Dialgebraic Semantics of Typed Object Calculi

JOHAN GLIMMING

Licentiate Thesis
Stockholm, Sweden, 2005

TRITA-NA-0511
ISSN 0348-2952
ISRN KTH/NA/R--05/11--SE
ISBN 91-7178-031-9

Stockholms Universitet / KTH
Numerisk analys och datalogi
SE-100 44 Stockholm
SWEDEN

Akademisk avhandling som med tillstånd av Stockholms Universitet framlägges till offentlig granskning för avläggande av licentiatavhandling 19 september, 2005 i Sal D3, Kungl Tekniska högskolan, Lindstedsvägen 5 (entré vid Stora Valvet), Stockholm.

Tryck: Universitetsservice US AB

Corrections added 19 September, 2005.

Abstract

Algebraic data type theory has a notion of structural recursion. Coalgebraic data types similarly have a notion of structural corecursion. In this thesis we study a third form of recursion: direcursion. The other two notions have been used in program derivations, correctness proofs, and in foundations of functional and class-based languages. Direcursion, on the other hand, has not been extensively studied in the context of programming languages, and not at all in the context of algebraic techniques for object-oriented programming languages or typed object calculi. Yet, every object in object calculi is equipped with this recursion principle, and we will demonstrate that this principle can be used in foundations and in programming (as a powerful and general way of computing with objects), and when reasoning with object calculi programs, e.g. in correctness proofs.

The family of *object calculi* developed by Abadi and Cardelli [3] is one of several proposed foundations for object-oriented programming languages. It is one of the more general frameworks available, and arguably the most general framework which fully supports subtyping. The study of direcursion involves dealing with several aspects of object calculus, with contributions ranging from giving an operational (natural) semantics of a typed object calculus and interpreting this semantics into fixed point calculus while proving soundness and adequacy results to directly constructing a denotational semantics of typed object calculi. As a result, this thesis lays a foundation for algebraic programming techniques and laws for typed object calculi based on direcursion.

Contents

Contents **iv**

1 Introduction **3**
1.1 Motivation 4
1.2 Contributions 5
1.3 Overview 5

2 Background **7**
2.1 Category Theory 7
2.2 Theory of Data Types 14
2.3 Semantics of Programming Languages 22
2.4 Object Calculi and other Foundations 28

3 Soundness and Adequacy of System S^- **39**
3.1 First-Order Object Calculus without Subtyping 39
3.2 Fixed Point Calculus 46
3.3 Translating Object Calculus in FPC 48
3.4 Soundness and Adequacy 51

4 Direcursion **55**
4.1 Denotational Semantics 55
4.2 Wrappers 58

5 Conclusions and Further Work **65**

Bibliography **67**

Acknowledgments

I would like to thank my supervisor Professor Karl Meinke for taking me on as his PhD student at Stockholm University, and for having continuously supported and guided me during my postgraduate studies. Professor Meinke also monitored my graduate education. He has trusted me to take over his own graduate course at Stockholm University.

I would also like to express my appreciation to my coauthor, Dr. Neil Ghani. Dr. Ghani has been an excellent colleague as well as a capable teacher in category theory. We share an enthusiasm for the field and cooperate well. The present thesis is in many ways a result of our joint efforts.

The support from Professor Faron Moller and Professor Björn Lisper, goes all the way back to my time as an undergraduate student at Uppsala University. My postgraduate studies began with a joint paper with Professor Moller, and his support has been constant and has also led me to Professor Lisper and in recent years to Professor Meinke.

I would also like to express my gratitude to Professor Roland Backhouse at University of Nottingham for allowing me to visit his research group during an extensive period of time while working on this thesis. Likewise, I want to thank Professor Richard S. Bird for taking me on as his graduate student at Maths Institute, University of Oxford.

On the personal side, I am deeply grateful to my parents Eva and Kent, and to my sister Ida. There are not words to express their importance to me. Last but not least, my much loved girlfriend Pia should be thanked for putting up with me when my mind has been elsewhere and my working hours long.

Chapter 1

Introduction

Algebraic data type theory has a notion of structural recursion. Coalgebraic data types similarly have a notion of structural corecursion. In this thesis we study a more general notion of recursion that occurs in object calculus: direcursion. This notion of direcursion has not been extensively studied in the context of object-oriented programming languages, and in particular it has not been investigated as a foundation for formal methods for object-oriented programs with laws and equational reasoning. Yet, every object in object calculus is equipped with this recursion principle, and we will develop a denotational model for typed object calculi such that direcursion becomes a universal property for each object type. The overall aim is to provide a foundation for a logic that enables equational reasoning with object calculi programs, e.g. in correctness proofs, by means of direcursion.

Our investigation of direcursion is driven by an ambition to establish an "algebra of objects", i.e. a methodology and formal method for construction of object-oriented programs much like the so called Bird-Meertens formalism [10, 5, 55, 7, 8]. In this work we therefore propose a foundation for this new line of work, and in particular we have found a promising model of objects as categorical data types, which we hope can serve as a basis for expressing desirable recursion schemes as special-cases of the very general and powerful notion of direcursion. The combination of object-oriented programming and a theory of object types of this form seems to have a role to play, particularly considering the interest in expressing recurring idioms such as design patterns in the object-oriented community. Our work has the potential of providing a vehicle for such design patterns, allowing them to be formally expressed, and facilitating formal reasoning with them in actual programs.

The family of *object calculi* developed by Abadi and Cardelli [3] is one of several proposed foundations for object-oriented programming languages. It is one of the more general frameworks available, and arguably the most general that fully supports subtyping. While exploring direcursion we will cover several aspects of object calculus, ranging from giving an operational semantics of a typed object calculi, interpreting this semantics into fixed point calculus while proving soundness results, to directly constructing a denotational semantics of object calculi. As a result, this thesis lays a foundation for algebraic programming techniques and laws for typed object calculi based on direcursion, and further work

is needed to instantiate this powerful recursion principle for actual programs. To this end, we give in this thesis an example based on an object that represents natural numbers. The simplicity of the construction suggests that further object-oriented idioms can be treated in a similar style.

1.1 Motivation

In this section we will further position the research presented in this thesis, and give a more elaborate motivation for why this research is important.

Firstly, programming languages need a semantical definition in order to eliminate insecurities and unsound features. This can be witnessed by for example the Simula type system, Eiffel, and Smalltalk, all of which had significant type insecurities [18, 61, 3]. These insecurities caused ”message not understood”-errors while a well-typed object-oriented program was executing, which is exactly what a typing discipline tried to prevent (in other words these languages failed to satisfy a type soundness result). On the other hand, a strong typing discipline renders some programs invalid, specifically those programs which are not well-typed. In order to allow more programs to be written, typically programs which are more general (e.g. works on more inputs, accepts more subtypes, etc), one wants to extend the type system. This is the well known conflict between strong typing and flexibility. An example of features that increase the flexibility in this sense are parametric polymorphism and subtype polymorphism. It should be mentioned that one trend in programming is to write more general programs. For example, we would like to write down textbook design patterns in a programming language, such that they can be used and instantiated in as many contexts as possible. Extensions to the typing discipline with various forms of polymorphism, and extensions of the programming constructs themselves, can help to increase the expressiveness to make it possible to express such programs. A third and central motivation for this research is the correctness of programs with respect to a specification. Even when a programming language has a sound basis (semantics) it is possible to write incorrect programs. A correct program is one that satisfies the specification of the problem it was written to solve.

In this thesis we study foundational (semantical) aspects of a recursion principles in the context of object-oriented programming languages. The recursion principle requires extensions to the typing discipline and to language constructs. We have taken object calculi as our core operational language, but much of our work will be denotational. In fact we have left as further work to refine the operational semantics to reflect the extensions we develop mathematically in the denotational semantics. Therefore this research is centered around an extension of object-oriented programs that increases flexibility while preserving type soundness. This extension aims at capturing recurring programming idioms in a more succinct and general way. The extension is also tailored towards supporting correctness calculi, i.e. formal derivation of correct programs. The reason for this is that when objects are defined using the computation principle herein, various laws for optimisations are at the programmer’s disposal.

1.2 Contributions

In this section we will describe the contributions of this thesis, and briefly explain how our work is related to other related work.

Our first contribution is to link work on generic data type theory to object calculi by means of higher-order data types and self-application. Although the self-application interpretation is well-known it seems that we are the first ones to put it in this context. The closest previous work is the work of Reus and Streicher [68], but this work is based on untyped object calculus, does not give the link to direcursion and fusion laws, and has different scientific aims. Given this interpretation we show that the generic programming discipline (a.k.a. Bird-Meertens formalism and polytypic programming) applies to the object calculi setting by spelling out a recursion principle on the associated object data types. The recursion principle itself is due to Freyd, and was introduced in a functional programming setting by Hutton and Meijer. We develop wrappers for algebraic and coalgebraic data types which translate ordinary (first-order) data types into object types. Since we are working in a self-application interpretation of object calculi, we investigate the denotations of object calculi types and terms in the category pCpo and discuss why this category is needed. The direct interpretation of object calculi into pCpo is novel. We further prove soundness and adequacy for a translation of a typed object calculus without subtyping into Fiore's fixed point calculus, in order to give a solid foundation for the self-application interpretation. These results are again original, and firmly link object calculi to fixed point calculus.

Most of the work in chapter three and four of this thesis was carried out as a collaboration project between Dr. Neil Ghani of University of Leicester and myself. My own contributions in these chapters includes a substantial part of the proofs, examples, and overall content of chapter 3. The fourth chapter is based on the paper "Difunctorial Semantics for Object Calculi" where my idea of using recursive types to model objects is investigated. Dr. Ghani and I decided to use the category pCpo, and we then developed the categorical interpretation jointly. The section on wrappers arose as a joint result while I visited Dr. Ghani in 2004.

1.3 Overview

The content of this thesis is based on research results, most of which have been published or have been submitted for publication. The second chapter gives background in category theory, semantics, and in object-oriented programming (particularly object calculi). It further reviews the basic categorical theory of data types, including basic recursion schemes such as catamorphism, and the type functors for polymorphic data types. This chapter is purely a survey and there are no new results reported in it.

The third chapter on soundness and adequacy is based on the paper "Soundness and Adequacy of Object Calculi" where we consider the self-application encoding into FPC. While this encoding is well-known, we provide the first adequacy and soundness proofs of

typed object calculus with regard to this model, and we therefore link typed object calculus to a plethora of semantic models, e.g. pCpo models.

Next, we study the pCpo interpretation in its own right, and pay particular attention to the link between object types given by dialgebras and (co)algebras. We demonstrate "wrappers" which give canonical encodings of algebraic datatypes as objects. We study in more detail one particular wrapper for natural numbers, and demonstrate that direcursion can be used to express operations on the constructed object. We give a correctness proof for this simple example.

The final, fifth chapter concludes and plans for further work.

Chapter 2

Background

2.1 Category Theory

In category theory, one studies mathematical structures solely by means of relationships between them. These relationships are represented by morphisms between objects inside a structure known as a category. An object, such as a group or the natural numbers, is not given by its internal (set theoretic) structure, but more abstractly by morphisms going to and from this object. In other words, an object is characterised (up to isomorphism) solely by how its surrounding morphisms compose with other morphisms, which is typically shown in a commuting diagram. It is this, the fact that arrow composition rather than, say, set membership, that is central, that allows us to abstractly treat many recurring themes in a single categorical notion.

Category theory has been termed ”abstract nonsense” both by its advocators and detractors. Indeed, Goguen [39], in his ”Categorical Manifesto”, admits that category theory can be abused, for example by excessive generalisation (such as describing Galois connections as adjoints without actually making use of the added generality). However, Goguen also gives a plethora of examples where category theory and category theoretic methods have proven successful in computer science. His examples range from automata and types to programs and program schemes, polymorphism, data refinement, to models of lambda calculus using cartesian closure, notions of computations using monads, initial algebra semantics, and graph theory. Goguen also tries to explain why category theory has been so successful, arguing that set theory has failed to provide a common agreed upon foundation for mathematics (e.g. Aczel suggested an Anti-Foundation Axiom to model non-well founded sets occurring in computing, e.g. in modelling Milner’s CCS). One reason why category theory has proven so useful in computing science is, according to Goguen, the fact that computing science is at an early stage, where the categorical style helps in driving the research forward.

In this thesis we apply category theory to the denotational semantics of object-oriented programming languages. We use category theory to express object types as higher-order data types. Indeed, category theory gives a suitable language to formalise notions of data

types, as shown by Lehmann and Smyth [47], Manes and Arbib [53], and many others. A significant amount of research has also been devoted to generic data type theory, e.g. [52], and to taking Bird and Meertens' calculus for derivation of algorithms from specifications (the so called Bird-Meertens formalism) into a more general framework of category theory. Malcolm showed that, in a category theoretic framework, program derivations can indeed benefit from category theory because the derivations themselves reach a level of genericity with respect to data types. For example notions of recursion show up as beautiful universal constructions which are amenable to formal reasoning (e.g. in program calculations).

More concretely, the merits of category theory will appear in the modelling of data types using functors. Functors give us a succinct way of expressing the signatures of algebras, coalgebras, and dialgebras, which has a value when we perform program calculations. Further, we require a non set-theoretic model (domain theoretic, expressed in an axiomatic categorical style) to model object calculi denotationally.

Definition 2.1.1 (Category) *A category C consists of a class Ob of objects and a class Ar of arrows, together with a typed binary composition operator $\circ$ which is associative and has identities. An arrow f has a domain, written $dom(f)$, and a codomain $cod(f)$, both of which are objects (we write $f : dom(f) \to cod(f)$). Any two arrows f, g compose into $g \circ f$ exactly when $cod(f) = dom(g)$. We require $f \circ (g \circ h) = (f \circ g) \circ h$ whenever f, g, h have the required domains and codomains. Finally, for any object A, there must exist a designated identity arrow $id_A : A \to A$, with $f \circ id_{dom(f)} = f$ and $id_{cod(f)} \circ f = f$ for any arrow f in Ar.*

We follow standard convention and write $C(A, B)$ or $[A, B]$ for the class of arrows f such that $dom(f) = A$ and $cod(f) = B$ (the *homset*). We write $Ar(C)$ when the category C is not clear from the context (similarly for Ob).

Typical examples of categories are the empty category **0**, the category **1** with one object and its identity arrow, the category Set of sets and total functions, and the category Cpo of ω-complete partial orders and continuous functions.

Functors and Natural Transformations

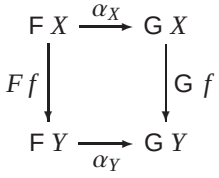
Definition 2.1.2 (Functor) *A functor $\mathsf{F} : C \to \mathcal{D}$ is a pair of total operations $\mathsf{F} : Ob(C) \to Ob(\mathcal{D})$ (the object map) and $\mathsf{F} : Ar(C) \to Ar(\mathcal{D})$ (the arrow map) such that domain, codomain, composition, and identities are preserved:*

(i) $\mathsf{F}\ f : \mathsf{F}(dom(f)) \to \mathsf{F}(cod(f))$
(ii) $\mathsf{F}\ f \circ \mathsf{F}\ g = \mathsf{F}\ (f \circ g)$
(iii) $\mathsf{F}\ id_A = id_{\mathsf{F}\ A}$

Definition 2.1.3 (Difunctor) *Let $\mathsf{F} : C^{op} \times C \to C$. Such F are called* (endo-) difunctors.

Definition 2.1.4 (Natural Transformation) *Given functors $\mathsf{F}, \mathsf{G} : C \to \mathcal{D}$, a natural transformation $\alpha : \mathsf{F} \dot{\longrightarrow} \mathsf{G}$ consists of a family of arrows $(\alpha_X : \mathsf{F}\ X \to \mathsf{G}\ X)_{X \in C}$ such*

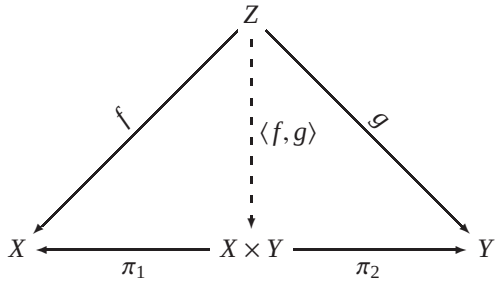
that for every arrow $f : X \to Y$ in C:



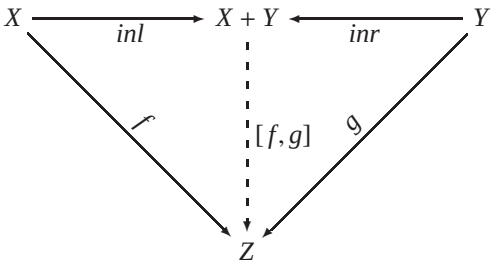
Cartesian closure, Limits and Colimits

Most interesting categories have additional structure. For example, the category Set has finite cartesian products and finite coproducts (also known as disjoint sum) etc. The notions of limits and colimits generalise such structural properties for arbitrary categories.

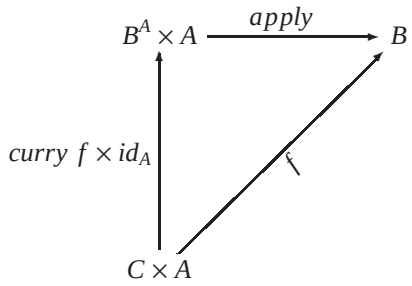
Definition 2.1.5 (Product) *A product of objects X and Y is an object $X \times Y$ together with arrows ("projections") $\pi_1 : X \times Y \to X$ and $\pi_2 : X \times Y \to Y$ such that for any arrows $f : Z \to X, g : Z \to Y$ there exists a unique arrow $\langle f, g \rangle$ making the following diagram commute:*



Definition 2.1.6 (Coproduct) *A coproduct of objects X and Y is an object $X + Y$ together with arrows ("injections") $inl : X \to X + Y \to X$ and $inr : Y \to X + Y$ such that for any arrows $f : X \to Z, g : Y \to Z$ there exists a unique arrow $[f, g]$ making the following diagram commute:*



Definition 2.1.7 (Exponentials) *Let C be a category with finite products and terminal object. The exponential of B by A is an object B^A (also written $[A, B]$) together with arrow $apply : B^A \times A \to B$ such that for every $f : C \times A \to B$ there exists an arrow $curry\ f : C \to B^A$ such that the following diagram commutes:*



The denotational semantics of typed lambda calculus can be given by Henkin models [43]. A Henkin model is a many-sorted algebraic structure with a (typed) map *app* (a typed applicative structure), satisfying two conditions: extensionality (i.e. if $app\ f\ x = app\ g\ x$ for all x, then $f = g$), i.e. *app* is one-to-one [59]) and an environment model condition (a total meaning function is definable). However, in a category theoretic setting we generalise Henkin models into the notion of cartesian closed categories:

Definition 2.1.8 (Cartesian closed categories) *A* cartesian closed category *(CCC) is a category with finite products, exponentials, and a terminal object.*

Partial Cartesian Closure

In this work we will work in a category pCpo rather than, say, the category of small complete pointed partial orders (posets with least elements closed under lubs of ω-chains) and continuous functions. The category pCpo gives a direct treatment of non-termination by means of partial continuous maps. This category consists of ω-complete partial orders and partial continuous functions. It is formally defined as follows:

Definition 2.1.9 (pCpo**)** *The category* pCpo *consists of small complete partial orders (posets, possibly without least element, closed under lubs of ω-chains), and partial continuous functions, i.e. partial functions $f : P \rightharpoonup Q$ such that:*

- Monotonicity*: for every $x, x' \in P$, if $x \sqsubseteq_P x'$, then either $f(x)$ is undefined, or $f(x) \sqsubseteq_P f(x')$ with $f(x')$ defined.*
- Continuity*: for every ω-chain x_i, $i \in I$ in P,*

$$\sqcup_k f(x_k) \simeq f(\sqcup_k x_k)$$

where $e \simeq e'$ means that either e and e' are both undefined, or else they are equal. The $\sqcup$ notation is intended to be undefined if every $f(x_k)$ is undefined; or else denote $\sqcup_{k \geqslant k_0}$ where k_0 is any index for which $f(x_{k_0})$ is defined.

A more abstract representation of partiality can be given by taking partial maps as total maps together with a *domain of definition*, i.e. work in the Kleisli category of a lifting

monad. The standpoint for such a development is to assume a category of values (which means total maps) and next consider a notion of computation (here: non-termination) as an additional structure on the category of values. Technically, one arrives at representing partial maps by introducing a subcategory of admissable monos (subobjects) giving the domains of definition for the partial maps. In this thesis we have chosen to work directly with the above more direct definition of pCpo.

We denote by Cpo the subcategory of pCpo consisting of all cpos and total continuous functions. The salient facts about the categories pCpo and Cpo can be found in [64]. Cpo is cartesian closed with finite coproducts. We give a brief summary of the structure of pCpo:

- **Zero object:** The empty cpo is a zero object in pCpo. That is, it is both an initial object and a terminal object.
- **Coproducts:** If A and B are cpos, their disjoint union is the coproduct of A and B in pCpo.
- **Partial Products:** If A and B are cpos, the cartesian product of the underlying sets is their partial product. It is not a product as the domain of definition of the pairing (f, g) is the intersection of the domains of f and g and hence $fst(f, g) \neq f$ etc. We denote the partial product by $A \otimes B$ to remind ourselves it is not a product. pCpo has partial products given via the base category Cpo for a pair of partial maps $(u, u') : (P, P') \rightharpoonup (Q, Q')$ by

$$u \otimes u' : P \times P' \rightharpoonup U \times U : (x, x') \mapsto \begin{cases} (u(x), u'(x')), & \text{if } u(x) \downarrow \text{ and } u'(x') \downarrow \\ \text{undefined}, & \text{otherwise} \end{cases}$$

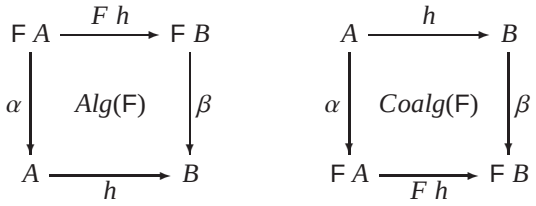
- **Kleisli/Partial Exponentials:** If A and B are cpos, then the set of partial continuous functions from A to B forms a cpo as usual. We denote this cpo $A \rightharpoonup B$ or $[A, B]$. As expected, partial exponentials are right adjoint to the partial product. $- \otimes A \dashv A \rightharpoonup -$: $\mathsf{Cpo} \longrightarrow \mathsf{pCpo}$. Note the domains and codomains for the functors involved in this adjunction.
- **Compactness:** pCpo is algebraically compact in that all locally continuous functors have coinciding initial algebras and final coalgebras [36].

The adjoint situation for Kleisli exponentials gives an isomorphism such that $\mathsf{pCpo}(A, B) \cong \mathsf{pCpo}(1 \otimes A, B) \cong \mathsf{Cpo}(1, A \rightharpoonup B)$. Note that the last homset gives the 1-elements a k a global elements of Cpo, so that this adjunction corresponds to our intuition and indeed shows that $A \rightharpoonup B$ internalises the partial maps $A \rightharpoonup B$ into pCpo.

Algebras, Coalgebras, and Functors

In pCpo, a covariant endofunctor $\mathsf{F} : C \longrightarrow C$ has a fixed point given by an object $A \cong FA$ which is the initial F-algebra or final F-coalgebra.

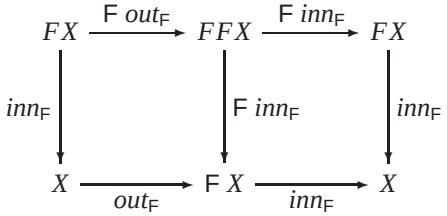
Definition 2.1.10 (Algebra, Coalgebra) *Given a functor* F *we say that an arrow* $\alpha : \mathsf{F}\,A \to A$ *is an* F*-algebra with carrier A. Such* F*-algebras are the objects in a category* **Alg**(F) *for every functor* F*. The dual notion is that of* F*-coalgebra, i.e. reversed arrows* $\alpha : A \to \mathsf{F}\,A$*. The arrows between (co)algebras are* F*-homomorphisms, i.e. arrows h such that, for* F*-algebras the left diagram below commutes and, for* F*-coalgebras the right diagram below commutes:*



We write inn_{F} for the initial F-algebra and out_{F} for the final F-coalgebra. In pCpo we have $inn_{\mathsf{F}}{}^{-1} = out_F$ for locally continuous F.

Lemma 2.1.11 (Lambek) *An initial algebra* $FX \xrightarrow{f} X$ *is an isomorphism.*

Proof



$inn_{\mathsf{F}} \circ out_{\mathsf{F}} = 1$ by uniqueness. $out_{\mathsf{F}} \circ inn_{\mathsf{F}} = F\ inn_{\mathsf{F}} \circ F\ out_{\mathsf{F}} = F(inn_{\mathsf{F}} \circ out_{\mathsf{F}}) = F1 = 1$ □

Dialgebras and Difunctors

The equation $D \cong [D, D]$ suggests, by cardinality grounds, that the existence of such recursively defined types is not at all obvious, e.g. there is clearly no set D such that $D \cong [D, D]$. The key feature of this example is that the mapping of an object D to the object $[D, D]$ is not a functor in that the left occurrence of D in the expression $[D, D]$ occurs *contravariantly* while the right occurrence is *covariant*. Such mappings are called *difunctors*.

Definition 2.1.12 (Difunctor) *If C is a category, a difunctor is a functor* $\mathsf{F} : C^{op} \times C \longrightarrow C$*. A fixed point of such a difunctor is an object X such that* $X \cong \mathsf{F}\ X\ X$

There has been much research on finding fixed points for difunctors. The classic paper [71] defines a category of embedding and projection pairs where the functor F acts covariantly and from which a fixed point of F can be derived. More recently, [37, 36, 28, 30] have used the more axiomatic setting of *algebraically compact* categories. i.e. categories

where all (in a suitably qualified sense) covariant functors have an initial algebra, the inverse of whose structure map is the final coalgebra. The related, but weaker, property of *algebraic completeness* merely requires all (again, in a suitably qualified sense) covariant functors to have an initial algebra.

The axiomatic approach is potentially easier to apply to non-domain theoretic models such as realizability models and models containing intensional features. Since we do not wish to over commit ourselves to a specific semantic setting at this stage, we therefore implicitly follow the axiomatic setting of [28, 30] in working in the Kleisli category of a lifting monad. However, for concreteness, we are explicitly working in the canonical model of the category pCpo described above.

It is worth making the observation here that, apart from compactness, we would have liked our ambient category to be cartesian closed and have finite coproducts so that we could manipulate polynomial functors and their (co-)algebras using the standard techniques. Indeed, settling for partial products and Kleisli exponentials may seem like a poor alternative. However, any compact category has a zero object (induced as the fixed point of the identity functor) and a CCC with a zero object is inconsistent as

$$A \cong A \times 1 \cong A \times 0 \cong 0$$

Hence we cannot get away from working in a non-cartesian closed setting. Nevertheless, the subcategory Cpo (where values take their denotation) is, of course, still cartesian closed.

So given a category like pCpo, how does one find fixed points for difunctors? When working with difunctors, algebras and coalgebras generalise to *dialgebras*. Note the presence of both covariance and contravariance in a difunctor means that we have no need for the dual notion of a dialgebra. The term dialgebra has several definitions in the literature (see for example [37, 6, 67, 42]), and we will use the following definition which is due to Freyd [37]:

Definition 2.1.13 (Dialgebras) *A* G-dialgebra *for difunctor* $\mathsf{G} : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C}$ *is a pair of objects* A, B *together with an associated pair of arrows* $f : \mathsf{G}\,A\,B \to B$ *and* $g : A \to \mathsf{G}\,B\,A$.

The category of dialgebras has maps between dialgebras given as follows

Definition 2.1.14 (Dialgebra Maps) *Given* G*-dialgebras* (A, B, ϕ, ψ) *and* (A', B', ϕ', ψ'), *a* G-homomorphism *is a pair of arrows* $(g : B \to B', h : A' \to A)$ *such that the following diagrams commute:*

$$\begin{array}{ccc} \mathsf{G}\,A\,B & \xrightarrow{\phi} & B \\ {\scriptstyle \mathsf{G}\,g\,h}\downarrow & & \downarrow{\scriptstyle g} \\ \mathsf{G}\,A'\,B' & \xrightarrow[\phi']{} & B' \end{array} \qquad \begin{array}{ccc} A & \xrightarrow{\psi} & \mathsf{G}\,B\,A \\ {\scriptstyle h}\uparrow & & \uparrow{\scriptstyle \mathsf{G}\,h\,g} \\ A' & \xrightarrow[\psi']{} & \mathsf{G}\,B'\,A' \end{array}$$

A key idea in axiomatic domain theory is to use algebraic compactness to find fixed points for difunctors. Here is a sketch of the construction:

Lemma 2.1.15 *Let* $\mathsf{G} : C^{op} \times C \longrightarrow C$ *be a difunctor on an algebraically compact category* C*. Then* G *has a fixed point.*

Proof Form the functor $\mathsf{G}' : C^{op} \times C \to C^{op} \times C$ by following the *doubling trick* proposed by Freyd:

$$\mathsf{G}'\ X\ Y \triangleq (\mathsf{G}(Y, X), \mathsf{G}(X, Y))$$

Since C is algebraically complete, so is $C^{op} \times C$ and thus G' has an initial algebra, say $\mathsf{G}'(X, Y) \to (X, Y)$, which is given by maps $inn_{\mathsf{G}} : X \to \mathsf{G}(Y, X)$ and $out_{\mathsf{G}} : \mathsf{G}(X, Y) \to Y$. By Lambek's lemma, inn_{G} and out_{G} are isomorphisms. Next, the pair $(out_{\mathsf{G}}, inn_{\mathsf{G}}) : (Y, X) \to \mathsf{G}'(Y, X)$ is easily seen to be the final G'-coalgebra. Since C is algebraically compact, so is $C^{op} \times C$ and hence the initial G'-algebra and final G'-coalgebra coincide. Thus $X = Y$ and we have a G-fixed point as required.

□

Of course, while the above proof may seem simple, much of the work is hidden in proving that i) algebraic completeness and compactness are preserved by taking products and opposite categories; ii) formalising exactly the class of difunctors which are to be considered; and iii) proving that certain categories are algebraically complete and compact. Further subtle and technical issues arise, e.g. that these fixed points should be suitably parameterised etc, but for this presentation we have decided to gloss over the details. See [28] for details. Having said this, the modularisation of the construction of fixed points is very elegant. Notice also that more is true than we claimed. In particular we constructed a specific fixed point of a difunctor with a universal property, namely, the initial dialgebra. We shall put this universal property to use later.

2.2 Theory of Data Types

Algebraic data types (without parameters), such as lists or trees, are modelled as least fixed points of functors. These fixed points appear as initial algebras of the associated functor. Dually, coalgebraic datatypes (without parameters), such as streams, are modelled as greatest fixed points and appear as final coalgebras.

In the present work, we use will categorical higher-order data types to model object types. In this section we will review the definitions and results for the first-order cases, which we later aim to generalise to a specific higher-order case of object types.

Bird-Meertens Formalism

The generic theory of data types we review in this section is often referred to as Bird-Meertens formalism [54, 7, 9, 72]. This generic theory of data types serves the purpose of supporting formal methods for program construction, because correctness proofs become particularly short and also amenable to automation [52]. In addition, this generic theory

of data types provides a mathematical analysis that generates laws for program transformations, and reveals and classifies the structure of recurring programming idioms in a rich mathematical setting of category theory.

It was Malcolm [52] that made the program calculation community aware of Hagino's ideas on giving categorical semantics to data types [41, 42]. Essentially, Malcolm demonstrated that the category theoretic approach lent itself well to program calculation. Later, Fokkinga [35] pursued Malcolm's work further, and developed topics such as algebras with laws, mutumorphism, and hylomorphism. More recently Uustalu and Vene have developed additional recursion schemes such as primitive (co)recursion [73].

One striking advantage of the categorical approach to data types and recursion, is the fact that data types (and therefore programs) come with laws that can be used for transformation. Such laws for example give conditions for when we can remove intermediate data structures or improve program efficiency by simplifying programs. These laws include the so called *fusion laws*. We will review some of these laws in this section, but much of the application of such fusion laws to our model of object calculi is, albeit very interesting, outside the scope of the present thesis.

Notions of Recursion

Definition 2.2.1 (Catamorphism) *Let* $(\mu\mathsf{F}, inn_{\mathsf{F}})$ *be the initial* F*-algebra, and* $\phi : \mathsf{F}\,A \to A$ *an arbitrary* F*-algebra.* $(\!|\phi|\!) : \mu\mathsf{F} \longrightarrow A$ *is defined to be the unique homomorphism in the following commuting diagram:*

$$\begin{array}{ccc} \mathsf{F}\,\mu\mathsf{F} & \xrightarrow{inn_{\mathsf{F}}} & \mu\mathsf{F} \\ {\scriptstyle \mathsf{F}\,(\!|\phi|\!)}\downarrow & & \downarrow{\scriptstyle (\!|\phi|\!)} \\ \mathsf{F}\,A & \xrightarrow[\phi]{} & A \end{array}$$

We will give some examples of catamorphism, and for this we must choose some object A and B. Objects are data types, and one of the simplest data types is *Nat*, the natural numbers. The naturals are given as the initial algebra of the functor $\mathsf{F}\,X = 1 + X$. Hence there are operations $inn : 1 + Nat \to Nat$ and $inn^{-1} : Nat \to 1 + Nat$. Because we are in an algebraically compact setting, we write *out* for inn^{-1} since the inverse of the initial algebra will be the final coalgebra. This model of algebraic data types unifies all the data type constructors in one single operator *inn* by means of a coproduct. However we have $inn \equiv [zero, succ]$, i.e. the ordinary constructors *zero* and *succ* are in fact defined by the initial algebra. Now that we have given the data type for naturals, we can easily use catamorphism (structural recursion on natural numbers) to define sum and product of two naturals, and the predecessor function:

$$\begin{array}{lll} add\ \ n\ m & \triangleq & (\!|[\lambda x.m, succ]|\!)\ n \\ mult\ \ n\ m & \triangleq & (\!|[\lambda x.zero, \lambda x.add\ m\ x]|\!)\ n \\ pred & \triangleq & (\!|[id + [zero, succ]]|\!) \end{array}$$

Next, we review the three properties satisfied by any catamorphism. These three properties are useful in program calculation since they tell us how to replace expressions involving catamorphisms with more simple expressions:

Corollary 2.2.2 (Properties of catamorphism) *Let* $(\mu\mathsf{F}, inn_{\mathsf{F}})$ *be the initial* F*-algebra.*

- ***Cancellation:*** *For any other* F*-algebra* $\phi : \mathsf{F}\ A \to A$ *we have*

$$(\!|\phi|\!) \circ inn_{\mathsf{F}} = \phi \circ \mathsf{F}(\!|\phi|\!) \qquad ((\!|\cdot|\!)\text{-S})$$

- ***Reflection:***

$$id = (\!|inn_{\mathsf{F}}|\!) \qquad ((\!|\cdot|\!)\text{-R})$$

- ***Fusion:*** *For any* F*-algebras* $\phi : \mathsf{F}\ A \to A$ *and* $\xi : \mathsf{F}\ B \to B$*, and arrow* $f : A \to B$

$$f \circ \phi = \xi \circ \mathsf{F}\ f \quad \Rightarrow \quad f \circ (\!|\phi|\!) = (\!|\xi|\!) \qquad ((\!|\cdot|\!)\text{-F})$$

The dual of catamorphism is called anamorphism. Anamorphism is the notion of structural corecursion, and is associated to each coalgebraic data type. Since we are in an algebraically compact setting where coalgebraic data types and algebraic data types coincide, anamorphism is simply another (co)recursion principle on data types. In this setting, both naturals and the dual "conaturals" (given as the associated final coalgebra) contain the infinite natural number. Note also that the function *pred* above is in fact the final coalgebra for *Nat*. For full generality we will write $\nu\mathsf{F}$ for the carrier of the final coalgebra, although in the algebraically compact setting we are assuming, we have in fact $\nu\mathsf{F} = \mu\mathsf{F}$:

Definition 2.2.3 (Anamorphism) *Let* $(\nu\mathsf{F}, out_{\mathsf{F}})$ *be the final* F*-coalgebra, and* $\psi : A \to \mathsf{F}\ A$ *an arbitrary* F*-coalgebra.* $[\!(\psi)\!] : A \longrightarrow \nu\mathsf{F}$ *is defined to be the unique homomorphism in the following commuting diagram:*

$$\begin{array}{ccc} \nu\mathsf{F} & \xrightarrow{out_{\mathsf{F}}} & \mathsf{F}\ \nu\mathsf{F} \\ {\scriptstyle [\!(\psi)\!]}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\ [\!(\psi)\!]} \\ A & \xrightarrow[\psi]{} & \mathsf{F}\ A \end{array}$$

A stream is an example of a coalgebraic data type. The stream of natural numbers is given as the final coalgebra of the functor $\mathsf{F}\ X = Nat \times X$. Such a stream is always a natural number paired with the stream of all remaining numbers, which is witnessed by the type $Stream \to Nat \times Stream$ of the corresponding map *out*. The destructors *head* and *tail* are not visible in this form, but are in fact defined by $\langle head, tail \rangle \equiv out$.

We can use anamorphisms to create values for coalgebraic data types. For example, to create a stream of all natural numbers greater than some number *n*, we can use an

anamorphism. Similarly, the function that zips together two streams into a stream of pairs, and the stream that repeatedly applies a function f to its argument creating a stream of numbers n, $f(n)$, $f(f(n))$, ..., are examples of anamorphisms:

$$\begin{array}{lll} nats & \triangleq & [\![\langle id, succ\rangle]\!] \\ zip & \triangleq & [\![\langle \pi_1 \times \pi_1, \pi_2 \times \pi_2\rangle \circ (out \times out)]\!] \\ iterate\ \ f & \triangleq & [\![\langle id, f\rangle]\!] \end{array}$$

Just like catamorphism, anamorphism satisfies some useful properties:

Corollary 2.2.4 (Properties of anamorphism) *Let* (A, out_{F}) *be the final* F*-coalgebra.*

- ***Cancellation:*** *For any other* F*-coalgebra* $\psi : B \to \mathsf{F}\ B$ *we have*

$$out_{\mathsf{F}} \circ [\![\psi]\!] = \mathsf{F}[\![\psi]\!] \circ \psi \qquad ([\![\cdot]\!]\text{-S})$$

- ***Reflection:***

$$id = [\![out_F]\!] \qquad ([\![\cdot]\!]\text{-R})$$

- ***Fusion:*** *For any* F*-coalgebras* $\psi : B \to \mathsf{F}\ B$ *and* $\xi : C \to \mathsf{F}\ C$*, and arrow* $f : B \to C$

$$\psi \circ f = \mathsf{F}\ f \circ \psi \ \Rightarrow\ [\![\psi]\!] \circ f = [\![\xi]\!] \qquad ([\![\cdot]\!]\text{-F})$$

Catamorphism and anamorphism are sufficient to express at least every primitive recursive function, and in a setting with exponentials also functions such as the Ackerman function [73]. Still, it can sometimes be cumbersome to write functions as catamorphism. Typical examples of the problems arise when we consider simple primitive recursive functions on the naturals, for example the factorial function. The factorial function must be defined using *tupling* in the following way:

$$fact \quad \triangleq \quad \pi_1 \circ (\!|[\lambda x.\langle 1, 0\rangle, \lambda\langle f, n\rangle.\langle (n+1) * f, n+1\rangle]|\!)$$

The problem is that at every recursive step, *fact* depends not only on the value computed at the preceding recursive step, but also multiplies this value with a counter that has counted the number of recursive steps that have occurred so far, i.e. $n + 1$ is multiplied by *fact* n to produce *fact* $n + 1$. This does not immediately fit into the form of a catamorphism, unless we use the tupling trick. Therefore, it is practical to also have a variation of catamorphism that captures precisely primitive recursion:

Definition 2.2.5 (Paramorphism) *Given* $\phi : \mathsf{F}(A \times \mu\mathsf{F}) \to A$*, the paramorphism* $(\!|\phi|\!) : \mu\mathsf{F} \longrightarrow A$ *is defined to be the unique arrow making the following diagram commute:*

$$\begin{array}{ccc} \mathsf{F}\mu\mathsf{F} & \xrightarrow{inn_{\mathsf{F}}} & \mu\mathsf{F} \\ {\scriptstyle \mathsf{F}\langle (\!|\phi|\!), id_{\mu\mathsf{F}}\rangle}\downarrow & & \downarrow{\scriptstyle (\!|\phi|\!)} \\ \mathsf{F}(A \times \mu\mathsf{F}) & \xrightarrow[\phi]{} & A \end{array}$$

Now factorial is more simply defined by $fact = (\!|[succ \circ zero, \lambda\langle f, n\rangle.mult\langle succ\ n, f\rangle]|\!)$, which is using precisely primitive recursion on natural numbers, with the two cases separated in a coproduct. However, as proved by Meertens [56], a paramorphism is still nothing but a catamorphism in disguise:

Lemma 2.2.6 (Meertens [56]) $(\!|\phi|\!) = \pi_1 \circ (\!|\langle \phi, inn \circ \mathsf{F}\pi_2\rangle|\!)$

Again the recursion scheme comes with some basic properties:

Corollary 2.2.7 (Properties of paramorphism) *Let* (A, inn_{F}) *be the final* F*-algebra.*

- ***Cancellation:*** *For any arrow* $\phi : \mathsf{F}(A \times \mu\mathsf{F}) \to A$ *we have*

$$(\!|\phi|\!) \circ inn = \phi \circ \mathsf{F}\langle (\!|\phi|\!), id\rangle \qquad ((\!|\cdot|\!)\text{-S}\ )$$

- ***Reflection:***

$$id = (\!|inn_F \circ \mathsf{F}\pi_1|\!) \qquad ((\!|\cdot|\!)\text{-R}\ )$$

- ***Fusion:*** *For any arrows* $\phi : \mathsf{F}(A \times \mu\mathsf{F}) \to A$, $\psi : \mathsf{F}(B \times \mu\mathsf{F}) \to B$ *and* $f : A \to B$ *we have*

$$f \circ \phi = \psi \circ \mathsf{F}(f \times id) \ \Rightarrow\ f \circ (\!|\phi|\!) = (\!|\psi|\!) \qquad ((\!|\cdot|\!)\text{-F}\ )$$

Parametric Data Types

So far we have not considered data types which have parameters, e.g. lists or trees of *arbitrary* element types. Such data types are however easy to model in the same categorical framework. Instead of having one fixed point associated to each data type, we introduce bifunctors $\mathsf{F} : C \times C \to C$ and use them to give a family of fixed points indexed over the type parameter:

Theorem 2.2.8 (Data Functors [52]) *Suppose* $\mathsf{F} : C \times C \to C$ *is a bifunctor such that for any object* X *there exists an initial* F_X*-algebra* $(\mu\mathsf{F}_X, inn_{\mathsf{F}_X})$*. Then the mapping* $\mathsf{T}\ X = \mu\mathsf{F}_X$ *can be extended to an endofunctor on* C *by defining*

$$\mathsf{T}\ f = (\!|inn \circ \mathsf{F}(f, id)|\!) \qquad (\text{map-D}\ )$$

The functor $\mathsf{T} : C \to C$ *is called the* data functor *of* F.

Proof We must show that T is indeed a functor, i.e. that it preserves identities and composition. First, identities:

$$
\begin{array}{ll}
 & \mathsf{T}\ id \\
= & \quad \{\ \text{map-D}\quad \} \\
 & (\!|inn \circ \mathsf{F}(id, id)|\!) \\
= & \quad \{\ \mathsf{F}\ \text{is bifunctor}\ \} \\
 & (\!|inn|\!) \\
= & \quad \{\ (\!|\cdot|\!)\text{-R}\quad \} \\
 & id
\end{array}
$$

Next, composition:

$$
\begin{array}{ll}
 & \mathsf{T} f \circ \mathsf{T} g \\
= & \quad \{\ \text{map-D}\quad \} \\
 & \mathsf{T} f \circ (\!|inn \circ \mathsf{F}(g, id)|\!) \\
= & \quad \{\ (\!|\cdot|\!)\text{-F}\quad \} \\
 & (\!|inn \circ \mathsf{F}(f \circ g, id)|\!) \\
= & \quad \{\ \text{map-D}\quad \} \\
 & \mathsf{T}(f \circ g)
\end{array}
$$

Note that we could use $(\!|\cdot|\!)$-F because the condition for the rule was satisfied, namely:

$$
\begin{array}{ll}
 & \mathsf{T} f \circ inn \circ \mathsf{F}(g, id) \\
= & \quad \{\ \text{map-D}\quad \} \\
 & (\!|inn \circ \mathsf{F}(f, id)|\!) \circ inn \circ \mathsf{F}(g, id) \\
= & \quad \{\ (\!|\cdot|\!)\text{-S}\quad \} \\
 & inn \circ \mathsf{F}(f, id) \circ \mathsf{F}(id, (\!|inn \circ \mathsf{F}(f, id)|\!)) \circ \mathsf{F}(g, id) \\
= & \quad \{\ \mathsf{F}\ \text{bifunctor}\ \} \\
 & inn \circ \mathsf{F}(f \circ g, id) \circ \mathsf{F}(id, (\!|inn \circ \mathsf{F}(f, id)|\!)) \\
= & \quad \{\ \text{map-D}\quad \} \\
 & inn \circ \mathsf{F}(f \circ g, id) \circ \mathsf{F}(id, \mathsf{T}\ f)
\end{array}
$$

□

The previous theorem immediately dualises to the setting of coalgebraic data types:

Corollary 2.2.9 (Codata Functors) *Suppose* $\mathsf{F} : C \times C \to C$ *is a bifunctor such that for any object* X *there exists an final* F_X*-coalgebra* $(\nu \mathsf{F}_X, out_{\mathsf{F}_X})$*. Then the mapping* $\mathsf{T}\ X = \nu \mathsf{F}_X$

can be extended to an endofunctor on C by defining

$$\mathsf{T}\ f = (\!|\mathsf{F}(f, id) \circ out_{\mathsf{F}}|\!) \qquad \text{(comap-D)}$$

The functor $\mathsf{T} : C \to C$ *is called a* codata functor *of* F.

Polytypic Programming

The above generic theory of data types makes it possible to prove programs correctness *generically*, i.e. by considering the functor F representing the data type to be a parameter, we can derive correctness proofs which are independent of the particular choice we make for the functor. A natural question to ask is if we can make the same generalisation in programming languages, and thus make it possible to actually define the associated *generic* functional programs inside the programming language itself.

The answer to this question is affirmative, and there have been several languages that attempts to support "polytypic programming" (in one style or another) which means parameterizing programs with respect to the "pattern" functor F for data types. We will review three such languages in this section: Charity, PolyP, and Generic Haskell.

Charity

The programming language Charity [23] automatically provides functions such as catamorphism for each user-defined data type. As a programming language, Charity is functional and strongly normalizing, which means that all programs are terminating. From the perspective of this thesis we are primarily interested in recursive types which are higher-order, whereas Charity can only model inductive/coinductive (first-order) data types. However, an extension, Higher-order Charity [70], has been proposed which introduces exponential types. However, the mixed variant data types that we use in this thesis *cannot* be modelled even in this extension of Charity (only covariant positions are allowed).

PolyP

The programming language PolyP takes a different approach than Charity. In PolyP the programmer can define their own functions by induction over the structure of the pattern functor F. This means that given a concrete data type, such as list, a function such as map_{F} can be instantiated. Accordingly, PolyP automatically generates one map function for each user-defined data type, given only one single definition of *map*. In PolyP's standard library the map function is first defined using a polytypic definition (polytypic meaning exactly induction over the structure of pattern functors):

```
polytypic fmap2 :: (a → c) → (b → d) → f a b → f c d
  = λp r → case f of
      g ⊕ h   → fmap2 p r -⊕- fmap2 p r
      g ⊗ h   → fmap2 p r -⊗- fmap2 p r
      Unit    → Const ()
```

$$
\begin{array}{ll}
Par & \rightarrow p \\
Rec & \rightarrow r \\
d \odot g & \rightarrow pmap\ (fmap2\ p\ r) \\
Const\ t & \rightarrow id
\end{array}
$$

The function *fmap2* replaces the built-in Haskell function *fmap*, which must be user-defined for every new data type. Given a user-defined data type D, PolyP allows both to use the constructor for this data type by means of generic functions *inn* and *out* (also generated for all user-defined data types). The above *fmap2* extends these constructor functions into functors, where the action on functions corresponds to the data functors given above categorically.

Now that we have *inn*, *out*, and *fmap2* for any data type, we can define the map function *pmap*, as well as the catamorphism *cata* and anamorphism *ana*:

$$
\begin{array}{l}
pmap :: (a \rightarrow b) \rightarrow D\ a \rightarrow D\ b \\
pmap\ f = inn \circ fmap2\ f\ (pmap\ f) \circ out
\end{array}
$$

$$
\begin{array}{l}
cata :: (F\ a\ b \rightarrow b) \rightarrow D\ a -> b \\
cata\ \phi = \phi \circ fmap2\ id\ (cata\ \phi) \circ out
\end{array}
$$

$$
\begin{array}{l}
ana :: (b \rightarrow F\ a\ b) \rightarrow b \rightarrow D\ a \\
ana\ \psi = inn \circ fmap2\ id\ (ana\ \psi) \circ \psi
\end{array}
$$

Generic Haskell

In Generic Haskell [44, 51], polytypic functions are defined by induction over kinds, and so have kind-indexed types. Here is an example of a kind-indexed type definition of the map function:

$$
\begin{array}{l}
type\ \mathsf{Map}\ \{\!|\star|\!\}\ s\ t = s \rightarrow t \\
type\ \mathsf{Map}\ \{\!|\kappa \rightarrow \nu|\!\}\ s\ t = \forall\ a\ b\ .\mathsf{Map}\{\!|\kappa|\!\}\ a\ b \rightarrow \mathsf{Map}\{\!|\nu|\!\}\ (s\ a)\ (t\ b)
\end{array}
$$

Instead of the type case statement inside PolyP, the definition of *gmap* (the map function) has a clause for each kind-index:

$$
\begin{array}{lll}
gmap\ \{\!|t :: \kappa|\!\} :: Map\ \{\!|\kappa|\!\}\ t\ t & & \\
gmap\ \{\!|\oplus|\!\}\ gmapA\ gmapB\ (Inl\ a) & = & Inl\ (gmapA\ a) \\
gmap\ \{\!|\oplus|\!\}\ gmapA\ gmapB\ (Inr\ b) & = & Inr\ (gmapB\ b) \\
gmap\ \{\!|\otimes|\!\}\ gmapA\ gmapB\ (a \otimes b) & = & gmap\ A\ a \otimes gmapB\ b \\
gmap\ \{\!|Unit|\!\} \qquad\qquad Unit & = & Unit \\
gmap\ \{\!|Con\ c|\!\}\ gmapA \qquad (Con\ a) & = & Con\ (gmapA\ a)
\end{array}
$$

Generic Haskell is more powerful than PolyP in that it supports a larger class of data types. In addition to the unary regular data types supported by PolyP, Generic Haskell

allows polytypic functions to be defined over potentially non-regular data types of arbitrary kinds [60]. However, in allowing more data types, more is also required when defining a polytypic function. In particular, Generic Haskell cannot directly define *ana* and *cata* since datatypes are not represented by fixed points [60].

2.3 Semantics of Programming Languages

In the first chapter we gave some brief motivation for why one needs formal semantics when considering extensions of programming languages. In this section we will give an introduction to the basic notions that occur in formal semantics of programming languages, particularly the distinction between operational and denotational semantics, and the theorems and results that are associated with such semantics. Several of this theorems will be proved in a particular setting in Chapter 3.

We have explained that the motivation to this work was both semantical foundations for object-oriented programming languages and also correctness of (object-oriented) programs with respect to specifications. In particular, we claimed that the notion of recursion we study in this thesis comes with laws for program optimisation/transformation (such laws are well-known from the Bird-Meertens formalism). Because of this, formal semantics plays another important rolê. It is the source in which we give mathematical foundations to the laws, i.e. to the logic that establishes how we can formally derive/reason/calculate with object-oriented programs. Of primary important to us is denotational semantics, since we will use universal properties of category theory to express generic laws about object types.

Operational Semantics

In operational semantics one evaluates a formal language on an abstract machine by means of one or more evaluation/reduction relations. The meaning of a term in the formal language is defined to be the *value* (also known as *canonical form* or *normal form*), i.e. to the term (*contractum*) to which the term (called *redex* in this context) reduce after all possible reductions have been applied. In other words, a value cannot be further reduced. Hence, it is crucial that it does not matter in which order we apply the reduction rules, since otherwise a single term would have multiple meanings. This desirable property is known as *confluence* (or *Church-Rosser property*) of the reduction relation.

There are several approaches to operational semantics, e.g. big-step (natural) semantics and small-step (reduction) semantics. In small-step semantics, we do not immediately reduce to canonical forms, although the transitive closure of small-step reduction must yield canonical forms when such exist. In small-step semantics we distinguish between *strongly normalising* and *weakly normalising* reduction relations, the former means that canonical forms always arise no matter what order we apply the reductions to a term. Weak normalisation merely requires that for any given term there exists *some* reduction sequence that gives a canonical form. Typed object calculi studied in this thesis have confluent small-step semantics given by Abadi and Cardelli [3]. But typed object calculi can encode untyped lambda calculus, and therefore we cannot hope to have neither weak nor strong normalisation (consider e.g. the term $\Omega = (\lambda x.xx)(\lambda x.xx)$).

We will be solely be concerned with *typed* calculi (or calculi for which there is a type assignment system). There are several reasons for the interest in typed calculi as opposed to untyped calculi. First of all, a type system gives a lightweight method for proving absence of certain bad program behavior, such as "method not understood" errors when invoking a method that is in fact not available in a particular class. A type system can indeed be seen as giving an - often static - approximation to the intended behavior of the computer program [62]. Types also ensure a disciplined programming. For example, in this thesis we use recursive types to make precise that an object is given semantics when it is applied to value of its own type. This condition could not have been asserted in an untyped system. As a result, more structure is associated to object types, for example, the recursion principle that is central to this entire thesis. There are additional reasons for using typed calculi, such as efficiency, language safety, and so on. However, we will not have reason to delve into these additional topics in the present thesis.

When giving an operational semantics one must take care to define the sets of terms and types. Typically, this is done first with inductive definitions for raw terms and types, and then by selecting the well-formed types and well-typed terms, and quotient with respect to a suitable notion of alpha congruence. The resulting smaller sets are termed well-formed terms and well-formed types, and the rules are called judgements. For example, the typing judgements determine the set of well-typed terms, and kinding judgements the set of well-kinded types. The latter notion will not be needed in the present thesis. Instead, a well-formed type consists of a sequence of distinct type variables (a type context) together with a type whose free type variables appear in the sequence. Similarly, a well-formed term consists of a type context and a similar sequence of variables with type assignment of the form $x : \tau$ (term context), again such that all the free (term) variables occur in the context. However, for terms we require additional constraints to hold, namely that every term can be formed by a finite number of applications of typing judgements. The following judgements are used in typing rules (we postpone explaining them):

Definition 2.3.1 (Judgements)

$\vdash \Theta$	well-formed type context
$\vdash \Gamma$	well-formed term context
$\Theta, \Gamma \vdash \sigma \leq \tau$	type/subtype judgement
$\Theta, \Gamma \vdash \upsilon\sigma \leq \upsilon'\tau$	subtype judgement with variance
$\Theta, \Gamma \vdash m : \tau$	typing judgment

A Framework for Operational Semantics

When we give an operational semantics in this thesis, we will often need some basic rules and definitions for context formation. For convenience we have therefore chosen to give a fragment that we call $\Delta_{\vdash}$. This fragment sets up the basic notions of well-formed contexts

and a typing judgement for projecting out a variable from the context (the V x rule below). The fragment assumes that there exists sets V of type variables, U of term variables, and sets of terms ($\mathcal{M}$) and types ($\mathcal{T}$) (we regard $\Delta_{\vdash}(\mathcal{T}, \mathcal{M})$ as function to allow arbitrary notation for these two sets). These sets will however be defined differently depending on the calculus the fragment is included in, and the definitions are in fact by mutual induction on the sets of terms and types used in any particular calculus. Like $\Delta_{\vdash}$, operational semantics given in this thesis will also assume some countable sets of method labels (L), type variables (V), and term variables (U). For the elements in these sets we will use symbols $\ell_0, ..., \ell_n$ for labels, $x_0, ..., x_n$ for term variables and $X_0, ..., X_n$ for type variables. A well-formed type consists of a sequence of distinct type variables (a type context) together with a type whose free type variables appear in the sequence. The well-formed type contexts are given in definition 2.3.2.

Note that operational semantics is given in a meta-language which has a notion of tuples (e.g. $\Diamond$ for the empty tuple, and $\langle \Theta, X \rangle$) and sets. We write $i \in I$ and $i \in I - \{j\}$ to say that i is an arbitrary index from a index set I, and, for the second case, from the same set but without the element j. We will also use a universal quantifier, $\forall i \in I$, to indicate that a premise of a rule is in fact an abbreviation for a list of premises of slightly more complex form. In order to simplify contexts where we have $X \preceq \top$ where $\top$ is a designated "greatest type", we will allow the shorthand X for $X \preceq \top$.

When we give an operational semantics we want to prove certain soundness results. First of all for types:

- **Free variables are in context**: at any reduction step all well-typed terms/types with free (type) variables must be formed such that every free variable is listed in the context. This is a very basic requirement, but it still needs to be checked.

- **Substitution lemma for types**: the denotation of a term where a type variable has been substituted by some concrete type should satisfy a homomorphism property. Informally, the property means that this substitution can either be done denotationally or in the operational semantics, with the same result.

- **Unique/minimal type property**: in a system without subtyping, unique typing is desirable. Unique typing means that for any term there is exactly one type that can be inferred from the typing judgements. In a system with subtyping, the weaker property of type minimality is instead desirable. This property states that the least possible subtype exists and can be inferred from the rules.

- **Type preservation**: typings must be preserved during reduction. This notion is also termed subject reduction.

- **Type progress**: while reducing a term we will either reach a normal form ("value"), or the term will take a step according to the reduction rules. Type progress is a statement about "stuck terms", i.e. terms that are neither values, nor does the rules tell us how to reduce them. A difference between big-step and small-step semantics appears here. If a term is in normal form in big-step semantics, then the rules are

Definition 2.3.2 ($\Delta_{\vdash}(\mathcal{T}, \mathcal{M})$)

Type contexts Θ and term contexts Γ are generated by the following two rules

$$\Theta \quad ::= \quad \Diamond \mid \langle \Theta, X \preceq \tau \rangle \quad \text{where } X \in V \text{ and } \tau \in \mathcal{T}$$
$$\Gamma \quad ::= \quad \Diamond \mid \langle \Gamma, x : \tau \rangle \quad \text{where } x \in U \text{ and } \tau \in \mathcal{T}$$

and well-formed contexts are given by the rules:

$$\text{C}\ \emptyset \qquad\qquad \text{C}\ \preceq$$
$$\frac{}{\vdash \Diamond} \qquad\qquad \frac{\vdash \Theta}{\vdash \langle \Theta, X \preceq \tau \rangle}$$

where $X \in V, X \notin \Theta, \tau \in \mathcal{T}$

$$\text{V-C}\ \emptyset \quad \frac{\vdash \Theta}{\Theta \vdash \Diamond} \qquad \text{V-C}\ x \quad \frac{\Theta \vdash \tau, \Gamma}{\Theta \vdash \langle \Gamma, x : \tau \rangle} \qquad \text{V}\ x \quad \frac{\Theta \vdash \langle \Gamma, x : \sigma \rangle}{\Theta, \langle \Gamma, x : \sigma \rangle \vdash x : \sigma} \qquad \text{T}\ X \quad \frac{\vdash \langle \Theta, X \preceq \tau \rangle}{\langle \Theta, X \preceq \tau \rangle \vdash X}$$

where $x \in U, x \notin \Gamma$

applicable ad infinitum (values reduce to values). For small-step semantics, a term in normal form is stuck by definition. Type progress (for a small-step reduction relation) states that the reduction never gets ”stuck” for terms other than values, and thus gives a certain completeness result for our reduction rules. Absence of “stuck states” can also be proved for a big-step reduction relation e.g. by defining an algorithm and proving that this algorithm always computes a correct result given well-typed input (c.f. Abadi and Cardelli [3]). Progress and preservation is known together as *type soundness* or *type safety*.

Further, after giving a reduction relation we want to prove:

- **Substitution lemma for terms**: this is similar to the substitution lemma for types, only here we replace a term variable with some term instead of a type variable with a type. Again, the lemma makes a statement of a homomorphism property of substitution.

- **Soundness:** if a term t reduces to some term t', then the denotations of these terms are equal.

- **Adequacy:** if the denotations of term t and a term v in normal form are equal, then t reduces to v.
- **Computational soundness**: this is the property that termination is reflected in the denotational semantics, i.e. that any term that reduces to a normal form will have a terminating (e.g. total) denotation in the denotational semantics.
- **Computational adequacy**: this is the property that a term which has terminating/total denotation will reduce to a value in the operational semantics.
- **Full abstraction**: identified denotations give *observationally equivalent* terms and vice versa, e.g. meaning that there is no way to write a program that can distinguish those two terms, i.e. which reduces differently, depending on which of the two terms the program was built from.

The above explanation makes clear that soundness/adequacy is concerned with equality in the operational semantics and the denotational semantics (the former relation should be contained in the latter and vice versa) and thus reduction in general. Computational soundness/adequacy, on the other hand, is concerned with normalization (i.e. reduction to values). In this thesis we will prove computational soundness and computational adequacy for a particular typed calculi.

In this thesis we will give a denotational semantics but also *interpret* one operational semantics into another operational semantics. In the latter case the above theorems are relative to an operational semantics rather than a denotational model. We will see in chapter 3 that the above proof obligations take a particular form in the context of such an interpretation.

Subtyping

Subtyping is a reflexive and transitive relation on the universe of types that allows the use of any term of τ-subtype in any context where a τ-typed term is expected. The corresponding semantical rule is called *subsumption*.

We distinguish between *width subtyping* and *depth subtyping*. The former refers to allowing object types with more methods to be used in contexts where objects which fewer methods are expected, whereas the second refers to the ability to subtype the method bodies individually and thus create an object subtype. In other words, with depth subtyping, the type of each method may be subtyped, and as a result the entire object type will generate a subtype. This second notion of subtyping is identified with a *covariant object type* in [3].

We will here give a fragment $\Delta_{\leq}$ and a fragment $\Delta_{\nu\leq}$ of a conceived typed object calculi, e.g. **S** or **FOb** of [3]. The first of these adds standard subtyping judgements (subsumption, reflexivity, transitivity, variable subtyping, and $\top$-subtyping).

The rules for reflexivity and transitivity establishes that the subtype relation is a partial order on types. The rule for $\top$ establishes that there exists one type which is the subtype of any other type. The rule S -X projects out a subtype constraint from the type context. The rule for subsumption states that a values of subtype to τ is also of type τ.

Definition 2.3.3 ($\Delta_{\leq}$)

$$\text{S R}\ \frac{\Theta \vdash \tau}{\Theta \vdash \tau \leq \tau} \qquad \text{S T}\ \frac{\Theta \vdash \alpha \leq \beta \qquad \Theta \vdash \beta \leq \gamma}{\Theta \vdash \alpha \leq \gamma} \qquad \text{S -}\top\ \frac{\Theta, \Gamma \vdash \tau}{\Theta, \Gamma \vdash \tau \leq \top}$$

$$\text{S X}\ \frac{\vdash \Theta, X \leq \tau, \Theta'}{\Theta, X \leq \tau, \Theta' \vdash X \leq \tau} \qquad \text{S}\ \frac{\Theta, \Gamma \vdash m : \alpha \qquad \Theta \vdash \alpha \leq \beta}{\Theta, \Gamma \vdash m : \beta} \qquad \text{T T}\ \frac{\vdash \Theta}{\Theta \vdash \top}$$

We will consider an object calculus which extends the above rules with three additional rules based on *variance annotations*. A variance annotation is one of the symbols $^{\circ}, ^{+}, ^{-}$ which makes precise how a type can be subtyped (details will be given in a later section).

Definition 2.3.4 ($\Delta_{\leq \upsilon}$)

$$\text{S I}\ \frac{\Theta \vdash \tau}{\Theta \vdash {}^{\circ}\tau \leq {}^{\circ}\tau} \qquad \text{S C}\ \frac{\Theta \vdash \alpha \leq \beta \qquad \upsilon \in \{^{\circ}, ^{-}\}}{\Theta \vdash \upsilon\alpha \leq {}^{+}\beta} \qquad \text{S C}\ \frac{\Theta \vdash \beta \leq \alpha \qquad \upsilon \in \{^{\circ}, ^{-}\}}{\Theta \vdash \upsilon\alpha \leq {}^{-}\beta}$$

Recursive Types

Next we will consider recursive types and their interplay with subtyping. The interaction between these two notions is "delicate". The complication is that the fixed point operator μ binds a type variable X. This type variable can occur positively and/or negatively in a type τ. For example, X occurs negatively in the body of $\mu(X)X \to Y$, the function type with domain X and codomain Y. The problem with this situation is apparent from the following example:

$$\begin{aligned} P_1 &\triangleq \mu(X)[x : Int, mv_x : Int \to X] \\ P_2 &\triangleq \mu(X)[x, y : Int, mv_x, mv_y : Int \to X] \end{aligned}$$

In object calculi (e.g. $\mathbf{FOb}_{1<:\mu}$ of [3]) we derive an inconsistency from assuming $P_2 <: P_1$ [3], i.e. the inclusion of this rule is unsound (its inclusion invalidates type soundness). The reason for this is that there is a contravariant occurrence hiding in the object type, since every method receives X as an implicit argument (the self variable). The rule (S R) in definition 2.3.5 is still sound, because in $\mathbf{FOb}_{1<:\mu}$ we require all method bodies to be invariant with regard to the subtype relation (i.e. we can neither specialise a type expression occurring in an individual method, nor generalise it). Therefore, no interesting subtype

Definition 2.3.5 (Δ_μ [3])

T R

$$\frac{\langle \Theta, X \rangle \vdash \tau}{\Theta \vdash \mu(X)\tau}$$

S R

$$\frac{\Theta \vdash \mu(Y)\tau \qquad \Theta \vdash \mu(X)\tau \qquad \langle \Theta, Y, X \preceq Y \rangle \vdash \alpha \preceq \beta}{\Theta, \Gamma \vdash \mu(X)\alpha \preceq \mu(Y)\beta}$$

V

where $X \notin \Theta$

$$\frac{\Theta, \Gamma \vdash m : \tau\{\mu(X)\tau/X\}}{\Theta, \Gamma \vdash inn_{\mu(X)\tau}(m) : \mu(X)\tau}$$

V

$$\frac{\Theta, \Gamma \vdash m : \mu(X)\tau}{\Theta, \Gamma \vdash out(m) : \tau\{\mu(X)\tau/X\}}$$

relations are established for object types using this rule (at least when we have a negatively occurring self type, which must be invariant). In order to go around this problem, we will review another calculus with primitive covariant self type in a later section. We give the rules for recursive types in definition 2.3.5.

2.4 Object Calculi and other Foundations

It is common to distinguish between object-based and class-based programming, where class-based programming is perhaps the most well-known kind (represented by languages such as Java, C++, Smalltalk, Simula, and Eiffel). In this thesis we will entirely consider object-based programming (including delegation-based or prototype-based programming, and represented by languages such as Emerald and Self) since languages in this family can be regarded as more foundational (classed-based features can be reduced to object-based features e.g. on object calculus). In the systems we study, class-based languages are encoded as an often straightforward special case. We will use the term *object-oriented* to mean the family of class-based or object-based languages. In this section we will survey some characteristics typical to the programming languages in this family.

Following [34] and [3], we consider the following characteristics as typical for an object-oriented language:

- **Encapsulation**: an object typically contains a local state together with operations acting on that state. In some cases, such as for object calculi, the distinction between local state and operations are blurred, but the main idea is that a single object encapsulates both data and actions on that particular data. Typically there are ways of hiding selected components of the object, e.g. the components that should be conceived as the local state, and only present an interface which allows to abstractly operate on those hidden (protected) components.

- **Dynamic dispatch**: when we operate on an object by invoking some of its methods, the actual selection of an appropriate method is taken dynamically. This at least

means that it is not statically decided that an object of type τ has a particular set of method bodies. In our setting, it additionally means that the method bodies are allowed to change at run-time and that invocations of methods will refer to the current method bodies. There exists a generalisation of method invocation where the the selection of method may depend on the arguments passed to a method. Such generalisations (known as multiple dispatch) are not considered in this thesis.

- **Inheritance**: it is allowed for an object to copy method bodies from another object. In class-based languages this may happen statically, whereas in the object calculi that we use in this thesis, method updates provide inheritance and run-time (but there are some restrictions due to the unsoundness of arbitrary method extraction).

- **Subtyping and subsumption**: subtyping is a relation on the universe of types that formally defines when we are entitled to replace terms of type τ with terms of some other subtype σ of τ. Those types that satisfy such subsumption are said to be in subtype relation, written $\sigma <: \tau$.

Given the above five characteristic features, one is entitled to ask what distinguishes object-oriented programming languages from, for example, functional programming languages such as Standard ML or Haskell. We will see in the next sections that although there exists encodings into typed lambda calculi (thus functional languages), these encodings are often far from trivial. The main problem are the presence of subtyping, and the recursion inherent in objects (i.e. the ability of one method to refer to other methods inside the same object).

This thesis takes Abadi and Cardelli's *object calculi* [1, 2, 3] as a viable class of formal models for object-oriented programming, i.e. as a foundation that satisfies all of the characteristic we have given for an object-oriented programming language. In the remainder of this section we will describe these object calculi in some detail (particularly those calculi in this family that we will be most interested in, the first-order typed calculi).

Abadi and Cardelli's object calculi (hence forth referred to just as "object calculi") are also contrasted to Mitchell and Fisher's *lambda calculus of objects*. This is really also a family of calculi that has undergone several refinements and extensions over the last ten years, ranging from the original system of Mitchell [58], the refined system of [32], the system of Fisher's PhD thesis [31], and more recent extensions with subtyping and matching [50, 14, 13]. These two approaches embody axiomatic approaches to object-oriented foundations, and are in a later subsection contrasted to *encodings* into typed lambda calculi.

Superficially, the differences between object calculi and lambda calculus of objects can be summarised in the following table:

		Lambda Calculus of Objects	
	Object Calculus [3]	[31]	[12]
Typing	*Church-style*	*Curry-style*	*Church-style*
Overriding	*Method update*	*Method update*	*Method update*
Extension	*None*	*Method addition*	*Method addition*
Subtyping	*Width/Depth*	*None*	*Width**

* *subject to additional constraints*

The table compares a particular object calculus, namely system **S** of [3] with two different systems of lambda calculi of objects. However, recent work [12, 49] show that the two systems can be unified in a single Church-style system with method update, method addition, and limited subtyping.

Object Calculi

In section we will give the formal definition of system **S** of [3]. This is an object calculus with subtyping, variance annotations, structural typing rules, and a powerful covariant self type. The presentation is basically Abadi and Cardelli's original presentation, The only difference is that we have replaced the unit type with an empty object type, separate the type context from the term context, and have a type judgement of the form $X \preceq \top$ instead of X, i.e. we avoid type judgements by using subtype judgement also in the case of no subtype bound on variable. I.e. subtype judgements generalise type judgements.

Types and Terms

We will now formally define system **S** from [3]. This system embodies complicated recursive types (although they are implicit in the semantics, rather than given using fixed point binders). An object type may recursively refer to its own type σ (the self type [3], also called *MyType* [18]). This means that object types are abstracted over their self type σ. In addition, objects are abstracted over the value of self, although the actual self type is also abstracted (it may in fact be a subtype in a particular context). This is termed a primitive covariant self type [3] and also *MyType* polymorphism [18]. It is a crucial feature to allow object types to have proper subtypes also in cases where they have σ-valued methods (i.e. methods that return the self type).

To make the presentation more convenient, we follow [3] and write $[\ell_i : \tau_i]^{i\in I}$ for $[\ell_1 : \tau_1, ..., \ell_n : \tau_n]$ with $n \in \mathbb{N}$ and equate object types which are equivalent under permutation of the order of labels.

Definition 2.4.1 (S-types)

The set $\mathcal{T}_{\mathbf{S}}$ is defined by induction with

τ	::=	X	type variable
		$\top$	greatest type
		$Obj(X)[\ell_i \upsilon_i : \tau_i(X)]^{i\in I}$	object types (ℓ_i distinct)

where $X \in V$ (where V is some infinite set of type variables) and for each i, $\ell_i \in L$ (L some infinite set of labels, for which meta notation ℓ_i is used), $\nu_i \in \{\circ, +, -\}$ (the variance annotations).

Except for the type $\top$ which is inhabited by any possible value (it is the supertype of all other types), the only other types in this system are object types, of which the empty object type, written $Obj(X)[]$ is a special case.

Next, figure 2.4.2 gives the syntax for the terms in **S**. The terms are built up from variables x_i, objects (the empty object $Obj(X = \sigma)[]$ is written separately for clarity of its syntax), and method updates and invocations.

Definition 2.4.2 (Syntax of S-terms)

The set $\mathcal{M}_{\mathbf{S}}$ is defined by induction with

m	::=	$Obj(X = \sigma)[]$	empty object
		x_i	term variables
		$Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I}$	object (ℓ_i distinct)
		$m_1.l \Leftarrow (Y \preceq \sigma, y : Y)\varsigma(x : Y)m_2$	method update
		$m.l$	method invocation

where for each i, $x_i \in U$, $X \in V$, $\sigma, \tau_i \in \mathcal{T}_{\mathbf{S}}$, and $\ell_i \in L$.

An object has the form $Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I}$ where σ is some instantiation of the self type. Variance annotations are present in the associated type. Method update is written with the $\Leftarrow$ operator, and takes in addition to a new method body also the old self (denoted y) and the unknown self type Y. Finally, method invocations are written using dot notation, and corresponds to substituting the current self for the argument $x : X$ in the associated method body. The precise meaning will appear in the associated reduction rules.

Next, we give the definition of free variables for a term and for a type (scoping), and the substitution of a term for a free variable occurring in a term.

System **S** decorates the labels of any object type with a variance annotation. The annotations are $^+, ^-, ^\circ$ for positive, negative, and invariant components, and is interpreted as admitting subtyping, admitting reversed subtyping (or *supertyping*), or admitting no subtyping at all. Figure 2.4.5 gives an extended definition of a predicate determining if a variable occurs with a given variance in some given **S**-type.

This notion of variant occurrences should be read as follows: $B\{X^+\}$ means that X occurs *at most* positively in B, and similarly $B\{X^-\}$ means that X occurs *at most* negatively in B.

Given the syntax for types and type-annotated terms, we are now ready to give the judgements for **S** which determine the well-typed terms and the well-formed types. These are shown in figure 2.4.6. Since we already have given certain fragments of these judgements, we simply refer to these fragments in the present definition.

We will give explanations for the rules in $\Delta_{\mathbf{S}}$ excluding the fragments which have already been given some explanation in a previous section. The rule T O makes

Definition 2.4.3 (Object Scoping)

$$
\begin{array}{ll}
FV(\varsigma(x:\tau)b) & = FV(b) \setminus \{x\} \\
FV(x) & = FV(x) \\
FV(Obj(X=\sigma)[\ell_i = \varsigma(x_i:X)b_i]^{i\in I}) & = \bigcup_{i\in I} FV(\varsigma(x:\tau_i)b_i) \\
FV(m.\ell_i) & = FV(m) \\
FV(m.\ell \Leftarrow \varsigma(x:\tau)b) & = FV(m) \cup FV(\varsigma(x:\tau)b)
\end{array}
$$

Definition 2.4.4 (Object Substitution)

$$
\begin{array}{lcl}
(\varsigma(x:\tau)b)\{c/x\} & = & \varsigma(y:\tau)(b\{c/x\}) \\
\quad \textit{where } y \neq x, y \notin FV(\varsigma(x:\tau)b), y \notin FV(c) & & \\
x\{c/x\} & = & c \\
y\{c/x\} & = & x \textit{ for } y \neq x \\
Obj(X=\sigma)[\ell_i = \varsigma(x_i:X)b_i]^{i\in I}\{c/x\} & = & Obj(X=\sigma)[\ell_i = (\varsigma(x_i:\tau_i)b_i)\{c/x\}^{i\in I}] \\
(m.\ell)\{c/x\} & = & m\{c/x\}.\ell \\
(m.\ell \Leftarrow \varsigma(x:\tau)b & = & (m\{c/x\}).\ell \Leftarrow ((\varsigma(x:\tau)b)\{c/x\})
\end{array}
$$

precise how to form an object type from legal types of method bodies. Note that the variable X must occur positively in all body types τ_i.

Similarly, the rule V O makes precise how to form an object term from well-formed method bodies of correct type. The structure of these two first rules are the same for all typed object calculi we will consider in this thesis.

The rule V S is a *structural* rule because it makes assumptions about the structure of object types [3]. The rule shows how the self type variable X is being replaced by a known type σ which is assumed to also be an object type. This assumption is operationally sound, but one must take care for it to hold in a denotational model [3]. This known type σ is allowed to be any subtype of the true type σ'. The substitution allows for the return type of a method to be parametric. In this case a method can, for example, return something of type σ. It is this form of parametricity that gives **S** its notion of "primitive covariant" self type.

The rule V U is a bit more involved due to the possibility of updating an object which has been subsumed, and due to the fact that a method body may want to refer to the previous method body. This means that we must require the updating method body to work with the partially known self type, and again we have a notion of parametricity inherent in **S**. It also explains the syntax for method update which included $y : Y$ for the old self. The rule V U assumes that the true type of the object to be updated is σ'. However, we are updating this object in a context where it is subsumed to type σ, a subtype of σ'. The parametricity required for the self-returning method body is as follows: the method must return the self $x : X$, the old self $x : \sigma$, or a modification of these.

The rule S O allows a "wider" object to be subsumed for a "narrower" object.

Definition 2.4.5 (Variant Occurrences)

$Y\{X^+\}$	whether $X = Y$ or $X \neq Y$
$\top\{X^+\}$	always
$Obj(Y)[\ell_i \upsilon_i : \tau_i(X)]^{i\in I}\{X^+\}$	if $X = Y$, or for all $i \in I$:
	if $\upsilon_i \equiv {}^+$, then $B_i\{X^+\}$
	if $\upsilon_i \equiv {}^-$, then $B_i\{X^-\}$
	if $\upsilon_i \equiv {}^\circ$, then $X \notin FV(B_i)$
$Y\{X^-\}$	if $X \neq Y$
$\top\{X^-\}$	always
$Obj(Y)[\ell_i \upsilon_i : \tau_i(X)]^{i\in I}\{X^-\}$	if $X = Y$, or for all $i \in I$:
	if $\upsilon_i \equiv {}^+$, then $B_i\{X^-\}$
	if $\upsilon_i \equiv {}^-$, then $B_i\{X^+\}$
	if $\upsilon_i \equiv {}^\circ$, then $X \notin FV(B_i)$
$A\{X^\circ\}$	if neither $A\{X^+\}$ nor $A\{X^-\}$

The length here refers to the number of methods. In addition, variance annotations allow us to subtype an object type while changing also the type of individual method bodies, provided that the variance restrictions are satisfied.

We say that a syntactically correct term in **S** is well-typed if there exist well-formed contexts Θ, Γ and a well-formed type $\Theta \vdash \tau$ such that $\Theta, \Gamma \vdash m : \tau$ is derivable. We let $\mathcal{M}_{\mathbf{S}}$ denote the set of well-typed terms up to permutations of method labels.

Despite the failure of unique types (due to subtyping), the following lemma for minimality of types holds for **S**. Unfortunately, with the inclusion of variant types (in a later section) we will fail to have the unique type property even without subtyping.

Lemma 2.4.7 (Minimal Type) *If $\vdash m : \tau$, then there exists a type σ such that $\vdash m : \sigma$ and, for any τ', if $\vdash m : \tau'$, then $\vdash \sigma \leq \tau'$.*

This is proved by induction on the derivations of typing judgements [3]. Next, the definition of substitution respects types:

Lemma 2.4.8 (Substitution) *If $\Theta, \langle \Gamma, x : \tau' \rangle \vdash t : \tau$ and $\Theta, \Gamma \vdash t' : \tau'$ are derivable then so is $\Theta, \Gamma \vdash t\{t'/x\} : \tau$.*

Again, the proof is by induction. The following result establishes type soundness for System **S** (the progress lemma is omitted):

Lemma 2.4.9 (Subject Reduction) *If $\vdash m : \tau$ and $m \rightsquigarrow v$, then $\vdash v : \tau$.*

The proof is an induction on $m \rightsquigarrow v$, c.f. [3].

Definition 2.4.6 (S-judgements - Δ_S)

All the judgements for system **S** is given by $\Delta_{\vdash}, \Delta_{\upsilon\leq}$ together with the following additional rules:

$$\text{T}\quad\text{O}\qquad \frac{\ell_i \text{ distinct}, \upsilon_i \in \{^\circ, ^-, ^+\} \qquad \langle \Theta, X\rangle, \Gamma \vdash \tau_i(X^+) \qquad i \in I}{\Theta \vdash Obj(X)[\ell_i \upsilon_i : \tau_i(X)]^{i\in I}}$$

$$\text{V}\quad\text{O}\qquad \frac{\sigma \equiv Obj(X)[\ell_i \upsilon_i : \tau_i(X)]^{i\in I} \qquad \Theta, \langle \Gamma, x_i : \sigma\rangle \vdash b_i\{\sigma\} : \tau_i\{\sigma\} \qquad \forall i \in I}{\Theta, \Gamma \vdash Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I} : \sigma}$$

$$\text{S}\quad\text{O}\qquad \frac{\alpha \equiv Obj(X)[\ell_i \upsilon_i : \tau_i(X)]^{i\in I\cup J} \qquad \beta \equiv Obj(X)[\ell_i \upsilon_i' : \tau_i'(X)]^{i\in I} \qquad \Theta \vdash \alpha \qquad \Theta \vdash \beta \qquad \langle \Theta, Y \preceq \alpha\rangle \vdash \upsilon_i \tau_i\{Y\} \preceq \upsilon_i' \tau_i'\{Y\}}{\Theta \vdash \alpha \preceq \beta}$$

$$\text{V}\quad\text{S}\qquad \frac{\sigma' \equiv Obj(X)[\ell_i \upsilon_i : \tau_i(X)]^{i\in I} \qquad \Theta, \Gamma \vdash m : \sigma \qquad \Theta \vdash \sigma \preceq \sigma' \qquad \upsilon_j \in \{^\circ, ^+\} \qquad j \in I}{m.\ell_j : \tau_j\{\sigma\}}$$

$$\text{V}\quad\text{U}\qquad \frac{\sigma' \equiv Obj(X)[\ell_i \upsilon_i : \tau_i(X)]^{i\in I} \qquad \Theta, \Gamma \vdash m : \sigma \qquad \Theta \vdash \sigma \preceq \sigma' \qquad \langle \Theta, Y \preceq \sigma\rangle, \langle \Gamma, y : Y, x : Y\rangle \vdash b : \tau_j\{Y\} \qquad \upsilon_j \in \{^\circ, ^-\} \qquad j \in I}{\Theta, \Gamma \vdash m.\ell_j \Leftarrow (Y \preceq \sigma, y : Y)\varsigma(x : Y)b : \sigma}$$

Operational Semantics

The operational semantics for **S** is given in [3]. We give an overview of the reduction rules are given in figure 2.4.10. These rules will reappear for a related operational semantics in chapter 3, but in that setting there is no subtyping so the operational meaning will be slightly different. Here, all rules involve substitutions of the self parameters. The rule R U replaces the old self, whereas R S replaces the self variable inside a particular method body. Also type substitutions are carried out to handle presence of type variables in objects (replacing the self type variable X with the actual type). The other rules are trivial (variables, and the statement that an object is a canonical form).

We have now completely defined the syntax and evaluation rules for **S**. We could extend this system with quantifiers, but **S** is already rich enough to express many examples. In fact, there is, as we mentioned, already a notion of parametricity in **S**: the self type. However, adding parametric types by means of universal type quantifiers gives the system better features with regard to inheritance and method reuse.

S satisfies many "nice" properties. In particular, it has type soundness and satisfies various substitution lemmas [3]. The details are omitted in this survey, but we will prove

Definition 2.4.10 (Operational Semantics for S)

$$\text{R}\quad x \rightsquigarrow x$$

$$\text{R O}\quad v \rightsquigarrow v \quad (v \equiv Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I})$$

$$\text{R S}\quad \frac{m \rightsquigarrow v' \qquad b_i\{v', \sigma\} \rightsquigarrow v}{m.\ell_i \rightsquigarrow v} \quad (v' \equiv Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I})$$

$$\text{R U}\quad \frac{m \rightsquigarrow v \qquad j \in I}{m.\ell_j \Leftarrow (Y \le \sigma', y : Y)\varsigma(x : Y)b\{Y, y\} \rightsquigarrow Obj(X = \sigma)[l_j = \varsigma(x : X)b\{X, v\}, l_i = \varsigma(x_i : X)b_i]^{i\in I-\{j\}}} \quad (v \equiv Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I})$$

soundness and adequacy for an interpretation of a simplified version of **S** (dropping subtyping) in a later chapter of this thesis.

We conclude this section by giving a small example of a "program" expressed in **S**:

Example 2.4.11 *A movable point and coordinate can be defined elegantly in* **S***:*

$$\begin{array}{lcl} \mathit{Point} & \triangleq & \mathit{Obj}(X)[x : \mathit{Int},\ mv_x : \mathit{Int} \to X] \\ \mathit{Coordinate} & \triangleq & \mathit{Obj}(X)[x, y : \mathit{Int},\ mv_x : \mathit{Int} \to X,\ mv_y : \mathit{Int} \to X] \end{array}$$

Our subtyping rules can be used to prove Coordinate $\le$ *Point (simply by means of the* S O *rule since Coordinate contains more methods, and all "inherited" methods have equal types, i.e. invariance). Now, we may define a value of type Coordinate:*

$$\mathit{origin} \triangleq \mathit{Obj}(X = \mathit{Coordinate})[x = 0,\ y = 0\ , mv_x = ...,\ mv_y = ...]$$

At some point, we may update the move method to change the behavior of origin:

$$\mathit{origin}.mv_x \Leftarrow (Y \le \sigma, y : Y)\varsigma(x : Y)t$$

The method body t may use the old self y and therefore refer to the previous definition of mv_x *to allow incremental change of an object. Further, a method is always given the present self as a parameter. This self parameter x has a parametric type. Its type Y may be any subtype of Coordinate.*

Lambda Calculus of Objects

Lambda calculus of objects [58, 32] is similar to System **S**, but has a rule for method addition which means that we can extend an object with more methods. This is also makes the

type system more complicated and technical restrictions must be added to support subtyping. The original system of [58, 32] did not support any form of subtyping. However, in a later work, Fisher and Mitchell [33] proposed a system with two different syntactical entities: objects and prototypes. Prototypes are objects which support method addition, but not subtyping. In this way, they could get around the problem with subtyping. Another idea was studied by Bono and Liquori [15]. Here the type system was extended with labeled types to allow limited subtyping. The restriction is that we can apply width subtyping (i.e., hiding a method from an object type), if and only if this method is not used by any of the other methods.

Other Encodings

Object-oriented calculi are frequently studied by encoding them into more well-understood target calculi, typically some flavor of typed lambda calculus. Unfortunately, encoding object types and subtyping into a target language is a difficult task and may fail to explain object-oriented features (as expressed in the above two *axiomatic* approaches) as more primitive notions in typed lambda calculus [3]. Nevertheless, a considerable amount of research has proposed encodings of object types and subtyping into a plethora of target languages. These encodings are surveyed in [19] and also in [34, 59, 62].

Our starting point is second order typed lambda calculus, called System F. This system was developed by Girard [38] (in logic) and independently reinvented by Reynolds [69] (for programming languages). System F, while having impredicative polymorphic types, can encode inductive types (but not full recursive types) using an encoding [11, 48].

System F can also encode subtyping as coercions. However, the author believes, together with Philip Wadler and Benjamin C. Pierce (personal communication, 2005), that it is an open problem if System F can simultaneously encode inductive types and coercions between them.

Cardelli [22] proposed an extension of this system to support subtyping. The resulting system is called $F_{<:}$. In addition to subtyping it has bounded quantification (e.g. $\forall(X \leq \tau)\sigma$ is a type). When we discuss encodings of object calculi below, some of the encodings assume recursive types in addition to inductive types, and fixed point operators on elements/values. In other words, some encodings may require non-trivial extensions of $F_{<:}$.

The first approach of giving semantics to objects and object types was the *recursive record semantics* [20, 21, 24, 25, 26, 16]. This semantics is based on a typed lambda calculus with records and record types, and objects are values which are defined using a fixed point operator *on terms* (i.e. to create an object we apply the fixed point operator on a record-valued function abstracted on self). For example, an object *Point* would be defined in Haskell like this:

$$\begin{array}{l}\textbf{data}\ \mathit{Point} = \mathit{Point}\{x :: \mathit{Integer}, dx :: \mathit{Integer} \to \mathit{Integer}\}\\ \mathit{fix} :: (a \to a) \to a\\ \mathit{fix}\ f = \textbf{let}\ x = f\ x\ \textbf{in}\ x\\ \mathit{makePoint} :: \mathit{Integer} \to \mathit{Point}\\ \mathit{makePoint}\ x0 = \mathit{fix}\ (\mathit{pointF}\ x0)\end{array}$$

$$
\begin{array}{l}
\textbf{where} \\
\quad \mathit{pointF} :: \mathit{Integer} \to \mathit{Point} \to \mathit{Point} \\
\quad \mathit{pointF}\ x1\ s = \mathit{Point}\{x = x1, dx = \lambda x2 \to ((x\ s) - x2)\}
\end{array}
$$

In this encoding of objects, there is no immediate need for recursive types since references to "self" are captured under the *fix* combinator. However, and as pointed out by Cardelli [20], if the *Point* type contains also a method that returns a new *Point* value, we still will require a recursive type. Fortunately, the required recursive type has a "nice" form (the self variable occurs positively), and subtyping can be validated [3]. This model also can be extended to handle classes (the extended model, which was largely developed in [75, 25, 26], is sometimes referred to as the "generator model" [12]). Less satisfactory with the recursive record approach is the hard-wiring of the self variable that occurs when an object is created. As a consequence only internal or *self-inflicted* method updates can be modelled with this approach [3]. Another limitation is that recursive records cannot be combined with method addition [34].

A different approach is taken in the *self-application* encoding of objects, which was proposed for Smalltalk-80 [45, 46] some years after Cardelli had developed the recursive record model. An object in this approach is abstracted from the self variable (using a lambda or sigma binder), and remains so until a method invocation occurs, at which point the self is applied to the invoked method. Both lambda calculus of objects and object calculi are variations of this approach, and an object type is required to be a special form of recursive type. However, this encoding fails to support subtyping when we interpret lambda calculus of objects or object calculi into $F_{<:}$. Nevertheless, this thesis is dedicated to precisely this form of encoding, but in a denotational setting, so we will continue to study it in the following chapters. This gives a denotational setting where coercions and thus subtyping can later be studied.

In order to translate both method update and subtyping, Abadi et al [4] proposed the split-method interpretation, which extends recursive record encoding such that also external method updates are possible.

$$
\begin{array}{l}
\textbf{data}\ \mathit{Point} = \mathit{Point}\{x \quad :: \mathit{Integer}, dx :: \mathit{Integer} \to \mathit{Integer}, \\
\qquad\qquad\qquad\qquad \mathit{upd}_x\ :: (\mathit{Point} \to \mathit{Integer}) \to \mathit{Point}, \\
\qquad\qquad\qquad\qquad \mathit{upd}_{dx} :: (\mathit{Point} \to (\mathit{Integer} \to \mathit{Integer})) \to \mathit{Point}\} \\
\mathit{createF} :: \mathit{Point} \to (\mathit{Point} \to \mathit{Integer}) \to \\
\quad (\mathit{Point} \to (\mathit{Integer} \to \mathit{Integer})) \to \mathit{Point} \\
\mathit{createF}\ \mathit{self}\ b_x\ b_{dx} = \mathit{Point}\{x = b_x\ \mathit{self}, dx = b_{dx}\ \mathit{self}, \\
\quad \mathit{upd}_x\ = \lambda b \to \mathit{Point}\{x = b\ \mathit{self}, dx = b_{dx}\ \mathit{self}\} \\
\quad \mathit{upd}_{dx} = \lambda b \to \mathit{Point}\{x = b_x\ \mathit{self}, dx = b\ \mathit{self}\}\} \\
\mathit{create} = \mathit{fix}\ \mathit{createF} \\
\quad \textbf{where}\ \mathit{fix}\ f = \textbf{let}\ x = f\ x\ \textbf{in}\ x
\end{array}
$$

Other interpretations are typically based on interpreting objects into typed lambda calculus (see [19] for a more detailed survey). However, these are typically restricted to the class-based case where method update is inhibited.

Chapter 3

Soundness and Adequacy of System $\mathbf{S}^-$

In this chapter we will define System $\mathbf{S}^-$, a typed object calculus without subtyping, and interpret this calculus into Fixed-Point Calculus [65, 66, 64] using an eager self-application encoding, and prove soundness and adequacy of $\mathbf{S}^-$ with respect to this interpretation.

Self-application encodings have been studied for $F_{<:}$, but unfortunately $F_{<:}$ is too weak as a basis for object-oriented programming languages. $F_{<:}$ must be extended with a fixed-point operator on terms, recursive types, and either F-bounded quantification or higher-order functions from types to types, and some sort of record extension operator [17]. The reason for this requirement is that recursive types and subtyping require special techniques to work together in $F_{<:}$. Therefore we instead consider an encoding into FPC which has some of these features (recursive types and recursively-defined elements).

3.1 First-Order Object Calculus without Subtyping

Abadi and Cardelli have developed a family of Object Calculi, some of which are more powerful than others, e.g. by having subtyping, recursive types, variance annotations, polymorphism, or *Self*-type in addition to the standard first-order fragment. We will focus on first-order calculi and one particular higher-order calculus which has only the powerful *Self*-type and no polymorphism. Table 3.1.1 gives an overview of the calculi that we will study in this chapter, some similar or simpler typed object calculi, and the System **S** from the introductory chapter.

The table shows, for example, that variance annotations are not considered at all in this chapter (this is however no fundamental limitation since such annotations can easily be adjoined to an extended system). The main systems under consideration is $\mathbf{S}^-$, which will be defined in this section and which is an adaptation of **S** of [3]. This system has the *Obj*-binder, but no subtyping and thus no real *Self*-type. This system is $\mathbf{FOb}_{1\mu}$ of Abadi and Cardelli [3], but with *Obj*-binder instead of the μ-binder, and with the extensions listed in the diagram. We have chosen $\mathbf{S}^-$ instead of $\mathbf{FOb}_{1\mu}$ because $\mathbf{S}^-$ contains a larger subset of the syntax and rules for **S**, which is one of the most powerful typed object calculi. Since $\mathbf{S}^-$ is endowed with products and coproducts, FPC will contain a subset of the rules of $\mathbf{S}^-$.

Definition 3.1.1 Object Calculi

	$\mathbf{FOb_1}$	$\mathbf{FOb_{1\leq}}$	$\mathbf{FOb_{1\leq\mu}}$	$\mathbf{S}^-$	$\mathbf{S}$
Subtyping		•	•		•
Recursive types			•	•	•
Obj-binder				•	•
Self-type					•
Variance ann.					•
Products				•	
Coproducts				•	
Functions	•	•	•	•	

Note that $\mathbf{S}^-$ is not a subset of **S**, but contains some extensions such functions. These extensions can however also be given to System **S**, although Abadi and Cardelli's original presentation of **S** does not include them.

We will now define $\mathbf{S}^-$ and give some simple examples. We choose *n*-ary products and coproducts to simplify these examples. We give an operational semantics with a clear notion of values. Our choice of an operational approach permits us to prove soundness and adequacy with respect to a denotational model. These results could not be proven were we to have used the reduction based approach as certain reductions are in fact unsound. The reason for this is that a reduction-based semantics admits a degree of non-determinism in evaluations that invalidates the soundness proof. Notably, an object with some terminating methods and some non-terminating methods, is interpreted as a product of functions, such that even the terminating method may become non-terminating under some reduction strategies in FPC.

We assume a countable set of method labels L, type variables V, and term variables U. The types of $\mathbf{S}^-$ are given in definition 3.1.2. Notationally, we write $[\ell_i : \tau_i]^{i\in I}$ for $[\ell_1 : \tau_1, ..., \ell_n : \tau_n]$ with $n \in \mathbb{N}$ and equate object types which are equivalent under permutation of the order of labels or under the obvious notion of α-equivalence induced by the type binder Obj. We introduce shorthand $\tau_1 \times \tau_2 = \prod_{i\in\{1,2\}} \tau_i$ and similarly $\tau_1 + \tau_2 = \coprod_{i\in\{1,2\}} \tau_i$ for binary products and coproducts.

Definition 3.1.2 also gives the pre-terms of $\mathbf{S}^-$. We identify pre-terms which are equal up to the order of method label or are equivalent under the obvious notion of α-equivalence induced by the term-binders λ, ς, and *case* and the type-binder Obj. We use the standard definition of substitution which can be found in Abadi and Cardelli, and write $m\{a/x\}$ to mean that a is substituted for all free occurrences of x in m [3]. Further, $m(x)$ means x may occur free in m. We use similar notation for the substitution of types for type variables in both types and terms. When clear from the context, we eliminate the type or term variable being substituted for and simply write $m\{a\}$ and $m\{\tau\}$.

A type judgement consists of a sequence of distinct type variables (a type context) together with a type whose free type variables appear in the sequence. The formal definition of type contexts appear in Definition 3.1.3.

Definition 3.1.2 (Syntax of $\mathbf{S}^-$)

Syntax for Types
The set $\mathcal{T}_{\varsigma\mathbf{FOb}}$ is defined by induction with

τ	::=	X	type
		1	terminal type
		$\prod_{i\in I}\tau_i$	product types
		$\coprod_{i\in I}\tau_i$	coproduct types
		$\tau_1 \to \tau_2$	function types
		$Obj(X)[\ell_i : \tau_i(X)]^{i\in I}$	object types (ℓ_i distinct)

where $X \in V$, and for each i in a finite set I, $\ell_i \in L$ are pairwise distinct.

Syntax for Terms
The set $\mathcal{M}_{\mathbf{S}}$ is defined by induction with

m	::=	$\star$	unit
		x_i	term variables
		$\langle m_0, ..., m_n\rangle$	tupling
		$\pi_i\ m$	projections
		$case(m_0, x_1.m_1, ..., x_n.m_n)$	case
		$\iota_i\ m$	injections
		$m_0\ (m_1)$	λ-application
		$\lambda x : \tau.m$	λ-abstraction
		$Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I}$	object introduction
		$m_1.\ell \Leftarrow \varsigma(x : \tau)m_2$	method update
		$m.\ell$	method invocation

where for each i, $x_i \in U$, $X \in V$, $\sigma, \tau_i \in \mathcal{T}_{\varsigma\mathbf{FOb}}$, and $\ell_i \in L$.

Here are a couple of examples:

Example 3.1.4 *One may consider representing the Java-like interface*

interface *Point* {**public void** *bump*(); **public int** *val*(); }

as the following type in $\mathbf{S}^-$ *(assuming a type Int exists):*

$$Point = Obj(X)[val : Int, bump : X]$$

Example 3.1.5 *The Java-like interface*

interface *UnLam* {**public void** *bump*(); }

Definition 3.1.3 (Types and Type Contexts in $\mathbf{S}^-$)

Type Contexts
Type contexts are generated by the following rules

$$\frac{}{\vdash \Diamond}\ \mathrm{T\,C\quad E} \qquad \frac{\vdash \Theta}{\vdash \langle \Theta, X\rangle}\ \mathrm{T\,C\quad X}\ \text{where}\ X \in V, X \notin \Theta$$

Well-formed Types
The typing judgments $\Theta \vdash \tau$ are those generated by the following rules:

$$\frac{\vdash \Theta}{\Theta \vdash X}\ \mathrm{T\quad X}\ \text{where}\ X \in \Theta \qquad \frac{\vdash \Theta}{1}\ \mathrm{T\quad U} \qquad \frac{\Theta \vdash \tau_1 \quad \Theta \vdash \tau_2}{\Theta \vdash \tau_1 \to \tau_2}\ \mathrm{T\quad F}$$

$$\frac{\Theta, X \vdash \tau_i \qquad i \in I}{\Theta \vdash Obj(X)[\ell_i : \tau_i(X)]^{i\in I}}\ \mathrm{T\quad O} \qquad \frac{\Theta \vdash \tau_i \qquad i \in I}{\Theta \vdash \boxdot_{i\in I}\tau_i}\ \text{where}\ \boxdot \in \{\textstyle\prod, \coprod\}$$

gives rise to an object type of the form

$$UnLam = Obj(X)[bump : X]$$

Once we have the type judgements, we can define term contexts and then term judgements (Definition 3.1.7). As one would expect, terms are closed under substitution. That is, if $\Theta, \langle \Gamma, x : \tau' \rangle \vdash t : \tau$ and $\Theta, \Gamma \vdash t' : \tau'$ are derivable then so is $\Theta, \Gamma \vdash t\{\!|t'/x|\!\} : \tau$.

Example 3.1.6 *A point whose value is* 0 *and whose bump method adds* 1 *to the value can be represented in* $\mathbf{S}^-$ *as*

$$\begin{array}{rcl} p \triangleq Obj(X = Point)[\quad val & = & \varsigma(x : X)0, \\ \quad bump & = & \varsigma(x : X)x.val \Leftarrow \varsigma(y : X)x.val + 1\ \ ] \end{array}$$

Unlike Java, $\mathbf{S}^-$ makes no distinction between objects and classes. Therefore, a class is represented by an object, which can be cloned or copied into new objects which will (initially at least) have the same methods. There are other differences: object calculus allows methods to be updated, which is impossible in Java, and $\mathbf{S}^-$ has no imperative features. Since we have method updates, there is no need to have separate attributes. Attributes, like *val*, are instead identified with method bodies in which the self variable does not occur.

Definition 3.1.7 ($\mathbf{S}^-$-Typing judgments)

Term Contexts
Well-formed term contexts are given by the rules

$$\frac{\text{C}\quad\text{E}}{\vdash \Theta}{\Theta \vdash \langle\rangle} \qquad \frac{\text{C}}{\Theta \vdash \tau, \Gamma}{\Theta \vdash \langle \Gamma, x : \tau\rangle} \text{ where } x \in U, x \notin \Gamma$$

Well-typed Terms
The typing judgments of $\mathbf{S}^-$ are

$$\frac{\text{V}\quad\text{O}}{\sigma \equiv Obj(X)[\ell_i : \tau_i(X)]^{i\in I} \quad \Theta, \langle \Gamma, x_i : \sigma\rangle \vdash b_i\{\sigma\} : \tau_i\{\sigma\} \qquad \forall i \in I}{\Theta, \Gamma \vdash Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I} : \sigma} \qquad \frac{\text{V}\quad\text{S}}{\Theta, \Gamma \vdash m : Obj(X)[\ell_i : \tau_i(X)]^{i\in I} \qquad \forall i \in I}{\Theta, \Gamma \vdash m.\ell_i : \tau_i\{Obj(X)[\ell_i : \tau_i(X)]^{i\in I}\}}$$

$$\frac{\text{V}\quad\text{U}}{\sigma \equiv Obj(X)[\ell_i : \tau_i(X)]^{i\in I} \quad \Theta, \Gamma \vdash m : \sigma \qquad \Theta, \langle \Gamma, x : \sigma\rangle \vdash b\{\sigma\} : \tau_j\{\sigma\} \quad j \in I}{\Theta, \Gamma \vdash m.\ell_j \Leftarrow \varsigma(x : \sigma)b\{\sigma\} : \sigma}$$

$$\frac{\text{V}\quad\text{P}}{\Theta, \Gamma \vdash a : \prod_{i\in I} \tau_i \;\; j \in I}{\Theta, \Gamma \vdash \pi_j\, a : \tau_j} \qquad \frac{\text{V}\quad\text{P}}{\Theta, \Gamma \vdash a_1 : \tau_1 \qquad ... \qquad \Theta, \Gamma \vdash a_n : \tau_n}{\Theta, \Gamma \vdash \langle a_1, ..., a_n\rangle : \prod_{i\in I} \tau_i}$$

$$\frac{\text{V}\quad\text{S}}{\Theta, \Gamma \vdash a : \tau_j \;\; j \in I}{\Theta, \Gamma \vdash \iota_i\, a : \coprod_{i\in I} \tau_i} \qquad \frac{\text{V}\quad\text{C}}{\Theta, \Gamma \vdash m : \coprod_{i\in I} \sigma_i \qquad \Theta, \langle \Gamma, x_j : \sigma_j\rangle \vdash m_j : \tau \quad j \in I}{\Theta, \Gamma \vdash case(m, x_1.m_1, ..., x_n.m_n) : \tau}$$

$$\frac{\text{V}\quad\text{U}}{\Theta, \Gamma \vdash \star : 1} \qquad \frac{\text{V}}{\Theta \vdash \langle \Gamma, x_i : \tau_i, ...\rangle}{\Theta, \langle \Gamma, x_i : \tau_i, ...\rangle \vdash x_i : \tau_i}$$

$$\frac{\text{V}\quad\text{E}}{\Theta, \Gamma \vdash m_1 : \tau_1 \to \tau_2, m_2 : \tau_1}{\Theta, \Gamma \vdash m_1(m_2) : \tau_2} \qquad \frac{\text{V}\quad\text{F}}{\Theta, \langle \Gamma, x : \tau_1\rangle \vdash b : \tau_2}{\Theta, \Gamma \vdash \lambda x : \tau_1.b : \tau_1 \to \tau_2}$$

We say a pre-term $m \in \mathcal{N}_{\varsigma\mathbf{FOb}}$ is well-typed if there exists well-formed contexts Θ, Γ and a type judgement $\Theta \vdash \tau$ such that $\Theta, \Gamma \vdash m : \tau$ is derivable. We let $\mathcal{M}_{\mathbf{S}}$ denote the set of well-typed terms up to α-equivalence and permutations of method labels.

Operational Semantics

We have now defined the language of $\mathbf{S}^-$, and will give it an operational semantics. The semantics is call-by-value and, in particular, each component of a product must have a value for a projection of the product to attain a value. The rules for the non-object part of the calculus are standard while we feel that those for the object intro, elim and update rules are reasonable, e.g. one does not reduce under the binder in object intro terms and hence all object intro terms are values. This feeling is reinforced by the results we derive later on soundness and adequacy. The values (or normal/canonical forms) are as follows:

$$\begin{array}{rcl} v & ::= & x \mid 1 \mid \iota_i\, v \mid \langle v_1, ..., v_n\rangle \mid \lambda x : \tau.m \mid \\ & & Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)m_i]^{i\in I} \end{array}$$

The actual operational rules are given in Definition 3.1.10. Note that the values are precisely the terms v such that $v \rightsquigarrow v$, where $\rightsquigarrow$ means the reduction relation. This is the statement that values are irreducible in a formal sense. A program p is a term such that for some type τ we have $\vdash p : \tau$, i.e. a well-typed term with empty contexts. The key theorem which means that the implementation of the calculus, as given by its operational semantics, respects compile time type information is the preservation of types as shown in the next theorem.

Theorem 3.1.8 (Preservation) *If t is a well-typed term $\Theta, \Gamma \vdash t : \tau$ such that $t \rightsquigarrow t'$, then $\Theta, \Gamma \vdash t' : \tau$.*

Proof The proof is by induction on the derivation of $t \rightsquigarrow t'$ and is fairly routine. Suppose $\Theta, \Gamma \vdash t : \tau$ and $t \rightsquigarrow t'$. We have omitted trivial cases:

Case (R C): We have $t = case(m, x_1.m_1, ..., x_n.m_n)$ and $\Theta, \Gamma \vdash t : \tau$. Since t is well-typed we have $\Theta, \Gamma \vdash m : \coprod_{i\in I} \sigma_i$ and $\Theta, \langle \Gamma, x : \sigma\rangle \vdash m_i : \tau$ for $i \in I$. We have subderivation $m \rightsquigarrow \iota_k v$ and $m_k\{v/x_k\} \rightsquigarrow t'$. But by induction hypothesis this means, by Lemma 2.4.8, $t' : \tau$.

Case (R P): We have $t = \pi_i(m)$ and $\Theta, \Gamma \vdash t : \tau$, which is to say $m = \langle a_1, ..., a_n\rangle$ for some $\Theta, \Gamma \vdash a_i : \tau_i$. The result follows by induction hypothesis on the required component.

Case (R E): We have $t = m_1(m_2)$ and $\Theta, \Gamma \vdash t : \tau_2$. Therefore $\Theta, \Gamma \vdash m_1 : \tau_1 \to \tau_2$ and $\Theta, \Gamma \vdash m_2 : \tau_1$. That is to say $m_1 = \lambda x : \tau_2.b$. Now for $m_2 \rightsquigarrow v$ we have $\Theta, \Gamma \vdash b\{v/x\} : \tau_2$ and by induction hypothesis $t' : \tau_2$ as required.

Case (V S): we have $t = m.\ell_i$ and $\Theta, \Gamma \vdash t : \tau_i$ for $m = Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I}$ and $\Theta, \Gamma \vdash m : \sigma$ with $\sigma = Obj(X)[\ell_i : \tau_i(X)]^{i\in I}$. For $m \rightsquigarrow v'$ and $b_i\{v', \sigma\} \rightsquigarrow t'$ we have $\Theta, \Gamma \vdash t' : \tau_i\{\sigma\}$ by induction hypothesis.

Case (V O): we have $t = m.\ell_j \Leftarrow \varsigma(x : \sigma).b$ and $t : \sigma$. Since $\Theta, \langle \Gamma, x : \sigma\rangle \vdash b_j\{\sigma\} : \tau_j\{\sigma\}$ we have $\Theta, \Gamma \vdash t' : \sigma$ as required.

□

Corollary 3.1.9 (Type Soundness) $\mathbf{S}^-$ *satisfies type soundness.*

Definition 3.1.10 (Operational Semantics for S^-)

$$\text{R}\quad x \rightsquigarrow x \qquad\qquad \text{R}\quad \text{U}\quad \star \rightsquigarrow \star$$

$$\text{R}\quad \text{P}\quad \frac{m_1 \rightsquigarrow v_1 \quad \ldots \quad m_n \rightsquigarrow v_n}{\langle m_1, \ldots, m_2\rangle \rightsquigarrow \langle v_1, \ldots, v_n\rangle} \qquad \text{R}\quad \text{P}\quad \frac{m \rightsquigarrow \langle v_1, \ldots, v_n\rangle \quad 1 \leqslant i \leqslant n}{\pi_i(m) \rightsquigarrow v_i}$$

$$\text{R}\quad \text{S}\quad \frac{m \rightsquigarrow v}{\iota_j\, m \rightsquigarrow \iota_j\, v} \qquad \text{R}\quad \text{C}\quad \frac{m \rightsquigarrow \iota_j(v) \quad m_j\{v/x_j\} \rightsquigarrow v' \quad j \in [1,n]}{case(m, x_1.m_1, \ldots, x_n.m_n) \rightsquigarrow v'}$$

$$\text{R}\quad \text{F}\quad \lambda x : \tau.m \rightsquigarrow \lambda x : \tau.m \qquad \text{R}\quad \text{E}\quad \frac{m_1 \rightsquigarrow \lambda x : \tau.b \quad m_2 \rightsquigarrow v \quad b\{v/x\} \rightsquigarrow v'}{m_1(m_2) \rightsquigarrow v'}$$

$$\text{R}\quad \text{O}\quad \frac{v \equiv Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I}}{v \rightsquigarrow v} \qquad \text{R}\quad \text{S}\quad \frac{v' \equiv Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I} \quad m \rightsquigarrow v' \quad b_i\{v', \sigma\} \rightsquigarrow v}{m.\ell_i \rightsquigarrow v}$$

$$\text{R}\quad \text{U}\quad \frac{v \equiv Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I} \quad m \rightsquigarrow v}{m.\ell_j \Leftarrow \varsigma(x : \sigma)b\{\sigma\} \rightsquigarrow Obj(X = \sigma)[\ell_i = \varsigma(x : X)b_i, \ell_j = \varsigma(x : X)b]^{i\in I-\{j\}}}$$

where in the last rule we delete the j'th method from v and then add the updated method $\ell_j = \varsigma(x : X)b$.

3.2 Fixed Point Calculus

Our intention is to interpret object types as solutions of certain recursive equations. We do this i) syntactically by translating the Object Calculus into the Fixed-Point Calculus (FPC); and ii) semantically by giving denotational models for the object calculus using some sophisticated categorical models. Note that this a different encoding of objects as recursive types than is found in e.g. the recursive record semantics in the literature, e.g. [20, 3]. Notably, the recursive record semantics would give the following interpretation of the $p : \mathit{Point}$ object given in the previous examples:

$$p = Y\ \lambda p.\langle 0, \langle \pi_1\ p + 1, \pi_2\ p\rangle\rangle$$

where $Y : (\tau{\to}\tau){\to}\tau$ is a fixed point combinator (which can be encoded into FPC). The type of p is $\mu X.\mathit{Int}{\times}X$, but as seen in this example we cannot replace the first component of p without giving a completely new definition of p. We will give p the type $\mu X.(X{\to}\mathit{Int}) \times (X{\to}X)$. This means that p is denoted simply by a product which enjoys the ordinary projections on each component.

Our calculus of recursive types is known in the semantics literature as FPC. This system is originally due to Plotkin [65, 66, 64], but detailed expositions are given e.g. by Gunter [40] and Fiore [29]. FPC intuitively arises from $\mathbf{S}^-$ by deleting the types and terms related to objects and inserting types and terms related to fixed points of mixed variant type constructors. Thus FPC uses the same countable supplies U and V of type and term variables. We summarise the formal rules in Definition 3.2.1.

The notions of substitution, α-congruence, contexts, well-formed types, are all identical, except that we replace object type formation with the following rule for well formed recursive types. The preterms of FPC are exactly those of $\mathbf{S}^-$, omitting all terms derived from the object formation rule, method updates, and method invocation, and adding to the grammar terms of the form $\mathit{inn}_{\mu X.\tau}$ for μ-introduction (V I) and *out* for μ-elimination (V O). The term judgments for FPC are similarly obtained from those of $\mathbf{S}^-$, but the V O , V S and V U rules are replaced by two rules for typing recursive types. Finally, the operational semantics of FPC is obtained by deleting V O terms as values, removing the operational rules for R O , R S and R U and adding the following values and rules from definition 3.2.1 to cope with recursive types.

$$v ::= \ldots \mid \mathit{inn}_{\mu X.\tau}(v)$$

In addition to this *eager* (call-by-value) version of FPC, we will briefly also recall the *lazy* (call-by-name) operational semantics that can be given to this language.

Definition 3.2.1 (Eager FPC)

$$\frac{X \notin \Theta}{\Theta, \Gamma \vdash m : \tau\{\mu X.\tau/X\}}\ \mathrm{V\ I} \qquad \frac{\Theta, \Gamma \vdash m : \mu X.\tau}{\Theta, \Gamma \vdash out(m) : \tau\{\mu X.\tau/X\}}\ \mathrm{V\ O}$$

$$\frac{\langle \Theta, X \rangle \vdash \tau}{\Theta \vdash \mu X.\tau}\ \mathrm{T\ R} \qquad \frac{e \rightsquigarrow v}{inn(e) \rightsquigarrow inn(v)}\ \mathrm{R\ I} \qquad \frac{e \rightsquigarrow inn(v)}{out(e) \rightsquigarrow v}\ \mathrm{R\ O}$$

Definition 3.2.2 (Lazy FPC)

Operational Semantics
The following rules take the place of R P , R C , and R E and all other rules are the same:

$$\frac{m \rightsquigarrow \langle m_1, ..., m_n \rangle \qquad m_i \rightsquigarrow v}{\pi_i(m) \rightsquigarrow v}\ \mathrm{R\ L\ P} \qquad \frac{m \rightsquigarrow \iota_j(k) \quad m_j\{k/x_j\} \rightsquigarrow v' \quad j \in [1, n]}{case(m, x_1.m_1, .., x_n.m_n) \rightsquigarrow v'}\ \mathrm{R\ L\ C}$$

$$\frac{m_1 \rightsquigarrow \lambda x : \tau.b \quad b\{m_2/x\} \rightsquigarrow v}{m_1(m_2) \rightsquigarrow v}\ \mathrm{R\ L\ E}$$

Under a lazy semantics, we have more values than we had in the eager semantics. If $t_i, \lambda x.m$ are closed terms, the values now also include:

$$v ::= \ldots \ \iota_j\, t \ \mid \ \lambda x.m \ \mid \ \langle t_1, \ldots, t_n \rangle \ \mid \ inn_{\mu X.\tau}(v)$$

3.3 Translating Object Calculus in FPC

This section contain a translation of $\mathbf{S}^-$ into (eager) FPC. This translation is at the level of types, terms and operational semantics and we find an excellent fit whereby the operational semantics for FPC is both sound and complete. This allows us to transport the well-understood theory of FPC, in particular its denotational models (e.g. [29, 76, 40]), to $\mathbf{S}^-$.

First, recall the key feature of this encoding is that it reflects our intuition that the object types of $\mathbf{S}^-$ are fixed points of recursive type equations. More specifically, the recursion is over the self-parameter which occurs both covariantly and contravariantly. This intuition is clearly seen in the V O typing rule for $\sigma = Obj(X)[\ell_i : \tau_i(X)]^{i\in I}$ which suggests the i'th method will consume the self-parameter, which has type σ, to produce something of type τ_i where σ may occur free, e.g. also be produced. Thus, intuitively, the interpretation of σ should satisfy

$$\lceil\sigma\rceil \cong \lceil\sigma\rceil{\to}\lceil\tau_1\rceil \times \ \cdots \ \times \lceil\sigma\rceil{\to}\lceil\tau_n\rceil$$

where, as we mentioned above, each of the τ_i may contain σ. Hence the interpretation of σ should be the fixed point $\mu X.X{\to}\lceil\tau_1\rceil \times \cdots \times X{\to}[\![\tau_n]\!]$ where the τ_i may contain X free. Thus the interpretations of the object types *Point* and *UnLam* are

$$\begin{array}{lcl} \lceil Point\rceil & = & \mu X.(X{\to}X) \times (X{\to}Int) \\ \lceil UnLam\rceil & = & \mu X.X{\to}X \end{array}$$

Note the interpretation of this example shows how the type of untyped lambda terms arises naturally as an object. We do not need to translate type contexts since we have identified the sets of type variables. We thus begin by translating well-formed $\mathbf{S}^-$ types into FPC-types:

$$\begin{array}{lcl} \lceil X\rceil & \triangleq & X \\ \lceil 1\rceil & \triangleq & 1 \\ \lceil A{\to}B\rceil & \triangleq & \lceil A\rceil{\to}\lceil B\rceil \\ \lceil \prod_{i\in I} A_i\rceil & \triangleq & \prod_{i\in I}\lceil A_i\rceil \\ \lceil \coprod_{i\in I} A_i\rceil & \triangleq & \coprod_{i\in I}\lceil A_i\rceil y \end{array}$$

As mentioned above, the translation of object types is into the solution of a mixed variance recursive type equation.

$$\lceil Obj(X)[\ell_i : \tau_i(X)]^{i\in I}\rceil \quad \triangleq \quad \mu X.X{\to}\lceil\tau_1\rceil \times \ldots \times X{\to}\lceil\tau_n\rceil$$

Notice that the translation of types respects substitutions, that is $\lceil\tau[\sigma/X]\rceil = \lceil\tau\rceil[\lceil\sigma\rceil/X]$. We can now syntactically translate (term) contexts:

$$\begin{array}{lcl} \lceil\Theta \vdash \langle\rangle\rceil & \triangleq & \Theta \vdash \langle\rangle \\ \lceil\Theta \vdash \langle\Gamma, x:\tau\rangle\rceil & \triangleq & \Theta \vdash \langle\lceil\Gamma\rceil, x:\lceil\tau\rceil\rangle \end{array}$$

Now we extend our translation to typing judgments. The translations of terms in the intersection of the calculi are just by induction.

$$\begin{array}{lcl} \lceil\Theta,\Gamma \vdash x_i:\tau\rceil & \triangleq & \Theta,\lceil\Gamma\rceil \vdash x_i:\lceil\tau\rceil \\ \lceil\Theta,\Gamma \vdash \star:1\rceil & \triangleq & \Theta,\lceil\Gamma\rceil \vdash \star:1 \\ \lceil\Theta,\Gamma \vdash \langle m_0,...,m_n\rangle:\tau\rceil & \triangleq & \Theta,\lceil\Gamma\rceil \vdash \langle\lceil m_0\rceil,...,\lceil m_n\rceil\rangle:\lceil\tau\rceil \\ \lceil\Theta,\Gamma \vdash \pi_i\, m:\tau\rceil & \triangleq & \Theta,\lceil\Gamma\rceil \vdash \pi_i\,\lceil m\rceil:\lceil\tau\rceil \\ \lceil\Theta,\Gamma \vdash \iota_j\, m:\tau\rceil & \triangleq & \Theta,\lceil\Gamma\rceil \vdash \iota_j\,\lceil m\rceil:\lceil\tau\rceil \\ \lceil\Theta,\Gamma \vdash m_0\, m_1:\tau\rceil & \triangleq & \Theta,\lceil\Gamma\rceil \vdash \lceil m_0\rceil\lceil m_1\rceil:\lceil\tau\rceil \\ \lceil\Theta,\Gamma \vdash \lambda(x:\sigma)m:\tau\rceil & \triangleq & \Theta,\lceil\Gamma\rceil \vdash \lambda(x:\lceil\sigma\rceil)\lceil m\rceil:\lceil\tau\rceil \end{array}$$

$$\begin{array}{l} \lceil\Theta,\Gamma \vdash case(m_0, x_1.m_1, ..., x_n.m_n):\tau\rceil \\ \quad \triangleq \Theta,\lceil\Gamma\rceil \vdash case(\lceil m_0\rceil, x_1.\lceil m_1\rceil, ..., x_n.\lceil m_n\rceil):\lceil\tau\rceil \end{array}$$

We translate object introductions, method update, and object elimination (method invocation) in the obvious way once one recalls the translation of object types.

$$\begin{array}{l} \lceil\Theta,\Gamma \vdash m:\sigma\rceil \\ \quad \triangleq \Theta,\lceil\Gamma\rceil \vdash inn(\langle\lambda x:\lceil\sigma\rceil.\lceil b_1\{\sigma\}\rceil, ..., \lambda x:\lceil\sigma\rceil.\lceil b_n\{\sigma\}\rceil\rangle):\lceil\sigma\rceil \\ \lceil\Theta,\Gamma \vdash m.\ell_i\rceil \\ \quad \triangleq \Theta,\lceil\Gamma\rceil \vdash (\pi_i\ \alpha)(\lceil m\rceil):\lceil\tau_i\{\sigma\}\rceil \\ \lceil\Theta,\Gamma \vdash m.\ell_j \Leftarrow \varsigma(x:\sigma)b\{\sigma\}\rceil \\ \quad \triangleq \Theta,\lceil\Gamma\rceil \vdash inn(\langle\pi_1\alpha, ..., \pi_{j-1}\alpha, \lambda x:\lceil\sigma\rceil.\lceil b\rceil\{\sigma\}, \pi_{j+1}\alpha, ..., \pi_n\alpha\rangle):\lceil\sigma\rceil \\ \text{where} \\ \quad \alpha \equiv out(\lceil m\rceil) \\ \quad m \equiv Obj(X=\sigma)[\ell_i = \varsigma(x_i:X)b_i]^{i\in I} \\ \quad \sigma \equiv Obj(X)[\ell_i:\tau_i(X)]^{i\in I} \end{array}$$

Here π_{l_j} is the projection of a labeled product.

Let F_i be type expressions. We have interpreted object types as $\mu X.(X{\to}F_1\ X) \times ... \times (X{\to}F_n\ X)$ (where for method invocation, self is applied *after* projection) rather than $\mu X.(X{\to}F\ X)$. This is because the latter interpretation would break soundness. Consider, for example the interpretation of method invocation. For soundness, we need to prove that π_{ℓ_j} applied to a term reduces to a value in the case when $m.\ell_j$ reduces to a $\mathbf{S}^-$-value. However, the eager operational semantics of projection in FPC requires that all components of the tuple have a value, and we can easily construct an object for which this would not hold. However, given a lazy operational semantics for FPC (e.g. Winskel [76]) this argument would no longer apply, since partially evaluated terms (in particular products) are included as values.

We will now prove an important lemma which shows that our interpretation function $\lceil - \rceil$ is substitutive on terms (it is trivially substitutive on types):

Lemma 3.3.1 $\lceil m\{v/x, \sigma/\gamma\}\rceil = \lceil m\rceil\{\lceil v\rceil/x, \lceil\sigma\rceil/\gamma\}$

Proof The proof is by induction on the image of terms under $\lceil - \rceil$. We need only consider object intro, elim, update under $\lceil - \rceil$:

$$
\begin{array}{ll}
 & \lceil Obj(X=\sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I}\{v/x, \sigma/\gamma\} : \delta\rceil \\
= & \text{by definition} \\
 & inn(\langle \lambda x : \tau.\lceil b_1\{\delta, v/x, \sigma/\gamma\}\rceil, ..., \lambda x : \tau.\lceil b_n\{\delta, v/x, \sigma/\gamma\}\rceil\rangle) \\
= & \text{by induction hypothesis on } b_i \\
 & inn(\langle \lambda x : \tau.\lceil b_1\{\delta\}\rceil\{\lceil v\rceil/x, \lceil\sigma\rceil/\gamma\}, ..., \lambda x : \tau.\lceil b_n\{\delta\}\rceil\{\lceil v\rceil/x, \lceil\sigma\rceil/\gamma\}\rangle) \\
= & \text{since } \lceil - \rceil \text{ is substitutive on } inn\text{, tupling, and } \lambda \\
 & inn(\langle \lambda x : \tau.\lceil b_1\{\delta\}\rceil, ..., \lambda x : \tau.\lceil b_n\{\delta\}\rceil\rangle)\{\lceil v\rceil/x, \lceil\sigma\rceil/\gamma\} \\
= & \text{by definition} \\
 & \lceil Obj(X=\sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I}\rceil\{\lceil v\rceil/x, \lceil\sigma\rceil/\gamma\}
\end{array}
$$

The situation is similar for object elim and method update, in that $\lceil - \rceil$ will be substitutive on sub-terms formed according to the rules of FPC.

□

Our translation preserves types:

Lemma 3.3.2 *If* $\Theta, \Gamma \vdash t : \tau$ *then* $\lceil\Theta\rceil, \lceil\Gamma\rceil \vdash \lceil t\rceil : \lceil\tau\rceil$

Proof The proof is by induction on well-typed terms. We consider only Val Object, V S , and Val Update, since the other cases follow by induction. Suppose $\Theta, \Gamma \vdash Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I} : \sigma$ where $\sigma = Obj(X)[\ell_i : \tau_i(X)]^{i\in I}$. We must show that $\Theta, \lceil\Gamma\rceil \vdash inn(\langle \lambda x : \lceil\sigma\rceil.\lceil b_1\{\sigma\}\rceil, ..., \lambda x : \lceil\sigma\rceil.\lceil b_n\{\sigma\}\rceil\rangle) : \lceil\sigma\rceil$ where $\lceil\sigma\rceil = \mu X.X\to\lceil\tau_1\rceil\times ...\times X\to\lceil\tau_n\rceil$.

This follows if we can satisfy the premises of the (μI) rule, i.e. if

$$
\begin{array}{rl}
\Theta, \lceil\Gamma\rceil \vdash & inn(\langle \lambda x : \lceil\sigma\rceil.\lceil b_1\rceil\{\lceil\sigma\rceil\}, ..., \lambda x : \lceil\sigma\rceil.\lceil b_n\rceil\{\lceil\sigma\rceil\}\rangle) : \\
 & X\to\lceil\tau_1\rceil\times ...\times X\to\lceil\tau_n\rceil \\
 & \quad \{\mu(X)X\to\lceil\tau_1\rceil\times ...\times X\to\lceil\tau_n\rceil/X\}
\end{array}
$$

The premises of V O asserts $\Theta, \langle\Gamma, x_i : \sigma\rangle \vdash b_i\{\sigma\} : \tau_i\{\sigma\}$ which by induction hypothesis means $\Theta, \langle\lceil\Gamma\rceil, x_i : \lceil\sigma\rceil\rangle \vdash \lceil b_i\{\sigma\}\rceil : \lceil\tau_i\{\sigma\}\rceil$. We then have a FPC-term of the required type from the bodies $\lceil b_i\{\sigma\}\rceil = \lceil b_i\rceil\{\lceil\sigma\rceil\}$ by the substitution lemma. The V F and V P rules gives us $\langle \lambda x : X.\lceil b_1\rceil\{X\}, ..., \lambda x : X.\lceil b_n\rceil\{X\}\rangle\{\lceil\sigma\rceil/X\}$. Finally V I gives us the required type.

The case for V U is almost identical. For V S we assume $\Theta, \Gamma \vdash m : \sigma$ where $m = Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i\in I}$ and $\sigma = Obj(X)[\ell_i : \tau_i(X)]^{i\in I}$ and consider $\Theta, \Gamma \vdash m.\ell_i : \tau_i\{\sigma\}$. We want $\Theta, \lceil\Gamma\rceil \vdash (\pi_{l_i}\ out(\lceil m\rceil))(\lceil m\rceil) : \lceil\tau_i\{\sigma\}\rceil$. By induction hypothesis we have $\Theta, \lceil\Gamma\rceil \vdash \lceil m\rceil : \lceil\sigma\rceil$. Further $\Theta, \lceil\Gamma\rceil \vdash out(\lceil m\rceil) : \lceil\sigma\{\mu X.\sigma/X\}\rceil$ and after projection we have the body b_i of type $\tau_i\{\sigma\}$, and the result follows by applying the induction hypothesis to b_i.

□

3.4 Soundness and Adequacy

We will prove the soundness and adequacy of our translation of $\mathbf{S}^-$ into FPC. This means that not only does the translation of $\mathbf{S}^-$ into FPC work at the level of types and terms, but also that the operational semantics of FPC is strong enough to interpret the operational semantics of $\mathbf{S}^-$ while not being so strong as to give extra computations which were not present in $\mathbf{S}^-$.

We will show that $t \rightsquigarrow v$ implies $\ulcorner t \urcorner \rightsquigarrow \ulcorner v \urcorner$. This establishes that our translation is correct (soundness). We also prove an adequacy result of the operational semantics of $\mathbf{S}^-$. These two results establish that the denotational semantics of FPC given by Fiore is, via the self-application interpretation, a suitable mathematical setting for object calculus. For example, a category such as pCpo immediately gives us a denotational model of object calculus.

In what follows we assume that $[\![-]\!] : FPC \to \mathcal{M}$ is the interpretation of FPC into a sound and adequate model such as the one given by Fiore and Plotkin.

Definition 3.4.1 (Computational Soundness) *We say that an interpretation $\ulcorner - \urcorner$ of $\mathbf{S}^-$ into FPC is computationally sound if, for every $\Theta, \Gamma \vdash o : \tau$ such that $o \rightsquigarrow v$ where v is a value, we have that $[\![\ulcorner \Theta, \Gamma \vdash o : \tau \urcorner]\!]$ is a total map.*

Definition 3.4.2 (Computational Adequacy) *We say that an interpretation $\ulcorner - \urcorner$ of $\mathbf{S}^-$ into FPC is computationally adequate if given any $\Theta, \Gamma \vdash o : \tau$, whenever $[\![\ulcorner \Theta, \Gamma \vdash o : \tau \urcorner]\!]$ is a total map, we also have that $o \rightsquigarrow v$ for some value v.*

Theorem 3.4.3 (Soundness) *The interpretation $\ulcorner - \urcorner$ is computationally sound. That is, if $t \rightsquigarrow v$, then $\ulcorner t \urcorner \rightsquigarrow \ulcorner v \urcorner$*

Proof We only check the derivation rules V O , V S , and V U , since the result follows from induction for the other derivation rules. The translation of an V O term is an FPC value and hence the theorem holds for terms arising as the result of the V O rule.

For V S , suppose $m \rightsquigarrow v'$ and $b_i\{v', \sigma\} \rightsquigarrow v$, where $v' = Obj(X = \sigma)[l_1 = \varsigma(x_i : X).b_i]^{i \in I}$. We want to show that $\ulcorner m.\ell_i \urcorner \rightsquigarrow \ulcorner v \urcorner$. By induction $\ulcorner m \urcorner \rightsquigarrow \ulcorner v' \urcorner$ and hence

$$\pi_i(out\ulcorner m \urcorner) \rightsquigarrow \lambda x : \ulcorner \sigma \urcorner . b_i$$

Again, by induction, $\ulcorner m \urcorner \rightsquigarrow \ulcorner v' \urcorner$ and $\ulcorner b_i\{v', \sigma\} \urcorner \rightsquigarrow \ulcorner v \urcorner$. The result then follows by the substitution lemma since $\ulcorner b_i\{v', \sigma\} \urcorner = \ulcorner b_i \urcorner \{\ulcorner v' \urcorner, \sigma\}$.

For method update, suppose $m \rightsquigarrow v$ where $v = Obj(X = \sigma)[\ell_i = \varsigma(x_i : X)b_i]^{i \in I}$. In order to prove $\ulcorner m.\ell_j \Leftarrow \varsigma(x : \sigma).b \urcorner \rightsquigarrow \ulcorner v' \urcorner$ where $v' = Obj(X = \sigma)[l_i = \varsigma(x : X).b_i, l_j = \varsigma(x : X).b]^{i \in I}$ we must prove that

$$\begin{array}{l} inn(\langle \pi_1 \alpha, ..., \lambda x : \ulcorner \sigma \urcorner . \ulcorner b \urcorner, ..., \pi_n \alpha \rangle) \rightsquigarrow \\ inn(\langle \lambda x : \ulcorner \sigma \urcorner . \ulcorner b_1 \urcorner, ..., \lambda x : \ulcorner \sigma \urcorner . \ulcorner b \urcorner, ..., \lambda x : \ulcorner \sigma \urcorner . \ulcorner b_n \urcorner \rangle) \end{array}$$

where $\alpha = out(\lceil m \rceil)$. By induction

$$\lceil m \rceil \rightsquigarrow \lceil v \rceil = inn(\langle \lambda x : \lceil \sigma \rceil . \lceil b_1 \rceil, ..., \lambda x : \lceil \sigma \rceil . \lceil b_n \rceil \rangle)$$

and hence $\pi_i \alpha \rightsquigarrow \lambda x : \lceil \sigma \rceil . \lceil b_i \rceil$ as required.

□

We proceed with adequacy which shows that the operational semantics of FPC is not too strong with respect to the operational semantics of $\mathbf{S}^-$.

Theorem 3.4.4 (Adequacy) *The interpretation $\lceil - \rceil$ is computationally adequate, that is if $\lceil t \rceil \rightsquigarrow v$, then there is a v' such that $t \rightsquigarrow v'$ and $\lceil v' \rceil = v$*

Proof The proof is by induction on the derivation tree for $\lceil t \rceil \rightsquigarrow v$. If t is a variable or any of the terms related to the standard type constructors of λ-calculus, then the proof is as expected. If t is a V O term, then both t and $\lceil t \rceil$ are values and hence the theorem trivially holds. There are two more cases:

If t is given by V S , say $t \equiv m.\ell_j$, then $\lceil t \rceil \rightsquigarrow v$ must have the following form:

$$\dfrac{\dfrac{\dfrac{\lceil m \rceil \rightsquigarrow inn\langle ..., \lambda x.b, ... \rangle}{out\lceil m \rceil \rightsquigarrow \langle ..., \lambda x.b, ... \rangle}}{\pi_j out\lceil m \rceil \rightsquigarrow \lambda x.b} \qquad \lceil m \rceil \rightsquigarrow u \qquad \lceil b\{u/x, \sigma\} \rceil \rightsquigarrow v}{(\pi_j out\lceil m \rceil)(\lceil m \rceil) \rightsquigarrow v}$$

We see that the derivation for $\lceil b\{u/x, \sigma\} \rceil \rightsquigarrow v$ is contained in the above derivation. Therefore we can apply the induction hypothesis to it, and also to the term m. The premises of the rule V S are now satisfied, and we can conclude that $t \rightsquigarrow \xi$ for value ξ. It remains to be shown that $\lceil \xi \rceil = v$, but this is just the induction hypothesis for $\lceil b\{u/x\} \rceil$.

Finally, let $t = m.\ell \Leftarrow \varsigma(x : \sigma)b$ be a method update term given by V U , and $\lceil t \rceil \rightsquigarrow v$ for some value v. Such a term has the following derivation tree:

$$\dfrac{\dfrac{\lceil m \rceil \rightsquigarrow inn\, v}{out\lceil m \rceil \rightsquigarrow v \equiv \langle ..., \lambda x.b, ... \rangle}}{inn\langle \pi_1 out\lceil m \rceil, ..., \pi_{j-1} out\lceil m \rceil, \lambda x : \lceil \sigma \rceil . \lceil b \rceil\{\sigma\}, \pi_{j+1} out\lceil m \rceil, ..., \pi_n out\lceil m \rceil \rangle \rightsquigarrow v}$$

The derivation tree clearly shows that $\lceil t \rceil$ reduces to a value exactly when $\lceil m \rceil$ reduces to a value which means, by induction hypothesis, that we have $m \rightsquigarrow u$ for some value u. In other words, the premise of the V U rule is satisfied, so we have indeed that $t \rightsquigarrow u'$ for some value u'. It remains to be shown that $\lceil u' \rceil = v$. However, v has the form indicated in the derivation tree ($\langle ..., \lambda x.b, ... \rangle$), which is given as the interpretation of precisely the value $Obj(X = \sigma)[\ell_i = \varsigma(x : X)b_i, l_j = \varsigma(x : X)b]^i \in I$ to which t reduces to by the ($\Leftarrow$) rule.

□

Although adequacy holds, the stronger property of full abstraction does *not* hold. In order to discuss full abstraction we first need to define the notion of observation (contextual) equivalence with which we can define the full abstraction property:

Definition 3.4.5 (Contextual Equivalence) *Let $C[-]$ be a one-hole context such that $\Theta, \Gamma \vdash C[o] : \tau$ and $\Theta, \Gamma \vdash C[o'] : \tau$ (i.e. $C[o]$ and $C[o']$ are closed typeable terms). We say that o is contextually equivalent to o', written $o \cong o'$, if for all contexts C, we have $[\![\ulcorner \Theta, \Gamma \vdash C[o] : \tau \urcorner]\!]$ being total iff $[\![\ulcorner \Theta, \Gamma \vdash C[o'] : \tau \urcorner]\!]$ is total.*

Definition 3.4.6 (Full Abstraction) *Full abstraction is the property that for any terms o, o', if and only if $[\![\ulcorner \Theta, \Gamma \vdash o : \tau \urcorner]\!] = [\![\ulcorner \Theta, \Gamma \vdash o' : \tau \urcorner]\!]$, then $o \cong o'$, i.e. identified denotations correspond to observationally congruent terms.*

Now consider the counter-example in [74]. Let $a = Obj(X = \sigma)[\ell = \varsigma(x : X)x.\ell$ and $b = Obj(X = \sigma)[\ell = \varsigma(x : X)case(x.\ell, y.\iota_1 \star, y.\iota_2 \star)]$. Note $l : 1 + 1$ (representing a boolean type) in both a and b. Although, $a \cong b$, we do not have equal denotations, since self-application admits application of an object where the l method converges, which gives different function values.

Although most FPC models suffer from expressiveness of "parallel or" which cannot be defined operationally, there are in other words additional obstacles with self-application relative to FPC. What are the consequences of this additional obstacle? It means that with regard to self-applications of the "non-intended" form, i.e. applying a different object to another object, more properties will be valid than those which are provable/observable from the operational semantics. Viswanathan [74] avoids this problem by removing the dependence of bodies b_i of l_i, which gives n mutually recursive self-applicable functions, each using $n - 1$ labels in their method bodies, where n are the total number of labels. The corresponding interpretation satisfies full abstraction relative to $F_{<:}$, but is instead cluttered with technicalities needed to handle the mutual recursion in methods.

In summary, we have developed an interpretation of $\mathbf{S}^-$, a typed object calculi extended with functions, coproducts, and products, into FPC, and proved adequacy and soundness, while considering both eager and lazy variations of FPC. Full abstraction does not hold, but from a pragmatic point of view the simplicity of the interpretation is of greater importance than its precise characterisation of objects. The self-application interpretation into FPC that we have studied is simple but comes with a powerful recursion scheme, that will be studied further in the next chapter of this thesis.

Finally, we would like to remark that subtyping has not been studied in this chapter, but is certainly very important and need to be addressed. To this end, FPC can interpret subtyping using coercion functions, but the details are saved for future work.

Chapter 4

Direcursion

4.1 Denotational Semantics

In this section we give a denotational model of $\mathbf{FOb}_1$, first-order object calculus similar to $\mathbf{S}^-$, but without the extra notation for self type support. Our model is developed using the category pCpo. The key feature of this semantics is that it reflects our intuition that the object types of $\mathbf{FOb}_1$ are fixed points of recursive type equations. More specifically, the recursion is over the self-parameter which occurs negatively. This intuition is clearly seen in the object-intro typing rule for $\sigma = [l_1 : \tau_1, \ldots, l_n : \tau_n]$ which suggests the i'th method will consume the self-parameter, which has type σ, to produce something of type τ_i. Thus, intuitively, the interpretation of σ should satisfy

$$[\![\sigma]\!] \cong [\![\sigma]\!] \rightharpoonup [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!]$$

and hence the denotation of σ should be the fixed point of $\mu X.X \rightharpoonup [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!]$. Crucially, the following lemma shows that such an interpretation supports *self-application* [32] which our semantics both requires and supports. We state the lemma specifically for pCpo to make clear we are not using cartesian closure in the proof.

Lemma 4.1.1 *Let* $\mathsf{F} : \mathsf{pCpo} \longrightarrow \mathsf{pCpo}$ *be a covariant functor and* O *an object of* pCpo *satisfying* $O \cong [O, FO]$. *Then there is a self-application map* $sapp : O \to FO$.

Proof All isomorphisms are total and hence the isomorphism uncurries to give a map $O \otimes O \to FO$. Now precompose with the diagonal which partial products possess.

□

Notice how this differs with the recursive record semantics [3], where the recursion is in the output or covariant position while the contravariant occurrence of *self* is replaced by having a separate state type, and a fixed point operator at the level of terms. Our semantics also differs from other encodings such as various encoding with existentials [63, 19] where the contravariant occurrence is present but hidden under the existential quantifier. In our model of $\mathbf{FOb}_1$ we instead explicate the contravariant *self* parameter

and interpret all object types into more elaborate recursive types which, as we have seen, support self-application.

If $\mathcal{C}$ is a category we denote by $\hat{\mathcal{C}}$ the category $\mathcal{C}^{op} \times \mathcal{C}$ and note that $(\hat{\mathcal{C}})^n = (\hat{\mathcal{C}^n})$. The doubling trick used to obtain fixed points of difunctors assigns to each difunctor $F : \mathcal{C}^{op} \times \mathcal{C} \longrightarrow \mathcal{D}$ a functor $\hat{F} : \mathcal{C}^{op} \times \mathcal{C} \longrightarrow \mathcal{D}^{op} \times \mathcal{D}$. We call functors that arise in this way *symmetric* - see [28] for a full definition. Each symmetric functor F induces two functors F_1 and F_2 by post-composition with the projections Π_1 and Π_2 arising from the product on Cat. In fact the mapping $F \mapsto \hat{F}$ is a bijection between difunctors and symmetric functors with inverse sending F to F_2. This fact will be used below to define symmetric functors by giving difunctors. Finally let $\mathcal{P}$ be the category $\mathsf{pCpo}^{op} \times \mathsf{pCpo}$.

With this notation we can give a semantics to types as follows. If a type τ has n-free type variables[1], its interpretation is a symmetric functor

$$[\![\tau]\!] : \mathcal{P}^n \longrightarrow \mathcal{P}$$

Using the bijection mentioned above, we define the symmetric functor $[\![\tau]\!]$ by giving $[\![\tau]\!]_2$. The exceptions to this rule are for the interpretations of recursive types and object types.

$$\begin{array}{lcl}
[\![1]\!]_2 X & = & 1 \\
[\![\tau_1 + \tau_2]\!]_2 X & = & [\![\tau_1]\!]_2 X + [\![\tau_2]\!]_2 X \\
[\![\tau_1 \times \tau_2]\!]_2 X & = & [\![\tau_1]\!]_2 X \otimes [\![\tau_2]\!]_2 X \\
[\![\tau_1 \to \tau_2]\!]_2 X & = & [\![\tau_1]\!]_1 X \multimap [\![\tau_2]\!]_2 X \\
[\![\mu v.\tau]\!] X & = & ([\![\tau]\!] X)^\dagger
\end{array}$$

where $([\![\tau]\!]X)^\dagger$ is the fixed point of $[\![\tau]\!]X : \mathcal{P} \longrightarrow \mathcal{P}$. Finally, for an object type $\sigma = [l_1 : \tau_1, \ldots, l_m : \tau_m]$, we have

$$[\![[l_1 : \tau_1, \ldots, l_m : \tau_m]]\!] = [\![\mu v.\ v \to \tau_1 \times \cdots \times \tau_m]\!]$$

Unwinding the definition, we thus have

$$[\![\sigma]\!]_2 X \cong [\![\sigma]\!]_2 X \multimap [\![\tau_1]\!]_2 X \otimes \cdots \otimes [\![\tau_m]\!]_2 X$$

and note that, in this situation, Lemma 4.1.1 applies since we can take F to be the constant functor returning $[\![\tau_1]\!]_2 X \otimes \cdots \otimes [\![\tau_m]\!]_2 X$. Just as we gave an interpretation to types, so we give one to environments. If E is an environment with n-free type variables, then

$$[\![E]\!] : \mathcal{P}^n \longrightarrow \mathcal{P}$$

is the symmetric functor defined by

$$[\![x_1 : \tau_1, \ldots, x_m : \tau_m]\!]_2 X = [\![\tau_1]\!]_2 X \otimes \cdots \otimes [\![\tau_m]\!]_2 X$$

[1] At this point we play a slight price of informality for not indexing judgments by free type variables. However we previously gained by having less notationally cumbersome judgments. We leave the reader to decide if this was an appropriate choice.

Finally we come to the interpretation for term judgments. If $E \vdash e{:}\tau$ is a judgment using n-type variables, then its interpretation is an indexed family of morphisms

$$[\![E \vdash e{:}\tau]\!]_A : [\![E]\!]_2 A \longrightarrow [\![\tau]\!]_2 A$$

for each symmetric functor $A : \mathcal{P}^n$, i.e. for some $X : \mathsf{pCpo}^n$ the functor A is of the form $A = ((X_1, X_1), \ldots, (X_n, X_n))$. Since the semantic clauses for the term constructs associated with the basic types $1, +, \times, \rightarrow$ are as expected, we leave them as an exercise and focus instead on the judgments for object introduction, update and elimination which we take verbatim from Definition 3.4

- *Object Introduction:* By assumption we are given maps

  $$[\![E, x : \sigma \vdash b_i{:}\tau_i]\!]_A : [\![E, x : \sigma]\!]_2 A \longrightarrow [\![\tau_i]\!]_2 A$$

  in pCpo. Using the definition of $[\![E, x : \sigma]\!]_2$ and the the adjunction between partial product and and partial exponentials, these correspond to the following maps *in the category* Cpo:

  $$[\![E]\!]_2 A \longrightarrow ([\![\sigma]\!]_2 A \rightharpoonup [\![\tau_i]\!]_2 A)$$

  and hence we get, for each A, one map

  $$[\![E]\!]_2 A \longrightarrow ([\![\sigma]\!]_2 A \rightharpoonup [\![\tau_1 \times \cdots \times \tau_n]\!]_2 A)$$

  But, since $[\![\sigma]\!]_2 A \rightharpoonup [\![\tau_1 \times \cdots \times \tau_n]\!]_2 A$ is isomorphic to $[\![\sigma]\!]_2 A$, we are done.

- *Object Elimination:* We are given a family of maps

  $$[\![E \vdash a{:}\sigma]\!]_A : [\![E]\!]_2 A \longrightarrow [\![\sigma]\!]_2 A$$

  and want a map

  $$[\![E \vdash a{:}\sigma]\!]_A : [\![E]\!]_2 A \longrightarrow [\![\tau_j]\!]_2 A$$

  This can be constructed by postcomposing with the self-application map $[\![\sigma]\!]_2 A \longrightarrow [\![\tau_1]\!]_2 A \times \cdots [\![\tau_n]\!]_2 A$ and then the j'th projection.

- *Object Update:* Start with the map

  $$[\![E \vdash a{:}\sigma]\!]_A : [\![E]\!]_2 A \longrightarrow [\![\sigma]\!]_2 A$$

  Unwind the isomorphism defining $[\![\sigma]\!]_2 A$. Replace the j'th component of the tuple with

  $$[\![E, x : \sigma \vdash b{:}\tau_j]\!]_A : [\![E, x : \sigma]\!]_2 A \longrightarrow [\![\tau_j]\!]_2 A$$

  and then refold the isomorphism to get the required map.

4.2 Wrappers

We saw in the previous section how object types and the associated term judgments can be given a semantics by solving recursive equations of the form $O \cong O \to K$ for some constant K representing the types of the fields of the object type. There is thus an asymmetry in that the self parameter can be consumed by the methods but the methods can't produce new self's or objects. More generally one would like methods to be able to both consume and return the self parameter - this would make sense in both functional and imperative object calculi. Doing this means solving equations of the form

$$O \cong [O, \mathsf{F}\ O]$$

where F is some covariant functor. Such generalised objects are clearly supported by the semantics we have already developed. Also note by instantiating F with the identity functor we get the classic equation $D \cong [D, D]$.

We put this idea to use by asking the following question. Given that both the initial algebra and final coalgebra styles of programming have proven to be very popular in the functional world, can we incorporate them into the world of objects? More precisely, if F is a covariant functor with initial algebra $\mu\mathsf{F}$ and final coalgebra $\nu\mathsf{F}$, can we find an object O which supports the kind of programming enjoyed by $\mu\mathsf{F}$ and $\nu\mathsf{F}$. Of course, since we work in an algebraically compact category $\mu\mathsf{F} = \nu\mathsf{F}$.

We provide a positive answer to this question by choosing O to be the fixed point of the equation $O \cong [O, \mathsf{F}\ O]$. Note that our analysis is semantic in that we treat all covariant functors rather than retreating into some restricted syntactic class of functors such as polynomials. For the rest of this section, fix a covariant functor F and define the difunctor $G(X, Y) = X \to \mathsf{F}\ Y$. Also we write *inn* and *out* for the structure maps

$$inn : [O, FO] \longrightarrow O \qquad out : O \longrightarrow [O, FO]$$

of the initial G-dialgebra. Our first result is that objects can be "evaluated" into the final coalgebra and hence enjoy a notion of equality induced by bisimulation.

Lemma 4.2.1 *O is an F-coalgebra and hence there is a F-coalgebra homomorphism $O \longrightarrow \nu\mathsf{F}$.*

Proof From lemma 4.1.1, self application gives a coalgebra $O \longrightarrow FO$.

□

Not only is there a map from O to the final F-coalgebra, but also there is a map from the initial algebra to O

Lemma 4.2.2 *O is an F-algebra and hence there is a F-algebra homomorphism $\mu\mathsf{F} \longrightarrow O$.*

Proof We would like constructors for O, that is for O to be an F-algebra. Using the isomorphism defining O, the structure map $\mathsf{F}\ O \longrightarrow O$ can be given by a map $\mathsf{F}\ O \longrightarrow [O, \mathsf{F}\ O]$

which we take to be the first projection after uncurrying. Now that O is an F-algebra, the *fold* operation of the initial algebra defines an F-algebra homomorphism $\mu\mathsf{F} \longrightarrow O$.

□

That the composite $\mu\mathsf{F} \longrightarrow O \longrightarrow \nu\mathsf{F}$ is the canonical map induced by the initiality of $\mu\mathsf{F}$ and/or the finality of $\nu\mathsf{F}$ relies on the regularity of O. In this setting O is therefore a retract of $\mu\mathsf{F}$ showing it contains the elements of $\mu\mathsf{F}$ but a whole lot more as well.

Next, we wish to consider recursion principles. Initial algebras come with a canonical recursion operator *fold* (catamorphism) which arises as the unique map from the initial algebra to some other algebra. Similarly there is a recursion operator *unfold* (anamorphism) which arises as the unique map from some coalgebra to the final coalgebra. As we mentioned earlier, O has the universal property of being the initial dialgebra and hence comes with its own recursion principle for defining maps from O to any other dialgebra. Unwinding the definition of dialgebra etc, this gives the principle of *direcursion*.

Definition 4.2.3 (Direcursion) *Let (ϕ, ψ) be a dialgebra with types given in the diagram below. Define $(\!|\phi|\!) : O \longrightarrow B$ and $[\!(\psi)\!] : A \longrightarrow O$ to be the unique dialgebra homomorphism such that the following diagram commutes:*

$$\begin{array}{ccc} [O, \mathsf{F}\,O] & \xrightarrow{inn_{\mathsf{G}}} & O \\ {\scriptstyle \mathsf{G}\,[\!(\phi,\psi)\!]\,(\!|\phi,\psi|\!)}\downarrow & & \downarrow{\scriptstyle (\!|\phi,\psi|\!)} \\ [A, \mathsf{F}\,B] & \xrightarrow[\phi]{} & B \end{array} \qquad \begin{array}{ccc} O & \xrightarrow{out_{\mathsf{G}}} & [O, \mathsf{F}\,O] \\ {\scriptstyle [\!(\phi,\psi)\!]}\uparrow & & \uparrow{\scriptstyle \mathsf{G}\,(\!|\phi,\psi|\!)\,[\!(\phi,\psi)\!]} \\ A & \xrightarrow[\psi]{} & [B, \mathsf{F}\,A] \end{array}$$

By simply chasing the above diagram, one can extract the direcursion principle as two mutually recursive combinators:

Definition 4.2.4 (Direcursion - combinators)

$$\begin{aligned} (\!|\phi,\psi|\!)\ o &\triangleq \phi\ \ ((\mathsf{F}\ (\!|\phi,\psi|\!)) \circ (out_{\mathsf{G}}\ o) \circ [\!(\phi,\psi)\!]) \\ [\!(\phi,\psi)\!]\ a &\triangleq inn_{\mathsf{G}}\ \ ((\mathsf{F}\ [\!(\phi,\psi)\!]) \circ (\psi\ a) \circ (\!|\phi,\psi|\!)) \qquad \text{(direc-D)} \end{aligned}$$

Corollary 4.2.5 (Properties of direcursion) *Let $(O, inn_{\mathsf{G}}, out_{\mathsf{G}})$ be the initial G dialgebra.*

- ***Cancellation:*** *For any other G-dialgebra given by $(A, \phi : \mathsf{G}\ B\ A \to A)$ and $(B, \psi : B \to \mathsf{G}\ A\ B)$ we have*

$$\begin{aligned} out_{\mathsf{G}} \circ [\!(\phi,\psi)\!] &= \mathsf{G}(\!|\phi,\psi|\!)[\!(\phi,\psi)\!] \circ \psi \\ (\!|\phi,\psi|\!) \circ inn_{\mathsf{G}} &= \phi \circ \mathsf{G}[\!(\phi,\psi)\!](\!|\phi,\psi|\!) \qquad \text{(direc-S)} \end{aligned}$$

- ***Reflection:***

$$id = (\!|inn_F, out_F|\!)$$
$$id = [\!(inn_F, out_F)\!] \qquad \text{(direc-R)}$$

- ***Fusion:*** *Let* $\phi : \mathsf{G}\ A\ B \to B, \psi : A \to \mathsf{G}\ B\ A$ *and* $\phi' : \mathsf{G}\ A'\ B' \to B', \psi' : A' \to \mathsf{G}\ B'\ A'$ *be two* G*-dialgebras. Now, given a pair of arrows* $g : A' \to A, h : B \to B'$*, we have:*

$$\begin{cases} h \circ \phi = \phi' \circ \mathsf{G}\ g\ h \\ \psi \circ g = \mathsf{G}\ h\ g \circ \psi' \end{cases} \Rightarrow \begin{cases} h \circ (\!|\phi, \psi|\!) = (\!|\phi', \psi'|\!) \\ [\!(\phi, \psi)\!] \circ g = [\!(\phi', \psi')\!] \end{cases} \qquad \text{(direc-F)}$$

This recursion scheme has been developed as a programming tool by by [27, 57] and also opens the way for potential optimisations of programs based upon fusion, deforestation etc and gives laws for object-oriented programs á la *Algebra of Programming*-school.

Here, we use direcursion to show that O can be used to simulate the unfold operation of the final F-coalgebra. That is given any F-coalgebra $\alpha : A \to \mathsf{F}\ A$, we define a map from A to O. This can be done by instantiating the direcursion principle by taking B to be the one element cpo. The map ϕ must then be the unique total map, while the map $A \to [1, \mathsf{F}\ A]$ sends a to the total function returning $\alpha(a)$.

Wrapper Naturals

In this section we explicitly construct translation maps $k : A \to O$ and $w : O \to A$ for the well-known algebraic data type for naturals $\mathbb{N}$ (i.e. $\mu\mathsf{F}$ where $\mathsf{F}\ X = 1 + X$). The coalgebra $sapp : O \to \mathsf{F}\ O$ has a unique map to the final coalgebra $out_{\mathsf{F}} : \nu\mathsf{F} \to \mathsf{F}\ \nu\mathsf{F}$. This map is denoted w and defined as follows:

$$\begin{array}{rcll} w(o) & \triangleq & inl\ \star & \text{if } sapp\ o = inl\ \star \\ & \triangleq & inr(w\ o') & \text{if } sapp\ o = inr\ o' \end{array}$$

Here, we have written $\star$ for the element of 1 (which is unique up to isomorphism). We also write $\triangleq$ for definitional equality.

We know that, in C, μF is the initial F-algebra. We can equip O with an F-algebra structure by defining a map $\mathsf{F}\ O \to O$. We define a map $\hat{k} : \mathsf{F}\ O \to (O \to \mathsf{F}\ O)$ by $\hat{k} = \lambda x.\lambda y.x$. Note that $\hat{k}$ is the (typed) K-combinator. By composing $\hat{k}$ with $inn : (O \to \mathsf{F}\ O) \to O$ we have

$$inn \circ \hat{k} : F\ O \to O$$

as required. Thus we have a unique map $k : \mu F \to O$, an F-algebra homomorphism:

$$\begin{array}{lcll} k(0) & \triangleq & inn(\lambda o.inl\ \star) & \text{where } \lambda o.inl\ \star \text{ is the function } O \to 1 + O \\ k(n+1) & \triangleq & inn(\lambda o.inr(k\ n)) & \end{array}$$

We have a canonical map $\mu\mathsf{F} \to \nu\mathsf{F}$ which embeds finite terms into the set of finite and infinite terms:

Lemma 4.2.6 $w \circ k = id$

Proof The proof is by induction. For the base case, we show that $(w \circ k)\ 0 = w(inn(\lambda o.inl\ \star)$ by calculation:

$$\begin{array}{ll} sapp(inn(\lambda.inl\star)) & \\ & = out(inn(\lambda o.inl\ \star))(inn(\lambda o.inl\ \star)) \\ & = (\lambda o.inl\ \star)(inn(\lambda o.inl\ \star)) \\ & = inl\ \star \end{array}$$

For the induction step, we show $(w \circ k)\ (n+1) = (n+1)$. We have:

$$(w \circ k)\ (n+1) = w(inn(\lambda o.inr(k(n)) = inr(w(k(n)) = inr(n)$$

□

Under k, we see that the naturals $0, 1$, and also ω, are translated into wrappers as follows:

$$inn\ \lambda x.inl\ \star \qquad inn\ \lambda x.inr(inn\lambda x.inl\ \star) \qquad inn\lambda x.x$$

etc. The corresponding object type is:

$$NatO = \mu X.[zero\_or\_succ : 1 + X]$$

The wrapper naturals are hence the following $\mathbf{FOb}_1$-terms (setting $UNatO = out\ NatO$):

$$\begin{array}{lcl} zero & \triangleq & [zero\_or\_succ = \varsigma(x : UNatO)inl\ \star] \\ one & \triangleq & [zero\_or\_succ = \varsigma(x : UNatO)inr\ zero] \\ \vdots & & \\ \omega & \triangleq & [zero\_or\_succ = \varsigma(x : UNatO)x] \end{array}$$

We are now in a position where we would like to compare add function for naturals to an add function for wrapper naturals. In other words, given $m \in \mathbb{N}$ we wish to define a function $+_m : O \to O$ such that $+_m\ k(n) = k(m+n)$ for any $n \in \mathbb{N}$. For this function we will write $k(m) + k(n)$ although we are actually defining a section for fixed m. We proceed denotationally:

$$+_m\ o \quad \triangleq \quad (\!|\phi_m, \psi|\!)\ o$$

where

$$\phi_m\ o = \begin{cases} k(m), \text{ if } sapp\ o = inl\ \star \\ inn\ (\lambda q.o\star), \text{ otherwise} \end{cases}$$

and where we have $\psi = \lambda x.\lambda o.sapp\ k(0)$. We have instantiated direcursion such that $[\![\phi, \psi]\!] : 1 \to O$. To see how this computation proceeds we expand the definition of catamorphism for $(\!|\phi_m, \psi|\!)\ k(0)$:

$$(\!|\phi_m, \psi|\!)\ k(0) \quad = \quad \phi_m(\mathsf{F}\ (\!|\phi_m, \psi|\!) \circ (out\ k(0)) \circ [\!(\phi_m, \psi)\!])$$

This shows that the anamorphism is first evaluated. Since $k(0)$ discards its argument, this anamorphism will have no role to play provided that the composed function terminates even if its un-evaluated argument involves non-termination. Formally, we prove the following correctness property:

Lemma 4.2.7 $+_m\ k(n) = k(m + n)$

Proof By induction on the number n. Let m be an arbitrary but fixed natural number. First for the base case, we set $n = 0$ and reason:

$$\begin{array}{ll} & (\!|\phi_m, \psi|\!)\ k(0) \\ = & \{\ \text{by definition of catamorphism}\ \} \\ & \phi_m\ (\mathsf{F}\ (\!|\phi_m, \psi|\!) \circ out(k(0))) \circ [\!(\phi_m, \psi)\!] \\ = & \{\ \text{since } out(k(0)) \text{ discards its argument}\ \} \\ & \phi_m\ (\mathsf{F}\ (\!|\phi_m, \psi|\!) \circ out(k(0))) \\ = & \{\ \text{functor}\ \} \\ & \phi_m\ ((id_1 + (\!|\phi_m, \psi|\!)) \circ (\lambda o.inl\ \star)) \\ = & \{\ \text{composition with } id_1\ \} \\ & \phi_m(\lambda o.inl\ \star) \\ = & \{\ \text{definition of } \phi_m\ \} \\ & k(m) \end{array}$$

By similar reasoning, it holds for $n = 1$. Note the overloaded meaning of +, both the add function and in the derivation also coproduct. For the induction step we first assume that for $n - 1$ the statement holds, i.e. $+_m\ k(n - 1) = k(m + n - 1)$, and then derive the required equality:

$$\begin{array}{ll} & (\!|\phi_m, \psi|\!)\ k(n) \\ = & \{\ \text{by definition of catamorphism}\ \} \\ & \phi_m\ (\mathsf{F}(\!|\phi_m, \psi|\!) \circ out(k(n)) \circ [\!(\phi_m, \psi)\!]) \\ = & \{\ \text{since } out(k(n)) \text{ discards its argument}\ \} \\ & \phi_m\ (\mathsf{F}(\!|\phi_m, \psi|\!) \circ out(k(n))) \\ = & \{\ \text{functor}\ \} \\ & \phi_m\ ((id_1 + (\!|\phi_m, \psi|\!)) \circ out(k(n))) \\ = & \{\ \text{assuming } n > 0\ \} \\ & \phi_m\ (inr\ (\!|\phi_m, \psi|\!) \circ out(k(n))) \\ = & \{\ \text{since } \lambda x.inr(out(k(n-1))) = out(k(n))\ \} \\ & \phi_m\ (inr\ (\!|\phi_m, \psi|\!) \circ (\lambda x.inr(out(k(n-1))))) \\ = & \{\ \text{property of } k \text{ and } (\!|\cdot|\!), \text{ assuming } n > 1\ \} \\ & inn \circ (\lambda y.\lambda x.inr\ y) \circ out(\phi_m\ (inr\ (\!|\phi_m, \psi|\!) \circ out(k(n-1)))) \\ = & \{\ \text{by induction hypothesis we have} \\ & \quad k(m + n - 1) = (\!|\phi_m, \psi|\!)\ k(n-1) = \phi_m(inr(\!|\phi_m, \psi|\!) \circ out(k(n-1)))\ \} \\ & inn \circ (\lambda y.\lambda x.inr\ y) \circ out(k(m + n - 1)) \\ = & k(m + n) \end{array}$$

□

Lists

A second example is given by $\mathsf{F}\ X = \mathbb{N} + X \times \mathbb{N}$, the functor for lists of naturals, i.e. its fixed point given by the initial F-algebra is isomorphic to the natural numbers. The translation maps are now defined by:

$$\begin{array}{rcl} k(Nil) & = & inn(\lambda o.inl\ \star) \\ k(Cons\ a\ as) & = & inn(\lambda o.inr\ \langle k(a), as\rangle) \end{array}$$

$$\begin{array}{rcll} w(o) & = & Nil\ \star & \text{if } sapp\ o = inl\ \star \\ w(o) & = & Cons\ a\ as & \text{if } sapp\ o = inr\ \langle a, as\rangle \end{array}$$

We save for future work to more generally study the translation of folds on algebraic data types into folds on their corresponding wrappers. In addition, we would like to also consider laws for lifting a recursive function, for example the factorial function, into its canonical object and object type. Such "object introduction" rules are left as further work, and involves translation a recursive definition to its corresponding iterative object-oriented form. We anticipate that such introduction rules fits well into the idea of deriving object-oriented programs from (e.g. purely functional) specifications.

Of course the above examples merely scratch the surface of the envisioned applications of direcursion to objects. Firstly, the use of the fusion law could eliminate induction proofs such as the correctness proof for + given above. Secondly, we would like to go beyond natural numbers and lists and also consider design patterns and other canonical object-oriented phenomena together with their associated algebraic properties.

To summarise, we have defined a translation of some of the key features of initial algebra and final coalgebra programming into the world of objects. That is, we have defined an object type which contains the elements of the initial algebra, has constructors for pattern matching, can be evaluated into the final coalgebra, supports a notion of bisimulation and supports an unfold operator. That these constructions are quite simple suggests to us that these wrapper objects are natural and gives us hope that further concepts can be incorporated into the model without it becoming intractable.

Chapter 5

Conclusions and Further Work

In this licentiate thesis, I have demonstrated the foundations for "algebra of objects", a treatment of object-oriented/object-based programs in a style similar to Bird-Meertens formalism. This has been done by developing a dialgebraic semantics of object types such that objects can be interpreted as higher-order data types.

More work is required before we can claim to have established such a programming algebra for objects. It seems from the example in chapter 4, that it may already be possible to harvest some applications of direcursion to object-oriented programs. For example, it seems possible to define *k* and *w* themselves using direcursion, and next define additional methods inside the object representing a natural number using the same technique. However, the notion of direcursion is the subject of the author's current research and the aim is to find variations of direcursion. This follows precisely the development of Bird-Meertens formalism. For example, catamorphism cannot always be used to express the functions one want to use in functional programming, and as a result Meertens defined paramorphism which generalises catamorphism for natural numbers into a notion corresponding to primitive recursion, to give just one example. The case for direcursion is similar, and we leave as important future work to refine the notion of direcursion into notions such as parametric direcursion. Interestingly, objects have their special self-application semantics which means that they are a particular form of higher-order data types. As a result, specialisations and variations of direcursion seems plausible.

The present work has been mostly denotational in nature. It would be interesting to extend typed object calculi with the combinators $(\!| \cdot |\!)$ and $[\![\cdot]\!]$. This is, of course, a slightly different line of research, oriented more towards actual programming languages, than their denotational semantics.

We have studied a pCpo model of object calculi. Other models have been studied, e.g. the metric approach proposed by Abadi and Cardelli [3] which uses pers (partial equivalence relations). Future work should more extensively compare with such models.

The treatment of subtyping has been left as further work. This is indeed the subject of the author's current research, and notions of naturality/dinaturality appear together with embedding/projection pairs which are used for coercions. One goal of this line of research is to explain inheritance in terms of direcursion.

Bibliography

[1] Martín Abadi and Luca Cardelli. A semantics of object types. In *Ninth Annual IEEE Symposium on Logic in Computer Science, Paris, France*, pages 332–341, Los Alamitos, CA, July 1994. IEEE.

[2] Martín Abadi and Luca Cardelli. A theory of primitive objects — untyped and first-order systems. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 296–320. Springer-Verlag, April 1994.

[3] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[4] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 396–409, 1996.

[5] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming — an introduction. In *Lecture Notes in Computer Science*, volume 1608, pages 28–115. Springer-Verlag, 1999. Revised version of lecture notes for AFP'98.

[6] Roland Backhouse and Paul Hoogendijk. Final dialgebras: From categories to allegories. *Theoret. Informatics Appl.*, 33:401–426, 1999.

[7] Roland C. Backhouse. An exploration of the Bird-Meertens formalism. Technical Report Technical Report CS 8810, Department of Computer Science, Groningen University, 1988.

[8] John Backus. Can programming be liberated from the Von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21:613–641, 1978.

[9] Richard S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO ASI Series F*, pages 3–42. Springer-Verlag, 1987.

[10] Richard S. Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, 1997.

[11] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed Λ-programs on term algebras. *Theoretical Computer Science*, 39(2–3):135–154, August 1985.

[12] V. Bono, M. Bugliesi, and S. Crafa. Typed Interpretations of Extensible Objects. *ACM Transactions on Computational Logic (TOCL)*, 3(4):562–603, Agosto 2002. Revised and extended version of [**?**].

[13] V. Bono, M. Bugliesi, and L. Liquori. A Lambda Calculus of Incomplete Objects. In *Proc. of MFCS, International Symposium of Mathematical Foundation of Computer Science*, volume 1113 of *Lecture Notes in Computer Sciences*, pages 218–229. Springer Verlag, 1996. (38%) `http://www-sop.inria.fr/mirho/Luigi.Liquori/PAPERS/mfcs-96.ps.gz`.

[14] Viviana Bono and Michele Bugliesi. Matching constraints for the lambda calculus of objects. In *Proc. of TLCA*, volume 1210 of *Lecture Notes in Computer Science*, pages 46–62. Springer-Verlag, 1997.

[15] Viviana Bono and Luigi Liquori. A Subtyping for the Fisher-Honsell-Mitchell Lambda Calculus of Objects. In *Proc. of CSL, International Conference of Computer Science Logic*, volume 933 of *Lecture Notes in Computer Sciences*, pages 16–30. Springer Verlag, 1995.

[16] Gérard Boudol. The recursive record semantics of objects revisited. *Journal of Functional Programming*, 14(3):263–315, 2004.

[17] K. B. Bruce and G. Longo. A modest model of records, inheritance, and bounded quantification. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, pages 151–195. MIT Press, London, 1994.

[18] Kim B. Bruce. Paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, April 1994.

[19] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1-2):108–133, 1999.

[20] Luca Cardelli. A Semantics of Multiple Inheritance. In G. Kahn, D.B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types, International Symposium, Sophia-Antipolis, France*, pages 51–67. Springer-Verlag, 1984. Lecture Notes in Computer Science, volume 173.

[21] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, February/March 1988.

[22] Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, pages 431–507. Springer-Verlag, Berlin, 1991.

[23] Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report No. 92/480/18, Department of Computer Science, University of Calgary, June 1992.

[24] William R. Cook. A self-ish model of inheritance. Unpublished manuscript., 1987.

[25] William R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, Department of computer Science, Providence, Rhode Island, May 1989.

[26] William R. Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 433–443. ACM Press, 1989.

[27] Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Conf. Record 23rd ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages, POPL'96, St. Petersburg Beach, FL, USA, 21–24 Jan. 1996*, pages 284–294. ACM Press, New York, 1996.

[28] Marcelo P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1996.

[29] Marcelo P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps. Cambridge University Press, Distinguished Dissertations in Computer Science, 1996. Axiomatic Domain Theory in Categories of Partial Maps*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1996.

[30] Marcelo P. Fiore and Gordon D. Plotkin. An extension of models of axiomatic domain theory to models of synthetic domain theory. In *CSL*, pages 129–149, 1996.

[31] Kathleen Fisher. *Type Systems for object-oriented programming languages*. PhD thesis, Stanford University, Stanford, CA, USA, August 1996. STAN-CS-TR-98-1602.

[32] Kathleen Fisher, Furio Honsell, and John C. Mitchell. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[33] Kathleen Fisher and John Mitchell. A delegation-based object calculus with subtyping. In *Fundamentals of Computation Theory (FCT'95)*, volume 965 of *Lecture Notes in Computer Science*, pages 42–61. Springer Verlag, 1995.

[34] Kathleen Fisher and John C. Mitchell. Notes on typed object-oriented programming. In *TACS '94: Proceedings of the International Conference on Theoretical Aspects of Computer Software*, pages 844–885. Springer-Verlag, 1994.

[35] Maarten M. Fokkinga. Calculate categorically! *Formal Aspects of Computing*, 4(4):673–692, 1992.

[36] Peter J. Freyd. Algebraically complete categories. In *Proc. 1990 Como Category Theory Conference*, volume 1488 of *Lecture Notes in Mathematics*, pages 95–104. Springer-Verlag, 1990.

[37] Peter J. Freyd. Recursive types reduced to inductive types. In *Proceedings 5th IEEE Annual Symp. on Logic in Computer Science, LICS'90*, pages 498–507. IEEE Computer Society Press, June 1990.

[38] Jean-Yves Girard. *Interprétation functionelle et élimination des coupures dans l'arithmétique d'ordre supérieure*. PhD thesis, Université Paris VII, 1972.

[39] Joseph A. Goguen. A Categorical Manifesto. *Mathematical Structures in Computer Science*, 1(1):49–67, 1989.

[40] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, 1992.

[41] T. Hagino. A typed lambda calculus with categorical type constructors. In D. H. Pitt, A. Poigne, and D. E. Rydeheard, editors, *Category Theory and Computer Science*, volume 283 of *Lecture Notes in Computer Science*, pages 140–157. Springer-Verlag, 1987.

[42] T. Hagino. A categorical programming language. In M. Takeichi, editor, *Advances in Software Science and Technology*, volume 4, pages 111–135. Academic Press, 1993.

[43] Leon Henkin. Completeness in the theory of types. *J. Symb. Log.*, 15(2):81–91, 1950.

[44] Ralf Hinze. Generic programs and proofs. Habilitationsschrift, University of Bonn, October 2000.

[45] Samuel N. Kamin. Inheritance in Smalltalk-80: a denotational definition. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 80–87. ACM Press, 1988.

[46] Samuel N. Kamin and U. S. Reddy. Two semantic models of object-oriented languages. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, pages 463–495. MIT Press, London, 1994.

[47] Daniel J. Lehmann and Michael B. Smyth. Algebraic specification of data types: A synthetic approach. *Mathematical Systems Theory*, 14:97–139, 1981.

[48] Daniel Leivant. Reasoning about functional programs and complexity classes associated with type disciplines. In *Proceedings of the 24th Annual Symposium of Foundations of Computer Science*, pages 460–469. IEEE, 1983.

[49] Luigi Liquori. On object extension. In *Proceedings of ECOOP*, volume 1445 of *Lecture Notes in Computer Science*, pages 498–522, 1998.

[50] Luigi Liquori. Bounded polymorphism for extensible objects. *Lecture Notes in Computer Science*, 1657, 1999.

[51] Andres Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, 2004.

[52] Grant R. Malcolm. *Algebraic data types and program transformation*. Ph.D. thesis, Department of Computing Science, Groningen University, The Netherlands, 1990.

[53] Ernest G. Manes and Michael A. Arbib. *Algebraic approaches to program semantics*. Springer-Verlag, 1986.

[54] Lambert Meertens. Algorithmics – towards programming as a mathematical activity. In J. W. De Bakker, M. Hazewinkel, and J. K. Lenstra, editors, *Proceedings CWI Symposium on Mathematics and Computer Science*, number 1 in CWI Monographs, pages 289–334. North-Holland, 1986.

[55] Lambert Meertens. Constructing a calculus of programs. In J. L. A. Van de Snepscheut, editor, *Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 66–90. Springer-Verlag, 1989.

[56] Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

[57] Erik Meijer and Graham Hutton. Bananas in space: extending fold and unfold to exponential types. In S. Peyton-Jones, editor, *Functional Programming Languages and Computer Architecture*, pages 324–333. Association for Computing Machinery, 1995.

[58] John C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 109–124. ACM Press, 1990.

[59] John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1996.

[60] Ulf Norell. Implementing functional generic programming. Licentiate thesis, Chalmers University of Technology and Göteborg University, 2004.

[61] Benjamin C. Pierce. The essence of objects. *SIGSOFT Softw. Eng. Notes*, 25(1):69–71, 2000.

[62] Benjamin C. Pierce. *Types and programming languages*. The MIT Press, 2002.

[63] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–248, April 1994.

[64] G. D. Plotkin. Post-graduate lecture notes in advanced domain theory (incorporating the “Pisa Notes”). Dept. of Computer Science, Univ. of Edinburgh, 1981.

[65] G. D. Plotkin. A metalanguage for predomains. In *Workshop on the Semantics of Programming Languages*, pages 93–118. Programming Methodology Group, University of Göteborg and Chalmers University of Technology, 1983.

[66] G. D. Plotkin. Lectures on predomains and partial functions. Notes for a course given at the Center for the Study of Language and Information, Stanford, 1985.

[67] Erik Poll and Jan Zwanenburg. From algebras and coalgebras to dialgebras. In H. Reichel, editor, *Coalgebraic Methods in Computer Science (CMCS'2001)*, number 44 in ENTCS. Elsevier, 2001.

[68] Bernhard Reus and Thomas Streicher. Semantics and logic of object calculi. *Theoretical Computer Science*, 316(1):191–213, 2004.

[69] John C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation, Paris, France*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.

[70] Marc A. Schroeder. Higher-order Charity. Master's thesis, The University of Calgary, July 1997.

[71] M. Smyth and G. Plotkin. The category theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11(4):761–783, 1982.

[72] M. Spivey. A categorical approach to the theory of lists. In J. L. A. Van de Snepscheut, editor, *Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 399–408. Springer-Verlag, 1989.

[73] Varmo Vene. *Categorical programming with inductive and coinductive types*. PhD thesis, University of Tartu, 2000.

[74] Ramesh Viswanathan. Full abstraction for first-order objects with recursive types and subtyping. In *Proceedings of the Thirteenth Annual IEEE Symposium on Logic in Computer Science (LICS), 1998)*. IEEE Computer Society, 1998.

[75] Mitchell Wand. Type inference for objects with instance variables and inheritance. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, pages 97–120. MIT Press, London, 1994. Originally appeared as Northeastern University College of Computer Science Technical Report NU-CCS-89-2, February, 1989.

[76] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.