of types which is closed under sort constructors (e.g. $\times, \longrightarrow$, List) the polymorphic type of cons can be understood as the infinite intersection $\cap_{\tau\in U}(\tau \times \mathsf{List}\,\tau \longrightarrow \mathsf{List}\,\tau)$[3]. As a consequence, cons can be used in any context requiring a function of type $\tau \times \mathsf{List}\,\tau \longrightarrow \mathsf{List}\,\tau$, for an arbitrary τ. Cardelli and Wegner [26] refined this form of polymorphism in the presence of subtyping by restricting the range of α to subtypes of a given bound–type σ. This type of polymorphism is known as *bounded polymorphism.* Parametric polymorphism can be recovered from bounded polymorphism by taking σ to be the whole universe. For example,

[3] In a type discipline which does not contain intersections one usually speaks about the infinite product $\Pi_{\tau\in U}(\tau \times \mathsf{List}\,\tau \longrightarrow \mathsf{List}\,\tau)$.

$\mathsf{cons} : \forall\alpha \leq \mathsf{Point}.\ \alpha \times \mathsf{List}\ \alpha\ \longrightarrow \mathsf{List}\ \alpha$

restricts α to be a subtype of Point. Parametric polymorphism allows the definition of parametric object classes and its bounded form was essential to correctly type inheritance in [26]. We therefore also include bounded polymorphism in our type system.

As already implied by the definition of polymorphic types, we also include *intersection types* [84, 78]. However, unions and intersections are used in a very restricted way, allowing decidable type inference (more details are given in the next sections).

Finally we also include conditional types [82, 6]. A conditional type is written as $\tau_1\,?\,\tau_2$. It is void if τ_2 is void; otherwise it equals τ_1[4].

Combining the above types we can accurately express the connection between the input and the output of a case expression. For example:

$$p\ :\ \forall\alpha \leq \mathsf{x} \cup \mathsf{c}.\ \alpha \longrightarrow (\mathsf{Real}?\alpha \cap \mathsf{x})\ \cup\ (\mathsf{Color}?\alpha \cap \mathsf{c})$$

Similarly, if q is a point defined by $\mathcal{XCY}$ which is obtained by forgetting about the variants in the output of $\mathcal{XCY}'$ we have:

$$q\ :\ \forall\alpha \leq \mathsf{x} \cup \mathsf{c} \cup \mathsf{y}.\ \alpha \longrightarrow (\mathsf{Real}?\alpha \cap \mathsf{x})\ \cup\ (\mathsf{Color}?\alpha \cap \mathsf{c})\ \cup\ (\mathsf{Real}?\alpha \cap \mathsf{y})$$

Now we can use the subtyping rule for polymorphic types:

$$(\leq \forall)\quad \frac{S \vdash \tau_1' \leq \tau_1 \quad S, \alpha \leq \tau_1' \vdash \tau_2 \leq \tau_2'}{S \vdash (\forall\alpha \leq \tau_1.\tau_2)\ \leq\ (\forall\alpha \leq \tau_1'.\tau_2')}$$

where S is a set of *subsort assumptions.* Analogous to function types, polymorphic types are anti–monotonic in the "first argument" (τ_1 and τ_1') and monotonic in the "second one" (τ_2 and τ_2'). As for variants, anti–monotony is satisfied since $\mathsf{x} \cup \mathsf{c} \leq \mathsf{x} \cup \mathsf{c} \cup \mathsf{y}$. Moreover, in this case, if $\tau \leq \mathsf{x} \cup \mathsf{c}$, then the type of p equals the type of q when instantiated to τ because $\mathsf{Real}?\tau \cap \mathsf{y} = \emptyset$. Hence, by using $(\leq \forall)$ we obtain as desired that:

$$\begin{array}{l}\forall\alpha \leq \mathsf{x} \cup \mathsf{c} \cup \mathsf{y}.\alpha \longrightarrow (\mathsf{Real}?\alpha \cap \mathsf{x}) \cup (\mathsf{Color}?\alpha \cap \mathsf{c}) \cup (\mathsf{Real}?\alpha \cap \mathsf{y}) \leq \\ \forall\alpha \leq \mathsf{x} \cup \mathsf{c}. \qquad \alpha \longrightarrow (\mathsf{Real}?\alpha \cap \mathsf{x}) \cup (\mathsf{Color}?\alpha \cap \mathsf{c})\end{array}$$

Summarizing:

In order to correctly type objects we use a polymorphic type discipline which is based on union, intersection and conditional types.

[4] The void type contains only one element $\perp$, which denotes nontermination.

In the above discussion we did not consider the methods mv and eq. In order to correctly type the points in their presence, we also need recursive types. Instead of introducing them explicitly, we use a more general form of bounded quantification: $\forall\alpha_i.\tau$ **where** S, where α_i abbreviates the set $\{\alpha_i \mid 1 \leq i \leq n\}$. The quantified variables must satisfy the set of mutually recursive constraints S. For example, we obtain the following type for points p as defined by $\mathcal{P}$:

$$\forall\alpha.\alpha \longrightarrow (\mathsf{Real}?\alpha \cap \mathsf{x}) \cup (\mathsf{Color}?\alpha \cap \mathsf{c}) \cup (\mathsf{Bool}?\alpha \cap \mathsf{eq}(t)) \cup (t?\alpha \cap \mathsf{mv}(\mathsf{Real},\mathsf{Real}))$$

where

$$\begin{array}{lcl} \alpha & \leq & \mathsf{x} \cup \mathsf{c} \cup \mathsf{mv}(\mathsf{Real},\mathsf{Real}) \cup \mathsf{eq}(t) \\ t & = & \alpha \longrightarrow (\mathsf{Real}?\alpha \cap \mathsf{x}) \cup (\mathsf{Color}?\alpha \cap \mathsf{c}) \cup (\mathsf{Bool}?\alpha \cap \mathsf{eq}(t)) \cup (t?\alpha \cap \mathsf{mv}(\mathsf{Real},\mathsf{Real})) \end{array}$$

This justifies the next design decision:

The type discipline for objects has to support recursive types.

3.2 The Syntax

3.2.1 Notational Conventions

Usually a language is first described by a context free grammar. The terms generated by this grammar are called *rough terms* because among them there are also erroneous ones. The set of *well formed terms* i.e. the set of terms respecting the typing discipline are then filtered out by using a *context sensitive syntax.* We give this syntax by using a formalism based on logic. Specifically, we define expressions and their types simultaneously using axioms and inference rules. An inference rule has the form:

$$\frac{\Gamma \rhd e_1 : \sigma_1 \quad \ldots \quad \Gamma \rhd e_n : \sigma_n}{\Gamma \rhd e : \sigma}$$

It allows us to derive the conclusion $\Gamma \rhd e : \sigma$ if all the assumptions $\Gamma \rhd e_i : \sigma_i$ are true. The expressions $\Gamma \rhd e : \sigma$ with $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ are called *typing assertions*. They intuitively say that if variables $x_1, \ldots, x_n$ have types $\sigma_1, \ldots, \sigma_n$ then e is a well formed term of type σ. Γ is a *type assignment* or *type context* with no x_i occurring twice. We will write $\Gamma, x : \sigma$ for $\Gamma \cup \{x : \sigma\}$. In doing so we assume that x does not appear in Γ. An axiom is an inference rule with no assumptions.

Remark 3.1

> Alternatively, a typing assertion $\Gamma \rhd e : \sigma$ can be considered as the least three place relation $\rhd$ closed under the inference rules. We use these two definitions interchangeably. □

The function $dom(\Gamma)$ denotes the set of variables defined by Γ. The *range* of Γ is the collection of right–hand sides of bindings in Γ. $\Gamma(x)$ denotes the type of x in Γ, if it has one. The set of *free variables* of a term e is written as $\mathrm{FV}(e)$. We say that a term is *closed* with respect to the context Γ if $\mathrm{FV}(e) \subseteq dom(\Gamma)$. A typing assertion $\Gamma \rhd e : \sigma$ is closed if e is closed with respect to Γ. The set of *free type variables* of a type τ is written as $\mathrm{FTV}(\tau)$. The set $\mathrm{FTV}(\Gamma)$ of the free type variables of a context Γ is the union of the sets of free type variables of elements of the range of Γ.

In order to reason about subtyping, we use a second relation $\vdash$. In this case the expression $S \vdash \sigma \leq \tau$ is called a *subtyping assertion* where the *subtyping context* $S = \{\sigma_1 \leq \tau_1, \ldots, \sigma_n \leq \tau_n\}$ is a set of *proper*[5] *subtyping assumptions.*

A derivation of a typing or subtyping statement T is a proof tree, valid according to some collection of inference rules, whose root is T. We write $d :: T$ to indicate that d is a derivation of T.

[5] Proper sets of constraints are defined in Section 3.2.7.

In the next sections we present the context free and the context sensitive language of expressions together with their associated types. This language is based on [6]. However, the following modifications were necessary in order to make it appropriate for our object model.

- Add a new syntactical construct f **extend** $(p \Rightarrow e)$. This construct allows to extend a (case–) function with a new branch. As we show in Section 4.3 this construct is essential for modeling inheritance.
- Adapt the subtyping rules for a lazy semantics. This allows an appropriate treatment of message histories.

Moreover, in contrast to [6] we make a sharp distinction between the *type inference system* which is presented in Sections 3.2.2 through 3.2.6 and the *type inference algorithm* which is presented in Section 3.2.7. This considerably improves the clarity of the exposition. Also for the sake of clarity, we present the language incrementally. Each new production (or inference rule) extends the language introduced so far.

3.2.2 The Calculus $\lambda_{\leq}^{\rightarrow}$

This language is the core lambda calculus. On the level of expressions it contains only variables, function abstraction and function application. On the level of types it contains only type variables and the function space constructor.

Definition 3.1 Type expressions

The type expressions of $\lambda_{\leq}^{\rightarrow}$ are given by the following grammar:

$$\tau \quad ::= \alpha \mid \tau_1 \rightarrow \tau_2$$

□

Definition 3.2 Expressions

The context free language of terms of $\lambda_{\leq}^{\rightarrow}$ is given by the following grammar:

$$e \quad ::= \quad x \mid \lambda x.e \mid e_1 e_2$$

□

Remark 3.2 Type assignment systems

The absence of any type annotation for the bound variable occuring in the syntax of the λ–abstraction signals a fundamental design choice which we shall maintain throughout the thesis: all calculi we consider are *type assignment systems* as opposed to *explicitly typed systems.* These systems are compared in Section 3.3.1. □

Omitting type annotations allows the programmer to write programs as concise as in an untyped language. Moreover, like in ML, it is also easy to extend the system to allow *optional* type annotations. However, the absence of type annotations requires a very careful language design which assures decidability of type inference.

Definition 3.3 Context

A context is a pair $S \mid A$ where S is a subtyping context and A is a typing context. □

Definition 3.4 Context sensitive syntax

The context sensitive syntax for $\lambda^{\rightarrow}_{\leq}$ is given by the following inference rules:

$$(var) \quad \frac{}{S \mid x : \sigma \rhd x : \sigma}$$

$$(\rightarrow i) \quad \frac{S \mid A, x : \tau_1 \rhd e : \tau_2}{S \mid A \rhd \lambda x.e : \tau_1 \rightarrow \tau_2}$$

$$(\rightarrow e)^* \quad \frac{S \mid A \rhd e_1 : \tau_1 \rightarrow \tau_2 \quad S \mid A \rhd e_2 : \tau_1}{S \mid A \rhd e_1 e_2 : \tau_2}$$

$$(sub)^* \quad \frac{S \mid A \rhd e : \tau_1 \quad S \vdash \tau_1 \leq \tau_2}{S \mid A \rhd e : \tau_2}$$

□

The rule $(sub)^*$ allows a term of type τ_1 to be promoted (or coerced) to a type τ_2 whenever τ_1 is a subtype of τ_2. This rule is however not appropriate for a type inference algorithm where the next step of inference is determined by the term structure. One can eliminate $(sub)^*$ by modifying the rule $(\rightarrow e)^*$ to perform the necessary coercions as follows:

$$(\rightarrow e) \quad \frac{S \mid A \rhd e_1 : \tau_1 \quad S \mid A \rhd e_2 : \tau_2 \quad S \vdash \tau_1 \leq \tau_3 \rightarrow \tau_4 \quad S \vdash \tau_2 \leq \tau_3}{S \mid A \rhd e_1\ e_2 : \tau_4}$$

In our calculus, this rule replaces $(\rightarrow e)^*$ and $(sub)^*$.

In order to complete the definition of the context sensitive syntax, we have to define the subtype relation. Intuitively, a subtyping statement $S \vdash \sigma \leq \tau$ corresponds to the assertion that σ is a *refinement* of τ in the sense that every element of σ contains enough information to meaningfully be regarded as an element of τ. Actually, in the semantic model presented later, $\sigma \leq \tau$ simply means that σ is a *subset* of τ. This intended semantics for $\leq$ immediately implies that the subtype relation should be *reflexive* and *transitive* i.e. it should be a *preorder.* In order to obtain the necessary subtype relation for function spaces, consider that $f : \sigma \rightarrow \tau$. Regarding a type as a *constraint*, $\sigma \rightarrow \tau$ requires that f maps all values satisfying σ to values satisfying τ. If σ' is a weaker constraint, then informally, σ' denotes a larger set than σ. Weakening σ to σ' has the opposite effect on the type $\sigma \rightarrow \tau$ of functions from σ to τ because $f : \sigma' \rightarrow \tau$ requires f to map the larger set σ' to values satisfying τ. Hence $\sigma' \rightarrow \tau$ is a stronger constraint on f as $\sigma \rightarrow \tau$. On the other hand, weakening τ to τ' yields the weaker constraint $\sigma \rightarrow \tau'$ since each function satisfying $\sigma \rightarrow \tau$ also satisfies $\sigma \rightarrow \tau'$.

Definition 3.5 Subtyping

The subtyping relation of $\lambda_{\leq}^{\rightarrow}$ is given by the following rules:

$$(\leq ref) \quad \frac{}{S \vdash \tau \leq \tau}$$

$$(\leq tra) \quad \frac{S \vdash \tau_1 \leq \tau_2 \quad S \vdash \tau_2 \leq \tau_3}{S \vdash \tau_1 \leq \tau_3}$$

$$(\leq\rightarrow) \quad \frac{S \vdash \tau_1^{'} \leq \tau_1 \quad S \vdash \tau_2 \leq \tau_2^{'}}{S \vdash \tau_1 \rightarrow \tau_2 \leq \tau_1^{'} \rightarrow \tau_2^{'}}$$

□

Borrowing terminology from category theory one says that the function space constructor is *contravariant* in the first argument (i.e. it changes the direction of the subtype relation) and *covariant* in the second argument (i.e. it preserves the direction of the subtype relation).

Remark 3.3

The subtyping relation for arrow types is both *structural* and *compositional.* It is structural because the ordering on arrow types is completely determined by their left– and right–hand sides. It is compositional because the ordering on arrow types may be computed as a function of the ordering of their left– and right–hand sides. □

We say that σ is *equivalent* with τ, written as $S \vdash \sigma{=}\tau$ if and only if $S \vdash \sigma \leq \tau$ and $S \vdash \tau \leq \sigma$.

3.2.3 The Calculus $\lambda_{\leq}^{\rightarrow,c}$

Beside the function space constructor we also allow arbitrary *free data type constructors.* As we see later, they are very useful in defining functions by pattern matching.

Definition 3.6 Type expressions

The type expressions of $\lambda_{\leq}^{\rightarrow,c}$ are obtained from $\lambda_{\leq}^{\rightarrow}$ by adding the following production:

$$\tau \quad ::= c(\tau_1, \ldots, \tau_n)$$

□

Definition 3.7 Expressions

The context free syntax for the terms of $\lambda_{\leq}^{\rightarrow,c}$ is obtained from $\lambda_{\leq}^{\rightarrow}$ by adding the following production:

$$e \quad ::= c(e_1, \ldots, e_n)$$

□

Definition 3.8 Context sensitive syntax

The context sensitive syntax of $\lambda_{\leq}^{\rightarrow,c}$ is obtained from $\lambda_{\leq}^{\rightarrow}$ by adding the following inference rule:

$$(con) \quad \frac{S \mid A \rhd e_1 : \tau_1 \quad \ldots \quad S \mid A \rhd e_n : \tau_n}{S \mid A \rhd c(e_1, \ldots, e_n) : c(\tau_1, \ldots, \tau_n)}$$

□

Remark 3.4

It is assumed that both element and type constructors occur with a single arity within a programming context. □

The rules for the subtype relation basically express that a data type is different from any other type. This means that each type of the form $c(\tau_1, \ldots, \tau_n)$ is different from any other type $d(\tau_1', \ldots, \tau_m')$ and from any function type. Moreover, the subtyping relation between data types is required to be structural and compositional.

Definition 3.9 Subtyping

The subtyping relation of $\lambda_{\leq}^{\rightarrow,c}$ is obtained by adding to $\lambda_{\leq}^{\rightarrow}$ the following inference rules:

$$(\leq\rightarrow c)\quad \frac{}{S \vdash \tau_1 \rightarrow \tau_2 \not\leq c(\tau_1', \dots \tau_n')}$$

$$(\leq c \rightarrow)\quad \frac{}{S \vdash c(\tau_1', \dots \tau_n') \not\leq \tau_1 \rightarrow \tau_2}$$

$$(\leq cd)\quad \frac{}{S \vdash c(\tau_1, \dots \tau_n) \not\leq d(\tau_1', \dots \tau_m')}\ \{ \text{ if } c \neq d$$

$$(\leq cc)\quad \frac{S \vdash \tau_1 \leq \tau_1' \quad \dots \quad S \vdash \tau_n \leq \tau_n'}{S \vdash c(\tau_1, \dots \tau_n) \leq c(\tau_1', \dots \tau_n')}$$

□

Remark 3.5 Covariance

It can be observed from the rule ($\leq cc$) that constructors are covariant in all arguments. This corresponds to our intuition since if $\tau_i \leq \tau_i'$ than τ_i' contains more elements than τ_i and as a consequence the image $c(\tau_1', \dots, \tau_n')$ of $\tau_1', \dots, \tau_n'$ under c contains more elements than $c(\tau_1, \dots, \tau_n)$. □

3.2.4 The Calculus $\lambda_{\leq}^{\rightarrow,c,\cup,\cap,?}$

As we already mentioned in Section 3.1, in the presence of union types and constructors we can easily obtain variants. Since variant types are accompanied on the level of expressions by a *case functional*, we also expect to be able to define such a functional. Moreover, by using intersection types and conditional types we can type it more accurately as variants do. Beside these types we also introduce a *least type* 0 containing only the value ⊥, which denotes nontermination, and a *greatest type* 1 containing the entire semantic domain. More details are given in Section 3.3.

Definition 3.10 Type expressions

The type expressions of $\lambda_{\leq}^{\rightarrow,c,\cup,\cap,?}$ are obtained from $\lambda_{\leq}^{\rightarrow,c}$ by adding the following productions:

$$\tau \quad ::= \tau_1 \cup \tau_2 \mid \tau_1 \cap \tau_2 \mid \tau_1 ? \tau_2 \mid 0 \mid 1$$

□

In general when writing types we omit parenthesis and assume, the priority of constructors is from lower to upper as follows: $\rightarrow, \cup, ?, \cap, c$.

Definition 3.11 Expressions

The context free language of terms of $\lambda_{\leq}^{\rightarrow,c,\cup,\cap,?}$ is obtained from $\lambda_{\leq}^{\rightarrow,c}$ by adding the following productions:

$$\begin{array}{lcl} e & ::= & \textbf{case}\ e\ \textbf{of}\ p_1 \Rightarrow e_1 \ldots p_n \Rightarrow e_n \\ & \mid & e_1\ \textbf{extend}\ (p \Rightarrow e) \\ p & ::= & x \mid x\ \textbf{as}\ p \mid c(p_1, \ldots, p_n) \end{array}$$

□

Remark 3.6 Selectors

We do not provide selectors. Instead, functions can be defined by pattern matching. The syntactic category of patterns is given above by p. Patterns are either variables, data elements or variables restricted by patterns. □

Remark 3.7 Abbreviations

As a general rule for the syntax, we abbreviate an indexed set of expressions $\{e_i \mid i \in I\}$ with e_i and an indexed set of judgments $\{(S_i \mid A_i \rhd e_i : \tau_i) \mid i \in I\}$ with $S_i \mid A_i \rhd e_i : \tau_i$, if this causes no confusion. For example $\forall\{\alpha_i \mid 1 \leq i \leq n\}.\tau$ is written as $\forall\alpha_i.\tau$ and **case** e **of** $p_1 \Rightarrow e_1, \ldots, p_n \Rightarrow e_n$ is written as **case** e **of** $p_i \Rightarrow e_i$. □

In the type expressions given below we use two additional types. The *complement type* $\neg\tau$, which is the largest type such that $\tau \cap \neg\tau = 0$, and the *hat type* $\overline{\tau}$, which is the smallest monotype such that $\overline{\tau} \cup \neg\overline{\tau} = 1$ and for all substitutions ρ, $\rho(\tau) \leq \overline{\tau}$. If a pattern p has type τ then $\overline{\tau}$ intuitively denotes the set of all values that can match the pattern p. We did not provide these types in the type expressions' syntax because they can be written in terms of the others. More about their motivation will be said later.

Definition 3.12 Context sensitive syntax

The Context sensitive syntax of $\lambda_{\leq}^{\rightarrow,c,\cup,\cap,?}$ is obtained from $\lambda_{\leq}^{\rightarrow,c}$ by adding the following inference rules:

$$(cas)\quad \frac{\begin{array}{ll} S \mid A \rhd e : \tau & \\ S \mid A, A_{p_i} \rhd p_i : \tau_i', e_i : \tau_i & S \vdash \tau \leq \cup_{i=1}^{n} \tau_i' \end{array}}{S \mid A \rhd \textbf{case}\ e\ \textbf{of}\ p_i \Rightarrow e_i : \cup_{i=1}^{n} \tau_i ? \tau \cap \overline{\tau_i}'}$$

$$(ext)\quad \frac{\begin{array}{ll} S \mid A \rhd f : \tau'' & S \vdash \tau'' \leq \tau_1 \rightarrow \tau_2 \\ S \mid A, A_p \rhd p : \tau',\ e : \tau & S \vdash \tau_3 \leq \tau' \cup (\tau_1 \cap \neg\overline{\tau}') \end{array}}{S \mid A \rhd f\ \textbf{extend}\ (p \Rightarrow e) : \tau_3 \rightarrow (\tau ? \tau_3 \cap \overline{\tau}' \cup \tau_2 ? \tau_3 \cap \neg\overline{\tau}')}$$

$$(asp)\quad \frac{S \mid A \rhd x : \tau_1,\ p : \tau_2}{S \mid A \rhd x \ \mathbf{as}\ p : \tau_1 \cap \tau_2}$$

□

Remark 3.8 Patterns

Patterns are required to be linear (i.e. each variable occurs exactly once). In the rule (cas) they are also required to be pairwise disjoint. □

In the rules (cas) and (ext) A_p is the sort assignment for the free variables occuring in the pattern p. The rule (cas) is a generalization of the analogous rule for *sums* or *variants* because on the one side, p_i may be an arbitrary nested pattern and not only a label like inl or inr and on the other side, the expressions e_i are allowed to have different types. This generality is mainly achieved through the use of conditional types. The first constraint $\tau \leq \cup_{i=1}^{n}\tau_i'$ assures both an exhaustive analysis of the type τ and the propagation of the type information about e to the output type $\cup_{i=1}^{n}\tau_i?\tau \cap \overline{\tau_i}'$. If e has type τ_k then, because the patterns are required to be pairwise disjoint, only the intersection $\tau_k \cap \overline{\tau_k}'$ is different from 0 and the result is of type τ_k.

The rule (ext) allows to *extend* a (case–) function with a new branch $p \Rightarrow e$ which *overrides* any branch $p' \Rightarrow e'$ in f matching an instance of the pattern p. It is very similar in spirit with the record extension rule [25] but it directly allows to extend functions instead of their encoding as records. Moreover, f may have an arbitrary body and not only a case expression. The overriding effect can be read from the type $\tau_3 \rightarrow (\tau?\tau_3 \cap \overline{\tau}' \cup \tau_2?\tau_3 \cap \neg\overline{\tau}')$ where $\tau_3 \leq \tau' \cup (\tau_1 \cap \neg\overline{\tau}')$ as follows: if the argument is in $\overline{\tau}'$ (i.e. it matches p) then $\tau_3 \cap \overline{\tau}' \neq 0$ and the result is in τ; otherwise the argument is both in the complement $\neg\overline{\tau}'$ of $\overline{\tau}'$ and in τ_1 i.e. in $\tau_1 \cap \neg\overline{\tau}'$ and the result is in τ_2. The reason for using the disjoint union $\tau' \cup (\tau_1 \cap \neg\overline{\tau}')$ instead of $\tau' \cup \tau_1$ and more details about $\overline{\tau}$ and $\neg\overline{\tau}$ are given in Section 3.2.7.

The subtyping relation is defined by considering the properties of union, intersection and conditional types. It is also assumed that data constructors and functions are lazy i.e. that $\mathrm{c}(\perp, \ldots, \perp) \neq\perp$ and that $\lambda x.\ \perp\neq\perp$ respectively.

Definition 3.13 Subtyping

The subtyping relation of $\lambda_{\leq}^{\rightarrow,c,\cup,\cap,?}$ is obtained by adding to $\lambda_{\leq}^{\rightarrow,c}$ the following inference rules:

$$(\leq \cap r)\quad \frac{S \vdash \tau \ \leq\ \tau_1 \quad S \vdash \tau \ \leq\ \tau_2}{S \vdash \tau \ \leq\ \tau_1 \cap \tau_2}$$

$$(\leq \cap lb)\quad \frac{}{S \vdash \tau_1 \cap \tau_2 \ \leq\ \tau_i}\ \{\ i = 1, 2$$

$$(\leq \cup l) \quad \frac{S \vdash \tau_1 \ \leq \ \tau \quad S \vdash \tau_2 \ \leq \ \tau}{S \vdash \tau_1 \cup \tau_2 \ \leq \ \tau}$$

$$(\leq \cup gb) \quad \frac{}{S \vdash \tau_i \ \leq \ \tau_1 \cup \tau_2} \ \{ \ i = 1, 2$$

$$(\leq ?1) \quad \frac{S \vdash \tau_1 \ \leq \ \tau}{S \vdash \tau_1 ? \tau_2 \ \leq \ \tau}$$

$$(\leq ?2) \quad \frac{S \vdash \tau_2 \ \leq \ 0}{S \vdash \tau_1 ? \tau_2 \ \leq \ \tau}$$

$$(\leq 0l) \quad \frac{}{S \vdash 0 \ \leq \ \tau}$$

$$(\leq 0c) \quad \frac{}{S \vdash c(\tau_1, \ldots, \tau_n) \ \not\leq \ 0}$$

$$(\leq 0 \rightarrow) \quad \frac{}{S \vdash \tau_1 \rightarrow \tau_2 \ \not\leq \ 0}$$

□

3.2.5 The Calculus $\lambda_{\leq}^{\forall,\rightarrow,c,\cup,\cap,?}$

The notion of *bounded quantification* was introduced by Cardelli and Wegner [26] in the language Fun. This language integrates the Girard–Reynolds parametric polymorphism [42, 85] with Cardelli's first order calculus of subtyping [22, 23]. A bounded polymorphic type $\forall\alpha \leq \sigma.\ \tau$ constrains the type variable α to range only over subtypes of σ. Since we are mainly interested in the existence of a type inference algorithm, we retain the constraint idea, but use instead of the second order polymorphism of Fun, the simpler ML polymorphism [63, 38]. However, we generalize the constraint $\alpha \leq \sigma$ to proper sets of constraints.

Definition 3.14 Type expressions

The type expressions of $\lambda_{\leq}^{\forall,\rightarrow,c,\cup,\cap,?}$ are obtained from $\lambda_{\leq}^{\rightarrow,c,\cup,\cap,?}$ by adding the following productions:

$$\sigma \ ::= \tau \mid \forall\alpha_i.\tau \ \textbf{where} \ S$$

□

The type $\forall\alpha_i.\tau$ **where** S is said to be polymorphic.

Remark 3.9 Shallow polymorphism

> The universal quantifier binds sets of variables $\{\alpha_1, \ldots, \alpha_n\}$. It cannot be nested in the form $\forall S.\forall T.\tau$. Moreover, polymorphic types cannot occur in the argument positions of the type constructors. This is the reason why, this type of polymorphism is also called *shallow polymorphism.* □

As in [63, 38], to enable the definition and use of polymorphic terms, we extend the term language with the **let** construct.

Definition 3.15 Expressions

> The context free language of terms of $\lambda_{\leq}^{\forall,\rightarrow,c,\cup,\cap,?}$ is obtained from $\lambda_{\leq}^{\rightarrow,c,\cup,\cap,?}$ by adding the following production:
>
> $$e \;::=\; \textbf{let } x = e_1 \textbf{ in } e_2$$
>
> □

The variable x occuring in the **let** declaration is polymorphic. As a consequence, different occurrences of x in e_2 may have different types. Each of them must be an instance of the polymorphic one. This polymorphic type is given by a generalization operation $Gen(S, A, \tau) = \forall\alpha_i.\tau$ **where** S where $\alpha_i = \mathrm{FTV}(\tau) - \mathrm{FTV}(A)$. The instantiation of polymorphic types is given by a modified form of the rule (var). Finally, typing contexts A are extended to include polymorphic variables.

Definition 3.16 Context sensitive syntax

> The context sensitive syntax of $\lambda_{\leq}^{\forall,\rightarrow,c,\cup,\cap,?}$ is obtained from $\lambda_{\leq}^{\rightarrow,c,\cup,\cap,?}$ by adding the following inference rules[6]:
>
> $$(var)\quad \frac{x : \sigma \in A}{S[\tau_i/\alpha_i] \mid A \rhd x : \tau[\tau_i/\alpha_i]}\ \{\sigma = \forall\alpha_i.\tau \textbf{ where } S$$
>
> $$(let)\quad \frac{S \mid A \rhd e_1 : \tau_1 \quad S' \mid A, x : \sigma \rhd e_2 : \tau_2}{S, S' \mid A \rhd \textbf{let } x = e_1 \textbf{ in } e_2 : \tau_2}\ \{\sigma = Gen(S, A, \tau_1)$$
>
> □

If e is a closed term and $S \mid \emptyset \rhd e : \tau$, then we consider the type inferred for e to be $\sigma = Gen(S, \emptyset, \tau)$ and write $\rhd\ e : \sigma$. In general, given a typing context A and a subtyping context S such that $S \mid A \rhd e : \tau$, we write $A \rhd e : \sigma$ where $\sigma = Gen(S, A, \tau)$.

[6] The rule (var) replaces the old (var) rule.

3.2.6 Recursion

Recursive types are not included in the grammar for type expressions because they are definable using constraints. For example, the type of lists with elements of type β is the unique solution of the recursive constraint $\alpha = cons(\beta, \alpha) \cup nil$.

In general the set S from a type $\forall\alpha.\tau$ **where** S contains both recursive subtyping constraints and recursive equations. For example, if we define the following function:

```
pt = $\lambda xc.\lambda s.$
     case $s$ of
          cons(x,$t$) $\Rightarrow$ cons($xc$, pt $xc$ $t$)
```

then type of pt 1 is as follows:

$\forall\alpha.\alpha \rightarrow \mathsf{cons}(\mathsf{Int}, \beta)?\alpha \cap \mathsf{cons}(\mathsf{X}, 1)$
where
$\quad 0 \leq \alpha \leq \mathsf{cons}(\mathsf{X},\alpha)$
$\quad \beta = \mathsf{cons}(\mathsf{Int},\beta)\ ?\ \alpha \cap \mathsf{cons}(\mathsf{X},1)$

We say that α is free in S and that β is bound in S because if the type $\mathsf{cons}(\mathsf{Int},\beta)\ ?\ \alpha \cap \mathsf{cons}(\mathsf{X},1)$ is contractive in β then the above equation is guaranteed to have a *unique* fixed point which is written as:

$\mu\beta.\ \mathsf{cons}(\mathsf{Int},\beta)\ ?\ \alpha \cap \mathsf{cons}(\mathsf{X},1)$

Intuitively, a type constructor f is contractive if it decreases the difference (or distance) d between types, i.e. it exists an r with $0 < r < 1$ such that $d(f(\tau_1), f(\tau_2)) \leq r\, d(\tau_1, \tau_2)$.

Definition 3.17 Contractive types

> Let τ be a type expression and β a type variable. The predicate $\tau \succ \beta$ read as τ is *contractive* in β, is defined inductively on the structure of τ as follows:
>
> $$\begin{array}{ll} 0 \succ \beta & 1 \succ \beta \\ \overline{\tau} \succ \beta & \neg\overline{\tau} \succ \beta \\ c(\tau_1, \ldots, \tau_n) \succ \beta & \tau_1 \rightarrow \tau_2 \succ \beta \\ \alpha \succ \beta \Leftrightarrow \alpha \neq \beta & \tau_1 ? \tau_2 \succ \beta \Leftrightarrow \tau_1 \succ \beta \wedge \tau_2 \succ \beta \\ \tau_1 \cup \tau_2 \succ \beta \Leftrightarrow \tau_1 \succ \beta \wedge \tau_2 \succ \beta & \tau_1 \cap \tau_2 \succ \beta \Leftrightarrow \tau_1 \succ \beta \wedge \tau_2 \succ \beta \end{array}$$

□

The set of variables $\mathrm{TLV}(\tau) = \{\alpha \mid \tau \not\succ \alpha\}$ are called in [5] *top level variables*. As we shall see in Section 3.3 every set of *inductive* constraints $\{l_i \leq \alpha_i \leq u_i \mid 1 \leq i \leq n\}$

is *cascading* i.e. $\mathrm{TLV}(l_i) \cap \mathrm{TLV}(u_i) \subseteq \{\alpha_1, \ldots, \alpha_{i-1}\}$. We use this property in the rules given below. They are induction rules for proving the equality and inequality for recursive types. The rules are given in the form $S \mid A \vdash \tau_1 \leq \tau_2$ where A is a set of subtyping assumptions used in the induction.

Rules for Free Variables

$$(\leq vl) \quad \frac{}{S, l \leq \alpha \leq u \mid A, \alpha \leq \tau \vdash \alpha \leq \tau}$$

$$(\leq vr) \quad \frac{}{S, l \leq \alpha \leq u \mid A, \tau \leq \alpha \vdash \tau \leq \alpha}$$

$$(\leq \mu vl) \quad \frac{S, l \leq \alpha \leq u \mid A, \alpha \leq \tau \vdash u \leq \tau}{S, l \leq \alpha \leq u \mid A \vdash \alpha \leq \tau}$$

$$(\leq \mu vr) \quad \frac{S, l \leq \alpha \leq u \mid A, \tau \leq \alpha \vdash \tau \leq l}{S, l \leq \alpha \leq u \mid A \vdash \tau \leq \alpha}$$

$$(\leq \mu vlr)_1 \quad \frac{S, l \leq \alpha_j \leq u \mid A, \alpha_i \leq \alpha_j \vdash \alpha_i \leq l}{S, l \leq \alpha_j \leq u \mid A \vdash \alpha_i \leq \alpha_j} \; \{i \leq j$$

$$(\leq \mu vlr)_2 \quad \frac{S, l \leq \alpha_i \leq u \mid A, \alpha_i \leq \alpha_j \vdash u \leq \alpha_j}{S, l \leq \alpha_i \leq u \mid A \vdash \alpha_i \leq \alpha_j} \; \{j \leq i$$

In $(\leq \mu vlr)_1$ and $(\leq \mu vlr)_2$ we used the cascading property of inductive constraints. If $i \leq j$ then α_i may occur in the lower and respectively upper bounds of α_j.

Rules for Bound Variables

$$(\leq bl) \quad \frac{}{S, \alpha = \tau_1 \mid A, \alpha \leq \tau_2 \vdash \alpha \leq \tau_2}$$

$$(\leq br) \quad \frac{}{S, \alpha = \tau_1 \mid A, \tau_2 \leq \alpha \vdash \tau_2 \leq \alpha}$$

$$(\leq \mu bl) \quad \frac{S, \alpha = \tau_1 \mid A, \alpha \leq \tau_2 \vdash \tau_1 \leq \tau_2}{S, \alpha = \tau_1 \mid A \vdash \alpha \leq \tau_2} \; \{\tau_1 \succ \alpha$$

$$(\leq \mu br) \quad \frac{S, \alpha = \tau_1 \mid A, \tau_2 \leq \alpha \vdash \tau_2 \leq \tau_1}{S, \alpha = \tau_1 \mid A \vdash \tau_2 \leq \alpha} \; \{\tau_1 \succ \alpha$$

$$(\leq \mu blr)\quad \frac{S, \alpha = \tau_1, \beta = \tau_2 \mid A, \alpha \leq \beta \vdash \tau_1 \leq \tau_2}{S, \alpha = \tau_1, \beta = \tau_2 \mid A \vdash \alpha \leq \beta}\ \{\tau_1 \succ \alpha,\ \tau_2 \succ \beta$$

Note the similarity of the rule $(\leq \mu blr)$ with the rule given by Cardelli for recursive types [1]:

$$(\leq \mu)\quad \frac{S, \alpha \leq \beta \vdash \tau_1 \leq \tau_2}{S \vdash \mu\alpha.\tau_1 \leq \mu\beta.\tau_2}\ \{\tau_1 \succ \alpha,\ \tau_2 \succ \beta$$

3.2.7 The Type Inference Algorithm

Type inference is the general problem of transforming untyped or partially typed terms into well–typed terms by inferring missing type information. The motivation for type inference is pragmatic. On the one side it allows to program in a typed language, where type errors are detected at compile time. On the other side, the tedious process of declaring the type of every variable is made optional. This is particularly useful in polymorphic languages since polymorphism involves quite a bit of type information.

Given an untyped closed term e it is in general possible to infer more than one type τ such that $\emptyset \rhd e : \tau$. A type τ is *more general* than another type τ' if there is a substitution ρ such that $\tau' = \rho\tau$. For the ML language Milner proposed the algorithm W [63] which given a term e and a type assignment A computes a substitution ρ and a most general type (or principal type) τ such that $\rho A \rhd e : \tau$. This algorithm interleaves the production of equational constraints with their resolution by unification. For example, using the notation $\rho A \vdash^W e : \tau$ for $W(A, e) = (\rho, \tau)$ as suggested by Remy in [81], the rule $(\rightarrow e)$ is given as follows:

$$(\rightarrow e)\quad \frac{\rho A \vdash^W e_1 : \tau_1 \quad \rho'\rho A \vdash^W e_2 : \tau_2 \quad \rho'\tau_1 \overset{U}{=} \tau_2 \rightarrow \alpha}{U\rho'\rho A \vdash^W e_1\ e_2 : U\alpha}\ \{\ \alpha \text{ new}$$

However, as shown by Wand in [92], the generation and the resolution of the equational constraints can be separated by splitting the above algorithm in two parts: the first one which generates a set of constraints and the second one which solves them by unification. For example, writing $E \mid A \vdash^{W'} e : \tau$ for $W'(A, e) = (E, \tau)$ the generation and accumulation of constraints for $(\rightarrow e)$ can be given as follows:

$$(\rightarrow e)\quad \frac{E_1 \mid A \vdash^{W'} e_1 : \tau_1 \quad E_2 \mid A \vdash^{W'} e_2 : \tau_2}{E_1, E_2, \tau_1 = \tau_2 \rightarrow \alpha \mid A \vdash^{W'} e_1\ e_2 : \alpha}\ \{\ \alpha \text{ new}$$

If $W'(A, e) = (E, \tau)$ and U is the most general unifier for E (if this exists) then $(U, U\tau)$ is equal with (ρ, τ) as given by W. Thus, the type inference problem can be totally reduced to a unification problem.

The Constraint Accumulation Algorithm

We use a similar technique where the equational constraints are replaced by subtyping constraints. As an immediate consequence, the algorithm W is obtained as a particular case of ours. The algorithm Z, for accumulating constraints is given below. Note the similarity with the type inference system.

$$(var)^z \quad \frac{(x : \forall\alpha_i.\tau \ \mathbf{where}\ S) \ \in A}{S[\beta_i/\alpha_i] \mid A \vdash^Z x : \tau[\beta_i/\alpha_i]} \ \{\beta_i \text{ new}$$

$$(\rightarrow i)^z \quad \frac{S \mid A, x : \alpha \vdash^Z e : \tau}{S \mid A \vdash^Z \lambda x.e : \alpha \rightarrow \tau} \ \{\alpha \text{ new}$$

$$(\rightarrow e)^z \quad \frac{S_1 \mid A \vdash^Z e_1 : \tau_1 \quad S_2 \mid A \vdash^Z e_2 : \tau_2}{S_1, S_2, \tau_1 \leq \alpha \rightarrow \beta, \tau_2 \leq \alpha \mid A \vdash^Z e_1\ e_2 : \beta} \ \{\alpha, \beta \text{ new}$$

$$(let)^z \quad \frac{S_1 \mid A \vdash^Z e_1 : \tau_1 \quad S_2 \mid A, x : \sigma \vdash^Z e_2 : \tau_2}{S_1, S_2 \mid A \vdash^Z \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2} \ \{\sigma = Gen(S_1, A, \tau_1)$$

$$(con)^z \quad \frac{S_1 \mid A \vdash^Z e_1 : \tau_1 \quad \ldots \quad S_n \mid A \vdash^Z e_n : \tau_n}{S_1, \ldots, S_n \mid A \vdash^Z c(e_1, \ldots, e_n) : c(\tau_1, \ldots, \tau_n)}$$

$$(cas)^z \quad \frac{S \mid A \vdash^Z e : \tau \quad S_i \mid A, A_{p_i} \vdash^Z p_i : \tau_i', e_i : \tau_i}{\begin{array}{l} S, S_1, \ldots, S_n, \\ \tau \leq \cup_{i=1}^n \tau_i' \end{array} \mid A \vdash^Z \mathbf{case}\ e\ \mathbf{of}\ p_i \Rightarrow e_i : \cup_{i=1}^n \tau_i ? \tau \cap \overline{\tau_i}'}$$

$$(ext)^z \quad \frac{S'' \mid A \vdash^Z f : \tau'' \quad S \mid A, A_p \vdash^Z p : \tau',\ e : \tau}{\begin{array}{ll} S'', & \tau'' \leq \alpha \rightarrow \beta, \\ S, & \gamma \leq \tau' \cup (\alpha \cap \neg\overline{\tau}') \end{array} \mid A \vdash^Z \begin{array}{l} f\ \mathbf{extend}\ (p \Rightarrow e) : \\ \gamma \rightarrow (\tau ? \gamma \cap \overline{\tau}' \cup \beta ? \gamma \cap \neg\overline{\tau}') \end{array}} \ \{\alpha, \beta, \gamma \text{ new}$$

$$(asp)^z \quad \frac{S_1 \mid A \vdash^Z x : \tau_1 \quad S_2 \mid A \vdash^Z p : \tau_2}{S_1, S_2 \mid A \vdash^Z x\ \mathbf{as}\ p : \tau_1 \cap \tau_2}$$

Theorem 3.1 Termination

The algorithm Z always terminates.

Proof: Each rule generates subgoals involving terms smaller than the original. □

Definition 3.18 Minimal type

A closed term e has a minimal type σ iff $\triangleright e : \sigma$ and for any σ', if $\triangleright e : \sigma'$ then $\sigma \leq \sigma'$. □

Minimal types are related to but not the same as principal types. In the type systems presented so far, closed terms have minimal types but may not have principal types (e.g. because many type expressions denote the same type). A minimal type is the smallest derivable type in the semantical model.

Lemma 3.1 Minimal Type

If $S \mid A \vdash^{Z} e : \tau$ and S is a solvable set of constraints then $\sigma = Gen(S, A, \tau)$ is the minimal type for e.

Proof: The type σ has the form $\forall \alpha_i . \tau$ **where** S. By the inspection of the typing rules any other type σ' such that $A \triangleright e : \sigma'$ is equal to $\forall \alpha_i . \tau$ **where** S'. Since S contains the minimal set of subtyping constraints and these are in their most general form (with fresh type variables) it follows that the set of solutions $\mathrm{Sol}(S') \subseteq \mathrm{Sol}(S)$. But the meaning of a quantified type is the intersection of the meaning of τ in all solutions of the constraints. As a consequence $\sigma \leq \sigma'$. □

The Constraint Solving Algorithm

The above algorithm reduces the type inference problem for a term e to the resolution of the associated set of subtyping constraints S. The algorithm described in this section resolves S by attempting to transform it into an equivalent set of constraints $S' = \{l_i \leq \alpha_i \leq u_i \mid 1 \leq i \leq n\}$ which is inductive[7] and therefore is guaranteed to have solutions. If the transformation succeeds then the term t is well typed, otherwise it is not.

Each step of the transformation should replace a constraint $c \in S$ with the simpler set of constraints SC given by applying the corresponding subtyping rule backwards. By the soundness of the rules, S and $(S - \{c\}) \cup SC$, have the same solutions.

However, the subtyping rules for $\lambda_{\leq}^{\forall,\rightarrow,\cup,\cap,?}$ do not directly constitute an algorithm. In particular, for a constraint $\tau_1 ? \tau_2 \leq \tau$ one can either apply rule $(\leq ?1)$ or the rule $(\leq ?2)$ backwards to get the simpler constraints $\tau_1 \leq \tau$ and $\tau_2 \leq 0$. To take into

[7] Inductive constraints and sets of inductive constraints are defined in in Section 3.3.3

account both cases we write the rules in the form

$$\Gamma \mid S, \tau_1 ? \tau_2 \leq \tau \rightsquigarrow \Gamma \mid S, \tau_1 \leq \tau \mid S, \tau_2 \leq 0$$

where $\mid$ separates constraint systems and Γ is a list of constraint systems. Obviously the union of the solutions for the right–hand side systems is equivalent with the union of the solutions for the left–hand side systems.

The rule $(\leq tra)$ cannot effectively be applied backwards since this would involve "guessing" an appropriate value for the intermediate type. Instead of $(\leq ref)$ and $(\leq tra)$ we use the following reformulation:

$$(\leq ref) \quad \frac{}{S \vdash \alpha \leq \alpha}$$

$$(\leq tra) \quad \frac{S \vdash \tau_1 \leq \alpha \leq \tau_2}{S \vdash \tau_1 \leq \tau_2}$$

The reflexivity axiom is restricted to variables, and the transitivity rule is applied in the forward direction in a restricted form as shown below.

There are two cases of constraints which cannot be further decomposed by the rules given so far: $\tau_1 \cap \tau_2 \leq \tau$ and $\tau \leq \tau_1 \cup \tau_2$. For the last constraint one can try to "simplify" it by using set complement and write $\tau \leq \tau_1 \cup \tau_2 \Leftrightarrow \tau \cap (1 - \tau_1) \leq \tau_2$. However, in the semantical interpretation of type expressions given in Section 3.3, the set $1 - \tau_1$ may not be downward closed and consequently not a type. For example $1 - (1 \rightarrow 0)$ contains every function except the least one $\lambda x.\ \bot$. This motivates the following:

Definition 3.19 Complement

> The type expression $\neg\tau$ denotes the largest type such that $\tau \cap \neg\tau = 0$. □

Since $\neg\tau$ is not the set complement of τ one has to impose restrictions on the constraints $\tau_1 \cap \tau_2 \leq \tau$ and $\tau \leq \tau_1 \cup \tau_2$ that can be solved. For example it is not the case that $\tau \leq \tau_1 \cup \tau_2 \Leftrightarrow \tau \cap \neg\tau_1 \leq \tau_2$. The statement is however true if $\tau_1 \cup \neg\tau_1 = 1$.

Definition 3.20 Hat type

> The *hat type* $\overline{\tau}$ of a type τ is the smallest monotype such that $\overline{\tau} \cup \neg\overline{\tau} = 1$ and for all substitutions ρ, $\rho(\tau) \leq \overline{\tau}$ □

The above result about hat types can be used to provide a general way of decomposing constraints $\tau \leq \tau_1 \cup \tau_2$.

Lemma 3.2 Unions decomposition

Let $\tau \leq \tau_1 \cup \tau_2$ such that $\rho(\tau_1 \cap \tau_2) = 0$ for all substitutions ρ. Then:

$$\tau \leq \tau_1 \cup \tau_2 \Leftrightarrow (\tau \cap \neg\overline{\tau_1} \leq \tau_2) \ \wedge \ (\tau \cap \neg\overline{\tau_2} \leq \tau_1)$$

Proof: The proof is given in [5]. It basically uses the fact that $\tau \leq \overline{\tau}$, that $\tau \leq \overline{\tau_1} \cup \tau_2 \Leftrightarrow \tau \cap \neg\overline{\tau_1} \leq \tau_2$ and that $\forall\rho.\rho(\tau_1 \cap \tau_2) = 0$ iff $\overline{\tau_1 \cap \tau_2} = 0$. □

For the constraints $\tau_1 \cap \tau_2 \leq \tau$ if the intersection is restricted to the form $\tau_1 \cap \overline{\tau_2} \leq \tau$ then the following equivalence holds:

Lemma 3.3 Intersections decomposition

$$\tau_1 \cap \overline{\tau_2} \leq \tau \Leftrightarrow \tau_1 \leq (\overline{\tau_2} \cap \tau) \cup \neg\overline{\tau_2}$$

Proof:

$\tau_1 \ \cap \ \neg(\neg\overline{\tau_2}) \leq \tau \ \Leftrightarrow \ \tau_1 \leq \neg\overline{\tau_2} \ \cup \ \tau$

$\neg\overline{\tau_2} \ \cup \ \tau \ \Leftrightarrow \ \neg\overline{\tau_2} \ \cup \ \tau \cap (\neg\overline{\tau_2} \cup \overline{\tau_2}) \ \Leftrightarrow \ \neg\overline{\tau_2} \ \cup \ \tau \cap \overline{\tau_2}.$ □

The type τ is written above as $\overline{\tau_2} \cap \tau$ in order to get a disjoint union on the right which can be further simplified by using the previous lemma. In conclusion, the following restrictions apply for constraints:

- Unions on the right of constraints must be disjoint,
- Intersections on the left of constraints must have the form $\tau_1 \cap \overline{\tau_2}$.

The above restrictions are formalized by the following grammar [5]:

$$\begin{array}{lll} l & ::= 0 \mid \alpha \mid r \rightarrow l \mid c(l_1, \ldots, l_n) \mid l_1 \cap \overline{l_2} \mid l_1 \cup l_2 & \\ r & ::= 0 \mid \alpha \mid l \rightarrow r \mid c(r_1, \ldots, r_n) \mid r_1 \cap r_2 \mid r_1 \cup r_2 & \text{where } \overline{r_1 \cap r_2} = 0 \end{array}$$

Definition 3.21 Proper constraints

A system of *proper constraints* has the form $\{l_i \leq r_i \mid 1 \leq i \leq n\}$. □

Remark 3.10 Inference rules

The constraint accumulation algorithm generates proper constraints if all the primitive functions have l–types. This is not very restrictive since functions tend to have l–types. □

The type 1 does not appear in the grammar because it can be defined using proper constraints as $\alpha = (0 \to \alpha) \cup \bigcup_{c \in C} c(\alpha, \ldots, \alpha)$. Moreover, $\overline{\tau}$ and $\neg\overline{\tau}$ can be eliminated from type expressions. An algorithm for their simultaneous elimination is given in Appendix B.3. The constraint simplification rules given below also assume that the type expressions are in disjunctive normal form. A normalization algorithm is given in Appendix B.4.

$(\le 0l)$	$\Gamma \mid S, 0 \le r$	$\rightsquigarrow$	$\Gamma \mid S$
$(\le cc)$	$\Gamma \mid S, c(l_1, \ldots l_n) \le c(r_1, \ldots r_n)$	$\rightsquigarrow$	$\Gamma \mid S, l_1 \le r_1, \ldots, l_n \le r_n$
$(\le\to)$	$\Gamma \mid S, r_1 \to l_1 \le l_2 \to r_2$	$\rightsquigarrow$	$\Gamma \mid S, l_2 \le r_1, l_1 \le r_2$
$(\le \cup l)$	$\Gamma \mid S, l_1 \cup l_2 \le r$	$\rightsquigarrow$	$\Gamma \mid S, l_1 \le r, l_2 \le r$
$(\le \cap r)$	$\Gamma \mid S, l \le r_1 \cap r_2$	$\rightsquigarrow$	$\Gamma \mid S, l \le r_1, l \le r_2$
$(\le ?12)$	$\Gamma \mid S, l_1 ? l_2 \le r$	$\rightsquigarrow$	$\Gamma \mid S, l_1 \le r \mid S, l_2 \le 0$
$(\le \cup r)$	$\Gamma \mid S, l \le r_1 \cup r_2$	$\rightsquigarrow$	$\Gamma \mid S, l \cap \neg\overline{r_1} \le r_2, l \cap \neg\overline{r_2} \le r_1$
$(\le \mathit{ref})$	$\Gamma \mid S, \alpha \le \alpha$	$\rightsquigarrow$	$\Gamma \mid S$
$(\le \cap lb)$	$\Gamma \mid S, \alpha \cap \overline{l} \le \alpha$	$\rightsquigarrow$	$\Gamma \mid S$
$(\le \cap l)$	$\Gamma \mid S, \alpha \cap \overline{l} \le r$	$\rightsquigarrow$	$\Gamma \mid S, \alpha \le (r \cap \overline{l}) \cup \neg\overline{l}$

The Constraint Simplification Rules

Remark 3.11

The inequalities $(\le\to c), (\le c \to), (\le cd), (\le 0c), (\le 0 \to)$ are not included in the above rules. They are used by the algorithm given below to detect inconsistent systems of constraints. Moreover, the rule $(\le \cup gb)$ is replaced by $(\le \cup r)$ and the rule $(\le \cap lb)$ is rewritten as above. □

Definition 3.22 Inductive constraints

A constraint $\tau \le \alpha_i$ or $\alpha_i \le \tau$ is inductive iff $TLV(\tau) \subseteq \{\alpha_1, \ldots, \alpha_{i-1}\}$. □

Now, the constraint solving algorithm can be presented as follows.

Definition 3.23 The constraint solving algorithm

Given a set S of proper constraints, iterate the following steps until all constraints are inductive, no additional inductive constraints can be added and there are no inconsistent systems:

1. For any constraint that is not inductive use the first (top down) applicable constraint simplification rule.
2. For every pair of inductive constraints $l \leq \alpha_i$ and $\alpha_i \leq r$ in a system S add the transitive constraint $l \leq r$ to S.
3. Delete any system from Γ with a constraint
$$\begin{array}{lll} c(\tau'_1, \ldots \tau'_n) \leq \tau_1 \rightarrow \tau_2, & c(\tau_1, \ldots \tau_n) \leq d(\tau'_1, \ldots \tau'_m), & 1 \leq 0, \\ \tau_1 \rightarrow \tau_2 \leq c(\tau'_1, \ldots \tau'_n), & c(\tau_1, \ldots, \tau_n) \ \leq \ 0, & \tau_1 \rightarrow \tau_2 \ \leq \ 0 \end{array}$$
because it has no solutions.

Finally, for each $S \in \Gamma$ combine lower bounds $l_1 \leq \alpha,\ l_2 \leq \alpha$ into $l_1 \cup l_2 \leq \alpha$ and upper bounds $\alpha \leq r_1,\ \alpha \leq r_2$ into $\alpha \leq r_1 \cap r_2$. The result is a set of inductive systems. □

Theorem 3.2 Proper constraints

Every solvable system of proper constraints $\{l_i \leq r_i \mid 1 \leq i \leq n\}$ is equivalent to a finite set of inductive systems which is found by the constraint solving algorithm.

Proof: The algorithm presented above generates less constraint systems and detects more inconsistencies as the algorithm presented in [5] because our data constructors are lazy. For example we detect $S, c(\tau_1, \ldots \tau_n) \leq d(\tau'_1, \ldots \tau'_m)$ as inconsistent whereas in [5] this is reduced to $S, \tau_1 \ \leq \ 0 \ \mid \ \ldots \ \mid \ S, \tau_n \ \leq \ 0$. However, all our constraints are also generated in [5]. Since this algorithm always finds the equivalent set of inductive systems if this exists, so does ours. □

As an immediate consequence is the following lemma.

Lemma 3.4 Decidability

It is decidable whether systems of proper constraints have solutions. Furthermore, all solutions can be exhibited. □

3.3 The Semantics

3.3.1 Curry Semantics versus Church Semantics

Work in the semantics of typed programming languages and λ–calculi may be roughly divided into two philosophical camps. The first one, sometimes called *Curry–style semantics* takes the semantics of an expression to be the semantics of the pure λ–term found by erasing any type annotation it may contain. The other one, sometimes called *Church–style semantics* views the expressions of a typed λ–calculus as a linear shorthand for fully typed forms in which every phrase and sub–phrase is annotated with its typing; it is these fully explicit forms, i.e. the typing derivations of the calculus, to which a semantic interpretation is given. In general, Curry–style systems correspond to the left–hand side of the following diagram, while Church presentations correspond to the right hand side.

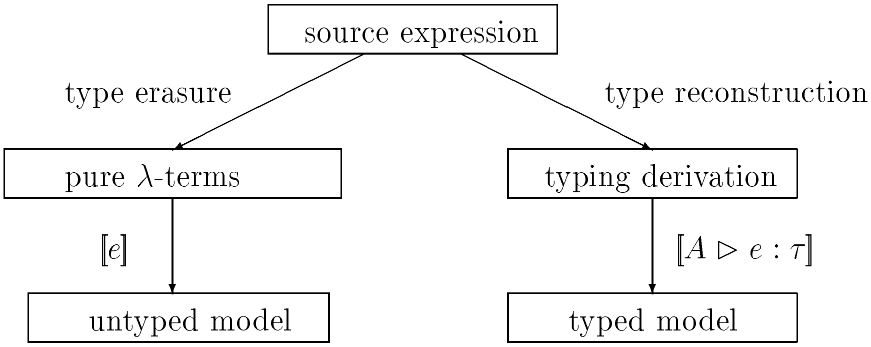


Figure 3.2: Church versus Curry semantics

The two perspectives have also been called the *epistemological* and the *ontological* views of types [55] since one is primarily concerned with *knowledge,* the other with *being;* extrinsic and intrinsic. Both views yield sensible and useful interpretations.

3.3.2 Typed Semantics

In Church–style type systems, commonly referred to as *typed λ–calculi* typing has behavioral force: it is not a *description* of semantics but *an integral part* of semantics. The interpretation function $[\![-]\!]$ is defined by induction on typing derivations, not on the underlying terms. In cases where typing derivations contain a subsidiary subtyping derivation, the latter is mapped into a function between semantic domains – a derivation whose conclusion is $\sigma \leq \tau$ is mapped into a *coercion function* from $[\![\sigma]\!]$ to $[\![\tau]\!]$.

In order to give a semantics to the implicitly typed language $\lambda_{\leq}^{\forall,\rightarrow,c,\cup,\cap,?}$ we can proceed in an analogous way as Mitchell and Harper did for Standard ML in [71].

First, we introduce an explicitly typed variant $\mathrm{T}\lambda_{\leq}^{\forall,\rightarrow,c,\cup,\cap,?}$ for $\lambda_{\leq}^{\forall,\rightarrow,c,\cup,\cap,?}$. The intention is that any implicitly typed term from $\lambda_{\leq}^{\forall,\rightarrow,c,\cup,\cap,?}$ may be regarded as a convenient shorthand for an explicitly typed term in $\mathrm{T}\lambda_{\leq}^{\forall,\rightarrow,c,\cup,\cap,?}$. Given a semantics for the target language, a translation process can be used to study the meaning of programs in the source language.

The relationship between the untyped and the typed languages can be pictured as in Figure 3.3. The two vertical arrows ϕ and ψ are functions mapping derivations to the typing that appear as their conclusion and the horizontal arrow $\rightsquigarrow$ represents the translation process.

$$\begin{tikzcd}
\boxed{\text{untyped } \lambda_{\leq}^{\forall,\rightarrow,c,\cup,\cap,?} \text{ derivations}} \arrow[r, "\rightsquigarrow"] \arrow[d, "\phi"'] & \boxed{\text{typed } \mathrm{T}\lambda_{\leq}^{\forall,\rightarrow,c,\cup,\cap,?} \text{ derivations}} \arrow[d, "\psi"] \\
\boxed{\text{untyped } \lambda_{\leq}^{\forall,\rightarrow,c,\cup,\cap,?} \text{ typings}} & \boxed{\text{typed } \mathrm{T}\lambda_{\leq}^{\forall,\rightarrow,c,\cup,\cap,?} \text{ typings}}
\end{tikzcd}$$

Figure 3.3: The relationship between typed and untyped derivations

Rather than giving separate definitions for the untyped calculus, the typed calculus and the translation between them, it is convenient to define all three using judgments of the form $S \mid A \rhd e \rightsquigarrow e' : \sigma$ where e is the untyped term and e' is its corresponding typed translation[8]. The rules are given below, where a coercion function c from σ to τ is written as $c : \sigma \leq \tau$ and interpreted as an *evidence* of the subtyping relation. Moreover, S is also extended to contain evidences.

$$(var) \quad \frac{}{S \mid x : \sigma \rhd x \rightsquigarrow x : \sigma}$$

$$(\rightarrow i) \quad \frac{S \mid A, x : \tau_1 \rhd e \rightsquigarrow e' : \tau_2}{S \mid A \rhd \lambda x.e \rightsquigarrow \lambda x : \tau_1.e' : \tau_1 \rightarrow \tau_2}$$

$$(\rightarrow e) \quad \frac{\begin{array}{ll} S \mid A \rhd e_1 \rightsquigarrow e_1' : \tau_1 & S \vdash c_1 : \tau_1 \leq \tau_3 \rightarrow \tau_4 \\ S \mid A \rhd e_2 \rightsquigarrow e_2' : \tau_2 & S \vdash c_2 : \tau_2 \leq \tau_3 \end{array}}{S \mid A \rhd e_1 \ e_2 \rightsquigarrow (c_1 e_1') \ (c_2 e_2') : \tau_4}$$

$$(con) \quad \frac{S \mid A \rhd e_1 \rightsquigarrow e_1' : \tau_1 \quad \ldots \quad S \mid A \rhd e_n \rightsquigarrow e_n' : \tau_n}{S \mid A \rhd c(e_1, \ldots, e_n) \rightsquigarrow c(e_1', \ldots, e_n') : c(\tau_1, \ldots, \tau_n)}$$

[8] A similar approach was used by Jones in [52] for Core ML.

$$(cas)\quad \frac{\begin{array}{l} S \mid A \rhd e \leadsto e' : \tau \\ S \mid A, A_{pi} \rhd p_i, e_i \leadsto p'_i : \tau'_i, e'_i : \tau_i \end{array} \qquad S \vdash c : \tau \leq \cup_{i=1}^{n} \tau'_i}{S \mid A \rhd \begin{array}{c} \textbf{case } e \textbf{ of } p_i \Rightarrow e_i \\ \leadsto \\ \textbf{case } ce' \textbf{ of } p'_i \Rightarrow e'_i \end{array} : \cup_{i=1}^{n} \tau_i ? \tau \cap \overline{\tau'_i}}$$

$$(ext)\quad \frac{\begin{array}{l} S \mid A \rhd f \leadsto f' : \tau'' \\ S \mid A, A_p \rhd p,\ e \leadsto p' : \tau',\ e' : \tau \end{array} \qquad \begin{array}{l} S \vdash c_1 : \tau'' \leq \tau_1 \rightarrow \tau_2 \\ S \vdash c_2 : \tau_3 \leq \tau' \cup \tau_1 ? \neg \overline{\tau}' \end{array}}{S \mid A \rhd \begin{array}{c} f \textbf{ extend } (p \Rightarrow e) \\ \leadsto \\ (c_1 f' \textbf{ extend } (p' \Rightarrow e')) \circ c_2 \end{array} : \tau_3 \rightarrow (\tau ? \tau_3 \cap \overline{\tau}' \cup \tau_2 ? \tau_3 \cap \neg \overline{\tau}')}$$

$$(asp)\quad \frac{S \mid A \rhd x, p \leadsto x : \tau_1,\ p' : \tau_2}{S \mid A \rhd x \textbf{ as } p \leadsto x \textbf{ as } p' : \tau_1 \cap \tau_2}$$

$$(gen)\quad \frac{S \mid A \rhd e \leadsto e' : \tau}{\emptyset \mid A \rhd e \leadsto \lambda \alpha_i . e' : \forall \alpha_i . \tau \textbf{ where } S}\ \{\alpha_i \notin FT(A)$$

$$(ins)\quad \frac{S \mid A \rhd e \leadsto e' : \forall \overline{\alpha} . \tau \textbf{ where } S'}{S, S'[\tau_i / \alpha_i] \mid A \rhd e \leadsto e' \tau_i : \tau[\tau_i / \alpha_i]}$$

$$(let)\quad \frac{S \mid A \rhd e_1 \leadsto e'_1 : \sigma \quad S' \mid A, x : \sigma \rhd e_2 \leadsto e'_2 : \tau}{S, S' \mid A \rhd \textbf{let } x = e_1 \textbf{ in } e_2 \leadsto \textbf{let } x = e'_1 \textbf{ in } e'_2 : \tau}$$

A starting point for the typed semantics can be the bounded polymorphic language with intersection types $F_\wedge$ as given by Pierce in [78]. However, we would have to extend it to also include union types and to generalize the constraints $\alpha \leq \tau$ to sets of proper constraints $\{\sigma_i \leq \tau_i \mid 1 \leq i \leq n\}$. Moreover we would have to prove the *coherence* of the translation i.e. we would have to prove that any translation e_1 and e_2 of a term e given by derivations $S \mid A \rhd e \leadsto e_1 : \tau$ and $S \mid A \rhd e \leadsto e_2 : \tau$ are semantically equivalent. Coherence results for systems with subtyping were given in [15, 28] and for systems with intersection types were given in [83].

We do not continue the development of this semantics and let it rather as a further research.

3.3.3 Untyped Semantics

Curry style systems are often called *type assignment systems.* The interpretation of a term e is some element m of an untyped model U, given by a semantical function $[\![-]\!]$ which is defined by induction on the structure of terms. Typing is a matter of *predication:* a typing statement involving a term e is an assertion about $[\![e]\!]$.

According to this point of view, the interpretation of a type τ is a predicate i.e. a set of elements for which the assertion expressed by τ is true. When σ and τ are regarded as sets, the assertion that $\sigma \leq \tau$ simply means $[\![\sigma]\!] \subseteq [\![\tau]\!]$. Similarly, $[\![\sigma \cap \tau]\!] = [\![\sigma]\!] \cap [\![\tau]\!]$ and $[\![\sigma \cup \tau]\!] = [\![\sigma]\!] \cup [\![\tau]\!]$.

A type–checker in this context can be thought as proving theorems about programs – theorems that show on the basis of a set of typing rules that are known to be sound descriptions of the semantics of terms, that the interpretations of terms behave in a certain way. A *type inference procedure* is a deterministic procedure for discovering a *principal theorem* – a theorem of which all other theorems about the behavior of the program are corollaries.

The untyped semantics presented below is known as the ideal model [63, 57]. It is a veritable example of the use of different mathematical constructions on different levels of the semantics to solve in principle the same problem: find a solution of a recursive equation $x = f(x)$. On the first level f is a function. Hence, this level gives a semantics to recursive functions. On the second level f is a domain constructor. As we see later, this provides a semantics for self application. Finally, on the third level f is a type constructor. Consequently, this level gives a meaning to recursive types.

Level 1 : Recursive Functions

The mathematical machinery used to provide a semantics for recursive functions is based on *complete partial orders* and *continuous functions.* Being standard stuff in domain theory we do not give their definitions here but include them in Appendix C.1. The basic result about continuous functions over cpo's is the *fixed point theorem.*

Theorem 3.3 Fixed points in ω–cpo's

> Suppose $\mathcal{U}$ is an ω–cpo and $f : \mathcal{U} \to \mathcal{U}$ is ω–continuous. If $x \sqsubseteq f(x)$ for some $x \in U$, then there is a least element y such that $y = f(y)$ and for all z, $x \sqsubseteq z \wedge z = f(z) \Rightarrow y \sqsubseteq z$. This element is the limit $\bigsqcup_{n \in \omega} f^n(x)$ of the chain $x \sqsubseteq f^1(x) \sqsubseteq f^2(x) \ldots$. □

If the cpo U has a least element $\perp$ then $\perp \sqsubseteq f(\perp)$ and $y = \bigsqcup_{n \in \omega} f^n(\perp)$.

Level 2: Self Application

In the untyped λ–calculus, any pure λ–expression can be used either as *a function* or as *an argument*. For example, in the expression $x(x)$, x is both a function i.e. $x \in U \to U$ and a value i.e. $x \in U$. This implies that U is equal with $U \to U$ i.e. U is the solution of the recursive cpo equation $U = U \to U$[9].

In this case, $\to$ is a cpo constructor and U is a cpo. The basic cpo constructors are function space $(. \to .)$, product $(. \times .)$, sum $(. \oplus .)$ and lifting $((.)_\perp)$. Their definitions are given in Appendix C.2.

In order to solve a cpo equation we would like to build a chain $U_1 \subseteq U_2 \subseteq \ldots$ and use the fixed point theorem to construct the fixed point. Although at first sight very appealing, taking $\subseteq$ for $\sqsubseteq$ is not satisfactory since the arrow constructor is, as we already pointed out, *not monotonic* in the first argument. Hence, it cannot be continuous.

The fix of this problem is to generalize the inclusions $U_i \subseteq U_j$ to continuous embeddings $\phi : U_i \to U_j$ (with corresponding projections $\psi : U_j \to U_i$) and the chains $(U_n)_{n\in\omega}$ to chains $(U_n, \phi_n)_{n\in\omega}$. In other words, posets are generalized to categories with objects cpo's and arrows embedding/projection pairs. Accordingly, the sort constructors become functors, since they do not only map cpo's but also the arrows between them. The challenge is now to provide for function space, product, sum and lifting, functors which preserve the direction of arrows (i.e. they "are monotonic") and preserve colimits (i.e. they "are continuous"). These functors are also given in Appendix C.2.

Definition 3.24 Embedding/projection pairs

A *continuous* function $\phi : U \to V$ is an *embedding* if there is a continuous function $\psi : V \to U$ such that:

$$\psi \circ \phi = id_U, \quad \phi \circ \psi \sqsubseteq id_V$$

The map ψ is called a *projection* and being uniquely determined by ϕ it is also written ϕ^R. □

The injections in_1 and in_2 are embeddings with projections out_1 and out_2. The identity is both an injection and a projection. Embeddings are preserved by composition and $(\phi_1 \circ \phi_0)^R = \phi_0^R \circ \phi_1^R$.

Definition 3.25 Basic embedding/projection pairs

If $\phi_0 : U_0 \to V_0$ and $\phi_1 : U_1 \to V_1$ are embeddings then we have the following

[9] In practice equality is weakened to *isomorphism* which is written as $U \cong U \to U$.

embedding/projection pairs between the constructed cpo's:

$$\begin{array}{llll}
\phi_0^R \to \phi_1 : & (U_0 \to U_1) \to (V_0 \to V_1), & \phi_0 \to \phi_1^R : & (V_0 \to V_1) \to (U_0 \to U_1) \\
\phi_0 \times \phi_1 : & (U_0 \times U_1) \to (V_0 \times V_1), & \phi_0^R \times \phi_1^R : & (V_0 \times V_1) \to (U_0 \times U_1) \\
\phi_0 \oplus \phi_1 : & (U_0 \oplus U_1) \to (V_0 \oplus V_1), & \phi_0^R \oplus \phi_1^R : & (V_0 \oplus V_1) \to (U_0 \oplus U_1) \\
(\phi_0)_\perp : & (U_0)_\perp \to (V_0)_\perp & (\phi_0^R)_\perp : & (V_0)_\perp \to (U_0)_\perp
\end{array}$$

□

Before writing the recursive cpo equation for our language let us make some additional observations. First, we want to distinguish between $\perp$ and $\lambda x.\ \perp$. Semantically, this can be expressed by using a lifted space $(U \to U)_\perp$ of continuous functions. Second, we want to distinguish between $\perp$ and $\mathrm{c}(\perp, \ldots, \perp)$ and between constructors with different names. Semantically this can be achieved by interpreting constructors with injective functions which map lifted n–ary tuples. We write U^n for an n–ary product and $|c|$ for the arity of the constructor c. The n–ary products and sums are trivial generalizations of their binary counterparts. Third, we introduce the value $*$ for wrong typed elements. Now, the cpo isomorphism for $\lambda_{\leq}^{\forall,\to,c,\cup,\cap,?}$ is:

$$U \cong \oplus_{c \in C}(U^{|c|})_\perp \oplus (U \to U)_\perp \oplus \{*\}_\perp$$

The cpo U is the result of a limiting process.

$$\begin{tikzcd}
 & U \arrow[r, "\theta"] & \oplus_{c \in C}(U^{|c|})_\perp \oplus (U \to U)_\perp \oplus \{*\}_\perp & & \\
U_0 \arrow[r, "\phi_0"'] \arrow[ur, "\mu_0"] & U_1 \arrow[r, "\phi_1"'] \arrow[u, "\mu_1"] \arrow[ur, "\nu_0"'] & U_2 \arrow[r, "\phi_2"'] \arrow[ul, "\mu_2"] \arrow[u, "\nu_1"'] & U_3 \arrow[r, "\phi_3"'] \arrow[ull, "\mu_3"] \arrow[ul, "\nu_2"'] & \cdots
\end{tikzcd}$$

Figure 3.4: The colimit construction

Definition 3.26 Colimit

Define cpo's U_n and the embeddings $\phi_n : U_n \to U_{n+1}$ as follows:

$$U_0 = \emptyset_\perp, \qquad U_{n+1} = \oplus_{c \in C}(U_n^{|c|})_\perp \oplus (U_n \to U_n)_\perp \oplus \{*\}_\perp$$

$$\phi_0 = \lambda x \in U_0.\ \perp, \quad \phi_{n+1} = \oplus_{c \in C}(\phi_n^{|c|})_\perp \oplus (\phi_n^R \to \phi_n)_\perp \oplus id_{\{*\}_\perp}$$

U is the *colimit* of the chain $<U_n, \phi_n>$ in the category of embeddings i.e. there is a cone[10] $\mu : <U_n, \phi_n> \to U$ such that $\mu \circ \mu^R$ is an increasing chain with $\sqcup_{n \geq 0}\ \mu_n \circ \mu_n^R = id_U$. The desired isomorphism θ is constructed via a cone

$$\nu_n : <U_{n+1}, \phi_{n+1}> \to \oplus_{c \in C}(U^{|c|})_\perp \oplus (U \to U)_\perp \oplus \{*\}_\perp$$

of embeddings where

$$\nu_n = \oplus_{c \in C}(\mu_n^{|c|})_\perp \oplus (\mu_n^R \to \mu_n)_\perp \oplus id_{\{*\}_\perp}$$

by putting $\theta = \sqcup\ \nu_n \circ \mu_{n+1}^R$ and its inverse $\theta^{-1} = \sqcup\ \mu_{n+1} \circ \nu_n^R$. The cones μ and ν are related by the equations

$$\nu_n = \theta \circ \mu_{n+1}, \qquad \mu_{n+1} = \theta^{-1} \circ \nu_n$$

□

As we already anticipated, terms are interpreted in U.

Definition 3.27 The semantics of terms

The semantics of terms is given by an interpretation function $\mathcal{E} : Exp \to Env \to U$ which is defined by induction on the term structure. $Env = Var \to U$ is the set of environments ranged by ρ. To save some tedious numbering for the injections into the sum, let us write $d : D$ for the injection of an element d into the D component of the sum.

$$\mathcal{E}[\![x]\!]\rho = \rho(x)$$

$$\mathcal{E}[\![\lambda.e]\!]\rho = \mathrm{up}(f) : (U \to U)_\perp$$

$$f(d) = \begin{cases} * & \text{if} \quad d = * \\ \mathcal{E}[\![e]\!]\rho[d/x] & \text{otherwise} \end{cases}$$

$$\mathcal{E}[\![e_1 e_2]\!]\rho = \begin{cases} \mathrm{down}(f)(\mathcal{E}[\![e_2]\!]\rho) & \text{if} \quad \mathcal{E}[\![e_1]\!]\rho = f : (U \to U)_\perp \\ * & \text{otherwise} \end{cases}$$

$$\mathcal{E}[\![c(e_1, \ldots, e_n)]\!]\rho = \begin{cases} \mathrm{up}(\mathcal{E}[\![e_1]\!]\rho, \ldots, \mathcal{E}[\![e_n]\!]\rho) : (U^{|c|})_\perp & \text{if} \quad \mathcal{E}[\![e_i]\!]\rho \neq * \\ * & \text{otherwise} \end{cases}$$

[10] To say that $\mu : <U_n, \phi_n> \to U$ is a cone means that for all n, $\mu_n : U_n \to U$ and that $\mu_n = \mu_{n+1} \circ \phi_n$.

$$\mathcal{E}[\![\textbf{case}\ e\ \textbf{of}\ \vec{p} \Rightarrow \vec{e}]\!]\rho = \begin{cases} \mathcal{E}[\![e_i]\!]\rho[\vec{a}/\vec{x}] & \text{if} \quad \mathcal{E}[\![e]\!]\rho = \mathcal{E}[\![p_i]\!]\rho[\vec{a}/\vec{x}] \wedge \vec{x} = FV(p_i) \\ \bot & \text{if} \quad \mathcal{E}[\![e]\!]\rho = \bot \\ * & \text{otherwise} \end{cases}$$

$$\mathcal{E}[\![x\ \textbf{as}\ p]\!]\rho = \mathcal{E}[\![p]\!]\rho$$

$$\mathcal{E}[\![f\ \textbf{extend}\ (p \Rightarrow e)]\!]\rho = \text{up}(g) : (U \to U)_\bot$$

$$g(d) = \begin{cases} \mathcal{E}[\![e]\!]\rho[\vec{a}/\vec{x}] & \text{if} \quad d = \mathcal{E}[\![p]\!]\rho[\vec{a}/\vec{x}] \wedge \vec{x} = FV(p) \\ \text{down}(h)(d) & \text{if} \quad \mathcal{E}[\![f]\!]\rho = h : (U \to U)_\bot \\ * & \text{otherwise} \end{cases}$$

$$\mathcal{E}[\![\textbf{let}\ x = e_1\ \textbf{in}\ e_2]\!]\rho = \begin{cases} * & \text{if} \quad \mathcal{E}[\![e_1]\!]\rho = * \\ \mathcal{E}[\![e_2]\!]\rho[\mathcal{E}[\![e_1]\!]\rho/x] & \text{otherwise} \end{cases}$$

□

Remark 3.12 Data constructors

The uniqueness of $\vec{a} = (a_1, \ldots, a_n)$ in the interpretation of **case** and **extend** is assured by the the injectivity of constructors which implies the injectivity of patterns. Patterns are injective because they contain only constructors and variables and the composition of injective functions is an injective function. □

Level 3: Types

Considering types as predicates over the untyped universe U i.e. as subsets of U which contain elements with similar structure (e.g. they are all functions or pairs) leads to the very intuitive interpretation of subtyping as inclusion and of union and intersection as set theoretical union and intersection. However, as we already pointed out, recursive type equations cannot be solved by taking inclusion as ordering and applying the fixed point theorem because of the anti–monotony of → in its first argument.

The key idea in this case is to use the cpo's U_n occuring in the colimit construction to define a *metric* that measures the *distance* (or difference) between types. Now, if the type constructors decrease this distance i.e. they are *contractive,* then by the Banach fixed point theorem they have a unique fixed point. The mathematical machinery for explaining recursion is in this case topological.

In order to formalize notions like "distance" and "same structure" it is convenient that each element $x \in U$ is the limit of a chain of *finite* elements which *approximate* x. The appropriate cpo's having this property are known as *bounded complete domains* (see Appendix C.3). New finite elements are created in the limiting process when proceeding from U_i to U_{i+1}.

The intuition about types is formalized by the following definition. It says that the structure of the elements should be preserved when we go down to approximations or when we go up to limits. Moreover, since $\emptyset \rhd (\lambda x.xx)(\lambda x.xx) : \forall\alpha.\alpha$ and the meaning of $(\lambda x.xx)(\lambda x.xx)$ is $\perp$ it follows that $\perp$ has every type. As a consequence, no type can be empty.

Definition 3.28 Ideals

A subset I of a partial order P is an *order ideal* iff

1. $I \neq \emptyset$
2. $\forall y \in I.\forall x \in P.x \sqsubseteq y \Rightarrow x \in I$

A subset I of a bc–domain U is an *ideal* if it is an order ideal and it additionally satisfies:

3. $\forall (x_n)_{n\in\omega}.(\forall n.x_n \in I) \Rightarrow (\sqcup_{n\in\omega} x_n \in I)$

The collection of all order ideals is written as $\mathcal{P}_o(P)$ and the collection of ideals as $\mathcal{P}(U)$. □

Ideals are determined by their finite elements i.e. there is an evident isomorphism between $(\mathcal{P}(U), \subseteq)$ and $(\mathcal{P}_o(U^o), \subseteq)$.

Definition 3.29 Rank, closeness, distance

The *rank* $\mathrm{r}(e)$ of a finite element $e \in U^\circ$ is the least $n \geq 0$ with $e \in \mu_n(U_n^\circ)$ where $r : U^\circ \to \mathbb{N}$. A *witness* for two ideals I and J is any element in their symmetric difference $(I - J) \cup (J - I)$. The *closeness* $\mathrm{c}(I, J)$ is the least possible rank of a witness for I and J and ∞ if none exists. The *distance* $\mathrm{d}(I, J) = 2^{-\mathrm{c}(I,J)}$ □

Proposition 3.4 Rank

1. $\mathrm{r}(u) = 0$ iff $u = \perp_U$
2. Any finite element u of $(U^{|c|})_\perp$ other than $\perp$ is equal to $\mathrm{up}\,(a_1, \ldots, a_{|c|})$ with a_i finite and $\mathrm{r}(a_i) < \mathrm{r}(u)$ for all i.
3. Any finite element u of $(U \to U)_\perp$ other than $\perp$ is equal to $\mathrm{up}\,(a_1 \Rightarrow b_1) \sqcup \ldots \sqcup (a_n \Rightarrow b_n)$ with a_i and b_i finite and $\mathrm{r}(a_i), \mathrm{r}(b_i) < \mathrm{r}(u)$ for all i.

$$\begin{tikzcd}
U \arrow[r, "\theta"] & \oplus_{c\in C}(U^{|c|})_\perp \oplus (U \to U)_\perp \oplus \{*\}_\perp \\
U_{n+1} = \oplus_{c\in C}(U_n^{|c|})_\perp \oplus (U_n \to U_n)_\perp \oplus \{*\}_\perp \arrow[u, "\mu_{n+1}"] \arrow[ur, "\nu_n = \oplus_{c\in C}(\mu_n^{|c|})_\perp \oplus (\mu_n^R \to \mu_n)_\perp \oplus \mathrm{id}_\perp"']
\end{tikzcd}$$

Proof: The proof is similar to the proof given in [57] but here we have to take care of lifting.

1. Immediate from the definition of r.

2. To say that $u \in (U^{|c|})_\perp$ means that $\theta(u) = \mathrm{up}\,(a_1, \ldots, a_{|c|}) \in (U^{|c|})_\perp$ (ignoring the injection into the sum). Since u is finite, so are a_i for all i. As $u \neq\perp$ we can assume that $\mathrm{r}(u) = n+1$ for some $n \geq 0$, and hence $u = \mu_{n+1}(d)$ for some $d \in U^o_{n+1}$. Then:

$\mathrm{up}\,(a_1, \ldots, a_{|c|}) = \theta(u) = \theta(\mu_{n+1}(d)) = \nu_n(d) = (\mu_n \times \ldots \times \mu_n)_\perp(d)$

and so $d = \mathrm{up}\,(a'_1, \ldots, a'_{|c|})$ for some $a'_i \in U^o_n$ with $\mathrm{up}\,(a_1, \ldots, a_{|c|}) = (\mu_n \times \ldots \times \mu_n)_\perp\ (\mathrm{up}\,(a'_1, \ldots, a'_{|c|})) = \mathrm{up}\,(\mu_n(a'_1), \ldots, \mu_n(a'_{|c|}))$. Therefore $a_i = \mu_n(a'_i)$ and $\mathrm{r}(a_i) \leq n$ for all i.

3. To say that $u \in (U \to U)_\perp$, means that $\theta(u) = \mathrm{up}(f) \in (U \to U)_\perp$ (ignoring the injection into the sum). Since u is finite so is f. Consequently it is represented as the limit of step functions, $f = (a_1 \Rightarrow b_1) \sqcup \ldots \sqcup (a_n \Rightarrow b_n)$ with a_i and b_i finite. As $u \neq\perp$ we can assume that $\mathrm{r}(u) = n+1$ for some $n \geq 0$, and hence $u = \mu_{n+1}(g)$ for some $g \in U^o_{n+1}$. Then:

$\mathrm{up}\,(f) = \theta(u) = \theta(\mu_{n+1}(g)) = \nu_n(g) = (\mu_n^R \to \mu_n)_\perp(g)$

and so $g = \mathrm{up}\,(a'_1 \Rightarrow b'_1) \sqcup \ldots \sqcup (a'_n \Rightarrow b'_n)$ with

$\mathrm{up}\,(a_1 \Rightarrow b_1) \sqcup \ldots \sqcup (a_n \Rightarrow b_n) =$

$(\mu_n^R \to \mu_n)_\perp\ (\mathrm{up}\,(a'_1 \Rightarrow b'_1) \sqcup \ldots \sqcup (a'_n \Rightarrow b'_n)) =$

$\mathrm{up}\,(\mu_n(a'_1) \Rightarrow \mu_n(b'_1)) \sqcup \ldots \sqcup (\mu_n(a'_n) \Rightarrow \mu_n(b'_n))$

Therefore $a_i = \mu_n(a'_i), b_i = \mu_n(b'_i)$ and $\mathrm{r}(a_i), \mathrm{r}(b_i) \leq n < \mathrm{r}(u)$. □

The analog of chains are in the topological setting the Cauchy sequences and the analog of cpo's are the complete metric spaces.

Definition 3.30 Cauchy sequence, complete metric space

A sequence of ideals $(I_n)_{n\geq 0}$ is called a *Cauchy sequence* if

$$\forall\epsilon > 0.\exists n.\forall i, j.(i > n \wedge j > n) \Rightarrow \mathrm{d}(I_i, I_j) < \epsilon$$

A metric space is *complete* if every Cauchy sequence converges. □

Theorem 3.5 Complete metric space

The metric space $<\mathcal{P}(U), d>$ is complete. The limit of $(I_n)_{n\geq 0}$ is I where $I^o = \{b \in U^o \mid b \text{ is in almost all } I_n\}$. □

For complete metric spaces the analog of continuous functions are the contractive ones.

Definition 3.31 Contractive, non–expansive

A *uniformly contractive* map $f : \vec{X} \to Y$ of metric spaces is one such that there is a real number $0 \leq r < 1$ such that

$\forall \vec{x}, \vec{y} \in \vec{X}.\mathrm{d}(f(\vec{x}), f(\vec{y})) \leq r \max\{\mathrm{d}(x_i, y_i) \mid 1 \leq i \leq n\}$

and is *non–expansive* if it holds with $r \leq 1$. □

The analog of the fixed point theorem is the Banach fixed point theorem.

Theorem 3.6 Banach

If X is a nonempty complete metric space and $f : X \to X$ is contractive then it has a unique fixed point, namely $lim_{n\geq 0} f^n(x_0)$ where x_0 is any point in X. □

Definition 3.32 Type constructors

The type constructors $\overset{|c|}{\times}_\perp$, $\to_\perp$ and ? and are defined as follows:

$$\overset{|c|}{\times}_\perp (I_1, \ldots, I_{|c|}) = (\overset{|c|}{\times} (I_1, \ldots, I_{|c|}))_\perp$$

$$I \to_\perp J = \{f \in U \to U \mid f(I) \subseteq J\}_\perp$$

$$I \ ? \ J = \begin{cases} I & \text{if} \quad J \neq \{\perp\} \\ \{\perp\} & \text{otherwise} \end{cases}$$

□

Theorem 3.7 Contractive type constructors

The type constructors $\overset{|c|}{\times}_\perp$ and $\to_\perp$ are contractive on ideals.

Proof: The proof is similar with the one given for × and → in [57]. Here we have to take care of lifting. First note that

$\mathrm{d}(f(\vec{x}), f(\vec{y})) \leq r \max\{\mathrm{d}(x_i, y_i) \mid 1 \leq i \leq n\}$ iff $\mathrm{c}(f(\vec{x}), f(\vec{y})) \geq \min\{\mathrm{d}(x_i, y_i) \mid 1 \leq i \leq n\}$

if some $x_i \neq y_i$. If $r < 1$ then the strict inequality $>$ must hold.

$\overset{|c|}{\times}_\perp$: Suppose u is a witness of minimum rank for $\overset{|c|}{\times}_\perp (I_1, \ldots, I_{|c|})$ and $\overset{|c|}{\times}_\perp (I'_1, \ldots, I'_{|c|})$ with say $u \in \overset{|c|}{\times}_\perp (I_1, \ldots, I_{|c|})$. Since $u \neq \perp$, by the proposition about ranks, $u = \mathrm{up}(a_1, \ldots, a_{|c|})$ where a_i are finite elements of I_i for all i and $\mathrm{r}(u) > \max\{\mathrm{r}(a_i) \mid 1 \leq i \leq |c|\}$. Since $u \notin \overset{|c|}{\times}_\perp (I'_1, \ldots, I'_{|c|})$ there must be an i such that $a_i \notin I'_i$. But this is a witness for I_i and I'_i of rank less then $\mathrm{r}(u)$ and

$$\mathrm{c}(\overset{|c|}{\times}_\perp(I_1, \ldots, I_{|c|}), \overset{|c|}{\times}_\perp(I'_1, \ldots, I'_{|c|})) = \mathrm{r}(u) > \mathrm{r}(a_i) \geq \mathrm{c}(I_i, I'_i) \geq \min\{\mathrm{c}(I_i, I'_i) \mid 1 \leq i \leq |c|\}$$

$\rightarrow_\perp$: Let u be a witness of minimum rank for $I \rightarrow_\perp J$ and $I' \rightarrow_\perp J'$, being say in the former ideal. Then $u \neq \perp$ and by the proposition about ranks, $u = \mathrm{up}\,(a_1 \Rightarrow b_1) \sqcup \ldots \sqcup (a_n \Rightarrow b_n)$ where $n > 0$ and a_i, b_i are finite elements of U with $\mathrm{r}(u) > \max\{\mathrm{r}(a_i), \mathrm{r}(b_i) \mid 1 \leq i \leq n\}$. Since $u \notin I' \rightarrow_\perp J'$ there must be an $x \in I'$ such that $\mathrm{down}(u)(x) \notin J'$. Let $a = \sqcup\{a_i \mid a_i \sqsubseteq x\}$ and $b = \sqcup\{b_i \mid a_i \sqsubseteq x\} = \mathrm{down}(u)(x)$. Then $a \in I'$ as $a \sqsubseteq x \in I'$ and $b \notin J'$. The rank $\mathrm{r}(a) = \mathrm{r}(\sqcup\{a_i \mid a_i \sqsubseteq x\}) < \mathrm{r}(u)$ and similarly $\mathrm{r}(b) < \mathrm{r}(u)$. Now there are two cases:

$a \notin I$: Then a is a witness for I and I' of rank less then $\mathrm{r}(u)$.

$a \in I$: Then $b = \mathrm{down}(u)(a) \in J$ because $u \in I \rightarrow_\perp J$ and thus b is a witness for J and J' of rank less than $\mathrm{r}(u)$.

In both cases we have:

$\mathrm{c}(I \rightarrow_\perp J, I' \rightarrow_\perp J') = \mathrm{r}(u) > \min\{\mathrm{c}(I, I'), \mathrm{c}(J, J')\}$ □

Theorem 3.8 Nonexpansive type constructors

The type constructors $\cup$, $\cap$ and ? are not contractive but non–expansive, considered as binary functions over ideals.

Proof: The proof for $\cup$ and $\cap$ is given in [57]. The proof for ? is given along the same lines. Noncontractiveness is shown by $\mathrm{d}(I?1, I'?1) = \mathrm{d}(I, I')$. The proof that ? is nonexpansive is done by a case analysis.

$J = 0, J' = 0:$ $\mathrm{c}(I?J, I'?J') = \mathrm{c}(0, 0) \geq \min\{\mathrm{c}(I, I'), \mathrm{c}(0, 0)\}$

$J = 0, J' \neq 0:$ $\mathrm{c}(I?J, I'?J') = \max\{\mathrm{c}(0, I'), \mathrm{c}(0, J')\} \geq \min\{\mathrm{c}(I, I'), \mathrm{c}(0, J')\}$

$J \neq 0, J' = 0:$ similarly to the previous case

$J \neq 0, J' \neq 0:$ $\mathrm{c}(I?J, I'?J') = \mathrm{c}(I, I') \geq \min\{\mathrm{c}(I, I'), \mathrm{c}(J, J')\}$

□

Definition 3.33 Semantics of type expressions

For the semantics of type expressions we define the semantic function $\mathcal{T} : Texp \rightarrow Tenv \rightarrow \mathcal{P}(U)$ where $Tenv = Tvar \rightarrow \mathcal{P}(U)$ is the set of environments ranged over by ν. The definition is by structural induction on the type expression structure.

$$
\begin{array}{lcl}
\mathcal{T}[\![\alpha]\!]\nu & = & \nu(\alpha) \\
\mathcal{T}[\![\tau_1 \rightarrow \tau_2]\!]\nu & = & \mathcal{T}[\![\tau_1]\!]\nu \rightarrow_\perp \mathcal{T}[\![\tau_2]\!]\nu \\
\mathcal{T}[\![c(\tau_1, \ldots, \tau_n)]\!]\nu & = & \overset{|c|}{\times}_\perp(\mathcal{T}[\![\tau_1]\!]\nu, \ldots, \mathcal{T}[\![\tau_n]\!]\nu) \\
\mathcal{T}[\![\tau_1 \cup \tau_2]\!]\nu & = & \mathcal{T}[\![\tau_1]\!]\nu \cup \mathcal{T}[\![\tau_2]\!]\nu \\
\mathcal{T}[\![\tau_1 \cap \tau_2]\!]\nu & = & \mathcal{T}[\![\tau_1]\!]\nu \cap \mathcal{T}[\![\tau_2]\!]\nu \\
\mathcal{T}[\![\tau_1 ? \tau_2]\!]\nu & = & \mathcal{T}[\![\tau_1]\!]\nu \ ? \ \mathcal{T}[\![\tau_2]\!]\nu \\
\mathcal{T}[\![0]\!]\nu & = & \{\perp\} \\
\mathcal{T}[\![1]\!]\nu & = & U - \{*\} \\
\mathcal{T}[\![\forall \alpha_i.\tau \textbf{ where } S]\!]\nu & = & \bigcap_{\substack{I_i \in \mathcal{P}(U) \\ \nu[I_i/\alpha_i] \models S}} \mathcal{T}[\![\tau]\!]\nu[I_i/\alpha_i]
\end{array}
$$

□

We write $\nu \models S$ iff for all constrains $l \leq r$ in S, $\mathcal{T}[\![l]\!]\nu \subseteq \mathcal{T}[\![r]\!]\nu$. In order to complete the semantics of type expressions we have to characterize S and to show how to find solutions ν such that $\nu \models S$.

Let us first define the syntactic class of recursive equations which are guaranteed to have unique fixed points in the semantics.

Definition 3.34 Contractive predicate

Let τ be a type expression and β a type variable. The predicate $\tau \succ \beta$ read as τ is *contractive* in β, is defined inductively on the structure of τ as follows:

$$
\begin{array}{ll}
0 \succ \beta & 1 \succ \beta \\
\overline{T} \succ \beta & \neg\overline{T} \succ \beta \\
c(\tau_1, \ldots, \tau_n) \succ \beta & \tau_1 \rightarrow \tau_2 \succ \beta \\
\alpha \succ \beta \Leftrightarrow \alpha \neq \beta & \tau_1 ? \tau_2 \succ \beta \Leftrightarrow \tau_1 \succ \beta \wedge \tau_2 \succ \beta \\
\tau_1 \cup \tau_2 \succ \beta \Leftrightarrow \tau_1 \succ \beta \wedge \tau_2 \succ \beta & \tau_1 \cap \tau_2 \succ \beta \Leftrightarrow \tau_1 \succ \beta \wedge \tau_2 \succ \beta
\end{array}
$$

□

Remark 3.13 Hat types

> Similarly to 0 and 1, the hat types and their negations are also contractive in β because they do not contain any type variables. □

The set of variables $\mathrm{TLV}(\tau) = \{\alpha \mid \tau \not\succ \alpha\}$ are called in [5] *top level variables*.

Definition 3.35 Contractive equations

> An equation $\alpha = E$ is *contractive* if $E \succ \alpha$. □

Definition 3.36 Cascading equations

> A set of equations $\{\alpha_1 = E_1, \ldots, \alpha_n = E_n\}$ is *cascading* if $\mathrm{TLV}(E_i) \subseteq \{\alpha_1, \ldots, \alpha_{i-1}\}$. □

Cascading equations allow the elimination of the recursive top level variables α_i by substituting E_i for α_i in E_{i+1} through E_n. Now, given an environment ν' for the other variables the above results guarantee the existence of a unique extension $\nu = \nu'[I_i/\alpha_i]$ of ν' such that $\mathcal{T}[\![\alpha_i]\!]\nu = \mathcal{T}[\![E_i]\!]\nu$. Hence, ν is a solution for E i.e. $\nu \models E$. A particular class of recursive inclusion constraints, the so called *inductive constraints* can be shown equivalent with a set of cascading equations.

Definition 3.37 Inductive constraints

> A constraint $\tau \leq \alpha_i$ or $\alpha_i \leq \tau$ is inductive iff $\mathrm{TLV}(\tau) \subseteq \{\alpha_1, \ldots, \alpha_{i-1}\}$. □

Definition 3.38 Inductive sets of constraints

> Let us denote by t_j the smallest type containing $U_j - \{*\}$. Then, a system S of constraints is inductive if the following three conditions hold:

1. $S = \{l_i \leq \alpha_i \leq u_i \mid 1 \leq i \leq n\}$
2. $\mathrm{TLV}(l_i) \cup \mathrm{TLV}(u_i) \subseteq \{\alpha_1, \ldots, \alpha_{i-1}\}$
3. $\forall 1 \leq i_0 \leq n . \forall j . \forall \nu .$

$$\left.\begin{array}{l}\forall 1 \leq i < i_0 . \\ \quad \mathcal{T}[\![l_i]\!]\nu \cap t_j \subseteq \nu(\alpha_i) \cap t_j \subseteq \mathcal{T}[\![u_i]\!]\nu \cap t_j \;\wedge \\ \forall i_0 \leq i \leq n . \\ \quad \mathcal{T}[\![l_i]\!]\nu \cap t_{j-1} \subseteq \nu(\alpha_i) \cap t_{j-1} \subseteq \mathcal{T}[\![u_i]\!]\nu \cap t_{j-1}\end{array}\right\} \Rightarrow \mathcal{T}[\![l_{i_0}]\!]\nu \cap t_j \subseteq \mathcal{T}[\![u_{i_0}]\!]\nu \cap t_j$$

□

The above definition makes it possible to build solutions ν with a double induction over (i, j). The indices (i, j) in $\nu_{i,j}$ say that at induction step j we already constructed the interpretation $\{\alpha_1 \to I_{1,j}, \ldots, \alpha_i \to I_{i,j}\}$ where $I_{k,j} \subseteq U_j$ are sets of finite elements in U_j. For the rest of the variables we have the interpretation constructed in the previous induction step i.e. $\{\alpha_{i+1} \to I_{i+1,j-1}, \ldots, \alpha_n \to I_{n,j-1}\}$ where $I_{k,j-1} \subseteq U_{j-1}$. Now, condition 2. allows at a given induction step j to assign sets $I_{i,j}$ successively to $\alpha_1, \ldots, \alpha_n$ because each α_i is constrained only by lower level variables at the top level. Condition 3. assures the existence of a $I_{i+1,j}$ satisfying the constraint $\mathcal{T}[\![l_{i+1}]\!]\nu_{i,j} \subseteq I_{i+1,j} \subseteq \mathcal{T}[\![u_{i+1}]\!]\nu_{i,j}$. The induction starts by taking $\nu_{n,0} = \{\alpha_1 \to \{\bot\}, \ldots, \alpha_n \to \{\bot\}\}$.

Theorem 3.9

Let $S = \{l_i \leq \alpha_i \leq u_i\}$ be an inductive set of constraints. Then S is equivalent to the cascading set of equations $S' = \{\alpha_i = l_i \cup (\beta_i \cap u_i)\}$ where β_i are fresh variables.

Proof: The proof is given in [5]. The first implication (from inequations to equations) is immediate by letting $\beta_i = \alpha_i$. The other implication uses the properties of inductive constraints. □

Since α_i and β_i are the only variables occuring in S' each environment $\nu' = [I_i/\beta_i]$ for β_i induces a unique solution ν such that $\nu \models S'$ which also implies that $\nu \models S$.

This completes the semantics because the remaining type expressions $1, \overline{\tau}$ and $\neg\overline{\tau}$ can be expressed in terms of the other type expressions.

Now we are ready to state the main results linking the syntax with the semantics.

Definition 3.39

$S \models_\nu \tau_1 \leq \tau_2$ means that $\mathcal{T}[\![\tau_1]\!]\nu \subseteq \mathcal{T}[\![\tau_2]\!]\nu \quad$ if $\nu \models S$,

$S \models \tau_1 \leq \tau_2$ means that $S \models_\nu \tau_1 \leq \tau_2$ for all $\nu \in TEnv$. □

Theorem 3.10 Subtyping soundness

If $S \vdash \tau_1 \leq \tau_2$ then $S \models \tau_1 \leq \tau_2$.

Proof: By induction on the height of the proof that $\tau_1 \leq \tau_2$.

$\leq\!\to$:

Given a type environment ν we have to prove for the rule $(\leq\!\to)$ that $S \models_\nu \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'$ under the assumptions that $S \models \tau_1' \leq \tau_1$ and $S \models \tau_2 \leq \tau_2'$. If

$$f \in \mathcal{T}[\![\tau_1 \to \tau_2]\!]\nu = \{g \in U \to U \mid g(\mathcal{T}[\![\tau_1]\!]\nu) \subseteq \mathcal{T}[\![\tau_2]\!]\nu\}_\bot$$

then:

$\mathrm{down}(f)(\mathcal{T}[\![\tau_1']\!]\nu) \subseteq \mathrm{down}(f)(\mathcal{T}[\![\tau_1]\!]\nu) \subseteq \mathcal{T}[\![\tau_2]\!]\nu \subseteq \mathcal{T}[\![\tau_2']\!]\nu$

so $f \in \mathcal{T}[\![\tau_1 \rightarrow \tau_2]\!]\nu$.

The rules $(\leq tra)$ and $(\leq ref)$ are trivially satisfied by interpreting types as sets.

$\leq \boldsymbol{c}$:
Given a type environment ν we have to prove for the rule $(\leq cc)$ that $S \models_\nu c(\tau_1, \ldots, \tau_n) \leq c(\tau_1', \ldots, \tau_n')$ under the assumptions that $S \models \tau_i \leq \tau_i'$ for $1 \leq i \leq |c|$. If

$a = \mathrm{up}\,(a_1, \ldots, a_{|c|}) \in \mathcal{T}[\![c(\tau_1, \ldots, \tau_n)]\!]\nu = \overset{|c|}{\times}_{\perp}(\mathcal{T}[\![\tau_1]\!]\nu, \ldots, \mathcal{T}[\![\tau_{|c|}]\!]\nu)$

then $a_i \in \mathcal{T}[\![\tau_i]\!]\nu \subseteq \mathcal{T}[\![\tau_i']\!]\nu$ so

$a \in \overset{|c|}{\times}_{\perp}(\mathcal{T}[\![\tau_1']\!]\nu, \ldots, \mathcal{T}[\![\tau_{|c|}']\!]\nu) = \mathcal{T}[\![c(\tau_1', \ldots, \tau_n')]\!]\nu.$

The inequalities $(\leq\rightarrow c)$, $(\leq c \rightarrow)$ and $(\leq cd)$ are true because the associated types are injected in U with different functions: $\_ : (U^{|c|})_\perp, \_ : (U^{|d|})_\perp$ and $\_ : (U \rightarrow U)_\perp$ respectively.

$\leq \ \cup \cap$:
The rules $(\leq \cap r)$, $(\leq \cap lb)$, $(\leq \cup l)$ and $(\leq \cup gb)$ are trivially satisfied by interpreting types as sets.

$\leq \mathbf{0}$:
The axiom $(\leq 0l)$ is trivial since each type contains $\perp$. The proofs of the axioms $(\leq 0c)$ and $(\leq 0 \rightarrow)$ use the fact that each lifted type contains at least two elements, $\perp$ and $\mathrm{up}\,(\perp)$, and it is therefore different from $\mathcal{T}[\![0]\!]\nu = \{\perp\}$.

$\leq$ **?**:
For $(\leq ?1)$ note that $t_1 \,?\, t_2 \subseteq t_1$ by the definition of ?. Hence if $\mathcal{T}[\![\tau_1]\!]\nu$ is included in $\mathcal{T}[\![\tau]\!]\nu$ so does $\mathcal{T}[\![\tau_1 \,?\, \tau_2]\!]\nu$. For $(\leq ?2)$, if $\mathcal{T}[\![\tau_2]\!]\nu \subseteq \{\perp\}$ then $\mathcal{T}[\![\tau_2]\!]\nu = \{\perp\}$ and therefore by the definition of ? we have that $\mathcal{T}[\![\tau_1 \,?\, \tau_2]\!]\nu = \{\perp\}$ which is included in every type. □

Definition 3.40

$\models_{\rho,\nu} (S|A)$ means that $\nu \models S$ and $\mathcal{E}[\![x]\!]\rho \in \mathcal{T}[\![\sigma]\!]\nu$ whenever $(x : \sigma) \in A$,

$S \mid A \models_{\rho,\nu} e : \tau$ means that $\mathcal{E}[\![e]\!]\rho \in \mathcal{T}[\![\tau]\!]\nu \quad$ if $\models_{\rho,\nu} (S|A)$,

$S \mid A \models e : \tau$ means that $S \mid A \models_{\rho,\nu} e : \tau$ for all $\rho \in Env$ and $\nu \in TEnv$. □

Lemma 3.5 Typing soundness

If $S \mid A \rhd e : \tau$ then $S \mid A \models e : \tau$.

Proof: The soundness of the rules $(\rightarrow i)$, $(\rightarrow e)$ and (let) is proven in [63]. Let us prove the soundness of (ext), (cas) and (con).

***(ext)* :**

For given ρ and ν such that $\models_{\rho,\nu} (S|A)$ we have to prove that

$S \mid A \models_{\rho,\nu} g$ **extend** $(p \Rightarrow e) : \tau_3 \rightarrow (\tau ? \tau_3 \cap \overline{\tau'} \ \cup \ \tau_2 ? \tau_3 \cap \neg\overline{\tau'})$

under the assumptions

$$\begin{array}{ll} S \mid A, A_p \models p : \tau', e : \tau & S \models \tau'' \leq \tau_1 \rightarrow \tau_2 \\ S \mid A \models g : \tau'' & S \models \tau_3 \leq \tau' \ \cup \ (\tau_1 \cap \neg\overline{\tau'}) \end{array}$$

Suppose $d \in \mathcal{T}\llbracket \tau_3 \rrbracket \nu$. Then either $d \in \mathcal{T}\llbracket \tau' \rrbracket \nu$ or $d \in \mathcal{T}\llbracket \tau_1 \rrbracket \nu \cap \mathcal{T}\llbracket \neg\overline{\tau'} \rrbracket \nu$.

$d \in \mathcal{T}\llbracket \tau' \rrbracket \nu$: Then by the injectivity of constructors there is a unique $\vec{a}$ such that $d = \mathcal{E}\llbracket p \rrbracket \rho[\vec{a}/\vec{x}]$. As a consequence $S \mid A, A_p \models_{\rho[\vec{a}/\vec{x}],\nu} p : \tau', e : \tau$ and by the semantical definition

$$\begin{array}{rl} f(d) = \mathcal{E}\llbracket e \rrbracket \rho[\vec{a}/\vec{x}] \in & \mathcal{T}\llbracket \tau \rrbracket \nu = \mathcal{T}\llbracket \tau \rrbracket \nu ? \mathcal{T}\llbracket \tau_3 \rrbracket \nu \cap \mathcal{T}\llbracket \tau' \rrbracket \nu \subseteq \\ & \mathcal{T}\llbracket \tau \rrbracket \nu ? \mathcal{T}\llbracket \tau_3 \rrbracket \nu \cap \mathcal{T}\llbracket \overline{\tau'} \rrbracket \nu \ \cup \ \mathcal{T}\llbracket \tau_2 \rrbracket \nu ? \mathcal{T}\llbracket \tau_3 \rrbracket \nu \cap \mathcal{T}\llbracket \neg\overline{\tau'} \rrbracket \nu \end{array}$$

$d \in \mathcal{T}\llbracket \tau_1 \rrbracket \nu \cap \mathcal{T}\llbracket \neg\overline{\tau'} \rrbracket \nu$: Then $d \in \mathcal{T}\llbracket \tau_1 \rrbracket \nu$ and $d \in \mathcal{T}\llbracket \neg\overline{\tau'} \rrbracket \nu$. Hence for $h = \mathcal{E}\llbracket g \rrbracket \rho$ we have that

$$\begin{array}{rl} f(d) = \mathrm{down}(h)(d) \in & \mathcal{T}\llbracket \tau_2 \rrbracket \nu = \mathcal{T}\llbracket \tau_2 \rrbracket \nu ? \mathcal{T}\llbracket \tau_3 \rrbracket \nu \cap \mathcal{T}\llbracket \neg\overline{\tau'} \rrbracket \nu \subseteq \\ & \mathcal{T}\llbracket \tau \rrbracket \nu ? \mathcal{T}\llbracket \tau_3 \rrbracket \nu \cap \mathcal{T}\llbracket \overline{\tau'} \rrbracket \nu \ \cup \ \mathcal{T}\llbracket \tau_2 \rrbracket \nu ? \mathcal{T}\llbracket \tau_3 \rrbracket \nu \cap \mathcal{T}\llbracket \neg\overline{\tau'} \rrbracket \nu \end{array}$$

Hence

$\forall d \in \mathcal{T}\llbracket \tau_3 \rrbracket \nu.\ f(d) \in \mathcal{T}\llbracket \tau \rrbracket \nu ? \mathcal{T}\llbracket \tau_3 \rrbracket \nu \cap \mathcal{T}\llbracket \overline{\tau'} \rrbracket \nu \ \cup \ \mathcal{T}\llbracket \tau_2 \rrbracket \nu ? \mathcal{T}\llbracket \tau_3 \rrbracket \nu \cap \mathcal{T}\llbracket \neg\overline{\tau'} \rrbracket \nu$

As a consequence

$\mathrm{up}(f) \in \mathcal{T}\llbracket \tau_3 \rightarrow (\tau ? \tau_3 \cap \overline{\tau'} \ \cup \ \tau_2 ? \tau_3 \cap \neg\overline{\tau'}) \rrbracket \nu$

(cas)

The proof for (cas) is similar with the proof for (ext).

(con)

For given ρ and ν such that $\models_{\rho,\nu} (S|A)$ we have to prove that

$S \mid A \models_{\rho,\nu} c(e_1, \ldots, e_n) : c(\tau_1, \ldots, \tau_n)$

under the assumptions

$S \mid A \models e_i : \tau_i \quad$ for $1 \leq i \leq |c|$.

From the semantical definition of constructors and from these assumptions we can easily derive the desired result, that

$$\mathrm{up}\,(\mathcal{E}\llbracket e_1 \rrbracket \rho, \ldots, \mathcal{E}\llbracket e_n \rrbracket \rho) : (U^{|c|})_\perp \in \overset{|c|}{\times}_\perp (\mathcal{T}\llbracket \tau_1 \rrbracket \nu, \ldots, \mathcal{T}\llbracket \tau_{|c|} \rrbracket \nu) : (U^{|c|})_\perp$$

that is that $\mathcal{E}[\![c(e_1, \ldots, e_n)]\!]\rho \in \mathcal{T}[\![c(\tau_1, \ldots, \tau_n)]\!]\nu$. □

Corollary 3.11

If e is a closed expression and $\triangleright\ e : \sigma$ then $\mathcal{E}[\![e]\!] \neq *$. □

Chapter 4

The Functional Object Model

4.1 Introduction

The OO methodology is very appealing because it uses concepts which seem to be very intuitive and closely related to the "real world". However, these concepts are often (deliberately) stated very informally and a precise formulation leads to very different models. A typical example is message passing. According to Coad and Yourdon [32], pp 149:

- a message connection is a mapping of one object to another object (or occasionally to a class) in which a *sender* sends a message to a *receiver* to get some processing done.

In this definition it is left open what happens after the message is sent. Does this message interrupt the receiver or does the sender wait for the answer?

A precise response to these questions not only separates the sequential and the parallel worlds but also distinguishes between different parallel models. Despite of the terminology *message passing,* most existing object-oriented languages are sequential in nature. This can be explained by the fact that they observe the following restrictions:

1. Execution starts with exactly one object being active,
2. Whenever an object sends a message, it waits until the result is returned,
3. An object is only active when it is executing a method in response to an incoming message.

Under these conditions at any moment there is exactly one active object, although control is transferred very often from one object to another. The most sensible ways to introduce parallelism to OO languages is to relax restrictions 2 or 3.

Relaxing restriction 2, an object is not forced anymore to wait for the result after sending a message, but is allowed instead to immediately proceed in doing its own activities. This is called *asynchronous communication.* In this way, the sender can execute in parallel with the receiver of the message. This scheme has been adopted most notably by the family of Actor languages [3, 4].

Relaxing restriction 3, objects have their own activity (also called body) regardless if they receive a message or not. Execution of the body is started as soon as the object is created and it takes place in parallel with other objects in the system. Communication is most naturally achieved in this case by explicitly indicating *rendezvous points* where the body can synchronize with another object in order to exchange a message. This is called *synchronous* communication and was used most notably in the POOL family [9] and in the OO languages based on π-Calculus like $\pi o\beta\lambda$ [50, 51] and Abacus [75, 74].

The approach presented in this section is *purely functional* and *asynchronous.* It is therefore most closely related to Actor languages. However, in contrast to Actor languages, we use a typed formalism. Moreover, the set of messages that are sent but not yet received i.e. the message histories are *explicitly* modeled by infinite lists of messages, also known as *streams* [17]. The processes are particular continuous functions (case functions) operating in a sequential manner on streams. The main advantage of this approach is that it effectively unifies the concepts of *object* and *process* into one concept, that of a functional entity which is *self contained* and provided with a *unified communication protocol.* Processes interact only at clearly defined points: only where messages are sent or answered. Moreover, the possible ways of interaction are limited: only parameters or results may be sent. The variables in each object are protected from direct access by other objects. Shared data can be put in an object on its own and accessed by the available object methods. Another important aspect is that inside an object everything happens sequentially. This sequential, deterministic inside is protected from the parallel, nondeterministic outside world by the message interface. We consider that nondeterminism inside an object would make the models too difficult to comprehend and to verify.

The major simplification done in this asynchronous, functional model is to disallow explicit manipulation of object identities. Objects themselves or sometimes the streams they deliver and not their identifiers are sent as messages. However, by sending streams instead of identifiers we achieve most of the power of languages with explicit identity manipulation in a much more simpler setting. Moreover, it may be argued that encapsulation, subtyping and inheritance are *the* essential features of object orientation while *identity manipulation* only distinguishes between functional and procedural languages where the last ones usually provide constructs for manipulating pointers (read identifiers).

This chapter is organized as follows. In Section 4.2 we look for an appropriate formalism which extends in a natural way the sequential world to the parallel one.

This allows us to clarify the concepts of class, object and message passing. Section 4.3 is devoted to inheritance and Section 4.4 to parametric classes. For each OO concept, we review the definition given in Chapter 2 and instantiate it for our model.

4.2 Classes, Objects and Messages

The purpose of this section is to give a precise meaning for objects, classes and message passing. For reference, let us review their informal definitions as given in Sections 2.1 and 2.2.

Definition Class

> A class is a parameterized object definition. Different instantiations of the parameters permit the creation of different objects. In other words, a class is the definition of an object creation function. □

Definition Object

> An object is a clearly delimited software entity which has:
>
> - a *state* i.e. it contains some private data,
> - a *behavior* i.e. it can execute certain procedures,
> - a *unique identity*.
>
> □

Definition Communication and messages

> Objects can interact by exchanging messages according to a precisely determined message interface. A message consists of a method name and actual parameters to be passed the the method. The receiver alone determines when and which method to execute in response to a message. The method can return a result which is passed back to the sender. □

4.2.1 The Sequential World

The sequential model of Cartesian point objects presented in Section 3.1 identifies objects with functions. Their *state* is given by the λ– and the **let**–bound variables and the scoping rules for bound variables assures their privacy. For the representation of object *methods* (i.e. object behavior) and *message passing* we provided two alternatives. The first one is the *record* variant.

$$
\begin{array}{l}
\mathcal{P} = \lambda myclass.\lambda(xc, yc).\lambda self\\
\qquad \begin{array}{lll}
\{\ \mathsf{x} & = & xc,\\
\ \ \mathsf{y} & = & yc,\\
\ \ \mathsf{mv} & = & \lambda(dx, dy).\ \mathsf{fix}\ myclass(self.\mathsf{x} + dx, self.\mathsf{y} + dy),\\
\ \ \mathsf{eq} & = & \lambda p.\,(self.\mathsf{x} == p.\mathsf{x}) \wedge (self.\mathsf{y} == p.\mathsf{y})\ \}
\end{array}
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{cart\_point} = \mathsf{fix}\ \mathcal{P} & --\,\textit{Class definition}\\
o = \mathsf{fix}\ \mathsf{cart\_point}\,(a,b) & --\,\textit{Object creation}\\
(o.\mathsf{mv})(dx, dx) & --\,\textit{Message passing}
\end{array}
$$

In this case *methods* are functional fields in a record and *messages* are not directly delivered to the objects. The message name is first used to select the appropriate field (i.e. method) which is then applied to the message arguments. In this functional view, objects are not distinguished from their identities. The objects themselves and not their identifiers are sent among objects. *Classes* are represented as parameterized object definitions (or object creation functions).

The second alternative is the *case* variant.

$$
\begin{array}{l}
\mathcal{P} = \lambda myclass.\lambda(xc, yc).\lambda self.\lambda m.\\
\qquad \mathbf{case}\ m\ \mathbf{of}\\
\qquad\qquad \begin{array}{lll}
\mathsf{x} & \Rightarrow & xc\\
\mathsf{y} & \Rightarrow & yc\\
\mathsf{mv}\,(dx, dy) & \Rightarrow & \mathsf{fix}\ myclass(self(\mathsf{x}) + dx,\ \ self(\mathsf{y}) + dy)\\
\mathsf{eq}\,(p) & \Rightarrow & (self(\mathsf{x}) == p(\mathsf{x})) \wedge (self(\mathsf{y}) == p(\mathsf{y}))
\end{array}
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{cart\_point} = \mathsf{fix}\ \mathcal{P} & --\,\textit{Class definition}\\
o = \mathsf{fix}\ \mathsf{cart\_point}\,(a,b) & --\,\textit{Object creation}\\
o\ \mathsf{mv}(dx, dy) & --\,\textit{Message passing}
\end{array}
$$

This variant differs from the previous one only in the *message passing* mechanism. Objects are applied in this case to the whole message and not only to its arguments. Using an OO terminology, the whole message is passed to the object.

In the next sections we analyze how can we extend these formalisms to obtain objects working on histories of messages.

4.2.2 Simplifying the Recursion Scheme

The double recursion scheme used in the above examples can be simplified by understanding $self$ as an abbreviation for a new object identical to the receiver object.

We already used a similar technique when we modeled updating of instance variables (as done by mv) with the creation of a new object, identical to the receiver one except for the updated variables. For example, we can rewrite the record variant of points as follows:

$$
\begin{aligned}
&\mathcal{P} = \lambda myclass.\lambda(xc, yc).\\
&\qquad \textbf{let } self = myclass(xc, yc) \textbf{ in}\\
&\qquad \{ \; \mathsf{x} \;\;\, = xc,\\
&\qquad \;\;\; \mathsf{y} \;\;\, = yc,\\
&\qquad \;\;\; \mathsf{mv} = \lambda(dx, dy).\ myclass(self.\mathsf{x} + dx, self.\mathsf{y} + dy),\\
&\qquad \;\;\; \mathsf{eq} \;\, = \lambda p.\ (self.\mathsf{x} == p.\mathsf{x}) \wedge (self.\mathsf{y} == p.\mathsf{y}) \; \}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{cart\_point} = \mathsf{fix}\ \mathcal{P} && --\textit{Class definition}\\
&o = \mathsf{cart\_point}\,(a,b) && --\textit{Object creation}
\end{aligned}
$$

Moreover, since $self.\mathsf{x} = xc$ and $self.\mathsf{y} = yc$ we can simplify again and write:

$$
\begin{aligned}
&\mathcal{P} = \lambda myclass.\lambda(xc, yc).\\
&\qquad \{ \; \mathsf{x} \;\;\, = xc,\\
&\qquad \;\;\; \mathsf{y} \;\;\, = yc,\\
&\qquad \;\;\; \mathsf{mv} = \lambda(dx, dy).\ myclass(xc + dx, yc + dy),\\
&\qquad \;\;\; \mathsf{eq} \;\, = \lambda p.\ (xc == p.\mathsf{x}) \wedge (yc == p.\mathsf{y}) \; \}
\end{aligned}
$$

A similar rewriting is also possible for the case–function variant. We therefore dispense from now on from the use of $self$ as a recursion variable.

4.2.3 An Appropriate Extension to Message Histories

As we already pointed out, the record variant cannot be reasonably extended on message histories because field selection would destroy the object. Moreover, enclosing the "instantaneous" record object with a function which decodes the messages and selects the appropriate record field, would also involve a case selection but also additionally a record field selection. Hence, the case variant seems to be more appropriate for the needed extension.

In this case there are two possible alternatives. The first one is the straightforward extension of the "sequential" case variant. It considers instead of a message m a *stream* s of messages. Note that the recursive type of streams with elements of type β is the solution of the recursive constraint $\alpha = \beta : \alpha$[1]. Such constraints are

[1] As in Haskell, ":" is the list constructor and ft takes the first element of a list. Lists are lazy i.e. their elements are evaluated only when they are needed. The list $\perp$ is the undefined list.

automatically inferred for recursive functions which use patterns only in the form $m : s$ as the one given below.

$$
\begin{array}{l}
\mathcal{Q} = \lambda myclass.\lambda(xc, yc).\lambda s \\
\qquad \textbf{case } s \textbf{ of} \\
\qquad\qquad \begin{array}{lcl}
\mathsf{x} : t & \Rightarrow & xc : myclass\,(xc, yc)\; t \\
\mathsf{y} : t & \Rightarrow & yc : myclass\,(xc, yc)\; t \\
\mathsf{mv}\,(dx, dy) : t & \Rightarrow & p : p\; t \textbf{ where } \{ \\
\end{array} \\
\qquad\qquad\qquad p = myclass\,(x + dx, y + dy)\} \\
\qquad\qquad \begin{array}{lcl}
\mathsf{eq}\,(p) : t & \Rightarrow & b : myclass\,(xc, yc)\; t \textbf{ where } \{ \\
\end{array} \\
\qquad\qquad\qquad b = (xc == \mathsf{ft}(p(\mathsf{x} : \bot)) \wedge yc == \mathsf{ft}(p(\mathsf{y} : \bot)))\ \}
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{scart\_point} = \mathsf{fix}\ \mathcal{Q} & --\textit{Class definition} \\
o = \mathsf{scart\_point}\,(a,b) & --\textit{Object creation} \\
o\ s & --\textit{Message passing}
\end{array}
$$

In this case, as in POOL, all messages sent to an object are stored in one queue in the order in which they arrive. A graphical illustration is given in Figure 4.1.

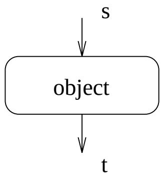


Figure 4.1: A point object with only one input channel

The second alternative is inspired by the record variant; instead of encoding an object as a record of stream processing functions it rather encodes it as a function taking as input a record of streams (the messages) and delivering as a result a record of output streams (the answers). Since the method names themselves are not included in the messages, we need a separate stream for each method. Each element of a stream contains the arguments of the instantaneous invocation of the method. In the case of nullary methods like x () and y () these elements contain just activation messages which we denote by (). A graphical illustration is given in Figure 4.2.

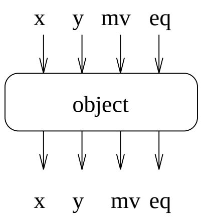


Figure 4.2: A point object with separate channels

This modeling of objects allows the parallel processing of the messages which do not alter the object state (the so called accessors).

$$\begin{array}{l}
\mathcal{Q} = \lambda myclass.\lambda(xc, yc).\lambda s \\
\quad \textbf{case } \mathsf{s} \textbf{ of} \\
\qquad \{\mathsf{x} = () : xs,\ \mathsf{y} = () : ys,\ \mathsf{mv} = mvs,\ \mathsf{eq} = p : eqs\} \Rightarrow \\
\qquad\quad \{\mathsf{x} = xc : r.\mathsf{x},\ \mathsf{y} = yc : r.\mathsf{y},\ \mathsf{mv} = r.\mathsf{mv},\ \mathsf{eq} = b : r.\mathsf{eq}\}\ \textbf{where}\ \{ \\
\qquad\quad r = myclass(xc, yc)\{\mathsf{x} = xs,\ \mathsf{y} = ys,\ \mathsf{mv} = mvs,\ \mathsf{eq} = eqs\}, \\
\qquad\quad b = (xc == \mathsf{ft}\ s.\mathsf{x}) \wedge (yc == \mathsf{ft}\ s.\mathsf{y}), \\
\qquad\quad s = p\ \{\ \mathsf{x} = ():\bot,\ \mathsf{y} = ():\bot,\ \mathsf{mv} = \bot,\ \mathsf{eq} = \bot\}\} \\
\\
\qquad \{\mathsf{x} = xs,\ \mathsf{y} = ys,\ \mathsf{mv} = (dx, dy) : mvs,\ \mathsf{eq} = eqs\} \Rightarrow \\
\qquad\quad \{\mathsf{x} = r.\mathsf{x},\ \mathsf{y} = r.\mathsf{y},\ \mathsf{mv} = p : r.\mathsf{mv},\ \mathsf{eq} = r.\mathsf{eq}\}\ \textbf{where}\ \{ \\
\qquad\quad p = myclass(xc + dx, yc + dy) \\
\qquad\quad r = p\ \{\mathsf{x} = xs,\ \mathsf{y} = ys,\ \mathsf{eq} = eqs,\ \mathsf{mv} = mvs\}\}
\end{array}$$

In contrast to the first variant, this object specification is nondeterministic, because the two record patterns occuring in the case expression are overlapping. Like e.g. in Ada, each entry (in this case corresponding to a method name) has its own queue and fairness between different queues is not necessarily guaranteed. In the first case however, nondeterminism and fairness concerns are pushed outside the objects by letting the environment to store all messages in one queue in the order in which they arrive. Because we consider nondeterminism inside an object more difficult to comprehend and verify, we prefer the first variant.

In the Cartesian points example each method was able to compute the result alone. However, in general a method needs further communication with other objects called *servers.* A graphical illustration is given in Figure 4.3.

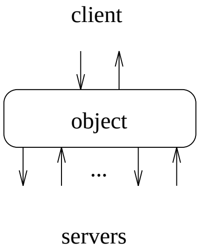


Figure 4.3: Object with many servers

In this case, the object has more than one input/output channel but it is case driven by the client input. In order to simplify things, we use this scheme also for more than one client as shown in Figure 4.4.

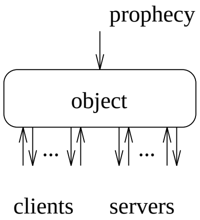


Figure 4.4: Object with many clients and servers

In this case the object is case–driven by the prophecy stream. Prophecy streams can be avoided by explicitly using nondeterministic *merge* components which collect the input from clients as shown in Figure 4.5.

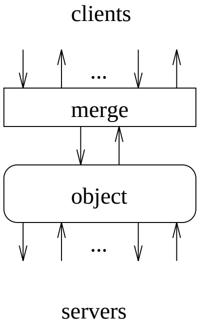


Figure 4.5: Object with nondeterministic merge

The behavior of merge components is usually given by predicates. The extension of our formalism to predicates and the usefulness of explicit merge components is considered as an important future work.

Now, we are ready to precisely state the meaning of classes, objects and message passing.

Definition 4.1 Objects

Objects are continuous functions having one input/output pair of streams on the client interface and possibly more output and input channels on the server interface.

- The attributes are the λ– and **let**–bound variables,
- The body of the functions is a case expression with one branch for each message type,
- Objects are their own identities.

□

Remark 4.1

Although channels usually occur in pairs, it is perfectly legal to have only one of them. For example, there could be objects which act as terminators (they eat the messages without delivering anything) or as initiators (they deliver spontaneously messages without any input). □

Definition 4.2 Communication and messages

Objects interact by exchanging messages asynchronously over message streams. A message is a data element consisting of a method name and actual parameters to be passed to the method. The receiver alone determines when and which method to execute in response to a message. The method can return a result to the sender. □

Definition 4.3 Classes

Classes are parameterized object definitions i.e. they are object creation functions. □

4.2.4 Ticks versus Objects

In the previous examples, the objects returned a copy of their modified versions in response to a `mv` message. In a more conservative approach one could consider `mv` as a "procedure" which updates the state of the object and does not return anything. In the semantical model, returning nothing actually means returning a special symbol $\surd$ also known as *tick*. As a consequence, streams are always defined and the merge anomaly is avoided. We decided to be more informative and return instead a copy of the object. This made it possible to define for example methods like `eq`.

4.3 Inheritance and Subtyping

Let us first review the informal definitions of inheritance and subtyping given in Section 2.4.

Definition Inheritance

Inheritance is a mechanism for incremental extension of recursive structures. □

Definition Subtyping

A type σ is a subtype of a type τ written as $\sigma \leq \tau$ if any expression of type σ is allowed in every context requiring an expression of type τ. □

Note that inheritance makes sense both in a typed or an untyped language, while subtyping makes sense only in a typed language i.e. in a language where every well formed term has an associated type. This type can be understood as a property or theorem about that term. For well formed objects this property says that they never deliver an answer of the form *message not understood* as do the objects in an untyped language like Smalltalk or Object Scheme. Beside this additional security, typing information inferred in the compilation process can be used to enhance program efficiency. Typing also supports data abstraction and modularity. We are therefore concerned with the inheritance mechanism in a typed language. The formal foundation for such a language was presented in Chapter 3.

4.3.1 Inheritance

Suppose we have stream points with only one coordinate xc and with only one method x. Their definition can be written as below.

$$
\begin{array}{l}
\mathcal{P} = \lambda pt.\lambda xc.\lambda s \\
\qquad \textbf{case } s \textbf{ of} \\
\qquad\qquad \mathsf{x} : t \quad \Rightarrow \; xc \; : \; pt\, xc\, t
\end{array}
$$

$$\mathsf{P} \stackrel{\text{def}}{=} \mathsf{fix}\ \mathcal{P}$$

The class P of stream point objects is the fixed point of $\mathcal{P}$. Now, we would like to define a class CP of colored points containing additionally a color attribute which is delivered in response to the new message c. This can be simply expressed in our framework as follows.

$$
\begin{array}{l}
\mathcal{CP} = \lambda cpt.\lambda xc.\lambda col. \\
\qquad \textbf{let } pt = \lambda x.cpt\,x\,col \textbf{ in} \\
\qquad \mathcal{P}\,pt\,xc \textbf{ extend} \\
\qquad\qquad \mathsf{c} : t \Rightarrow \; col : cpt\,xc\,col\;t \\
\mathsf{CP} \overset{\mathrm{def}}{=} \mathsf{fix}\,\mathcal{CP}
\end{array}
$$

Let us analyze this definition in more detail.

The State

The state of colored points contains an additional color attribute. This is simply modeled by adding a new λ–bound variable *col*.

Recursion

The recursion variable of $\mathcal{P}$ is unified with the recursion variable of $\mathcal{CP}$. Only after this unification is taken the fixed point to obtain the class CP. Note that in P the recursion variable is already bound. Hence, using P instead of $\mathcal{P}$ would not lead to the desired result since once receiving a message understood by P control would never be given back to CP. Now it is clear why we defined inheritance as a mechanism for incremental extension of recursive structures. Finally, the **let** clause is used to accommodate the difference in the attributes of $\mathcal{P}$ and $\mathcal{CP}$.

The Function Extension

If we expand $\mathcal{P}$ *pt xc* we obtain the following result.

$$
\begin{array}{l}
\mathcal{P}\;pt\;xc = \\
\qquad \textbf{case } s \textbf{ of} \\
\qquad\qquad \mathsf{x} : t \qquad\qquad \Rightarrow \; xc \;\; : \;\; cpt\,xc\,col\;t
\end{array}
$$

Hence $\mathcal{CP}$ is equivalent with the following definition.

$$
\begin{array}{l}
\mathcal{CP} = \lambda cpt.\lambda xc.\lambda col. \\
\qquad (\lambda s.\; \textbf{case } s \textbf{ of} \\
\qquad\qquad \mathsf{x} : t \qquad\qquad \Rightarrow \; xc \;\; : cpt\,xc\,col\;t \\
\qquad) \textbf{ extend}
\end{array}
$$

$$\mathsf{c} : t \qquad \Rightarrow \; col : cpt\,xc\,col\ t$$

The body of $\mathcal{CP}$ is in the form f **extend** $p \Rightarrow e$ where f is a stream processing function. It says that the function f is extended with the new branch $p \Rightarrow e$ which overrides any existing branch in f which matches an instance of p. Since no overriding takes place in $\mathcal{CP}$ the branch $\mathsf{c} : t \;\Rightarrow\; col : cpt\,xc\,col\,t$ simply extends the definition of $\mathcal{P}$. The obtained function has a new attribute and understands a new message. Moreover, this function is *guaranteed* to understand only the messages x and c. Any other message is caught at compile time and signaled as an error. As a consequence, an object created with CP never fails with the error "message not understood".

4.3.2 Subtyping

In the previous section we defined the class CP by inheriting (i.e. by extending) the definition of the class P. The correctness of the extension was assured by the calculus given in Chapter 3. A question which naturally arises, is if the objects generated by CP may also be used everywhere where objects generated by P may be, i.e. if the type of CP–objects is a subtype of the type of P–objects. This property is "proven" by the algorithm which infers the corresponding type of these objects. This is the key difference between inheritance and subtyping: the first one is *a construction* while the second one is *a property.*

Suppose we define a point $p = \mathsf{P}\,2$. The type of p is then as follows:

$$\begin{array}{l} \forall\gamma. \\ \gamma \;\rightarrow\; \mathsf{Int} : a \;?\; \gamma \cap \mathsf{x} : 1 \\ \textbf{where} \\ \qquad a = \mathsf{Int} : a \;?\; \gamma \cap \mathsf{x} : 1 \\ \qquad 0 \leq \gamma \leq \mathsf{x} : \gamma \end{array}$$

Since the class P is recursive, it is no wonder that this type is also recursive. It says that p accepts as input a stream of x messages and delivers as output a stream of integer messages. We write this type shortly as $\forall\gamma.\tau_1$ **where** S_1.

Before discussing the type of CP–objects, let us look to a somewhat "simpler" definition of colored points.

$$\begin{array}{l} \mathcal{Q} = \lambda pt.\lambda xc.\lambda col.\lambda s \\ \qquad \textbf{case } s \textbf{ of} \\ \qquad\qquad \mathsf{x} : t \qquad \Rightarrow \; xc \;:\; pt\,xc\,col\ t \end{array}$$

$$\mathsf{c} : t \qquad\qquad \Rightarrow \; col \; : \; pt\,xc\,col\; t$$

$\mathsf{Q} \overset{\text{def}}{=} \mathsf{fix}\ \mathcal{Q}$

A colored point q defined by $q = \mathsf{Q}\ 0\ \mathsf{red}$ has the following type:

$$\begin{array}{l}
\forall\gamma.\gamma \;\; \rightarrow (\\
\qquad \mathsf{Int} : b\ ?\ \gamma \cap \mathsf{x} : 1 \cup\\
\qquad \mathsf{red} : a\ ?\ \gamma \cap \mathsf{c} : 1)\\
\textbf{where}\\
\qquad 0 \leq \gamma \leq (\mathsf{x} \cup \mathsf{c}) : \gamma\\
\qquad a = \mathsf{Int} : b\ ?\ \gamma \cap \mathsf{x} : 1\ \cup\ \mathsf{red} : a\ ?\ \gamma \cap \mathsf{c} : 1\\
\qquad b = \mathsf{Int} : b\ ?\ \gamma \cap \mathsf{x} : 1\ \cup\ \mathsf{red} : a\ ?\ \gamma \cap \mathsf{c} : 1
\end{array}$$

This point accepts as input, streams $s \in \gamma$ such that $\gamma \leq (\mathsf{x} \cup \mathsf{c}) : \gamma$. In other words, it accepts any mixture of x and c messages including streams of purely x messages and streams of purely c messages. But if the input stream contains only x messages, i.e it has the type $\gamma' \leq \mathsf{x} : \gamma'$ then $\gamma' \cap \mathsf{c} : 1 = 0$ and the output stream has the type $b = \mathsf{Int} : b?\gamma' \cap \mathsf{x} : 1$. Hence, the type of q is a subtype of the type of p. More formally, if we write the type of q shortly as $\forall\gamma.\tau_2$ **where** S_2, then

$\forall\gamma.\tau_2$ **where** $S_2 \leq \forall\gamma.\tau_1$ **where** S_1

Although we did not provided an explicit rule for polymorphic types, this inequality can be proven generically by using the same instance variables for both types i.e. by proving that

$S_1, S_2 \mid \emptyset \vdash \tau_2 \leq \tau_1$

This is easily achieved by observing that $\gamma \cap \mathsf{c} : 1 = 0$ if $\gamma \leq \mathsf{x} : \gamma$ as required by S_1 and by using the subtyping rules given Section 3.2 (including those for free and bound variables).

The type inferred for an inherited object $r = \mathsf{CP}\ 0\ \mathsf{red}$ is more complicated as the type inferred for q because it additionally contains overriding information.

$$\begin{array}{l}
\forall\alpha,\gamma.\gamma \;\; \rightarrow (\\
\qquad \mathsf{Int} : b\ ?\ \alpha \cap \mathsf{x} : 1\ ?\ \gamma \cap \neg\mathsf{c} : 1 \cup\\
\qquad \mathsf{red} : a\ ?\ \gamma \cap \mathsf{c} : 1)\\
\textbf{where}\\
\qquad 0 \leq \alpha \leq \mathsf{x} : \gamma\\
\qquad 0 \leq \gamma \leq \mathsf{c} : \gamma\ \cup\ \alpha \cap \neg\mathsf{c} : 1\\
\\
\qquad a = \mathsf{red} : a?\gamma \cap \mathsf{c} : 1\ \cup\ \mathsf{Int} : b\ ?\ \alpha \cap \mathsf{x} : 1\ ?\ \gamma \cap \neg\mathsf{c} : 1\\
\qquad b = \mathsf{red} : a?\gamma \cap \mathsf{c} : 1\ \cup\ \mathsf{Int} : b\ ?\ \alpha \cap \mathsf{x} : 1\ ?\ \gamma \cap \neg\mathsf{c} : 1
\end{array}$$

However, by using the subtyping rules this type can also be proven as a subtype of $\forall\gamma.\tau_1$ **where** S_1. This means that r may be used in each context in which p is allowed.

In general an inherited type is not necessarily a subtype. For example if we have also defined the recursive methods corresponding to the messages mv and eq we would had obtained an inherited type for CP which was not a subtype of P (see [35]).

4.3.3 The Meaning of Super

The treatment of the pseudo–variable *super* which is used to invoke methods of the super class is modeled in our framework simply with an additional **let** variable. For example, suppose we have points with an equality test as below.

$$
\begin{array}{l}
\mathcal{Q} = \lambda pt.\lambda(xc,yc).\lambda s\\
\quad \textbf{case}\ s\ \textbf{of}\\
\qquad \begin{array}{lcl}
\mathsf{x} : t & \Rightarrow & xc : pt\,(xc,yc)\ t\\
\mathsf{y} : t & \Rightarrow & yc : pt\,(xc,yc)\ t\\
\mathsf{eq}\,(p) : t & \Rightarrow & b : pt\,(xc,yc)\ t\ \textbf{where}\ \{
\end{array}\\
\qquad\qquad b = (xc == \mathsf{ft}(p(\mathsf{x} : \bot)) \wedge yc == \mathsf{ft}(p(\mathsf{y} : \bot)))\ \}\\
\mathsf{Q} \overset{\text{def}}{=} \mathsf{fix}\ \mathcal{Q}
\end{array}
$$

Then we can define by inheritance colored points with equality test as follows.

$$
\begin{array}{l}
\mathcal{CQ} = \lambda cpt.\lambda(xc,yc,col).\\
\quad \textbf{let}\ pt = \lambda(x,y).cpt\,(x,y,col)\\
\qquad\ \ super = \mathcal{P}(pt)(xc,yc)\\
\quad \textbf{in}\ super\ \textbf{extend}\\
\qquad \begin{array}{lcl}
\mathsf{c} : t & \Rightarrow & col : cpt\,(xc,yc,col)\ t\\
\mathsf{eq}(p){:}t & \Rightarrow & (col == \mathsf{ft}(p(\mathsf{c} :\bot))) \wedge (\mathsf{ft}(super(\mathsf{eq}(p) :\bot)))
\end{array}\\
\mathsf{CQ} \overset{\text{def}}{=} \mathsf{fix}\ \mathcal{CQ}
\end{array}
$$

We used here the abbreviation:

$$
\begin{array}{l}
f\ \textbf{extend}\ p_1 \Rightarrow e_1, \ldots, p_n \Rightarrow e_n\\
\text{for}\\
((f\ \textbf{extend}\ p_1 \Rightarrow e_1)\ldots)\ \textbf{extend}\ p_n \Rightarrow e_n
\end{array}
$$

4.3.4 The Sequential World

A very important feature of our approach is that the sequential world can be expressed in the same formalism, without any additional construct. If $\mathcal{P}$ is the definition of points as given in Section 4.2 then we can define colored points as follows.

```
$\mathcal{CP} = \lambda cpt.\lambda(xc, yc, col).$
    **let** $pt = \lambda(x, y).cpt\,(x, y, col)$
        $super = \mathcal{P}(pt)(xc, yc)$
    **in** $super$ **extend**
        $\mathsf{c}$          $\Rightarrow$ $col$
        $\mathsf{eq}(p)$      $\Rightarrow$ $(col == p(\mathsf{c})) \wedge (super(\mathsf{eq}(p)))$
$\mathsf{CP} \overset{\text{def}}{=} \mathsf{fix}\ \mathcal{CP}$
```

4.4 Parametric Classes

The purpose of this section is to give a precise meaning for parametric classes. Let first review one of the definitions given in Section 2.5.

> [13] A generic class is a class that serves as a template for other classes, in which the template may be parameterized by other classes, objects and/or operations. A generic class must be instantiated (its parameters filled in) before objects can be created. Generic classes are typically used for container classes. The terms generic class and parameterized class are interchangeable.

Parameterization is expressed in our framework by parametric polymorphism. Since classes are functions, parameterized classes are simply polymorphic functions. Note that parametric polymorphism is not an exclusive feature of OO languages. In its most general form, as parameterized abstract data–type, parameterization was already used in languages like Ada, Obj or ML. In this case the parameters may be arbitrary abstract data types. If the parameters are restricted to be types, parameterization is known e.g. in ML as parametric polymorphism. To distinguish between these two forms of parameterization, parameterized ADTs are called in ML functors. In comparison with ML, our polymorphism is more powerful, because it can be combined with recursive subtyping constraints. This makes it even more powerful as the bounded polymorphism introduced by Cardelli and Wagner in [26] (see Section 3.1).

Remember that type annotations are in our framework optional and that the type inferred for terms is as general (i.e. as parametric) as possible. For example, the type inferred for the class Q defined in previous section,

$$
\begin{array}{l}
\mathcal{Q} = \lambda pt.\lambda xc.\lambda col.\lambda s \\
\qquad \textbf{case } s \textbf{ of} \\
\qquad\qquad \begin{array}{ll}
\mathsf{x} : t & \Rightarrow\ xc \ :\ pt\, xc\, col\ t \\
\mathsf{c} : t & \Rightarrow\ col \ :\ pt\, xc\, col\ t
\end{array}
\end{array}
$$

$$\mathsf{Q} \overset{\text{def}}{=} \mathsf{fix}\ \mathcal{Q}$$

is as follows:

$$
\begin{array}{l}
\forall\alpha,\beta,\gamma.\alpha \to \beta \to \gamma \ \ \to (\\
\qquad \alpha : b\ ?\ \gamma \cap \mathsf{x} : 1\ \cup \\
\qquad \beta : a\ ?\ \gamma \cap \mathsf{c} : 1) \\
\textbf{where} \\
\qquad 0 \leq \gamma \leq (\mathsf{x} \cup \mathsf{c}) : \gamma \\
\qquad a = \alpha : b\ ?\ \gamma \cap \mathsf{x} : 1\ \ \cup\ \ \beta : a\ ?\ \gamma \cap \mathsf{c} : 1 \\
\qquad b = \alpha : a\ ?\ \gamma \cap \mathsf{x} : 1\ \ \cup\ \ \beta : a\ ?\ \gamma \cap \mathsf{c} : 1
\end{array}
$$

The instance variables xc and col have the generic types α and β. These types are instantiated at object creation. For example an object where xc is an integer and col is a color can be created as $o = \mathsf{Q}(\mathsf{0})(\mathsf{red})$, and has the following type:

$$
\begin{array}{l}
\forall\gamma.\gamma \ \ \to (\\
\qquad \mathsf{Int} : b\ ?\ \gamma \cap \mathsf{x} : 1\ \cup \\
\qquad \mathsf{red} : a\ ?\ \gamma \cap \mathsf{c} : 1) \\
\textbf{where} \\
\qquad 0 \leq \gamma \leq (\mathsf{x} \cup \mathsf{c}) : \gamma \\
\qquad a = \mathsf{Int} : b\ ?\ \gamma \cap \mathsf{x} : 1\ \ \cup\ \ \mathsf{red} : a\ ?\ \gamma \cap \mathsf{c} : 1 \\
\qquad b = \mathsf{Int} : a\ ?\ \gamma \cap \mathsf{x} : 1\ \ \cup\ \ \mathsf{red} : a\ ?\ \gamma \cap \mathsf{c} : 1
\end{array}
$$

Note that there is no restriction for instantiation. We could have used even objects as actual parameters for Q as in $p = \mathsf{Q}(o)(o)$.

List classes are often cited as a typical example for parameterization. So let us give a list class definition in our framework before concluding this section.

$$
\begin{array}{l}
\mathcal{L} = \lambda lt.\lambda x.\lambda s \\
\qquad \textbf{case } s \textbf{ of} \\
\qquad\qquad \begin{array}{ll}
\mathsf{ft} : t & \Rightarrow \textbf{case } x \textbf{ of } \mathsf{cons}(a,b) \Rightarrow a\ \ :\ \ lt\, x\, t \\
\mathsf{rt} : t & \Rightarrow \textbf{case } x \textbf{ of } \mathsf{cons}(a,b) \Rightarrow lt\, b\, :\, lt\, b\, t \\
\mathsf{add}(a) : t & \Rightarrow lt\, \mathsf{cons}(a,x)\ \ :\ \ lt\, \mathsf{cons}(a,x)\, t
\end{array} \\
\mathsf{L} \overset{\text{def}}{=} \mathsf{fix}\ \mathcal{L}
\end{array}
$$

The bounded variable x stands for a finite list, with constructors cons and nil. When creating a new list, one would usually like to initialize it with a given element. In this case, the creation function has the form $\mathsf{L}\ \mathsf{cons}(a,\mathsf{nil})$. The type of a determines the type of the list elements. One can also imagine a creation function which delivers an empty list. In this case the creation function has the form $\mathsf{L}\ \mathsf{nil}$ and the type of the list elements is determined by the first add message.

Chapter 5

Object Configurations

5.1 Introduction

As we discussed in Section 2.6, objects are not very interesting in isolation. The major reason for using objects is to construct systems where they contribute to the overall behavior by interacting with one another. The static structure of such systems is usually described with *links* and *associations.*

Definition Link

> A link is a physical or conceptual connection between objects. □

Definition Association

> An association describes a group of links with common structure and common semantics. All the links in an association connect objects from the same classes. □

In Chapter 4 we modeled *objects* with *stream processing case–functions.* As a consequence, it is very natural to model *links* with *streams (or channels).*

Definition 5.1 Link

> A link is a channel (or stream) between two or more objects. □

Since streams are directed, our links are directed. Undirected links can be described as a pair of directed ones. Modeling links with channels give links an equal status to objects. This is very close to Rumbaugh recommendation: do not bury links as object attributes in the analysis phase. Object attributes (or references) are however a common practice in OO programming languages. Links versus object attributes are investigated in Section 5.2.4.

Methodologically, it is very important to classify object configurations in *aggregation systems* and *mobile systems.* They have quite different properties and consequently require quite different correctness proofs with respect to a requirement specification. Section 5.2 is devoted to aggregation networks. We give some typical examples and investigate their properties. Section 5.3 analyzes mobile networks. We review their properties and show how to express mobility with higher order streams. Finally, in Section 5.4 we give a calculus of mobile networks. This calculus does not view systems as functions but as collections of equations with designated input/output channels.

5.2 Aggregation Networks

Aggregation networks are hierarchical systems which can be treated as a whole. In Section 2.6.1 we defined them as below.

Definition Aggregates

> An *aggregate* is semantically an extended object that is treated as a unit in many operations, although physically it is made of several lesser objects. Aggregation is a special form of *transitive* and *antisymmetric* association where a group of component objects form a single semantic entity. Operations on an aggregate often propagate to the components. □

We also classified aggregates in fixed, variable and recursive:

- A *fixed aggregate* has a fixed structure. The *number* and *types* of subparts are predefined.
- A *variable aggregate* has a finite number of levels, but the *number* of parts may vary.
- A *recursive aggregate* contains directly or indirectly an instance of the same kind of aggregate; the number of potential levels is unlimited.

In our asynchronous, functional framework aggregates are very naturally modeled with Khan networks[1].

[1] Their name was given after the French scientist Gilles Kahn who first described them formally in [53].

Definition 5.2 Kahn Networks

A Kahn network has the following form:

$$\begin{aligned}
&f(\vec{x}) = \vec{z} \ \ \textbf{where}\\
&\quad \vec{y_1} = g_1(\vec{x}, \vec{y})\\
&\quad \ldots\\
&\quad \vec{y_n} = g_n(\vec{x}, \vec{y})
\end{aligned}$$

The variables $\vec{x} = (x_1, \ldots, x_n)$ represent the input channels and the variables $\vec{z} = (z_1, \ldots, z_m)$ represent the output channels of the whole network. Each network component is described by an equation $\vec{y_i} = g_i(\vec{x}, \vec{y})$. Similarly to the whole network, $\vec{y_i} \subseteq \vec{y}$[2] are the output channels and $\vec{x}$ and $\vec{y}$ the input channels of the i-th network component. Since these equations may be mutually recursive, the behavior of the whole network is given by their *least fixed point.* The network performs additionally a *hiding* operation by allowing only the channels $\vec{z} \subseteq \vec{y}$ to be visible as output outside the system. The channels $\vec{y} - \vec{z}$ are *local*. □

Let us consider some examples.

5.2.1 Fixed Aggregates

The Master/Slave Aggregate

A very important example of a fixed aggregate is the master/slave configuration.

$$\begin{aligned}
&\mathsf{ms}(\vec{x}) = \vec{z} \ \ \textbf{where}\\
&\quad (\vec{z}, \vec{y}) = \mathsf{m}(\vec{x}, \vec{u})\\
&\quad \vec{u} = \mathsf{s}(\vec{y})
\end{aligned}$$

Its class and object diagrams in OMT like notation are given in Figure 5.1.

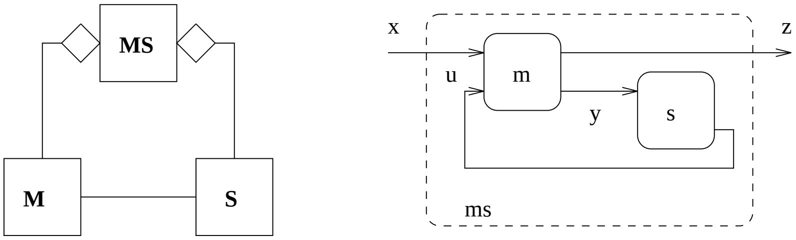


Figure 5.1: The class and object diagrams for master/slave

[2] We regard here $\vec{y}$ as a set.

The master/slave configuration consists of two components, m the master and s the slave. All the input of the slave comes via the master and all the output of the slave goes to the master. Viewing the master as the environment and the slave as the system, the master/slave configuration models a very general form of composition. Every network ms with a subnet s can be understood as a master/slave system where m denotes the surrounding net, the environment of s.

Note that fixed aggregates may contain loops i.e. they may contain recursive stream definitions. Both $\vec{u}$ and $\vec{y}$ are defined recursively. This stream recursion should not be confused with object recursion as present in recursive aggregates. Note also that the 1-1 association between the master and the slave in the class diagram is modeled as a pair of channels in the object diagram.

The fixed point of the equations contained by ms is taken simultaneously. Hence, we could alternatively define ms as follows:

$$\begin{array}{l}\mathsf{ms}(\vec{x}) = \vec{z} \;\; \textbf{where} \\ \quad ((\vec{z}, \vec{y}), \vec{u}) = (\mathsf{m}(\vec{x}, \vec{u}), \mathsf{s}(\vec{y}))\end{array}$$

This definition emphasizes more directly the *parallel composition* of the master and the slave. The corresponding object diagram, where only the physical position of m and s is modified is given in Figure 5.2.

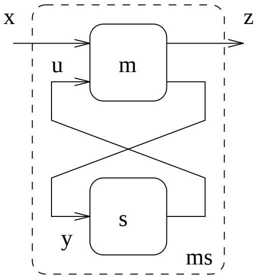


Figure 5.2: The master/slave configuration

The Summation Aggregate

A particular instance of the fixed master/slave configuration is the summation aggregate which is shown in Figure 5.3. It describes the stream processing function sum which takes as input a stream x of numbers and outputs each time the sum of the numbers already read. The summation aggregate description is as follows.

$$\begin{array}{l}\mathsf{sum}(x) = z \;\textbf{where} \\ \quad (z, y) = (\mathsf{add}(x, y), 0 : z)\end{array}$$

$$\mathsf{add}(a : as, b : bs) = (a + b) : \mathsf{add}(as, bs)$$

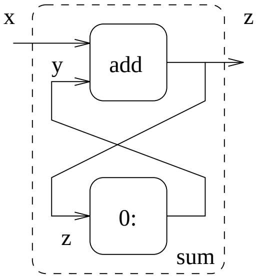


Figure 5.3: The summation aggregate

5.2.2 Recursive Aggregates

If the slave is taken to be the master/slave configuration itself, we obtain a recursive aggregate. A typical instance of such a configuration is the interactive queue shown in Figure 5.4.

$$\begin{array}{l} \mathsf{q}(\mathsf{add}(a):x) = z \textbf{ where} \\ \quad ((z,y),u) = (\mathsf{qc}\,a\,(x,u), \mathsf{q}(y)) \end{array}$$

$$\begin{array}{lll} \mathsf{qc}\,a\,(x,u) = \textbf{case } x \textbf{ of} & & \\ \quad \mathsf{add}(b):t & \Rightarrow (z, \mathsf{add}(b):y) \textbf{ where } (z,y) = \mathsf{qc}\,a\,(t,u) \\ \quad \mathsf{fst}:t & \Rightarrow (a:z,y) \textbf{ where } (z,y) = \mathsf{qc}\,a\,(t,u) \\ \quad \mathsf{rst}:t & \Rightarrow (u,t) \end{array}$$

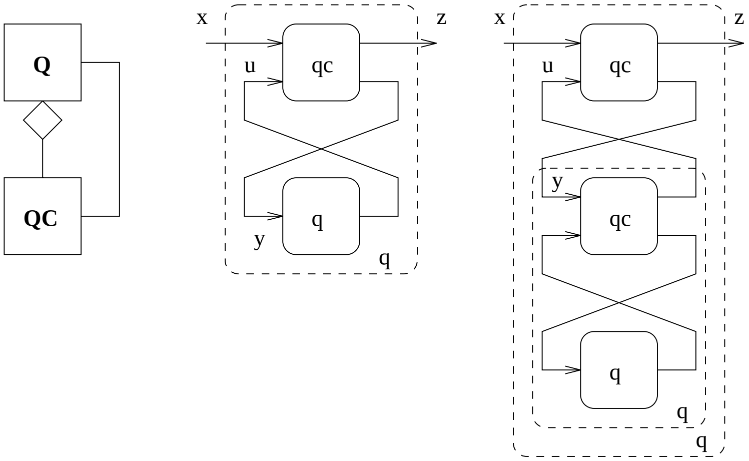


Figure 5.4: The interactive queue and one of its unfoldings

The interactive queue is capable to store in a queue cell qc only one element. Each time a new element arrives, the recursive definition is unfolded and a new cell qc is *created* to store that element. Unfolding is also shown in Figure 5.4. As a consequence, although the structure of the queue is predefined, the number of cells can vary dynamically.

5.2.3 The Static Structure of Kahn Networks

Let us analyse in more detail the master/slave configuration shown in Figure 5.5

ms*(x) = z* **where**
*((z,y), u) = (*m*(x,u),* s*(y))*

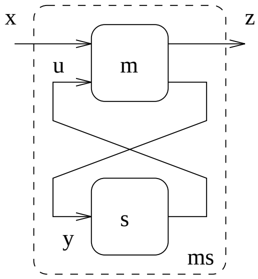


Figure 5.5: The master/slave configuration

Function composition, tupling and recursion can be interpreted for networks as *sequential composition, parallel composition* and *feedback* of network components. More precisely, denoting by $[1]$[3] the type of all streams and by $[1]^n$ the n-ary product $[1] \times \ldots \times [1]$ we can define [17]:

sequential composition

$$.;. : ([1]^n \to [1]^k) \to ([1]^k \to [1]^m) \to [1]^n \to [1]^m$$
$$(f;g)(x) = g(f(x))$$

parallel composition

$$.\|. : ([1]^k \to [1]^l) \to ([1]^m \to [1]^n) \to [1]^{k+m} \to [1]^{l+n}$$
$$(f\|g)(x,y) = (f(x), g(y))$$

feedback

$$\mu : ([1]^{n+m} \to [1]^m) \to [1]^n \to [1]^m$$
$$(\mu f)(x) = \mathsf{fix}\, \lambda y. f(x,y)$$

Using these forms of composition, the right hand side of the equation in ms represents the parallel composition of m and s. The channel variables z, y and u together with the feedback operator μ perform the necessary *wiring* between m and s. Variables actually allow to *multiplicate* a channel, to *exchange* the relative position of channels, or simply to *transmit* the values as they are. All these operations have a corresponding stream processing function [17]:

[3] Remember that 1 is the whole universe.

duplication

$\overset{n}{\gamma}\colon [1]^n \to [1]^{2n}$
$\overset{n}{\gamma}\; x = (x, x)$

exchange

$\overset{n\,m}{\chi}: [1]^{n+m} \to [1]^{m+n}$
$\overset{n\,m}{\chi}\; (x, y) = (y, x)$

identity

$I_n : [1]^n \to [1]^n$
$I_n(x) = x$

hiding
$\dagger^n : [1]^n \to [1]^0$

Hiding (†) is the unique stream processing function which consumes its input without producing any output.

The above forms of composition used in conjunction with the above stream processing functions allow us to describe the wiring of the master slave configuration without using variables. The corresponding configuration is given in Figure 5.6, where we use a slightly more powerfull exchange operator. This is actually $\overset{1\,1}{\chi} \parallel \overset{1\,1}{\chi}$.

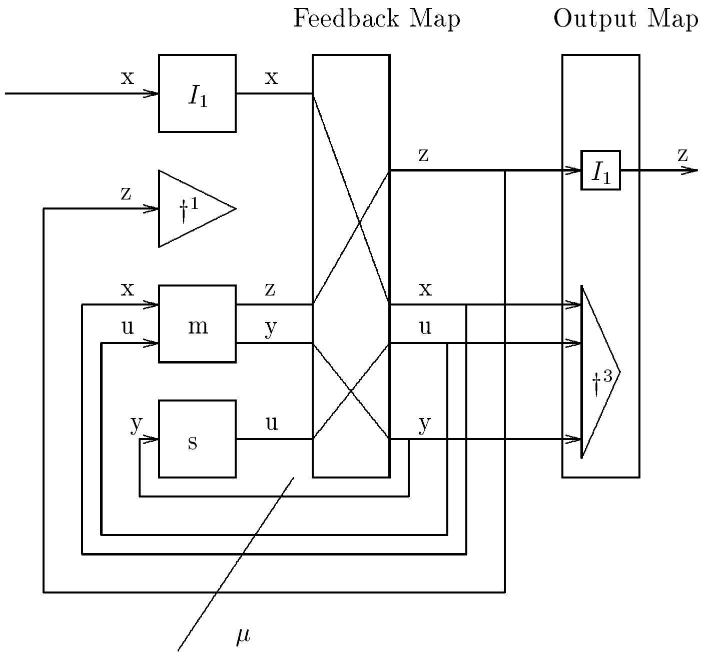


Figure 5.6: The structure of the master/slave configuration

The structure of the above figure is quite general. In fact, each Kahn network can be understood as consisting of 3 blocks:

- a *parallel composition* of the network components,
- a wiring component (the *feedback map*) which keeps track of the channel order and prepares the output for the feedback operator,
- a hiding component (the *output map*).

The interconnection of components is essentially determined by the feedback map which is *constant*. In this figure it is $\overset{11}{\chi} \parallel \overset{11}{\chi}$. This is the reason why Kahn networks have a static structure even if they are recursive, exactly as the aggregates do. Since the relationships between components are fixed once for all, aggregates do not need explicit manipulation of object identifiers. Each object has already the necessary channels to each of its clients and to each of its servers and no new clients or servers can appear anymore. Sharing of objects is also statically predefined. An object is shared by all of its clients.

5.2.4 Links versus Object Attributes

In the interactive queue definition from Section 5.2.2 we explicitly *linked* the rest of the queue to the cell being in front of it. Alternatively, we can understand this queue as an object attribute. This leads to a somewhat simpler definition.

$$
\begin{array}{l}
\mathsf{q}\,s = \textbf{case}\ s\ \textbf{of} \\
\qquad \begin{array}{ll}
\mathsf{add}(a) : t & \Rightarrow \mathsf{cell}\,a\,\mathsf{q}\,t
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{cell}\,a\,q\,i = \textbf{case}\ i\ \textbf{of} \\
\qquad \begin{array}{ll}
\mathsf{fst} : t & \Rightarrow a : \mathsf{cell}\,a\,q\,t \\
\mathsf{rst} : t & \Rightarrow q\,t : q\,t \\
\mathsf{add}\,(b) : t & \Rightarrow \mathsf{cell}\,a\,(q \ll \mathsf{add}(b))\,t : \mathsf{cell}\,a\,(q \ll \mathsf{add}(b))\,t\ \textbf{where} \\
\qquad q \ll \mathsf{add}(b) = \lambda s.\ q(\mathsf{add}(b) : s) &
\end{array}
\end{array}
$$

The case statement allows us to explicitly require that the first message a queue can ever receive is an add message. Each time the queue receives another add message, its object attribute is updated accordingly. This object replaces the current queue in response to a rst message.

In conclusion, object attributes allow to reduce the number of private servers which are explicitly linked to a client. This simplifies both the interface and the definition of the client.

5.3 Mobile Networks

5.3.1 Introduction

Aggregates are a very natural way to describe systems with a hierarchical structure. Such systems occur so often, that they received a special notation in object modeling. In principle, each system can be conceived as an aggregate.

However, encouraged by the OO paradigm, more and more researchers in the field of concurrent computation got interest in *mobile systems.*

Definition Mobile systems

> Systems in which every object can change its communication partners on the basis of computation and interaction are designated as *mobile.* □

To give an example of mobility, consider a simplified view of a *mobile telephone system* [65]. A Centre is in permanent contact with two Base stations, each in a different part of the country. A Car with a mobile telephone moves about the country; it should always be in contact with a Base. If it gets rather far from its current Base contact, then a hand–over procedure is initiated, and as a result the Car relinquishes contact with one Base and assumes contact with another. Figure 5.7 shows the possible configurations.

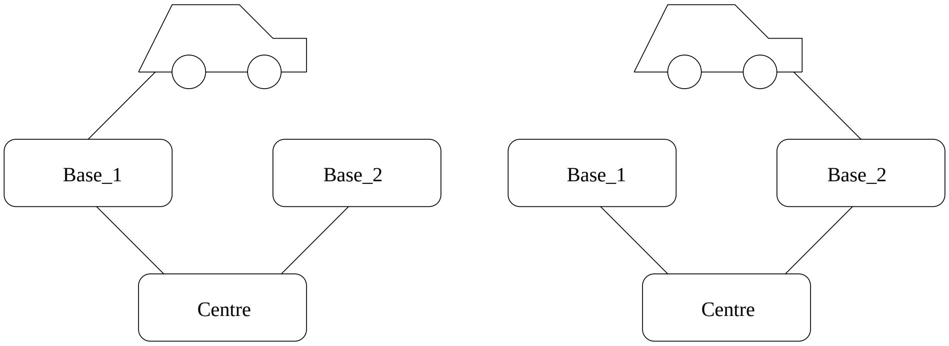


Figure 5.7: The mobile telephone example

Since the connection between the Car and the Bases is dynamic this network is mobile.

Probably the best known models which express mobility are the Actor Model of Hewitt [3, 4], the π–Calculus of Milner, Parrow and Walker [66, 67], the Chemical Abstract Machine of Berry and Boudol [12], the Rewriting Logic of Meseguer [59] and the Higher Order CCS of Bent Thomsen [89].

In these models mobility is achieved either by allowing *processes* to be passed as values in communication or by allowing *references* to processes to be communicated.

5.3.2 Mobility in π–Calculus

The π–Calculus [66, 67] is a way of describing and analyzing systems consisting of agents which interact among each other and whose configuration or neighborhood is continually changing. This calculus both *simplifies* CCS [64] by defining a structural congruence on processes and *extends* it by allowing to pass *link references* among processes.

The Syntax

The most primitive entity in π–calculus is a *name*. Names, infinitely many, are $x, y, \ldots \in \mathcal{X}$; they have no structure. In the basic version of π–calculus there is only one another kind of entity: *a process.* Processes P are built from names by the following syntax.

$\mathsf{P} ::= \mathsf{N} \mid \mathsf{P} \parallel \mathsf{Q} \mid (\nu)\mathsf{P} \mid !\mathsf{P}$
$\mathsf{N} ::= \pi.\mathsf{P} \mid \mathsf{0} \mid \mathsf{M} + \mathsf{N}$
$\pi ::= x(y) \mid \overline{x}y$

N are processes in *normal form.* They are either prefixed processes $\pi.\mathsf{P}$ or sums $\mathsf{M} + \mathsf{N}$ of normal processes. A nullary sum is written as $\mathsf{0}$.

In a summand $\pi.\mathsf{P}$ the prefix π represents an *atomic action,* the first action performed by $\pi.\mathsf{P}$. There are two basic forms of prefixes:

$x(y)$ which stands for reading some name on the channel x and calling it y. The variable y is bound in the prefixed process.

$\overline{x}y$ which stands for writing the name y on the channel named x. It does not bind y.

In each case x is *the subject* and y is *the object* of the action. The subject is *positive* for input and *negative* for output. A *name refers* to a link or a channel.

The sum $\mathsf{M} + \mathsf{N}$ represents a process able to take part in one – but only one – of the left and right alternatives. The choice is not made by the process.

$\mathsf{P} \parallel \mathsf{Q}$ means that P and Q are concurrently active, so they can act independently – but can also communicate.

$!\mathsf{P}$ means $\mathsf{P} \parallel !\mathsf{P}$. It is known as the replication operator or *bang*.

Finally $(\nu x)\mathsf{P}$ introduces a new name x and restricts its use to P. The variable x is bound by ν.

Processes with complementary prefixes can communicate with each other. For example in the process:

$\overline{x}\,y.0 \parallel x(u).\overline{u}\,v.0 \parallel \overline{x}\,z.0$

can occur one of two possible communications: between the first and the second component or between the third and the second component. If the first communication takes place, then the result configuration is:

$0 \parallel \overline{y}\,v.0 \parallel \overline{x}\,z.0$

The Structural Congruence

The intended semantics for the processes defined by the above syntax can be better understood by an anology with a *chemical solution* [12]. *Molecules* are normal form processes in a continous *Brownian* motion. If they get in contact then they *react* (i.e. communicate) with one another provided they have complementary prefixes (nested + processes can *rotate* and expose the prefix of the next summand). The hiding operator ν can be understood as *a membrane* which separates different solutions. This membrane is somewhat *porous* to allow communication between the encapsulated solution and its environment. The following congruence formalizes this behaviour.

The structural congruence $\equiv$ is the smallest congruence relation over the set $\mathcal{P}$ of processes such that the following laws hold:

(1) α–equivalence
(1.1) νx.P $\equiv$ νy.P[y/x] {y$\notin$fv(P)
(1.2) z(x).P $\equiv$ z(y).P[y/x] {y$\notin$fv(P)

(2) ($\mathcal{P}/\equiv$, $\parallel$, 0) is a symmetric monoid
(2.1) N $\parallel$ (M $\parallel$ P) $\equiv$ (N $\parallel$ M) $\parallel$ P
(2.2) N $\parallel$ 0 $\equiv$ N
(2.3) N $\parallel$ M $\equiv$ M $\parallel$ N

(3) Scoping rules for ν
(3.1) νx.0 $\equiv$ 0
(3.2) νx.νy. N = νy.νx. N
(3.3) νx. (N $\parallel$ M) $\equiv$ N $\parallel$ νx. M {x $\notin$ fv(N)

(4) Recursion (Bang operator)
!N $\equiv$ N $\parallel$!N

(5) ($\mathcal{N}/\equiv$, +, 0) is a symmetric monoid
(5.1) N + (M + P) $\equiv$ (N + M) + P
(5.2) N + 0 $\equiv$ N

(5.3) $\mathsf{N} + \mathsf{M} \equiv \mathsf{M} + \mathsf{N}$

Rules (1) are the usual α–conversion rules for bound variables. Rules (2) model the Brownian motion of the molecules and rules (5) their rotation. They also allow to discard the inactive process $\mathsf{0}$. Rules (3) model the membrane behavior. Among them, very important is rule (3.3) because it allows molecules to pass the membrane and interact with other molecules. As we will see later, reaction (i.e. communication) is allowed only inside the membrane. Finally, the rule (4) models tail recursion. Combined with link reference passing this is strongly enough to model general recursion.

The Reduction Rules

Communication in π calculus is given by defining a reduction relation $\rightarrow$ over processes $P \in \mathcal{P}$. The identification of expressions as required by $\equiv$ considerably simplifies the definition of this relation. $P \rightarrow P'$ means that P can be transformed into P' by a single computational step.

$$(comm)\quad \frac{}{(\ldots + x(y).P) \,\|\, (\ldots + \overline{x}z.Q) \;\rightarrow\; P[z/y] \,\|\, Q}$$

$$(par)\quad \frac{P \;\rightarrow\; P'}{P \,\|\, Q \;\rightarrow\; P' \,\|\, Q}$$

$$(res)\quad \frac{P \;\rightarrow\; P'}{(\nu x)P \;\rightarrow\; (\nu x)P'}$$

$$(struct)\quad \frac{Q \equiv P \quad P \rightarrow P' \quad P' \equiv Q}{Q \;\rightarrow\; Q'}$$

The only axiom is $(comm)$. It shows both how communication occurs between two atomic normal processes $\pi.\mathsf{P}$ which are complementary and how the other alternatives are discarded. The first two inference rules say that reduction can occur underneath composition and restriction. The third one simply says that structurally congruent terms have the same reductions.

The Mobile Telephone Example

Passing link refernces among processes allows to express mobility. Let us show this on the mobile telephone example. We use here the following abbreviations:

$x(y_1, \ldots, y_n)$	for	$x(w).w(y_1)\ldots w(y_n)$
$\overline{x}\, y_1 \ldots y_n$	for	$(\nu w)\,\overline{x}\, w.\overline{w}\, y_1 \ldots \overline{w}\, y_n$

We write x and $\overline{x}$ when $n = 0$. Note that w is a private channel between the communication partners so no other process can interrupt the sequence $y_1, \dots, y_n$. Beside the above abbreviations we also use recursive, parametric definitions in the form:

$K(\vec{x}) \overset{\text{def}}{=} P_K$

They can be expressed by using only ! and link passing (see [65]).

Now we are ready to give the example. The process configuration is shown in Figure 5.8.

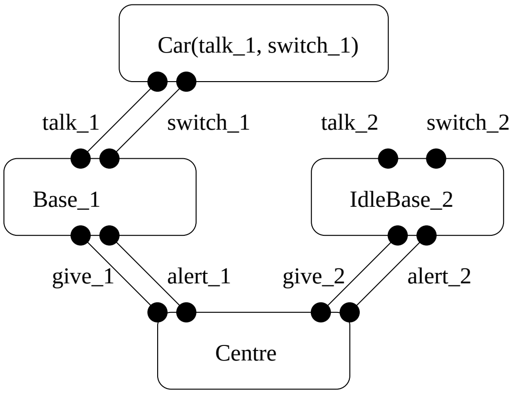


Figure 5.8: Mobile telephones in π calculus

$$\begin{aligned} \mathsf{System} \overset{\text{def}}{=} \; & (\nu\, tk_1,\, tk_2,\, sw_1,\, sw_2,\, gv_1,\, gv_2,\, al_1,\, al_2). \\ & (\mathsf{Centre} \,||\, \mathsf{Base}_1 \,||\, \mathsf{IdleBase}_2 \,||\, \mathsf{Car}(tk_1,\, sw_1)) \end{aligned}$$

$$\begin{aligned} \mathsf{Centre} &\overset{\text{def}}{=} \overline{gv_1}\, tk_2\, sw_2.\, \overline{al_2}.\, \mathsf{Centre}' \\ \mathsf{Centre}' &\overset{\text{def}}{=} \overline{gv_2}\, tk_1\, sw_1.\, \overline{al_1}.\, \mathsf{Centre} \end{aligned}$$

$$\begin{aligned} \mathsf{Base}(tk,\, sw,\, gv,\, al) \;\; & \overset{\text{def}}{=} \overline{tk}.\, \mathsf{Base}(tk,\, sw,\, gv,\, al) \\ & \;\; + gv\,(tk',\, sw').\overline{sw}\, tk'\, sw'.\, \mathsf{IdleBase}\,(tk,\, sw,\, gv,\, al) \\ \mathsf{IdleBase}(tk,\, sw,\, gv,\, al) & \overset{\text{def}}{=} al.\, \mathsf{Base}\,(tk,\, sw,\, gv,\, al) \end{aligned}$$

$$\begin{aligned} \mathsf{Car}\,(tk,\, sw) \;\; & \overset{\text{def}}{=} tk.\, \mathsf{Car}\,(tk,\, sw) \\ & \;\; + sw\,(tk',\, sw').\, \mathsf{Car}\,(tk',\, sw') \end{aligned}$$

$$\begin{aligned} \mathsf{Base}_1 &= \mathsf{Base}\,(tk_1,\, sw_1,\, gv_1,\, al_1) \\ \mathsf{Base}_2 &= \mathsf{Base}\,(tk_2,\, sw_2,\, gv_2,\, al_2) \end{aligned}$$

$$\begin{aligned} \mathsf{IdleBase}_1 &= \mathsf{IdleBase}\,(tk_1,\, sw_1,\, gv_1,\, al_1) \\ \mathsf{IdleBase}_2 &= \mathsf{IdleBase}\,(tk_2,\, sw_2,\, gv_2,\, al_2) \end{aligned}$$

The task of the Centre is (according to information which we do not model) to switch the communication between the Car and the Bases. If the Car is for example in contact with Base_1 then it sends the channels $talk_2$ and $switch_2$ to the Car and allerts Base_2 of this fact.

A Base can talk repeatedly with the Car. However, at any moment it can receive along its *give* channel two new channels which it should communicate to the Car and then become idle itself.

The Car is parametric upon a *talk* channel and a *switch* channel. On *talk* it can talk repeatedly; but at any time along *switch* it may receive two new channels which it must then start to use. In this way it relinquishes contact with the current Base and assumes contact with the new one. This dynamic switching makes the configuration mobile.

5.3.3 Mobility by Higher Order Streams

In Section 5.2.3 we discussed the inherently static structure of Kahn networks. The source of their static behavior is the way variables are used in the defining equations, a way which is equivalent to a *constant* feedback map. This is the reason why it is commonly accepted that Kahn networks are not adequate to describe mobile systems.

Fortunately, this is true only for first order systems i.e. for systems where messages themselves are not streams. However, in higher order systems i.e. in systems with *higher order streams* the constant feedback map only defines an *initial* configuration.

Sending streams is closely related to sending channel references as it happens in the π–calculus. To grasp its subtlety, let us *reuse* the Cartesian Points.

```
$\mathcal{P} = \lambda myclass.\lambda(xc).\lambda s$
    case $s$ of
        $\mathsf{x} : t \quad \Rightarrow \quad xc : myclass\ xc\ t$
        $\mathsf{mv}\,(dx) : t \quad \Rightarrow \quad p\ t : p\ t$ where $p = myclass\,(x + dx)$
```

The interesting *point* about this point definition is that the message returned in response to a mv request contains *the whole history produced after this request.* Regarding the output stream as a sequence of storage locations, this intuitively corresponds to returning a *reference* into this sequence. This reference advances each time a message is sent but is delivered only in response to a mv message.

It is therefore no wonder that with higher order streams we are able to model a wide class of mobile systems very similarly to π–Calculus. For example, let us formalize the mobile telephones with higher order streams. The process configuration is shown in Figure 5.9.

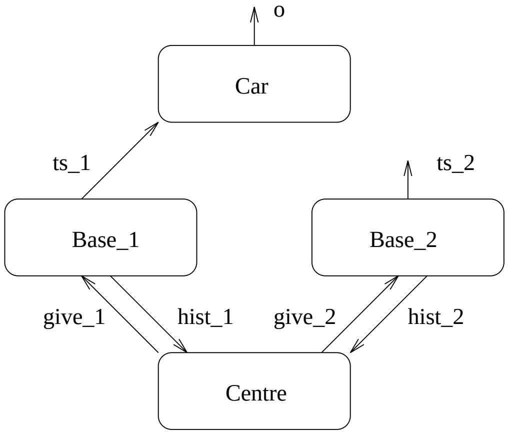


Figure 5.9: Mobile telephones with higher order streams

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{System}(p_1, p_2) = o \ \mathbf{where}} \\
\quad o & = \mathsf{Car}(ts_1) \\
\quad (ts_1, ht_1) & = \mathsf{Base}\, p_1\, gv_1 \\
\quad (gv_1, gv_2) & = \mathsf{Centre}'(ht_1, ts_2 : ht_2) \\
\quad (ts_2, ht_2) & = \mathsf{Base}\, p_2\, gv_2
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{Centre}\,(u_1 : h_1,\, h_2) & = (g_1,\, \mathsf{gv}\,(u_1) \,:\, g_2) \ \mathbf{where} \ (g_1,\, g_2) = \mathsf{Centre}'\,(h_1,\, h_2) \\
\mathsf{Centre}'\,(h_1,\, u_2 \,:\, h_2) & = (\mathsf{gv}\,(u_2) \,:\, g_1,\, g_2) \ \mathbf{where} \ (g_1,\, g_2) = \mathsf{Centre}\,(h_1,\, h_2)
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{Base}\, p\, (\mathsf{gv}(u) : g) = \mathbf{case} \ p \ \mathbf{of}} \\
\quad 0 \,:\, ps \ \Rightarrow & (\mathsf{tk} \,:\, t,\, h) & \mathbf{where}(t,\, h) = \mathsf{Base}\, ps\, (\mathsf{gv}(u) : g) \\
\quad 1 \,:\, ps \ \Rightarrow & (\mathsf{sw}\,(u) \,:\, t,\, t \,:\, h) & \mathbf{where}(t,\, h) = \mathsf{Base}\, ps\, g
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{Car}\, t = \mathbf{case} \ t \ \mathbf{of}} \\
\quad \mathsf{tk} \,:\, s & \Rightarrow \ \mathsf{out} \,:\, \mathsf{Car}\, s \\
\quad \mathsf{sw}\,(u) \,:\, s & \Rightarrow \ \mathsf{Car}\, u
\end{array}
$$

This formalization is different from the π–Calculus one in the following aspects:

- The nondeterministic choice operator "+" from the Base process is replaced by an arbitrary prophecy stream p,

- IdleBase and the channels alert and switch were removed because they are superflous in our asynchronous formalism.

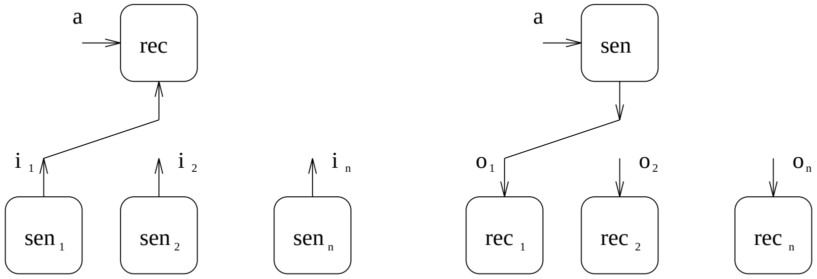
As before, the task of the Centre is (according to information which we do not model) to switch the communication between the Car and the Bases. If the Car is for example in contact with Base_1 then it sends the channel ts_2 i.e. the output produced by Base_2 to the Car.

A Base can talk repeatedly with the Car. However, at any moment it can receive along its *give* channel a new channel which it should communicate to the Car. In this case it also returns to the Centre on the channel *hist* a message containing *ts* i.e. its future output for the Car. The choice between talking or switching is controlled by the prophecy stream p.

The Car reads the messages from the current *ts* channel. If it receives a stream message sw(u) it discards the rest of the stream i.e. it relinquishes contact with the current base and uses subsequently the stream u i.e. it establishes contact withe the other base.

5.3.4 Higher Order Streams versus π–Calculus

The purpose of this section is to prove informally that each mobile system expressible in π–Calculus can also be expressed with higher order Kahn networks. The proof uses the observation that each mobile system is built in π–Calculus around two basic prefixes: *input–directed input* and *input–directed output*. They are shown in Figure 5.10.



rec = a(i).i(x).rec'

Input-directed Input

sen = a(o).$\bar{o}$ m.sen'

Input-directed Output

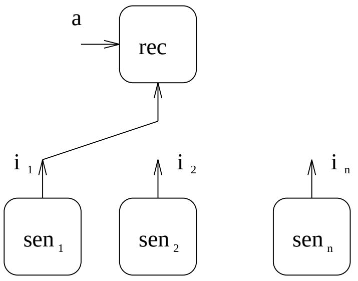
Figure 5.10: Basic building blocks for mobility in π–calculus

If we give a translation of this basic structures to higher order networks than we are done, because to each other π–calculus system corresponds a Kahn network given as the composition of basic translations.

It is important to see that both rec and sen have only one generic input and respectively output channel which is indirected by the data read on the channel a. Neither rec nor sen know in advance which are their communication partners. If trying to model this statically, one would have to allocate a separate channel for each possible partner.

Input–directed Input

Input–directed input is shown in Figure 5.11.



rec = a(i).i(x).rec'

Figure 5.11: Input–directed input

It was the essential construct used by the Car process in the mobile telephones example.

$$\begin{aligned}\mathsf{Car}\,(tk,\, sw) &\overset{\text{def}}{=} tk.\,\mathsf{Car}\,(tk,\, sw)\\ &+ sw\,(tk',\, sw').\,\mathsf{Car}\,(tk',\, sw')\end{aligned}$$

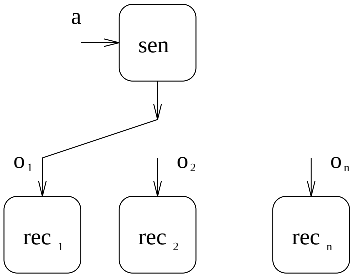
The translation to Kahn networks is as follows:

$$\mathrm{rec} = \mathrm{a}(i).i(x).\mathrm{rec}' \quad \rightsquigarrow \quad \mathrm{rec}(\mathrm{sw}(u) : t) = \mathrm{rec}(u)$$

Similarly to the π–calculus, the process rec from the Kahn network has only one (generic) input channel. However, over this channel it can receive as a message any of the other channels $i_2, \ldots, i_n$. In that case, it discards the rest of the stream and uses the stream contained in the message. This is equivalent to switching to the other channel. This switching happens again if the current stream, say i_k, contains a message of the form $\mathsf{sw}\,(i_m)$.

Input–directed Output

The dual configuration is the input–directed output as shown in Figure 5.12.



sen = a(o).ō m.sen'

Figure 5.12: Input–directed output

In this case the sender writes on a channel which is received as a message. It can in no way predict which is this channel. As in the previous case, the sender has only one generic output channel. We model this with a Kahn network having also only one channel. However, in this case the correspondence with the π–calculus is not so close beacuse it makes no much sense to receive as a message on the channel a an input channel. It seems to be more appropriate if Sen *receives as a message the actual receiver.* In this case it can write to this receiver and optionally return the receiver *continuation.* As a consequence, we give the following translation:

$$\begin{array}{llll} \mathrm{sen} = \mathrm{a}(o).\overline{o}\,m.\mathrm{sen}' & \rightsquigarrow & \mathrm{sen}(\mathrm{fn}(r) : t) = & (\mathrm{sw}(r\,(m : u)) \,:\, u,\, \mathrm{fn}(r \ll m) \,:\, v) \\ & & & \textbf{where} \\ & & & (u,\, v) = \mathrm{sen}\, t \end{array}$$

Note that the first projection of the output produced by the sender has the same structure as the input of the receiver in the input–directed input case.

As an example, suppose we want to model a system which is dual to the mobile telephones. There is a Centre knowing the input channels of the receivers Rec_1 and Rec_2. As before, it sends these channel refernces to a sender Sen which is parametric upon an output channel o. It can repeatedly talk on the channel o or receive a new output channel on a which it then starts to use. Finally, the receivers Rec_k consume the messages sent by Sen.

$$\mathsf{System} \overset{\mathrm{def}}{=} (\nu\, a,\, i_1,\, i_2).\, (\mathsf{Centre} \,||\, \mathsf{Sen}(i_1) \,||\, \mathsf{Rec}_1 \,||\, \mathsf{Rec}_2)$$

$$\mathsf{Centre} \overset{\mathrm{def}}{=} \overline{a}\, i_1.\, \mathsf{Centre}'$$
$$\mathsf{Centre}' \overset{\mathrm{def}}{=} \overline{a}\, i_2.\, \mathsf{Centre}$$

$$\mathsf{Sen}(o) \overset{\mathrm{def}}{=} \overline{o}\,\mathsf{talk}.\, \mathsf{Sen}(o) \;+\; a\,(o').\, \mathsf{Sen}(o')$$

$$\mathsf{Rec}_k \overset{\mathrm{def}}{=} i_k(m).\, \mathsf{Rec}_k$$

The corresponding configuration in π–calculus is shown in Figure 5.13.

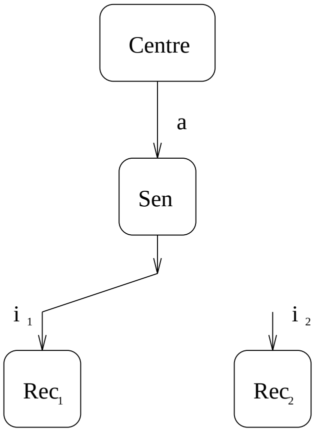


Figure 5.13: Input–directed output example

In this modeling the Centre contains as "attributes" the channel references i_1 and i_2 to the processes Rec_1 and Rec_2. In the modeling with higher order streams the processes Rec_1 and Rec_2. themselves are attributes of the Centre. The corresponding configuration with higher order streams is given in Figure 5.14.

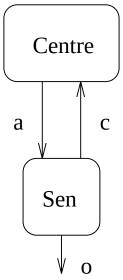


Figure 5.14: Input–directed output example

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{System}(p) \overset{\text{def}}{=} o \ \textbf{where}} \\
\quad (o, c) & = & \mathsf{Sen}\, p\, \mathsf{Rec}_1\, a \\
\quad a & = & \mathsf{Centre}\ \mathsf{Rec}_1\, \mathsf{Rec}_2\, c
\end{array}$$

$$\mathsf{Centre}\ r_1\, r_2\, (r : c) \overset{\text{def}}{=} r_1 \ :\ \mathsf{Centre}'\ r\, r_2\, c$$

$$\mathsf{Centre}'\ r_1\, r_2\, (r : c) \overset{\text{def}}{=} r_2 \ :\ \mathsf{Centre}\ r_1\, r\, c$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{Sen}\ p\ rec\ (r : t) \overset{\text{def}}{=} \textbf{case}\ p\ \textbf{of}} \\
\quad 0 : p' & \Rightarrow (\mathsf{ms}(m)\ :\ u, (rec \ll m)\ :\ v) & \textbf{where}(u, v) = \mathsf{Sen}\, p'\, (rec \ll m)\, (r : t) \\
\quad 1 : p' & \Rightarrow (\mathsf{sw}(r\, u)\ :\ u,\ r\ :\ v) & \textbf{where}(u, v) = \mathsf{Sen}\, p'\, r\, t
\end{array}$$

$$\mathsf{Rec}_k(x\ :\ t) \overset{\text{def}}{=} \mathsf{Rec}_k\, t$$

5.4 A Calculus of Mobile Networks

In the previous sections we described an object configuration as a Kahn network i.e. as a function having the form:

$$
\begin{array}{l}
f(\vec{x}) = \vec{z} \;\; \textbf{where} \\
\quad \vec{y_1} = g_1(\vec{x}, \vec{y}) \\
\quad \ldots \\
\quad \vec{y_n} = g_n(\vec{x}, \vec{y})
\end{array}
$$

One can alternatively dispense from f and describe a network simply as a *set of equations:*

$$
\begin{array}{l}
\vec{y_1} = g_1(\vec{x}, \vec{y}) \\
\ldots \\
\vec{y_n} = g_n(\vec{x}, \vec{y})
\end{array}
$$

As before $\vec{y}$ is the output and $\vec{x}$ is the input of the whole network and for each network component $\vec{y_i} = g_i(\vec{x}, \vec{y})$, $\vec{y_i}$ is the output and $\vec{x}, \vec{y}$ is the input. The hiding operation performed by f is achieved by introducing an *existential quantifier.* To simplify the syntax we also dispense from the set notation and compose network components with a *parallel composition operator* $\|$. Finally, *parametric, recursive networks* can be defined in the form $P(\vec{x}) = K_P$.

The motivation for this calculus is to define a platform for comparing the properties of Kahn networks and calculi expressing mobility (especially the π-calculus). It can be also understood as an instrument for investigating the logical properties necessary to express mobility.

5.4.1 The Syntax

The *context free syntax* of networks is given below where P is the name of a parameterized network:

$$\mathsf{N} ::= \mathsf{x} = \mathsf{t} \mid \mathsf{N} \parallel \mathsf{N} \mid \exists \mathsf{x}.\mathsf{N} \mid \mathsf{P}(\vec{x})$$

Hence, a non-recursive network is described in this case as

$$\exists u_1, \ldots, u_m.\; u_1 = \mathsf{g}_1(\vec{x}, \vec{u}) \parallel \ldots \parallel u_n = \mathsf{g}_n(\vec{x}, \vec{u})$$

where $m < n$. For the *context sensitive syntax* we must consider the following aspects:

1. The parallel composition $\mathsf{M} \parallel \mathsf{N}$ of two networks automatically *conects* the output channels of M to the input channels having the same name of N and reciprocally. These channels cease to be input channels for $\mathsf{M} \parallel \mathsf{N}$.

2. Networks can be composed only if they have *disjoint* output channels. Otherwise, common output channels would have to be merged in some way. We want to avoid this situation.

3. For parameterized networks $\mathsf{P}\,(\vec{x})$ we have to know which are the *input* and which are the *output* channels. This is given by the network *signature* which has the form $\mathsf{P} : \sigma_1 \times \ldots \times \sigma_n \overset{c_1 \times \ldots \times c_n}{\rightarrow} net$. Each σ_k defines the type of the channel k and c_k states if it is an input or an output channel ($c_k \in \{i, o\}$).

The rules given below have no axioms. This is because they are based on the context sensitive syntax for λ-terms. A typing assertion $\Gamma \rhd_\lambda e : \sigma$ says that e is a well formed λ-term of *stream type* σ with free stream variables in Γ. The contexts of the network terms contain additional information stating the input or the output nature of the channel variables. An output channel x of type σ is written as $\mathrm{o}(x) : \sigma$ to distinguish it from an input one which is written as $\mathrm{i}(x) : \sigma$. When we do not want to distinguish between input and output channels we write $c(x) : \sigma$. For a context $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ we write $c(\Gamma)$ for $\{c(x_1) : \sigma_1, \ldots, c(x_n) : \sigma_n\}$.

$$(eq) \quad \frac{\Gamma, x : \sigma \rhd_\lambda e : \sigma}{\mathrm{i}(\Gamma), \mathrm{o}(x) : \sigma \rhd x = e : net} \; \{\mathrm{dom}(\Gamma) = \mathrm{fv}(e) - \{x\}$$

$$(\| i) \quad \frac{\Gamma \rhd M : net \quad \Delta \rhd N : net}{\Gamma + \Delta \rhd M \| N : net} \; \{\Gamma + \Delta \text{ context}$$

$$(\exists i) \quad \frac{\Gamma, o(x) : \sigma \rhd N : net}{\Gamma \rhd \exists x. N : net}$$

$$(pn) \quad \frac{\Gamma \rhd_\lambda x_1 : \sigma_1 \quad \ldots \quad \Gamma \rhd_\lambda x_n : \sigma_n}{c_1(x_1) : \sigma_1, \ldots, c_n(x_n) : \sigma_n \rhd P(x_1, \ldots, x_n) : net} \; \Big\{ P \in \Sigma_{\sigma_1 \times \ldots \times \sigma_n \overset{c_1 \times \ldots \times c_n}{\rightarrow} net}$$

The operation + over contexts, takes care of the interconnections occurring in the parallel composition. It builds a multiset modulo the following interconnecting equations:

$$\{\mathrm{i}(x) : \sigma\} + \{\mathrm{i}(x) : \sigma\} = \{\mathrm{i}(x) : \sigma\}$$
$$\{\mathrm{i}(x) : \sigma\} + \{\mathrm{o}(x) : \sigma\} = \{\mathrm{o}(x) : \sigma\}$$

The first equation says that input channels are shared. The second one says that an output channel is connected to the input channel with the same name. The input channel is removed from the input/output channel context. If the same channel occurs with different types or twice as output then neither of the above rules can be applied. As a consequence the multiset cannot be reduced to a set i.e. to a valid context and the composition is undefined.

The annotation $c_1 \times \ldots \times c_n$ where $c_i \in \{i, o\}$ in the type of a parameterized network reflects its potential input/output behavior.

5.4.2 The Structural Congruence

As in π-calculus, network components can be understood as *molecules* in a Brownian motion which *react* if they have complementary channels. The existential quantifier plays the role of the *membrane*. It is interesting that these *solution* properties are a natural consequence of the intended interpretation for our calculus as a subset of the first order logic with equality and recursive predicates (read $||$ as $\wedge$).

Our calculus is axiomatic. As a consequence we do not define a separate reduction relation. Instead we have laws which correspond to the structural congruence of the π-calculus and laws corresponding to communication.

The congruence relation $\Leftrightarrow$ must satisfy the following axioms.

(1) α–equivalence

$\exists \mathsf{x.P} \Leftrightarrow \exists \mathsf{y.P[y/x]}$ $\quad\quad\quad\quad \{\mathsf{y} \notin \mathsf{fv(P)}$

(2) $(\mathcal{N}/\Leftrightarrow, ||, 0)$ is a symmetric monoid with $0 = \mathsf{true}$

(2.1) $\mathsf{N} \;||\; (\mathsf{M} \;||\; \mathsf{P}) \Leftrightarrow (\mathsf{N} \;||\; \mathsf{M}) \;||\; \mathsf{P}$
(2.2) $\mathsf{N} \;||\; 0 \Leftrightarrow \mathsf{N}$
(2.3) $\mathsf{N} \;||\; \mathsf{M} \Leftrightarrow \mathsf{M} \;||\; \mathsf{N}$

(3) Scoping rules for $\exists$

(3.1) $\exists \mathsf{x}.0 \Leftrightarrow 0$
(3.2) $\exists \mathsf{x}.\exists \mathsf{y}.\ \mathsf{N} \Leftrightarrow \exists \mathsf{y}.\exists \mathsf{x}.\ \mathsf{N}$
(3.3) $\exists \mathsf{x}.\ (\mathsf{N} \;||\; \mathsf{M}) \Leftrightarrow \mathsf{N} \;||\; \exists \mathsf{x}.\ \mathsf{M}$ $\quad\quad \{\mathsf{x} \notin \mathsf{fv(N)}$

(4) Recursion - For each definition $\mathsf{P}(\vec{x}) = \mathsf{K}$ add

$\mathsf{P}(\vec{y}) = \mathsf{K}[\vec{y}/\vec{x}]$

(5) Input and output neutrals

(5.1) $\mathsf{i}(x) \in \mathsf{Chan(N)} \Rightarrow (\mathsf{N} \Leftrightarrow \exists \mathsf{u.\ u = x} \;||\; \mathsf{N[u/x]})$
(5.2) $\mathsf{o}(y) \in \mathsf{Chan(N)} \Rightarrow (\mathsf{N} \Leftrightarrow \exists \mathsf{v.\ y = v} \;||\; \mathsf{N[v/y]})$

The graphical representation of the neutrals is given in Figure 5.15.

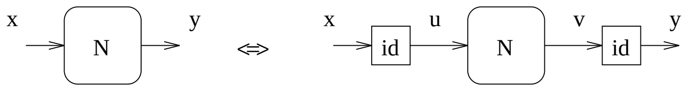


Figure 5.15: The input and output neutrals

5.4.3 Communication

In contrast to the π-calculus we have only one axiom. Requiring that $\Leftrightarrow$ is a congruence relation we automatically obtain the equivalents for $(par), (res)$ and $(struct)$. However, we can follow a similar path to the π-calculus when giving an *operational* semantics.

(6) Internal Communication

$\exists x.\ \mathsf{N} \parallel \mathsf{x} = \mathsf{a} : \mathsf{Q} \Leftrightarrow \exists x.\ \mathsf{N}[\mathsf{a} : x/x] \parallel x = \mathsf{Q}[\mathsf{a} : x/x]$

The graphical representation for (6) is given in Figure 5.16.

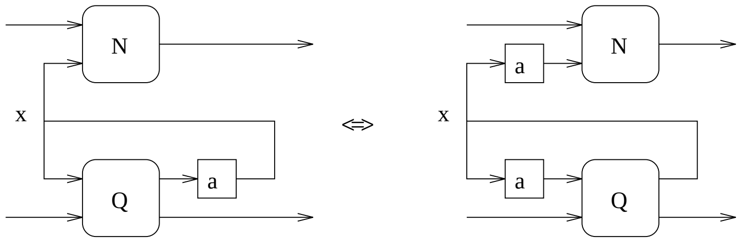
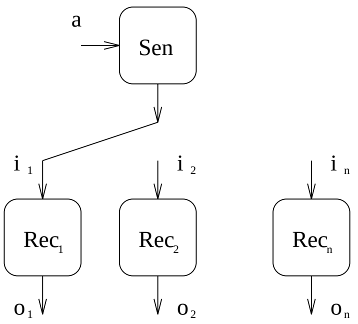


Figure 5.16: Local channel communication

It can be explained as follows. The rest Q of the stream x can be given a local name y i.e. $y = Q$. Then $x = a : y$. If we substitute x with $a : y$ in N and Q then x does not occur anymore in N and Q and it can therefore be discarded. Now using α-conversion we can rename y in x. A proof for the equivalences (5.1) and (6) in a natural deduction style is given in Appendix D. The communication axiom also gives the motivation for the rules (5). They prepare the network N for the *internal* communication.

5.4.4 The Mobile Nets versus the π-Calculus

The *input-directed output* example discussed in Section 5.3.4 can be expressed in the above calculus very similarly to the π-calculus description. The configuration is shown again in Figure 5.17.



Sen = a(o).ō m.Sen'

Figure 5.17: Input-directed output

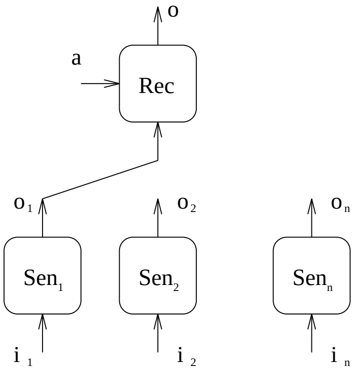
This can be described in the above syntax as follows:

$$\mathsf{Syst} \overset{\text{def}}{=} \exists\, a, i_1, \ldots, i_n, o_1, \ldots, o_n.$$
$$o_1 = \mathsf{Rec}_1(i_1) \parallel \ldots \parallel o_n = \mathsf{Rec}_n(i_n) \parallel \mathsf{Sen}(a)$$

$$\mathsf{Sen}(i : a(i)) = \exists u.\ (i = m : u) \parallel \mathsf{Sen}(a(u))$$

The stream a is not further specified. It should contain only channel names which are not already defined as outputs in the system configuration otherwise we get the merge problem. How to verify this condition statically is not treated in this thesis and it is left as a further research problem. This example also shows the definition of a recursive network.

The *input-directed input* example can be modeled as before. We split here for change the input of Rec in two channels. The configuration is shown in Figure 5.18.



Rec = a(i).i(x).Rec'

Figure 5.18: Input-directed input

It can be described in the above syntax as follows:

$$\begin{array}{l}
\mathsf{Syst}(p) \stackrel{\text{def}}{=} \exists\, o, i_1, \ldots, i_n, o_1, \ldots, o_n. \\
\qquad o_1 = \mathsf{Sen}_1(i_1) \parallel \ldots \parallel o_n = \mathsf{Sen}_n(i_n) \parallel \mathsf{o} = \mathsf{Rec}\, p\, o_1\, a
\end{array}$$

$$\begin{array}{l}
\mathsf{Rec}\, p\, (\mathsf{talk} : i)\, (j : a) = \textbf{case}\ p\ \textbf{of} \\
\qquad \mathsf{0} : ps \Rightarrow \mathsf{out} : \mathsf{Rec}\, ps\, i\, (j : a) \\
\qquad \mathsf{1} : ps \Rightarrow \mathsf{Rec}\, ps\, j\, a
\end{array}$$

As before, the channels a and $i_1, \ldots, i_n$ are not further specified here.

Chapter 6

Implementation

An implementation of the type inference algorithm given in Section 3.2 was written in common Lisp. It is an extension of the system Illyria to support lazy data types and function extension. Illyria was developed at IBM by Aiken and Wimmers and it is used to type the language FL, a successor of the Backus' language FP.

In the following examples we use some syntactic conventions of lisp. A quote suppresses evaluation; the value of a quoted expression is just the unevaluated expression. Terms must be quoted. Either a normal quote or a back-quote may be used. If an expression is back-quoted, then any subexpression ,x is replaced by the value of variable x. The prompt of the Common Lisp interpreter is <cl>. Finally, Illyria displays unions as | and intersections as &.

For example, the function which applies its argument on the argument itself can be defined and typed in this system as below.

```
<cl> (defvar t-half-bottom '(lam x (x x)))
T-HALF-BOTTOM

<cl> (typ t-half-bottom)
forall (a, c).c → a
where
0 ≤ c ≤ c → a
```

This type is more general than the familiar one which is recovered by making a = b.

The untyped fixed-point operator can be defined and typed as below.

```
<cl> (defvar t-y '(lam u ((lam x (u (x x))) (lam x (u (x x))))))
T-Y

<cl> (typ t-y)
forall (a, b).((b → a) & (b → b)) → a
```

As before this type is more general than the familiar one which is recovered by making a = b.

A simple point object with coordinates 0 and 1 can be defined and type checked as below. Constructor symbols are preceded by a colon.

```
<cl> (defvar xyD ‘(lam m
            (case m of
                  (:x 0)
                  (:y 1)
            )
      ))
XYD
```

```
<cl> (typ xyD)
forall (a).a → ((INT ? (a & X)) | (INT ? (a & Y)))
where
0 <= a <= X | Y
```

If we apply xyD on :x or :y we obtain the expected type INT.

```
<cl> (defvar xyA1 ‘(,xyD :x))
XYA1
```

```
<cl> (typ xyA1)
INT
```

We can extend this definition with a color as follows.

```
<cl> (defvar xycD ‘(,xyD extend
            (:c :red)
      ))
XYCD
```

```
<cl> (typ xycD)
forall (a, b).b →
      ((((INT ? (a & X)) | (INT ? (a & Y))) ? (b & NOT{C})) |
      (RED ? (b & C)))
where
      0 <= b <= C | (a & NOT{C})
      0 <= a <= X | Y
```

Applying xycD on :x and :c we get as expected the types INT and RED.

```
<cl> (defvar xycA1 ‘(,xycD :x))
```

```
<cl> (defvar xycA2 `(,xycD :c))
XYCA1
XYCA2

<cl> (typ xycA1)
INT
<cl> (typ xycA2)
RED
```

Similarly, we can override the definition for the y coordinate as follows.

```
<cl> (defvar xyDO `(,xyD extend
          (:y :red)
      ))
XYDO

<cl> (typ xyDO)
forall (a, b).b →
      ((((INT ? (a & X)) | (INT ? (a & Y))) ? ((b & NOT{Y}))) |
      (RED ? (b & Y)))
where
      0 <= b <= Y | (a & NOT{Y})
      0 <= a <= X | Y
```

Applying now xyDO to :x and :y we get as expected the types INT and RED.

```
<cl> (defvar xyDOa1 `(,xyDO :x))
<cl> (defvar xyDOa2 `(,xyDO :y))
XYDOA1
XYDOA2

<cl> (typ xyDOa1)
INT
<cl> (typ xyDOa2)
RED
```

Finally, a stream of :x messages can be defined and type-checked as follows.

```
<cl> (defvar stD `(,t-y (lam self (:cons :x self))))
STD

<cl> (typ stD)
CONS(X, a)
where
      a = CONS(X, a)
```

Chapter 7

Conclusions and Future Work

In this thesis we have developed a functional, asynchronous model for concurrent object oriented programming. This model is based on the implicitly typed calculus $\lambda_{\leq}^{\forall,\rightarrow,c,\cup,\cap,?}$ and it views an object configuration as a network of stream processing functions interacting over streams of messages.

Both concurrency aspects and OO issues influenced our object model and the properties of the calculus. Concurrency considerations determined us to abandon the record model in favor of the case-function one. Inheritance motivated the use of subtyping with its natural extension to union, intersection and conditional types. Parameterized or generic classes motivated the use of parametric polymorphism. Finally, the recursive nature of objects which is implicit in the use of the pseudo-variables $self$ and *myclass* motivated the use of recursive types. Recursive types were also necessary to model streams and provided the theoretical background for passing streams themselves as messages. Higher order streams proved to be very useful in expressing mobile systems. For the further investigation of mobility we also developed a network calculus.

From a theoretical perspective, some of the main contributions of this thesis are:

- A new way to model objects as case-functions. This modeling treats in a uniform manner both the sequential and the concurrent dialects of OO programming. Moreover it closes the gap between functions and objects and between objects and processes.

- A formal meaning for the basic OO concepts like objects and classes, links and associations, polymorphism and inheritance. Associated with objects we gave a meaning for object creation, encapsulation and message passing. Associated with classes we gave a meaning for inheritance and polymorphism. Finally, object configurations explained links and associations.

- An asynchronous model for concurrent OO programming. Associated with concurrency we showed how higher order streams can be used to model mobility. Moreover, for the further analysis of mobility we developed a network calculus.

From a practical perspective we mention:

- A step towards the integration of informal OO methodologies and formal methods. Providing the OMT or the Fusion method with our formal notation seems to be pragmatically feasible. In this respect note that our formalism makes the very desirable unification of the functional and of the dynamic aspects of a system.
- The practical verification of our ideas with an implementation of the inference algorithm.

As with any work of this nature there are aspects in this thesis which were deliberately ignored or simplified in order to keep its size and complexity between reasonable limits. The following sections describe additional areas for further study outlining some preliminary ideas in each case.

7.1 Theoretical Work

7.1.1 A First Order Logic

In chapter 3 we were mainly interested to develop a type theoretical framework for explaining OO issues like inheritance and polymorphism.

A first priority of the future activity is to extend this framework with the associated equational theory. This theory can be further embedded in the first order logic as done for example for the language SPECTRUM [45]. This would allow us to support system developments from abstract, non-executable specifications to concrete, executable implementations. This logic will also allow us to investigate the usefulness of OO concepts like inheritance and subtyping in the specification process.

7.1.2 Strictness Declarations

In order to keep things simple we used in this thesis only lazy constructors. This is also the case for lazy languages like Haskell or Gofer. However, in a first order logic it would be desirable to allow strictness declarations for various argument positions of constructors. One could imagine declarations of the form:

$\mathsf{cons}(\alpha, \beta)$ **strict in 1**

The modification of the type inference system, of the type inference algorithm and of the semantics is straight forward. Syntactically we have to replace the rule ($\leq cc$) by

$$
\begin{array}{lllll}
(\leq cc)' & \Gamma \mid S, \mathsf{c}(l_1, \ldots, l_n) \leq \mathsf{c}(r_1, \ldots, r_n) & \rightsquigarrow & \Gamma & \mid S, l_1 \leq r_1, \ldots, l_n \leq r_n \\
 & & & & \mid S, l_{i_1} \leq 0 \\
 & & & & \quad \ldots \\
 & & & & \mid S, l_{i_k} \leq 0
\end{array}
$$

where $i_1, \ldots, i_k$ are the strict positions of the constructor c.

Semantically a uniform treatment of constructors which are strict on some argument positions and lazy on others is achieved by using lifting and coalesced products. Instead of the type $(U \times \ldots \times U)_\perp$ we can use the isomorphic type $U_\perp \otimes \ldots \otimes U_\perp$. Strict positions are not lifted like in the product $U \otimes U_\perp$ which corresponds to cons.

7.1.3 PER Semantics

The ideal or inclusive sets model given in section 3.3 allows a very intuitive interpretation for types as sets and for subtyping as inclusion. A well known deficiency of this model is that the ξ rule is not sound and consequently extensionality fails. Although this does not influence our soundness lemma this could not be satisfactory when providing the equational theory.

The main reason that subset models do not form models of typed lambda calculus is that equality is untyped or independent of type. However, we can construct models in much the same spirit if in addition to a membership predicate we also associate an equivalence relation with each type. The equivalence says when two elements are to be regarded as equal with respect to that type. The membership relation and the equivalence relation can be combined by using Partial Equivalence Relations (PERs). In this case types can be interpreted as PERs and subtyping as inclusion of relations. PER models for recursive types known as CUPERs (Complete Uniform PERs) were developed in [7, 27, 8, 1].

7.1.4 Typed Semantics

The ideal and the CUPER models are type assignment semantics. However, following the directions presented in section 3.3.2 we can give a typed semantics for our calculus. The main technical difficulty in this case seems to be proving coherence.

7.1.5 Merge Component

In the network calculus presented in section 5.4 we allowed to compose networks only if they had disjoint output channels. In this way we avoided merging. In a practical implementation however, it could be necessary to explicitly use merging components. Further work should investigate the usefulness of explicit merging components and give an adequate formal treatment.

7.1.6 The Calculus of Mobile Networks

The calculus presented in section 5.4 has to be further developed. For example, in this calculus it is possible to describe network configurations both on the functional and on the predicative level. Since these descriptions are not completely independent we have to find out the right balance between them.

An important role in mobility is played by privacy constraints. In this respect we need syntactical criteria to assure the correctness of configurations like the input-directed output one.

Experience with this calculus should be gained by using it on particular examples. These should also point out if additional rules are necessary.

Last but not least, we should provide the calculus both with an operational and with a denotational semantics.

7.2 Practical Work

7.2.1 Type Simplification

The implementation of the type inference algorithm always returns the *least* type of an expression. This is not the *principal* type because it is often equivalent with other types which are not instances of this type. An important aspect is therefore to put this type in a “most-simple” form. This is currently done by a type simplification algorithm. However, this algorithm does not always deliver the expected results, especially when conditional types occur in conjunction with recursive ones. It is therefore necessary to carefully analyze the sources of these anomalies and to redesign the simplification part if necessary.

7.2.2 Theorem Proving

Theorem proving in presence of subtyping got much interest in the last period. It would be therefore very desirable to extend existing theorem provers like e.g. Isabelle with a type system along the lines presented in this thesis. Since Isabelle uses the Hindley-Milner type inference algorithm which is an instance of ours, this seems to be practically realizable.

7.2.3 Object Oriented Methodologies

An interesting practical consideration is to integrate our formal framework with an OO methodology like OMT or Fusion. Especially Fusion which is a "descendent" of OMT seems to be appropriate for this purpose because its operations specification and its asynchronous communication model are very close to ours.

Appendix A

The Type Inference System

A.1 Types

A.1.1 Type Expressions

$$\begin{array}{ll} \tau & ::= \alpha \mid \tau_1 \rightarrow \tau_2 \mid c(\tau_1, \ldots, \tau_n) \mid \tau_1 \cup \tau_2 \mid \tau_1 \cap \tau_2 \mid \tau_1 ? \tau_2 \mid 0 \mid 1 \\ \sigma & ::= \tau \mid \forall \alpha_i . \tau \textbf{ where } S \end{array}$$

A.1.2 Subtyping

Proper Constraints

$$\begin{array}{ll} l & ::= 0 \mid \alpha \mid r \rightarrow l \mid c(l_1, \ldots, l_n) \mid l_1 \cap \overline{l_2} \mid l_1 \cup l_2 \\ r & ::= 0 \mid \alpha \mid l \rightarrow r \mid c(r_1, \ldots, r_n) \mid r_1 \cap r_2 \mid r_1 \cup r_2 \qquad \text{where } \overline{r_1 \cap r_2} = 0 \\ & \\ S & ::= l \leq r, S \mid \epsilon \end{array}$$

Subtyping Rules

$$(\leq ref) \quad \frac{}{S \vdash \tau \leq \tau}$$

$$(\leq tra) \quad \frac{S \vdash \tau_1 \leq \tau_2 \quad S \vdash \tau_2 \leq \tau_3}{S \vdash \tau_1 \leq \tau_3}$$

$$(\leq\rightarrow) \quad \frac{S \vdash \tau_1^{'} \leq \tau_1 \quad S \vdash \tau_2 \leq \tau_2^{'}}{S \vdash \tau_1 \rightarrow \tau_2 \leq \tau_1^{'} \rightarrow \tau_2^{'}}$$

$$(\leq\to c) \quad \frac{}{S \vdash \tau_1 \to \tau_2 \not\leq c(\tau'_1, \ldots \tau'_n)}$$

$$(\leq c \to) \quad \frac{}{S \vdash c(\tau'_1, \ldots \tau'_n) \not\leq \tau_1 \to \tau_2}$$

$$(\leq cd) \quad \frac{}{S \vdash c(\tau_1, \ldots \tau_n) \not\leq d(\tau'_1, \ldots \tau'_m)} \; \{ \text{ if } c \neq d$$

$$(\leq cc) \quad \frac{S \vdash \tau_1 \leq \tau'_1 \quad \ldots \quad S \vdash \tau_n \leq \tau'_n}{S \vdash c(\tau_1, \ldots \tau_n) \leq c(\tau'_1, \ldots \tau'_n)}$$

$$(\leq \cap r) \quad \frac{S \vdash \tau \ \leq \ \tau_1 \quad S \vdash \tau \ \leq \ \tau_2}{S \vdash \tau \ \leq \ \tau_1 \cap \tau_2}$$

$$(\leq \cap lb) \quad \frac{}{S \vdash \tau_1 \cap \tau_2 \ \leq \ \tau_i} \; \{ \ i = 1, 2$$

$$(\leq \cup l) \quad \frac{S \vdash \tau_1 \ \leq \ \tau \quad S \vdash \tau_2 \ \leq \ \tau}{S \vdash \tau_1 \cup \tau_2 \ \leq \ \tau}$$

$$(\leq \cup gb) \quad \frac{}{S \vdash \tau_i \ \leq \ \tau_1 \cup \tau_2} \; \{ \ i = 1, 2$$

$$(\leq ?1) \quad \frac{S \vdash \tau_1 \ \leq \ \tau}{S \vdash \tau_1 ? \tau_2 \ \leq \ \tau}$$

$$(\leq ?2) \quad \frac{S \vdash \tau_2 \ \leq \ 0}{S \vdash \tau_1 ? \tau_2 \ \leq \ \tau}$$

$$(\leq 0l) \quad \frac{}{S \vdash 0 \ \leq \ \tau}$$

$$(\leq 0c) \quad \frac{}{S \vdash c(\tau_1, \ldots, \tau_n) \ \not\leq \ 0}$$

$$(\leq 0 \to) \quad \frac{}{S \vdash \tau_1 \to \tau_2 \ \not\leq \ 0}$$

A.1.3 Recursive Constraints

Contractive types

$$\begin{array}{ll}
0 \succ \beta & 1 \succ \beta \\
\overline{\tau} \succ \beta & \neg\overline{\tau} \succ \beta \\
c(\tau_1, \ldots, \tau_n) \succ \beta & \tau_1 \rightarrow \tau_2 \succ \beta \\
\alpha \succ \beta \Leftrightarrow \alpha \neq \beta & \tau_1 ? \tau_2 \succ \beta \Leftrightarrow \tau_1 \succ \beta \wedge \tau_2 \succ \beta \\
\tau_1 \cup \tau_2 \succ \beta \Leftrightarrow \tau_1 \succ \beta \wedge \tau_2 \succ \beta & \tau_1 \cap \tau_2 \succ \beta \Leftrightarrow \tau_1 \succ \beta \wedge \tau_2 \succ \beta
\end{array}$$

Rules for Free Variables

$$(\leq vl) \quad \frac{}{S, l \leq \alpha \leq u \mid A, \alpha \leq \tau \vdash \alpha \leq \tau}$$

$$(\leq vr) \quad \frac{}{S, l \leq \alpha \leq u \mid A, \tau \leq \alpha \vdash \tau \leq \alpha}$$

$$(\leq \mu vl) \quad \frac{S, l \leq \alpha \leq u \mid A, \alpha \leq \tau \vdash u \leq \tau}{S, l \leq \alpha \leq u \mid A \vdash \alpha \leq \tau}$$

$$(\leq \mu vr) \quad \frac{S, l \leq \alpha \leq u \mid A, \tau \leq \alpha \vdash \tau \leq l}{S, l \leq \alpha \leq u \mid A \vdash \tau \leq \alpha}$$

$$(\leq \mu vlr)_1 \quad \frac{S, l \leq \alpha_j \leq u \mid A, \alpha_i \leq \alpha_j \vdash \alpha_i \leq l}{S, l \leq \alpha_j \leq u \mid A \vdash \alpha_i \leq \alpha_j} \; \{i \leq j$$

$$(\leq \mu vlr)_2 \quad \frac{S, l \leq \alpha_i \leq u \mid A, \alpha_i \leq \alpha_j \vdash u \leq \alpha_j}{S, l \leq \alpha_i \leq u \mid A \vdash \alpha_i \leq \alpha_j} \; \{j \leq i$$

Rules for Bound Variables

$$(\leq bl) \quad \frac{}{S, \alpha = \tau_1 \mid A, \alpha \leq \tau_2 \vdash \alpha \leq \tau_2}$$

$$(\leq br) \quad \frac{}{S, \alpha = \tau_1 \mid A, \tau_2 \leq \alpha \vdash \tau_2 \leq \alpha}$$

$$(\leq \mu bl) \quad \frac{S, \alpha = \tau_1 \mid A, \alpha \leq \tau_2 \vdash \tau_1 \leq \tau_2}{S, \alpha = \tau_1 \mid A \vdash \alpha \leq \tau_2} \; \{\tau_1 \succ \alpha$$

$$(\leq \mu br) \quad \frac{S, \alpha = \tau_1 \mid A, \tau_2 \leq \alpha \vdash \tau_2 \leq \tau_1}{S, \alpha = \tau_1 \mid A \vdash \tau_2 \leq \alpha} \; \{\tau_1 \succ \alpha$$

$$(\leq \mu blr) \quad \frac{S, \alpha = \tau_1, \beta = \tau_2 \mid A, \alpha \leq \beta \vdash \tau_1 \leq \tau_2}{S, \alpha = \tau_1, \beta = \tau_2 \mid A \vdash \alpha \leq \beta} \; \{ \tau_1 \succ \alpha, \; \tau_2 \succ \beta$$

A.2 Expressions

A.2.1 Context free syntax

$$\begin{array}{lll}
e & ::= & x \mid \lambda x.e \mid e_1 e_2 \mid c(e_1, \ldots, e_n) \\
 & \mid & \textbf{case } e \textbf{ of } p_1 \Rightarrow e_1 \ldots p_n \Rightarrow e_n \\
 & \mid & e_1 \textbf{ extend } (p \Rightarrow e) \\
 & \mid & \textbf{let } x = e_1 \textbf{ in } e_2
\end{array}$$

$$p \quad ::= \quad x \mid x \textbf{ as } p \mid c(p_1, \ldots, p_n)$$

$$A \quad ::= \quad x : \sigma, A \mid \epsilon \qquad \{ \text{ no x occurs twice}$$

A.2.2 Context sensitive syntax

$$(var) \quad \frac{x : \sigma \in A}{S[\tau_i/\alpha_i] \mid A \rhd x : \tau[\tau_i/\alpha_i]} \; \{ \sigma = \forall \alpha_i.\tau \textbf{ where } S$$

$$(\rightarrow i) \quad \frac{S \mid A, x : \tau_1 \rhd e : \tau_2}{S \mid A \rhd \lambda x.e : \tau_1 \rightarrow \tau_2}$$

$$(\rightarrow e) \quad \frac{S \mid A \rhd e_1 : \tau_1 \quad S \mid A \rhd e_2 : \tau_2 \quad S \vdash \tau_1 \leq \tau_3 \rightarrow \tau_4 \quad S \vdash \tau_2 \leq \tau_3}{S \mid A \rhd e_1 \; e_2 : \tau_4}$$

$$(con) \quad \frac{S \mid A \rhd e_1 : \tau_1 \quad \ldots \quad S \mid A \rhd e_n : \tau_n}{S \mid A \rhd c(e_1, \ldots, e_n) : c(\tau_1, \ldots, \tau_n)}$$

$$(cas) \quad \frac{\begin{array}{ll} S \mid A \rhd e : \tau & \\ S \mid A, A_{pi} \rhd p_i : \tau_i', e_i : \tau_i & S \vdash \tau \leq \cup_{i=1}^{n} \tau_i' \end{array}}{S \mid A \rhd \textbf{case } e \textbf{ of } p_i \Rightarrow e_i : \cup_{i=1}^{n} \tau_i ? \tau \cap \overline{\tau_i}'}$$

$$(ext) \quad \frac{\begin{array}{ll} S \mid A \rhd f : \tau'' & S \vdash \tau'' \leq \tau_1 \rightarrow \tau_2 \\ S \mid A, A_p \rhd p : \tau', \; e : \tau & S \vdash \tau_3 \leq \tau' \cup (\tau_1 \cap \neg \overline{\tau}') \end{array}}{S \mid A \rhd f \textbf{ extend } (p \Rightarrow e) : \tau_3 \rightarrow (\tau ? \tau_3 \cap \overline{\tau}' \cup \tau_2 ? \tau_3 \cap \neg \overline{\tau}')}$$

$$(asp)\quad \frac{S \mid A \rhd x : \tau_1,\ p : \tau_2}{S \mid A \rhd x\ \mathbf{as}\ p : \tau_1 \cap \tau_2}$$

$$(let)\quad \frac{S \mid A \rhd e_1 : \tau_1 \quad S' \mid A, x : \sigma \rhd e_2 : \tau_2}{S, S' \mid A \rhd \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2}\ \{\sigma = Gen(S, A, \tau_1)$$

Appendix B

The type inference algorithm

B.1 The constraint accumulation algorithm

$$(var)^z \quad \frac{(x : \forall \alpha_i . \tau \textbf{ where } S) \ \in A}{S[\beta_i/\alpha_i] \mid A \vdash^Z x : \tau[\beta_i/\alpha_i]} \ \{\beta_i \text{ new}$$

$$(\rightarrow i)^z \quad \frac{S \mid A, x : \alpha \vdash^Z e : \tau}{S \mid A \vdash^Z \lambda x.e : \alpha \rightarrow \tau} \ \{\alpha \text{ new}$$

$$(\rightarrow e)^z \quad \frac{S_1 \mid A \vdash^Z e_1 : \tau_1 \quad S_2 \mid A \vdash^Z e_2 : \tau_2}{S_1, S_2, \tau_1 \leq \alpha \rightarrow \beta, \tau_2 \leq \alpha \mid A \vdash^Z e_1 \ e_2 : \beta} \ \{\alpha, \beta \text{ new}$$

$$(con)^z \quad \frac{S_1 \mid A \vdash^Z e_1 : \tau_1 \quad \ldots \quad S_n \mid A \vdash^Z e_n : \tau_n}{S_1, \ldots, S_n \mid A \vdash^Z c(e_1, \ldots, e_n) : c(\tau_1, \ldots, \tau_n)}$$

$$(cas)^z \quad \frac{S \mid A \vdash^Z e : \tau \quad S_i \mid A, A_{p_i} \vdash^Z p_i : \tau_i', e_i : \tau_i}{\begin{array}{l} S, S_1, \ldots, S_n, \\ \tau \leq \cup_{i=1}^n \tau_i' \end{array} \mid A \vdash^Z \textbf{case } e \textbf{ of } p_i \Rightarrow e_i : \cup_{i=1}^n \tau_i ? \tau \cap \overline{\overline{\tau_i}}'}$$

$$(ext)^z \quad \frac{S'' \mid A \vdash^Z f : \tau'' \quad S \mid A, A_p \vdash^Z p : \tau', \ e : \tau}{\begin{array}{ll} S'', & \tau'' \leq \alpha \rightarrow \beta, \\ S, & \gamma \leq \tau' \cup (\alpha \cap \neg \overline{\overline{\tau}}') \end{array} \mid A \vdash^Z \begin{array}{l} f \textbf{ extend } (p \Rightarrow e) : \\ \gamma \rightarrow (\tau ? \gamma \cap \overline{\overline{\tau}}' \cup \beta ? \gamma \cap \neg \overline{\overline{\tau}}') \end{array}} \ \{\alpha, \beta, \gamma \text{ new}$$

$$(asp)^z \quad \frac{S_1 \mid A \vdash^Z x : \tau_1 \quad S_2 \mid A \vdash^Z p : \tau_2}{S_1, S_2 \mid A \vdash^Z x \textbf{ as } p : \tau_1 \cap \tau_2}$$

$$(let)^z \quad \frac{S_1 \mid A \vdash^Z e_1 : \tau_1 \quad S_2 \mid A, x : \sigma \vdash^Z e_2 : \tau_2}{S_1, S_2 \mid A \vdash^Z \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2}\ \{\sigma = Gen(S_1, A, \tau_1)$$

B.2 Constraint Simplification

$$\begin{array}{llcl}
(\leq 0l) & \Gamma \mid S, 0 \leq r & \leadsto & \Gamma \mid S \\
(\leq cc) & \Gamma \mid S, c(l_1, \ldots l_n) \leq c(r_1, \ldots r_n) & \leadsto & \Gamma \mid S, l_1 \leq r_1, \ldots, l_n \leq r_n \\
(\leq \rightarrow) & \Gamma \mid S, r_1 \rightarrow l_1 \leq l_2 \rightarrow r_2 & \leadsto & \Gamma \mid S, l_2 \leq r_1, l_1 \leq r_2 \\
(\leq \cup l) & \Gamma \mid S, l_1 \cup l_2 \leq r & \leadsto & \Gamma \mid S, l_1 \leq r, l_2 \leq r \\
(\leq \cap r) & \Gamma \mid S, l \leq r_1 \cap r_2 & \leadsto & \Gamma \mid S, l \leq r_1, l \leq r_2 \\
(\leq ?12) & \Gamma \mid S, l_1 ? l_2 \leq r & \leadsto & \Gamma \mid S, l_1 \leq r \mid S, l_2 \leq 0 \\
(\leq \cup r) & \Gamma \mid S, l \leq r_1 \cup r_2 & \leadsto & \Gamma \mid S, l \cap \neg\overline{r_1} \leq r_2, l \cap \neg\overline{r_2} \leq r_1 \\
(\leq ref) & \Gamma \mid S, \alpha \leq \alpha & \leadsto & \Gamma \mid S \\
(\leq \cap lb) & \Gamma \mid S, \alpha \cap \overline{l} \leq \alpha & \leadsto & \Gamma \mid S \\
(\leq \cap l) & \Gamma \mid S, \alpha \cap \overline{l} \leq r & \leadsto & \Gamma \mid S, \alpha \leq (r \cap \overline{l}) \cup \neg\overline{l}
\end{array}$$

B.3 Elimination Algorithm for $\neg\overline{(.)}$ and $\overline{(.)}$

$$
\begin{array}{ll}
\overline{\tau_1 \cap \tau_2} & \rightsquigarrow \overline{\tau_1} \cap \overline{\tau_2} \\
\overline{\tau_1 \cup \tau_2} & \rightsquigarrow (\overline{\tau_1} \cap \neg\overline{\tau_2}) \cup (\overline{\tau_1} \cap \overline{\tau_2}) \cup (\neg\overline{\tau_1} \cap \overline{\tau_2}) \\
\overline{\tau_1 \rightarrow \tau_2} & \rightsquigarrow 0 \rightarrow 1 \\
\overline{c(\tau_1, \dots, \tau_n)} & \rightsquigarrow c(\overline{\tau_1}, \dots, \overline{\tau_n}) \\
\overline{\alpha} & \rightsquigarrow 1 \\
\overline{0} & \rightsquigarrow 0
\end{array}
$$

$$
\begin{array}{ll}
\neg\overline{\tau_1 \cap \tau_2} & \rightsquigarrow (\overline{\tau_1} \cap \neg\overline{\tau_2}) \cup (\neg\overline{\tau_1} \cap \neg\overline{\tau_2}) \cup (\neg\overline{\tau_1} \cap \overline{\tau_2}) \\
\neg\overline{\tau_1 \cup \tau_2} & \rightsquigarrow \neg\overline{\tau_1} \cap \neg\overline{\tau_2} \\
\neg\overline{\tau_1 \rightarrow \tau_2} & \rightsquigarrow \cup_{c \in C} c(1, \dots, 1) \\
\neg\overline{c(\tau_1, \dots, \tau_n)} & \rightsquigarrow (0 \rightarrow 1) \cup \\
 & \rightsquigarrow \cup_{d \in C - \{c\}} d(1, \dots, 1) \\
 & \rightsquigarrow \cup c(\neg\overline{\tau_1}, 1, \dots, 1) \cup c(\overline{\tau_1}, \neg\overline{\tau_2}, \dots, 1) \cup c(\overline{\tau}_1, \dots, \overline{\tau}_{n-1}, \neg\overline{\tau}_n) \\
\neg\overline{\alpha} & \rightsquigarrow 0 \\
\neg\overline{0} & \rightsquigarrow 1
\end{array}
$$

B.4 Putting Types in Disjunctive Normal Form

$$
\begin{array}{lcl}
\tau \cap 0 & \rightsquigarrow & 0 \\
 & & \\
c(\tau_1, \dots, \tau_n) \cap c(\tau'_1, \dots, \tau'_n) & \rightsquigarrow & c(\tau_1 \cap \tau'_1, \dots, \tau_n \cap \tau'_n) \\
c(\tau_1, \dots, \tau_n) \cap d(\tau'_1, \dots, \tau'_m) & \rightsquigarrow & 0 \\
c(\tau_1, \dots, \tau_n) \cap \tau'_1 \rightarrow \tau'_2 & \rightsquigarrow & 0 \\
 & & \\
\tau_1 \rightarrow \tau_2 \cap 0 \rightarrow 1 & \rightsquigarrow & \tau_1 \rightarrow \tau_2 \\
 & & \\
\tau \cup \tau & \rightsquigarrow & \tau \\
\tau \cap \tau & \rightsquigarrow & \tau \\
(\tau_1 \cup \tau_1) \cap \tau_3 & \rightsquigarrow & (\tau_1 \cap \tau_3) \cup (\tau_2 \cap \tau_3) \\
(\alpha \cap \tau_1) \cap \tau_2 & \rightsquigarrow & \alpha \cap (\tau_1 \cap \tau_2)
\end{array}
$$

Appendix C

Domains

C.1 Complete Partial Orders

The following definitions are standard definitions from domain theory [46]. We include them here to get a self contained presentation.

Definition 3.1 Partial order

A partial order $\mathcal{U}$ is a pair $(U, \sqsubseteq)$ where U is a set and $(\sqsubseteq) \subseteq U \times U$ is a reflexive, transitive and antisymmetric relation. □

Definition 3.2 ω-Cpo

A partial order $\mathcal{U}$ is (countably) ω-complete iff every chain $a_1 \sqsubseteq \ldots \sqsubseteq a_n \sqsubseteq \ldots$, $n \in \omega$ has a least upper bound in $\mathcal{U}$. We denote it by $\sqcup_{i \in \omega} x_i$. □

Definition 3.3 ω-Pcpo

A chain complete partial order $\mathcal{U}$ is pointed iff it has a least element. In the sequel we denote this least element by $\perp$. □

Definition 3.4 Monotonic functions

Let $\mathcal{U} = (U, \sqsubseteq_U)$ and $\mathcal{V} = (V, \sqsubseteq_V)$ be two pcpo's. A function[1] $f \in V^U$ is *monotonic* iff

$$d \sqsubseteq_U d' \;\Rightarrow\; f(d) \sqsubseteq_V f(d')$$

□

[1] We write V^U for all functions from U to V.

Definition 3.5 ω**-Continuous functions:**

A monotonic function between ω-pcpo's $\mathcal{U}$ and $\mathcal{V}$ is continuous iff for every chain $a_1 \sqsubseteq \ldots \sqsubseteq a_n \sqsubseteq \ldots$ in $\mathcal{U}$:

$$f(\bigsqcup_{i\in\omega} a_i) = \bigsqcup_{i\in\omega} f(a_i)$$

□

C.2 Cpo Constructors

Let $\mathcal{U} = (U, \sqsubseteq_U)$ and $\mathcal{V} = (V, \sqsubseteq_V)$ be two pcpo's.

Function Pcpo

The Set

$U \xrightarrow{c} V$ is the set of all continuous functions from U to V.

The Ordering

$f \sqsubseteq_{U \xrightarrow{c} V} g \quad \text{iff} \quad \forall a \in U. f(a) \sqsubseteq_V g(a),$
$\bot_{U \xrightarrow{c} V} = \lambda x{:}U.\bot_V$

The Universal Property

External Axiomatization

1. $\forall f : (W \times U) \rightarrow V.\ f = \text{apply} \circ (\text{Curry}(f) \times \text{id}_U)$
2. $\forall g : W \rightarrow (U \rightarrow V).\ g = \text{Curry}(\text{apply} \circ (g \times \text{id}_U))$

Internal Axiomatization

1. $\beta - reduction:\quad (\lambda x.e)(e') = e[e'/x]$
2. $\eta - reduction:\quad \lambda x.e(x) = e \quad x \notin FV(e)$

The Functor

$f \rightarrow g : (U \rightarrow V) \rightarrow (U' \rightarrow V')$
$f \rightarrow g = \lambda h : U \rightarrow V.\ g \circ h \circ f$

1. $\text{id}_U \rightarrow \text{id}_V = \text{id}_{U \rightarrow V}$
2. $(f \circ g) \rightarrow (g' \circ f') = (g \rightarrow g') \circ (f \rightarrow f')$

$$\begin{tikzcd}
(U \to V) \times U \arrow[d, "\text{apply}"'] & & W \times U \arrow[ll, "g \times id_U"'] \arrow[dll, "f"] \\
V & &
\end{tikzcd}
\qquad
\begin{tikzcd}
U \arrow[r, "h"] & V \arrow[d, "g"] \\
U' \arrow[u, "f"] \arrow[r, "(f \to g)(h)"'] & V'
\end{tikzcd}$$

Figure C.1: The Universal Property and The Functor

Product Pcpo

The Set

$U \times V = \{(u, v) \mid u \in U \wedge v \in V\}$

The ordering

$(d, e) \sqsubseteq_{U \times V} (d', e')$ iff $(d \sqsubseteq_U d') \wedge (e \sqsubseteq_V e')$
$\perp_{U \times V} = (\perp_U, \perp_V)$

The Universal Property

External Axiomatization

1. $\forall f_1 : V \to U_1, f_2 : V \to U_2, i \leq 2.\ \pi_i \circ (f_1, f_2) = f_i$
2. $\forall h : V \to U_1 \times U_2.\ h = (\pi_1 \circ h, \pi_2 \circ h)$

Internal Axiomatization

1. $\forall x_1 : U_1, x_2 : U_2.\ \pi_i(x_1, x_2) = x_i$
2. $\forall x : U_1 \times U_2.\ x = (\pi_1(x), \pi_2(x))$

The Functor

$f_1 \times f_2 : U_1 \times U_2 \to U_1' \times U_2'$
$f_1 \times f_2 = (f_1 \circ \pi_1, f_2 \circ \pi_2)$

1. $id_{U_1} \times id_{U_2} = id_{U_1 \times U_2}$
2. $(g_1 \circ f_1) \times (g_2 \circ f_2) = (g_1 \times g_2) \circ (f_1 \times f_2)$

$$\begin{tikzcd}
 & V \arrow[dl, "f_1"'] \arrow[d, dashed, "(f_1, f_2)"] \arrow[dr, "f_2"] & \\
U_1 & U_1 \times U_2 \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & U_2
\end{tikzcd}
\qquad
\begin{tikzcd}
U_1 \arrow[d, "f_1"'] & U_1 \times U_2 \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] \arrow[d, "f_1 \times f_2"] & U_2 \arrow[d, "f_2"] \\
U_1' & U_1' \times U_2' \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & U_2'
\end{tikzcd}$$

Figure C.2: The Universal Property and The Functor

This definitions may be generalized to n-ary products in a straight forward way.

Coalesced Sum Pcpo

The Set

$$U \oplus V = \{(0,u) \mid u \in U, u \neq \bot_U\} \cup \{(1,v) \mid v \in V, v \neq \bot_V\} \cup \{\bot\}$$

The Ordering

$$x \sqsubseteq_{U \oplus V} y \quad \text{iff} \begin{cases} x = \bot & \vee \\ x = (0,x') \wedge y = (0,y') \wedge x' \sqsubseteq_U y' & \vee \\ x = (1,x') \wedge y = (1,y') \wedge x' \sqsubseteq_V y' & \end{cases}$$

The Universal Property

External Axiomatization

1. $\forall f_1 : U_1 \circ\!\!\rightarrow V, f_2 : U_2 \circ\!\!\rightarrow V, i \leq 2.\ [f_1, f_2] \circ in_i = f_i$
2. $\forall h : U_1 \oplus U_2 \circ\!\!\rightarrow V.\ h = [h \circ in_1, h \circ in_2]$

Internal Axiomatization

$out_i : U_1 \oplus U_2 \circ\!\!\rightarrow V$
$out_1 = [\lambda_\bot x : U_1.x, \bot], \quad out_2 = [\bot, \lambda_\bot x : U_2.x]$

$is_i : U_1 \oplus U_2 \circ\!\!\rightarrow Bool$
$is_1 = [\lambda_\bot x_1.tt, \lambda_\bot x_2.ff], \quad is_2 = [\lambda_\bot x_1.ff, \lambda_\bot x_2.tt]$

1. $\forall x : U_i.\ x = out_i(in_i(x))$
2. $\forall x : U_1 \oplus U_2.\ x = is_1(x) \rightarrow in_1(out_1(x)) \mid is_2(x) \rightarrow in_2(out_2(x)) \mid \bot$

The Functor

$f_1 \oplus f_2 : U_1 \oplus U_2 \rightarrow (U_1' \oplus U_2')$
$f_1 \oplus f_2 = [in_1 \circ f_1, in_2 \circ f_2]$

1. $id_{U_1} \oplus id_{U_2} = id_{U_1 \oplus U_2}$
2. $(g_1 \circ f_1) \oplus (g_2 \circ f_2) = (g_1 \oplus g_2) \circ (f_1 \oplus f_2)$

$$\begin{tikzcd}
U_1 \arrow[r, "in_1"] \arrow[dr, "f_1"'] & U_1 \oplus U_2 \arrow[d, dashed, "{[f_1, f_2]}"] & U_2 \arrow[l, "in_2"'] \arrow[dl, "f_2"] \\
& V &
\end{tikzcd}
\qquad
\begin{tikzcd}
U_1 \arrow[r, "in_1"] \arrow[d, "f_1"'] & U_1 \oplus U_2 \arrow[d, "f_1 \oplus f_2"] & U_2 \arrow[l, "in_2"'] \arrow[d, "f_2"] \\
U_1' \arrow[r, "in_1"'] & U_1' \oplus U_2' & U_2' \arrow[l, "in_2"]
\end{tikzcd}$$

Figure C.3: The Universal Property and The Functor

This definitions may be generalized to n-ary sums in a straight forward way.

Lift Pcpo

The Set

$U_\perp = U \times \{0\} \ \cup \ \{\perp_{U_\perp}\}$ where $\perp_{U_\perp}$ is not a pair.

The Ordering

$(x, 0) \sqsubseteq_{U_\perp} (y, 0)$ iff $x \sqsubseteq_U y$
$\forall z \in U_\perp . \perp_{U_\perp} \sqsubseteq_{U_\perp} z$

The Universal Property

External Axiomatization

1. $\forall f : U \to V.\ f = \text{lift}(f) \circ \text{up}$
2. $\forall h : U_\perp \circ\!\!\to V.\ h = \text{lift}(h \circ \text{up})$

Internal Axiomatization

down : $U_\perp \circ\!\!\to U$
down = lift(id_U)

$\delta : U_\perp \circ\!\!\to \mathit{Bool}$
$\delta = \text{lift}(\lambda x : U.tt)$

1. $\forall x : U.\ x = \text{down}(\text{up}(x))$
2. $\forall y : U_\perp.\ y = \delta(x) \to \text{up}(\text{down}(y)) \mid\perp$

The Functor

$(.)_\perp : U \to U_\perp$
$f_\perp = \text{lift}(\text{up} \circ f)$

1. $(id_U)_\perp = id_{U_\perp}$
2. $(g \circ f)_\perp = g_\perp \circ f_\perp$

$$\begin{tikzcd}
U \arrow[r, "\text{up}"] \arrow[dr, "f"'] & U_\perp \arrow[d, dashed, "\text{lift}(f)"] \\
& V
\end{tikzcd}
\qquad
\begin{tikzcd}
U \arrow[r, "\text{up}"] \arrow[d, "f"'] & U_\perp \arrow[d, "f_\perp"] \\
V \arrow[r, "\text{up}"'] & V_\perp
\end{tikzcd}$$

Figure C.4: The Universal Property and The Functor

C.3 Bounded Complete Domains

Definition 3.6 ω-Finite

An element of a cpo is ω-finite iff whenever it is less than the limit $\sqcup_{n\in\omega} x_n$ of a chain $x_1 \sqsubseteq x_2 \ldots$ it is less than some element of the chain. The ω-finite elements in any subset X of a cpo are denoted by X^o. □

Definition 3.7 **ω-Algebraic**

A cpo is ω-algebraic iff

- it has countable many ω-finite elements,
- for every element x, there is an increasing chain $(x_n)_{n\geq 0}$ of finite elements such that $x = \sqcup_{n\geq 0} x_n$.

□

The problem with ω-algebraic cpo's is that given algebraic cpo's U and V the function space $U \to V$ is not algebraic. However, an additional condition removes this problem.

Definition 3.8 **Bounded Complete cpo**

A cpo U is bounded or consistently complete if every bounded subset of U has a least upper bound. □

Definition 3.9 **Bc-domain**

A bounded complete domain is a bounded complete algebraic cpo. □

All countable flat cpo's are bc–domains and all the constructions send bc–domains to bc–domains.

Proposition 3.1 **Finite elements**

$$(U \times V)^o = U^o \times V^o \quad (U \oplus V)^o = U^o \oplus V^o \quad (U_\perp)^o = (U^o)_\perp$$ □

Definition 3.10 **Continuous step function**

For the function space $U \to V$ the continuous *step function* $(a \Rightarrow b)$ for any $a \in U^o,\ b \in V^o$ is defined by:

$$(a \Rightarrow b)(x) = \begin{cases} b & \text{if } a \sqsubseteq x \\ \perp & \text{otherwise} \end{cases}$$ □

Definition 3.11 **Finite functions**

Finite functions are the finite lubs of step functions.

$$f = (a_1 \Rightarrow b_1) \sqcup \ldots \sqcup (a_n \Rightarrow b_n)$$
$$f(x) = \sqcup\{b_i \mid a_i \sqsubseteq x\}$$

The lub $(a_1 \Rightarrow b_1) \sqcup \ldots \sqcup (a_n \Rightarrow b_n)$ exists if and only if whenever $\{a_{i_1}, \ldots, a_{i_k}\}$ is a subset of $\{a_i, \ldots, a_n\}$ with an upper bound, then $\{b_{i_1}, \ldots, b_{i_k}\}$ has an upper bound too. □

Appendix D

Network Calculus Proofs

D.1 Internal Communication (6)

"⇒"

$$\dfrac{\exists x.x = a^\frown Q[x/y] \wedge N[x/y] \qquad \dfrac{x = a^\frown Q[x/y] \wedge N[x/y]}{\exists x.x = Q[a^\frown x/y] \wedge N[a^\frown x/y]}\,(A)}{\exists x.x = Q[a^\frown x/y] \wedge N[a^\frown x/y]}\,(\exists e)$$

The proof A is given below:

$$\dfrac{\dfrac{\exists z.z = Q[x/y] \quad x = a^\frown Q[x/y] \wedge N[x/y]}{\exists z.z = Q[x/y] \wedge x = a^\frown Q[x/y] \wedge N[x/y]}\,(\wedge i,\exists\wedge) \qquad \dfrac{\dfrac{\dfrac{z = Q[x/y] \wedge x = a^\frown Q[x/y] \wedge N[x/y]}{z = Q[x/y] \wedge x = a^\frown z \wedge N[x/y]}\,(\wedge ei,subst)}{z = Q[a^\frown z/y] \wedge N[a^\frown z/y]}\,(\wedge e,subst)}{\exists x.x = Q[a^\frown x/y] \wedge N[a^\frown x/y]}\,(\exists i,\alpha)}{\exists x.x = Q[a^\frown x/y] \wedge N[a^\frown x/y]}\,(\exists e)$$

"⇐"

$$\dfrac{\exists x.x = Q[a^\frown x/y] \wedge N[a^\frown x/y] \qquad \dfrac{x = Q[a^\frown x/y] \wedge N[a^\frown x/y]}{\exists x.x = a^\frown Q[x/y] \wedge N[x/y]}\,(B)}{\exists x.x = a^\frown Q[x/y] \wedge N[x/y]}\,(\exists e)$$

The proof B is given below:

$$\dfrac{\dfrac{\exists z.z = a{}^\frown x \quad x = Q[a{}^\frown x/y] \wedge N[a{}^\frown x/y]}{\exists z.z = a{}^\frown x \wedge x = Q[a{}^\frown x/y] \wedge N[a{}^\frown x/y]}\,{\scriptstyle(\wedge i,\exists\wedge)} \qquad \dfrac{\dfrac{\dfrac{z = a{}^\frown x \wedge x = Q[a{}^\frown x/y] \wedge N[a{}^\frown x/y]}{z = a{}^\frown x \wedge x = Q[z/y] \wedge N[z/y]}\,{\scriptstyle(\wedge ei,subst)}}{z = a{}^\frown Q[z/y] \wedge N[z/y]}\,{\scriptstyle(\wedge e,subst)}}{\exists x.x = a{}^\frown Q[x/y] \wedge N[x/y]}\,{\scriptstyle(\exists i,\alpha)}}{\exists x.x = a{}^\frown Q[x/y] \wedge N[x/y]}\,{\scriptstyle(\exists e)}$$

D.2 Input Neutral (5.1)

“⇒”

$$\dfrac{\dfrac{\exists u.u = x \quad N[x/y]}{\exists u.u = x \wedge N[x/y]}\,{\scriptstyle(\wedge i,\exists\wedge)} \qquad \dfrac{\dfrac{u = x \wedge N[x/y]}{u = x \wedge N[u/y]}\,{\scriptstyle(\wedge ei,subst)}}{\exists u.u = x \wedge N[u/y]}\,{\scriptstyle(\exists i)}}{\exists u.u = x \wedge N[u/y]}\,{\scriptstyle(\exists e)}$$

“⇐”

$$\dfrac{\exists u.u = x \wedge N[u/y] \qquad \dfrac{u = x \wedge N[u/y]}{N[x/y]}\,{\scriptstyle(subst)}}{N[x/y]}\,{\scriptstyle(\exists e)}$$

Bibliography

[1] M. Abadi and L. Cardelli. A theory of primitive objects: untyped and first-order systems. In *Proc. Theor. Aspects of Computer Software*, Sendai, Japan, 1994.

[2] N. Adams and J. Rees. Object-oriented programming in Scheme. In *Proc. ACM Symp. Lisp and Functional Programming Languages*, pages 277–288, July 1988.

[3] G. Agha and C. Hewitt. Concurrent programming using actors. In A. Yonezawa and M. Tokoro, editors, *Object Oriented Concurrent Programming.* MIT Press, 1988.

[4] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. Towards a theory of actor computation. In *The Third International Conference on Concurrency Theory (CONCUR '92)*, volume 630 of *Springer LNCS*, pages 565–579. Springer Verlag, aug 1992.

[5] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Conf. on Functional Programming and Computer Architecture*, 1993.

[6] A. Aiken, L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proc. ACM Symp. on Principles of Programming Languages*, 1994.

[7] R.M. Amadio. Recursion over realizability structures. *Information and Computation*, 91(1):55–86, 1991.

[8] R.M. Amadio and L. Cardelli. Subtyping recursive types. In *Proc. ACM Symp. Principles of Programming Languages*, pages 104–118, 1991.

[9] P. America. Issues in the design of a parallel object oiented language. *Formal Aspects of Computing*, 1(4):366–411, October 1989.

[10] P. America. A parallel object oriented language with inheritance and subtyping. In *OOPSLA/ECOOP '90*, pages 161–168, 1990.

[11] P. America. Pool - design and experience. *OOPS Mesenger*, 2(2), 1991.

[12] G. Berry and G. Boudol. The chemical abstract machine. In *Proc. ACM Symp. Principles of Programming Languages*, 90.

[13] G. Booch. *Object Oriented Design.* The Benjamin/Cummings Publishing Company, 1991.

[14] G. Booch. *Object Oriented Analysis and Design with Applications.* The Benjamin/Cummings Publication Company, 1994.

[15] V. Breazu-Tannen, T. Coquand, C.A. Gunter, and A. Scedrov. Inheritance as explicit coercion. *Information and Computation*, 93(1):172–221, 1991.

[16] Encyclopaedia Britannica. *Articles on "Behaviour, Animal", "Classification Theory" and "Mood".* Encyclopaedia Britannica, Inc., 1986.

[17] M. Broy. Compositional Refinement of Interactive Systems. Technical Report 89, DEC Systems Research Center, 130 Lytton Avenue, Paolo Alto, California 94301, July 1992.

[18] K. Bruce. Safe type checking in a statically-typed object-oriented programming language. In *Proc 20th ACM Symp. Principles of Programming Languages*, pages 285–298, 1993.

[19] K. Bruce and G. Longo. A modest model of records, inheritance and bounded quantification. *Information and Computation*, 87(1/2):196–240, 1990.

[20] K. Bruce and J.C. Mitchell. PER models of subtyping, recursive types and higher-order polymorphism. In *Proc. 19th ACM Symp. on Principles of Programming Languages*, pages 316–327, January 1992.

[21] P. Canning, W. Cook, W. Hill, J. Mitchell, and W. Olthoff. F-bounded quantification for object-oriented programming. In *Functional Prog. and Computer Architecture*, pages 273–280, 1989.

[22] L. Cardelli. A semantics of multiple inheritance. In *Proc. Int. Symp. on Semantics of Data Types, Sophia-Antipolis (France)*, pages 51–68, Berlin, June 1984. Springer LNCS 173.

[23] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[24] L. Cardelli. Extensible records in a pure calculus of subtyping. Technical Report 81, DEC Systems Research Center, 1992.

[25] L. Cardelli and J.C. Mitchell. Operations on records. *Math. Structures in Computer Science*, 1(1):3–48, 1991. Summary in *Math. Foundations of Prog. Lang. Semantics,* Springer LNCS 442, 1990, pp 22–52.

[26] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[27] F. Cardone. Relational semantics for recursive types and bounded quantification. In *ICALP*, pages 164–178, Berlin, 1989. Springer LNCS 372.

[28] Castagna, Ghelli, and Longo. A calculus for overloaded functions with subtyping. In *1992 ACM Conf. Lisp and Functional Programming*, pages 182–192, 1992.

[29] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *SIGPLAN '89 Conf. on Programming Language Design and Implementation*, pages 146–160, 1989.

[30] P. Chen. The entity-relationship model – toward a unified view of data. *ACM Trans. on Database Systems*, 1(1):9–36, March 1976.

[31] P. Coad and E. Yourdan. *Object–Oriented Design.* Prentice Hall Internat., 1992.

[32] P. Coad and E. Yourdan. *Object Oriented Analysys.* Prentice Hall Internat., 1992.

[33] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremes. *Object-Oriented Development: The Fusion Method.* Prentice-Hall International, Inc., 1994.

[34] W. Cook. Object oriented programming versus abstract data types. In *Proc. on Foundations of Object Oriented Programming*, volume 489 of *Springer LNCS*, 1991.

[35] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Proc. 17th ACM Symp. on Principles of Programming Languages*, pages 125–135, January 1990.

[36] W.R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. In *ACM Conf. Object-oriented Programming: Systems, Languages and Applications*, pages 1–15, 1992.

[37] B. J. Cox. *Object Oriented Programming.* Addison Wesley, 1991.

[38] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[39] Webster's New Twentieth Century Dictionary. Collins World, 1977.

[40] Ed Downs, Peter Clare, and Ian Coe. *Structured Systems Analysis and Design Method — Application and Context.* Prentice Hall, 1992.

[41] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Symposium on Principles of Programming Languages*, pages 52–66, 1984.

[42] J.-Y. Girard. The system F of variable types, fifteen years later. *Theor. Comp. Sci.*, 45(2):159–192, 1986.

[43] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison Wesley, 1983.

[44] R. Grosu and D. Nazareth. Towards a new way of parameterization. In *MC-SEAI*, pages 383–393, April 1994.

[45] R. Grosu and F. Regensburger. The Logical Framework of SPECTRUM. Technical Report TUM-I9402, Institut für Informatik, Technische-Universität München, 1994.

[46] C.A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* MIT Press, Cambridge, MA, 1992.

[47] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Proc. 18th ACM Symp. Principles of Programming Languages*, pages 131–142, January 1991. Extended version available as Carnegie Mellon Technical Report CMU-CS-90-157.

[48] I. Jacobson. *Object Oriented Software Engineering: A Use Case Driven Approach.* Addison Wesley, 1992.

[49] L. Jategaonkar and J.C. Mitchell. ML with extended pattern matching and subtypes. In *Proc. ACM Symp. Lisp and Functional Programming Languages*, pages 198–212, July 1988.

[50] C. B. Jones. An object–based design method for concurrent programs. Technical Report UMCS-92-12-1, University of Manchester, December 1992.

[51] C. B. Jones. Process-algebraic foundations for an object-based design notation. Technical Report UMCS-93-10-1, Manchester University, October 1993.

[52] M. Jones. Coherence for qualified types. Technical Report YALEU/DCS/RR-989, Yale University, September 1993.

[53] G. Kahn. The semantics of a simple language for parallel programming. *Information Processing*, 74:471–475, 1974.

[54] S. E. Keene. *Object Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley Publishing Co., 1989.

[55] D. Leivant. Typing and computational properties of λ-expressions. *Theoretical Computer Science*, 44:51–68, 1986.

[56] B. Liskov et al. *CLU Reference Manual*. Springer LNCS 114, Berlin, 1981.

[57] D. MacQueen, G Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1/2):95–130, 1986.

[58] J. Martin and J. Odell. *Object Oriented Analysis and Design*. Prentice Hall, Englewood Cliffs, Nj, 1992.

[59] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. Technical Report SRI-CSL-91-05, SRI, 1991.

[60] J. Meseguer. A logical theory of concurrent objects. In *ECOOP-OOPSLA*, pages 101–115, 90.

[61] B. Meyer. *Object Oriented Software Construction*. Prentice Hall, 1988.

[62] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[63] R. Milner. A theory of type polymorphism in programming. *JCSS*, 17:348–375, 1978.

[64] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[65] R. Milner. The polyadic π-calculus: A tutorial. Technical Report ECS-LFCS-91-180, University of Edinburgh, 91.

[66] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I. *Information and Computation*, 100(1):1–40, 1992.

[67] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part II. *Information and Computation*, 100(1):41–77, 1992.

[68] J. C. Mitchell. Foundations for object oriented programming. TOOLS Tutorial, July 1992.

[69] J.C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proc. 17th ACM Symp. on Principles of Programming Languages*, pages 109–124, January 1990.

[70] J.C. Mitchell. *Introduction to Programming Language Theory*. MIT Press, to be published 1994.

[71] J.C. Mitchell and R. Harper. The essence of ML. In *Proc. 15th ACM Symp. on Principles of Programming Languages*, pages 28–46, January 1988.

[72] J.C. Mitchell, F. Honsell, and K. Fisher. A lambda calculus of objects and method specialization. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 26–38, 1993.

[73] J.C. Mitchell and G.D. Plotkin. Abstract types have existential types. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988. Preliminary version appeared in *Proc. 12th ACM Symp. on Principles of Programming Languages,* 1985.

[74] O. Nierstrasz. Towards an object calculus. In *Proc. Object Based Concurrent Computing*, volume 612 of *Springer LNCS*, 1992.

[75] O. Niestrasz and M. Papathomas. Viewing objects as patterns of communicating agents. In *OOPSLA*, 1990.

[76] Tobias Nipkow and Christian Prehofer. Type checking type classes. In *Proc. 20th ACM Symp. Principles of Programming Languages*, pages 409–418, 1993.

[77] B. Pierce and D.N. Turner. Object-oriented programming without recursive types. In *Proc 20th ACM Symp. Principles of Programming Languages*, pages 299–312, 1993.

[78] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, December 1991.

[79] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 1993. To appear; a preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title “Object-Oriented Programming Without Recursive Types”.

[80] U.S. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Proc. ACM Symp. Lisp and Functional Programming Languages*, pages 289–297, July 1988.

[81] D. Rémy. Typechecking records and variants in a natural extension of ML. In *16th ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.

[82] J. C. Reynolds. Automatic computation of data set definitions. *Information Processing*, 68:456–461, 1969.

[83] J. C. Reynolds. The coherence of languages with intersection types. In *Theoretical Aspects of Computer Software*, volume 526 of *Springer LNCS*, 1991.

[84] J.C. Reynolds. Towards a theory of type structure. In *Paris Colloq. on Programming*, pages 408–425, Berlin, 1974. Springer-Verlag LNCS 19.

[85] J.C. Reynolds. Types, abstraction, and parametric polymorphism. In *Information Processing '83*, pages 513–523. North-Holland, Amsterdam, 1983.

[86] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object–Oriented Modeling and Design.* Prentice Hall, 1991.

[87] G.L. Steele and G.J. Sussman. Scheme: an interpreter for the extended lambda calculus. Technical Report 349, MIT Artificial Intelligence Laboratory, 1975.

[88] B. Stroustrop. *The C^{++} Programming Language.* Addison-Wesley, 1986.

[89] B. Thomsen. A calculus of higher order communicating systems. In *Proc. ACM Symp. Principles of Programming Languages*, 1989.

[90] P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *16th ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.

[91] M. Wand. Complete type inference for simple objects. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 37–44, 1987. Corrigendum in *Proc. IEEE Symp. on Logic in Computer Science,* page 132, 1988.

[92] M. Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, 10:115–122, 1987.

[93] M. Wand. Type inference for record concatenation and multiple inheritance. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 92–97, 1989. To appear in *Information and Computation.*

[94] A. Yonezawa and M. Tokoro. *Object Oriented Concurrent Programming.* MIT Press, 1987.

[95] E. Yourdon. *Modern Structured Analysis.* Englewood Cliffs, Yourdon Press, New Jersey, 1989.

[96] E. Yourdon and L. Constantine. *Structured Design.* Englewood Cliffs, New Jersey: Yourdon Press, 1979.

[97] S. N. Zilles. Procedural encapsulation: A linguistic protection mechanism. *SIGPLAN Notices*, 8(9):142–146, 1973.