Object-Oriented Programming in Explicit Mathematics: Towards the Mathematics of Objects

Thomas Studer
von Werthenstein

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

Leiter der Arbeit:
Prof. Dr. G. Jäger
Institut für Informatik und angewandte Mathematik

Object-Oriented Programming in Explicit Mathematics: Towards the Mathematics of Objects

Thomas Studer
von Werthenstein

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

Leiter der Arbeit:
Prof. Dr. G. Jäger
Institut für Informatik und angewandte Mathematik

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Bern, 3. April 2001

Der Dekan
Prof. Dr. P. Bochsler

Contents

Prologue

> Computer Science is no more about computers than astronomy is about telescopes.
>
> Edgar W. Dijkstra

> Few persons care to study logic, because everybody conceives himself to be proficient enough in the art of reasoning already.
>
> Charles S. Pierce

This thesis deals with the mathematical meaning of object-oriented programming languages. We study denotational semantics for type systems with overloading and late-binding; and based on these investigations, we present a recursion-theoretic interpretation of Featherweight Java. The main tool for our research are systems of explicit mathematics with their recursion-theoretic and set-theoretic models.

A lot of theoretical work has been carried out in order to gain a better insight into the concepts of object-oriented languages, see for instance the collection of papers edited by Gunter and Mitchell [50]. Over the past few years, many different object models have been proposed, such as the recursive record encoding (Cook, Hill and Canning [26] as well as Kamin and Reddy [69]), the existential encoding (Pierce and Turner [78]), the matching-based encoding (Bruce [11]) and several calculi which take "object" as a primitive notion (Abadi and Cardelli [1]). All these object models are based on Cardelli's [16] "objects as records" analogy according to which an object is modeled by a record encapsulating the methods which are defined for that object.

Our work is concerned with another approach to objects, taking *overloading* and *late-binding* as basic rather than encapsulation. Theoretically speaking, overloading denotes the possibility that several functions f_i with respective types $S_i \to T_i$ may be combined to a new overloaded function f of type $\{S_i \to T_i\}_{i \in I}$. We then say f_i is a *branch* of f.

If an overloaded function f is applied to an argument x, then the type of the argument selects a branch f_i, and the result of $f(x)$ is $f_i(x)$, i.e. the chosen

branch is applied to x. If the type at compile time selects the branch, then we speak of *early-binding*. If the selection of the branch is based on the run-time type, i.e. the type of the fully evaluated argument, then we call this discipline *late-binding*. Postponing the resolution of overloaded functions to run-time would not have any effect if types cannot change during the computation. Therefore, we need the concept of subtyping in order to obtain the real power of overloading. Then types can evolve during the execution of a program and this may affect the final result.

Ghelli [46] first defined typed calculi with overloading and late-binding for the study of object-oriented programming languages. This approach was further developed by Castagna, Ghelli and Longo [24]. They introduce $\lambda\&$, a calculus for overloaded functions with subtyping. It was pointed out by Castagna [21] that this calculus provides a foundation for both Simula's and CLOS's style of programming.

In his Ph.D. Thesis, Tsuiki [99] introduces a typed λ calculus with subtyping and a *merge operator*, which provides a way of defining overloaded functions. However, it models just *coherent* overloading which has the restriction that the definition of branches with related input types must be related. For example, if we have an overloaded function with two branches $M_1 : \mathsf{Int} \to T$ and $M_2 : \mathsf{Real} \to T$, then coherent overloading requires that for all $N : \mathsf{Int}$ we have $M_1 N = M_2 N$, since Int is a subtype of Real.

The construction of a semantics for calculi with overloading and late-binding turned out to be surprisingly difficult. Castagna, Ghelli and Longo [23] presented a category-theoretical model for a predicative variant of $\lambda\&$ with early-binding. However, there was no interpretation known for late-bound overloaded functions.

Moreover, they could not solve the problem of *impredicativity* in $\lambda\&$, either. Consider a term M with type $\{\{S \to T\} \to T, S \to T\}$. Since this type is a subtype of $\{S \to T\}$, the term M also belongs to $\{S \to T\}$; and therefore it is possible to apply M to itself, i.e. the type $\{\{S \to T\} \to T, S \to T\}$ is a part of its own domain. Hence, the interpretation of that type refers to itself. For this reason, it is not possible to give a semantics of $\lambda\&$ by induction on the type structure and it was not known how to deal with this form of impredicativity encountered in $\lambda\&$.

Tsuiki also meets the problem of impredicativity in his merge calculus. He can give a mathematical meaning to it thanks to the strong relation of the various branches required by the coherence condition. In [100] he presents a computationally adequate model for overloading via domain-valued functors. However, he only deals with early-binding and a very restricted form

of coherent overloading. Actually, he does not consider a subtype relation between basic types like Int and Real and he states that it would be difficult to extend his model so that it could deal with such a subtype relation.

In this thesis we employ systems of *explicit mathematics*, or *theories of types and names*, to tackle the problems of late-binding and of impredicativity occurring in $\lambda\&$. Explicit mathematics has originally been introduced by Feferman [29, 30, 31] to formalize Bishop style constructive mathematics. In the sequel, these systems have gained considerable importance in proof theory, particularly for the proof-theoretic analysis of subsystems of second order arithmetic and set theory. More recently, theories of types and names have been employed for the study of functional programming. In particular, they have been shown to provide a unitary axiomatic framework for representing programs, stating properties of programs and proving properties of programs. Important references for the use of explicit mathematics in this context are Feferman [33, 34, 35], Stärk [88, 89] and Turner [101, 102]. Beeson [9] and Tatsuta [96] make use of realizability interpretations for systems of explicit mathematics to prove theorems about program extraction.

Theories of explicit mathematics are formulated in a two sorted language. The first-order part, consisting of so-called *applicative theories*, is based on partial combinatory logic, cf. e.g. Jäger, Kahle and Strahm [60]. Types build the second sort of objects. They are extensional in the usual set-theoretic sense, but a special naming relation due to Jäger [56] allows us to deal with *names* of the types on the first-order level. Hence, they can be used in computations, which is the key to model overloading and late-binding. Moreover, we will obtain a solution to the problem of *loss of information* for free in our model.

This problem introduced by Cardelli [16] can be described as follows: when we apply for example the identity function $\lambda x.x$ of type $T \to T$ to an argument a of type S, a subtype of T, then we can usually only derive that $(\lambda x.x)a$ has type T. In many calculi with subtyping, the information that a was of type S gets lost although we have only applied the identity function. This is not the case in our model, as we will show later. At this point we have to mention that Castagna [21] is developing a second order calculus with overloading and late-binding in order to deal with the problem of loss of information in the context of type dependent computations. Our work is also a first step towards a better understanding of that system and the integration of overloading and parametric polymorphism.

Furthermore, we show how the problem of impredicativity can be solved by means of *power types*. They were introduced in explicit mathematics by

Feferman [31], but their use has always been doubted. Glass [48] showed that weak power types add nothing to the proof-theoretic strength of various systems of explicit mathematics without join, and Jäger [59] proved the inconsistency of strong power types with elementary comprehension. Our work provides a first example of an application of power types in explicit mathematics.

Recursive programs are usually modeled with *fixed point combinators*. Hence, we need a powerful principle involving these combinators in order to prove statements about the represented recursive programs. Probably the most famous such principle is fixed point induction introduced by Scott [86], which is based on a CPO interpretation of terms. For a good overview of Scott's induction principle and its connection to CPO models see for example Mitchell [72].

Looking at the untyped λ calculus, we find that in continuous λ-models, such as $\mathsf{P}\omega$ or D_∞, fixed point combinators are interpreted by the *least* fixed point operator in the model, cf. e.g. Amadio and Curien [4] or Barendregt [7]. This fact makes it possible to prove semantically many properties of recursively defined programs.

However, if we look at the purely syntactical side of formal frameworks which are used to analyze programming languages, we often do not find any direct account to least fixed points. In particular, the untyped λ calculus allows to define a fixed point combinator, but there is no possibility to express the leastness of a fixed point, cf. Curry, Hindley, Seldin [27], Hindley, Seldin [51] or Barendregt [7]. Also in the typed λ calculus, we can have fixed point combinators, but the question of leastness, which corresponds to termination, is answered from the outside by the use of normalization proofs. Comparing this with functional programming languages, we see that in a type free language, like Scheme, we can define a fixed point operator which "solves" recursive equations; and in typed languages, like ML, such operators are usually built in. However, there is no way to guarantee on the syntactical level that the solution produced by these operators will be the least fixed point. This is only given by the semantical interpretation, see for example Reade [82].

We are going to introduce an applicative theory with *computability axioms*, which allows to define a least fixed point operator. Hence, it is possible to prove in this theory that certain recursive programs will not terminate. Still, our theory has a recursion-theoretic interpretation.

The question of the mathematical meaning of a program is usually asked to gain more insight into the language the program is written in. This may be

to bring out subtle issues in language design, to derive new reasoning principles or to develop an intuitive abstract model of the programming language under consideration so as to aid program development. Moreover, a precise semantics is also needed for establishing certain properties of programs (often related to some aspects of security) with mathematical rigor.

As far as the Java language is concerned, most of the research on its semantics is focused on the operational approach (see Börger, Schmid, Schulte and Stärk [10], Cenciarelli, Knapp, Reus and Wirsing [25], Drossopoulou, Eisenbach and Khurshid [28], Nipkow and Oheimb [74], and Syme [95]). Notable exceptions are Oheimb [77] who introduces a Hoare-style calculus for Java as well as Alves-Foss and Lam [3] who present a denotational semantics which is, as usual, based on *domain-theoretic* notions, cf. e.g. Fiore, Jung, Moggi, O'Hearn, Riecke, Rosolini and Stark [43] for a recent survey on domains and denotational semantics. Also, the projects aiming at a verification of Java programs using modern CASE tools and theorem provers have to make use of a formalization of the Java language (cf. e.g. the KeY approach by Ahrendt, Baar, Beckert, Giese, Habermalz, Hähnle, Menzel and Schmitt [2] as well as the LOOP project by Jacobs, van den Berg, Huisman, van Berkum, Hensel and Tews [55]).

We will examine a *recursion-theoretic* denotational semantics for Featherweight Java, called FJ. Igarashi, Pierce and Wadler [54, 53] have proposed this system as a minimal core calculus for Java, making it easier to understand the consequences of extensions and variations. For example, they employ it to prove type safety of an extension with generic classes as well as to obtain a precise understanding of inner classes. Ancona and Zucca [6] present a module calculus where the module components are class declarations written in Featherweight Java.

Often, models for statically typed object-oriented programming languages are based on a highly *impredicative* type theory. Bruce, Cardelli and Pierce [13] for example use $\mathsf{F}^{\omega}_{<:}$ as a common basis to compare different object encodings, all of them based on the "objects as records" analogy. We will see that only a *predicative* variant of the overloading based object model is needed in order to interpret object-oriented programming languages. This will be our starting point for constructing a denotational semantics for Featherweight Java in a theory of types and names.

Feferman [33] claims that impredicative comprehension principles are not needed for applications in computational practice. Further evidence for this is also given by Turner [102] who presents computationally weak but highly expressive theories, which suffice for constructive functional programming.

In the present thesis we provide constructive foundations for Featherweight Java in the sense that our denotational semantics for FJ will be formalized in a constructive theory of types and names using the predicative object model of Castagna, Ghelli and Longo [24]. This supports Feferman's thesis that impredicative assumptions are not needed. Although our theory is proof-theoretically weak, we can prove the soundness of our semantics with respect to subtyping, typing and reductions. Moreover, the theory of types and names we use has a recursion-theoretic interpretation. Hence, computations in FJ will be modeled by ordinary computations. For example, a non-terminating computation is not interpreted by a function which yields $\perp$ as result, but by a *partial* function which does not terminate, either.

The plan of this thesis is as follows. We start with a chapter about object-oriented programming, where the concepts of overloading and late-binding are presented. In Chapter 2 we introduce our base system of explicit mathematics with elementary comprehension and join. Then the thesis is divided into four parts, which may be read independently. Chapter 3 is concerned with a semantics for predicative overloading whereas impredicative overloading is dealt with in Chapter 4. An applicative theory that allows us to prove the non-termination of recursive programs is introduced in Chapter 5. The fourth part of this thesis consists of Chapter 6, where Featherweight Java is discussed, and also of Chapter 7, where we construct a recursion-theoretic denotational semantics for FJ. This part depends on the results of non-termination obtained in Chapter 5. However, we will restate them (without proofs) in order to keep the parts independent. We conclude the thesis with an epilogue.

Finally, let us mention that throughout this thesis we make free use of the papers by Kahle and Studer [67] as well as by Studer [92, 93, 94].

Acknowledgments

Gerhard Jäger really deserves the first position here. It was him who educated me in logic and who taught me to work with formal systems. He introduced me to explicit mathematics and he encouraged me to study the mathematics of objects. His steady support contributed a lot to this thesis.

I am also grateful to Thomas Strahm. He was always there to discuss my problems, even when he had to postpone his own work. This thesis owes many things: he often helped me to find the right formulations of the theorems and he corrected many little mistakes in the proofs.

I benefited greatly from the conversations with Reinhard Kahle. We often collaborated: in particular, Chapter 5 is based on joint work with him.

Without the discussions about λ calculi in Jürg Schmid's workshops, Chapter 5 of this thesis and the papers by Kahle and Studer [67] as well as by Probst and Studer [81] would not have been written.

Giorgio Ghelli pointed out an error in an early version of Chapter 3. Moreover, he and Luca Cardelli explained to me many important things about type systems for object-oriented programming languages such as the problem of sound record update.

Great help in checking the last-minute changes of the thesis was provided by Dieter Probst.

Geoff Ostrin reviewed this thesis with respect to the English language. All elegant formulations are due to him. I am to blame for the rest.

The Swiss National Science Foundation financially supported my work on the mathematics of objects.

Last but not least I would like to thank all my friends and colleagues who I met over the past years for all the discussions about logic, computer science and the world. Especially, I thank all the present and former members of our research group and the people from the coffee break for the good time.

Thank you all!

Chapter 1

Object-Oriented Programming

> The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music.
>
> Donald E. Knuth

In this chapter we will not give an introduction to object-oriented programming. We assume that the reader is familiar with a typed object-oriented programming language like Java, C++ or Eiffel or else that she has some experience with type theory. The plan of this chapter is as follows: in the first section we will show which aspects of object-oriented programming we are most interested in, namely overloading and late-binding. Then we present $\lambda^{\{\}}$ which is a typed λ calculus based on these two principles; and we discuss the difficulties of giving a semantics for this calculus. The last section is concerned with the overloading based object model. It explains the relationship between object-oriented programming and the $\lambda^{\{\}}$ calculus.

1.1 Overloading and Late-Binding

Polymorphism is one of the concepts to which the object-oriented paradigm owes its power. Since the early work of Strachey [90] the distinction is made between parametric (or universal) and "ad hoc" polymorphism. Using parametric polymorphism, a function can be defined which takes arguments of a range of types and works uniformly on them. "Ad hoc" polymorphism allows the writing of functions that can take arguments of several different types which may not exhibit a common structure. These functions may execute a different code depending on the types of the arguments. The proof theory

and the semantics of parametric polymorphism have been investigated by many researchers, while "ad hoc" polymorphism has had little theoretical attention.

In object-oriented programming "ad hoc" polymorphism denotes the possibility that two objects of different classes can respond differently to the same message. Castagna, Ghelli and Longo [23] illustrate this by the following example: the code executed when sending a message *inverse* to an object of type *matrix* will be different from the code executed when the same message is sent to an object representing a real number. Nevertheless, the same message behaves uniformly on all objects of a certain class. This behavior is known as *overloading* since we overload an operator (here *inverse*) by different operations. We say the function consists of several *branches* and the selection of the actual operation depends on the types of the operands.

The real gain in power with overloading occurs only in programming languages which compute with types. They must be computed during the execution of the program and this computation may affect the final result of the computation. Selecting the branch to be executed at compile-time does not involve any computation on types. Postponing the resolution of an overloaded function to run-time, would not have any effect if types cannot change during the computation. Only if types can change, we obtain the real power of overloading. Hence, we need the concept of subtyping in order to have types that are able to evolve during the execution of a program. In such languages, an expression of a certain type can be replaced by another one of a smaller type. Thus, the type of an expression may decrease during the computation. This can affect the final result of a computation if the selection of the branch to be executed is based on the types at a specific moment in the computation. We talk of *early-binding* if the selection of the branch is based on the types at compile-time. If we use the types of the fully evaluated arguments to decide which branch should be executed, then we call this discipline *late-binding*. The introduction of overloading with early binding does not significantly influence the underlying language. However, overloaded functions combined with subtyping and late-binding show the real benefits of object-oriented programming.

The following example will stress the importance of late-binding in the context of object-oriented programming. Assume we have a program for a chess computer. This program will include a class `Piece` which models the general behavior of any piece of the chess game. The class `Piece` will have several subclasses like `King, Queen, Pawn`, etc. which implement the specific features of the different kinds of pieces. In particular, `Piece` will declare a method `move` which will be employed to perform moves. However, since

different pieces move according to different rules, the method `move` will be *abstract* in `Piece` and only be implemented in its subclasses, where we know which kind of piece we are moving. Otherwise, it would not be possible for this method to test whether it is a legal move that should be performed. Observe that it is important that `move` is already declared in `Piece`. During the execution of the program, there will occur instructions like 'take the piece on C4 and move it to D5'. If this instruction gets executed, then the piece on C4 is taken and we have to look at its run-time type in order to know which kind of piece it is. Only then we can decide which `move` method has to be called. Hence, only at run-time we can tell which method will be executed and this is exactly where late-binding enters the stage.

One usually uses higher order lambda calculi to model parametric polymorphism. These systems allow abstraction with respect to types and applications of terms to types. However, computations in these systems do not truly depend on types, i.e. the semantics of an expression does not depend on the types appearing in it. This fact is nicely exposed in a forgetful interpretation of such calculi. Hence, parametricity allows us to define functions that work on many different types, but always in the same way. On the other hand, overloading characterizes the possibility of executing different codes for different types. Thus, we have two different kinds of polymorphism.

The subject of higher order lambda calculus originates from the work of Girard [47] who introduced his system F for a consistency proof of analysis. For this reason, system F is highly impredicative. Independently, Reynolds [83] rediscovered it later and used it for applications in programming languages. Historically, Girard [47] and Troelstra [97] were the first to discover a semantic model for system F. Based on an abstract form of this model, Feferman [33] gives an interpretation of system F in a theory of explicit mathematics and he discusses in detail the advantages of representing programs in theories of types and names.

Until now, there are only a few systems available featuring "ad hoc" polymorphism. Ghelli [46] first defined typed calculi with overloading and late-binding in order to model object-oriented programming. This approach was further explored in Castagna, Ghelli and Longo [24]. In the present thesis we will use $\lambda^{\{\}}$ presented by Castagna [21, 22]. This calculus is designed for the study of the main properties of programming languages with overloading and late-binding. It is a minimal system, in which there is a unique operation of abstraction and a unique form of application. Hence, we only have overloaded functions and consider ordinary functions as overloaded with only one branch defined.

Castagna, Ghelli and Longo [23] present a category-theoretic semantics for $\lambda\&-$early which is a calculus with overloading and early-binding. In this calculus the types of the arguments of an overloaded function are "frozen"; the same goes for compile-time and run-time. Furthermore, its type system is stratified in order to avoid the problem of impredicativity. In Chapter 3 we will present a semantics for a stratified subsystem of $\lambda^{\{\}}$, which can handle not only overloading but also late-binding. Our model-construction will be carried out in a predicative theory of explicit mathematics. Later we will also investigate impredicative overloading in theories of types and names.

1.2 The $\lambda^{\{\}}$ Calculus

In this section we introduce Castagna's $\lambda^{\{\}}$ calculus. This minimal system, implementing overloading and late-binding, was first presented in [21, 22]. The goal was to use as few operators as possible. Terms are built up from variables by abstraction and application. Types are generated from a set of basic types by a constructor for overloaded types. Ordinary functions are considered as overloaded functions with just one branch.

Pretypes. First we define the set of pretypes. Later we will select the types from among the pretypes, meaning, a pretype will be a type if it satisfies certain conditions on good type formation. We start with a set of *atomic types* A_i, from which the pretypes are inductively defined as follows:

1. Every atomic type is a pretype.
2. If $S_1, T_1, \ldots, S_n, T_n$ are pretypes, then $\{S_1 \to T_1, \ldots, S_n \to T_n\}$ is also a pretype. Often we will employ a notation with index sets and write $\{S_i \to T_i\}_{i\in I}$ for $\{S_1 \to T_1, \ldots, S_n \to T_n\}$ where I is the set $\{1, \ldots, n\}$.

Subtyping. We define a subtyping relation $\leq$ on the pretypes. This relation will be used to define the types. We start with a predefined partial order $\leq$ on the atomic (pre)types and extend it to a preorder on all pretypes by the following subtyping rule:

$$\frac{(\forall i \in I)(\exists j \in J)(U_i \leq S_j \text{ and } T_j \leq V_i)}{\{S_j \to T_j\}_{j\in J} \leq \{U_i \to V_i\}_{i\in I}}$$

If the subtyping relation $\leq$ is decidable on the atomic types, then it is decidable on all pretypes. Note that $\leq$ is just a preorder and not an order. For instance, $U \leq V$ and $V \leq U$ do not imply $U = V$. As an example, assume $S' \leq S$, then we have $\{S \to T\} \leq \{S' \to T\}$, and thus, both $\{S \to T\} \leq \{S \to T, S' \to T\}$ and $\{S \to T, S' \to T\} \leq \{S \to T\}$ hold.

We will give an example how this subtyping rule works. Assume we have two basic pretypes Int and Real with $\mathsf{Int} \leq \mathsf{Real}$ since every integer is also a real number. Let us first consider ordinary functions, that is overloaded functions with just one branch. With the subtyping rule we obtain

$$\{\mathsf{Real} \to \mathsf{Int}\} \leq \{\mathsf{Int} \to \mathsf{Real}\}.$$

Of course, this inference is sound since every function from the reals to the integers maps an integer to a real number. Now let us look at the overloaded function type $\{\mathsf{Real} \to \mathsf{Real}, \mathsf{Int} \to \mathsf{Int}\}$. This type is a subtype of $\{\mathsf{Int} \to \mathsf{Int}\}$ and also of $\{\mathsf{Real} \to \mathsf{Real}\}$ since it extends these types with an additional branch. It is also a subtype of $\{\mathsf{Int} \to \mathsf{Real}\}$. Further, the subtyping rule implies that $\{\mathsf{Real} \to \mathsf{Int}\} \leq \{\mathsf{Real} \to \mathsf{Real}, \mathsf{Int} \to \mathsf{Int}\}$ since $\{\mathsf{Real} \to \mathsf{Int}\}$ is a subtype of every branch of the overloaded super-type.

Types. Although the selection of the branch is based on run-time types, the static typing must ensure that no type-errors will occur during a computation. We define the set of types as follows: we call a pretype S a *minimal element* of a set U of pretypes if S is an element of U and if there does not exist a pretype $T \neq S$ in U such that $T \leq S$. The set of $\lambda^{\{\}}$ types contains all atomic types of $\lambda^{\{\}}$ as well as all pretypes of the form $\{S_i \to T_i\}_{i \in I}$ that satisfy the following three *consistency conditions* concerning good type formation:

1. S_i and T_i are types for all $i, j \in I$,
2. $S_i \leq S_j$ implies $T_i \leq T_j$ for all $i, j \in I$,
3. if there exists $i \in I$ and a pretype S such that $S \leq S_i$, then there exists a unique $z \in I$ such that S_z is a minimal element of $\{S_j | S \leq S_j \wedge j \in I\}$.

The first condition simply states that every overloaded type is built up by making use of other types. The second condition is a consistency condition which ensures that a type may only decrease during a computation. If we have an overloaded function f of type $\{U_1 \to V_1, U_2 \to V_2\}$ with $U_1 \leq U_2$ and we apply it to an argument n with type U_2 at compile-time, then the expression $f(n)$ will have type V_2 at compile-time; but if the run-time type of n is U_1, then the run-time type of $f(n)$ will be V_1. Therefore $V_1 \leq V_2$ must hold. The third condition concerns the selection of the correct branch. It assures the existence and uniqueness of a branch to be executed. If, for example, we apply a function of type $\{S_i \to T_i\}_{i \in I}$ to a term of type U, then the third condition states that there exists a unique $z \in I$ such that S_z is a minimal element of the set $\{S_i \mid U \leq S_i\}$, i.e. S_z best approximates U and

the z^{th} branch will be chosen. Hence, this condition deals with the problem of multiple inheritance. It assures that there will be no ambiguity in the selection of the branch.

Terms. Terms are built up from variables by λ abstraction and application:

$$M \quad ::= \quad x \mid \lambda x(M_1 : S_1 \Rightarrow T_1, \ldots, M_n : S_n \Rightarrow T_n) \mid M_1 M_2,$$

where $n \geq 1$ and $S_1, T_1, \ldots, S_n, T_n$ are types. Variables are not indexed by types, because in a term for an overloaded function such as

$$\lambda x(M_1 : S_1 \Rightarrow T_1, \ldots, M_n : S_n \Rightarrow T_n),$$

the variable x should be indexed by different types. Thus, indexing is avoided and in the typing rules typing contexts are introduced. A *context* Γ is a finite set of typing assumptions $x_1 : T_1, \ldots, x_n : T_n$ with no variable appearing twice.

Type system. The following rules define the typing relation between terms and types.

$$\Gamma, x : T \vdash x : T$$

$$\frac{\Gamma, x : S_1 \vdash M_1 : U_1 \quad \cdots \quad \Gamma, x : S_n \vdash M_n : U_n}{\Gamma \vdash \lambda x(M_i : S_i \Rightarrow T_i)_{i \in \{1,\ldots,n\}} : \{S_i \to T_i\}_{i \in \{1,\ldots,n\}}}$$

where $U_i \leq T_i$ holds for all $i \in \{1, \ldots, n\}$, and

$$\frac{\Gamma \vdash M : \{S_i \to T_i\}_{i \in I} \quad \Gamma \vdash N : S \quad S_j = \min_{i \in I}\{S_i \mid S \leq S_i\}}{\Gamma \vdash MN : T_j}$$

A term M is called *well-typed*, if there exists a typing context Γ and a type T so that $\Gamma \vdash M : T$ is derivable.

Reduction. The notion of reduction in $\lambda^{\{\}}$ is quite complex. It was introduced by Castagna [22] and is discussed in detail in his book [21]. There one also finds all the proofs of the fundamental properties of this reduction relation like Church-Rosser and strong normalization of some important sub-calculi. Here, we confine ourselves to sketching only the main ideas of this reduction relation and to stating some basic properties without proofs.

When we apply an overloaded function to an argument, then the argument type selects the branch of the overloaded function which will be executed. Since we work in a calculus with late-binding, it is the run-time type of the argument which selects this branch. Hence, in each application we first have to evaluate the argument in order to know its run-time type. The reduction rules of $\lambda^{\{\}}$ therefore state that an application can only be reduced

if the argument is in normal form, that is fully evaluated, or when it is clear which branch will be chosen. On the other hand, the definition of normal form is based on the notion of reduction. Hence, we have to define the *notion of reduction* and the *terms in normal form* by simultaneous recursion. Reductions in $\lambda^{\{\}}$ will not only be performed when the argument is fully evaluated but also when one is sure that however the computation evolves the selected branch is always the same. That means an application can be reduced if either the argument is closed and in normal form or it is clear which branch will be chosen. This is stated in the following reduction rule. Since the argument of an application may be an open term, that is it may contain free variables, reduction will depend on a typing context Γ.

Before we can state our reduction rule, we have to define the set of *free variables* $\mathsf{FV}(M)$ of a term M. This set is given by:

1. $\mathsf{FV}(x) = \{x\}$, for any variable x,
2. $\mathsf{FV}(MN) = \mathsf{FV}(M) \cup \mathsf{FV}(N)$, for two terms M and N,
3. $\mathsf{FV}(\lambda x(M_1 : S_1 \Rightarrow T_1, \ldots, M_n : S_n \Rightarrow T_n))$ is defined as

$$(\mathsf{FV}(M_1) \cup \ldots \cup \mathsf{FV}(M_n)) \setminus \{x\},$$

for a variable x and terms $M_1, \ldots, M_n$.

A term M is *closed* if and only if $\mathsf{FV}(M)$ is empty. Otherwise we call M an *open term*.

We define the reduction relation $M \rhd_\Gamma N$ and the property of a term M being in normal form with respect to Γ by simultaneous recursion on the term structure of M.

1. We have the following *notion of reduction*:
   Let M be $\lambda x(M_i : S_i \Rightarrow T_i)_{i\in\{1,\ldots,n\}}$ and $\Gamma \vdash M : \{S_i \to T_i\}_{i\in\{1,\ldots,n\}}$ and $\Gamma \vdash N : S$, where $S_j = \min_{i\in I}\{S_i \mid S \leq S_i\}$. If N is a closed term in normal form with respect to Γ or $\{S_i \mid i \in I, S_i \leq S_j\} = \{S_j\}$, then

$$\lambda x(M_i : S_i \Rightarrow T_i)_{i\in I} N \rhd_\Gamma M_j[N/x],$$

   where $M_j[N/x]$ denotes the substitution of x in M_j by N. Then there are rules for the compatible closure: let M be an application PN and let $\Gamma \vdash P : \{S_i \to T_i\}_{i\in I}$, $\Gamma \vdash N : S$ and assume there exists an $i \in I$ with $S \leq S_i$, then

$$\frac{P \rhd_\Gamma P'}{PN \rhd_\Gamma P'N} \qquad \frac{N \rhd_\Gamma N'}{PN \rhd_\Gamma PN'}$$

Let M be $\lambda x(M_i : S_i \Rightarrow T_i)_{i\in I}$ and $\Gamma \vdash M : \{S_i \to T_i\}_{i\in I}$, then

$$\frac{M_i \rhd_{\Gamma,x:S_i} M_i'}{\lambda x(\cdots M_i : S_i \Rightarrow T_i \cdots) \rhd_\Gamma \lambda x(\cdots M_i' : S_i \Rightarrow T_i \cdots)}$$

2. A term M is in *normal form* with respect to Γ if there does not exist a term N such that $M \rhd_\Gamma N$. That is one of the following cases holds.

   (a) The term M is a variable.

   (b) The term M is of the form $\lambda x(M_i : S_i \Rightarrow T_i)_{i\in I}$ and each M_i $(i \in I)$ is in normal form with respect to $\Gamma, x : S_i$.

   (c) The term M is an application $\lambda x(M_i : S_i \Rightarrow T_i)_{i\in\{1,\dots,n\}}N$, the subterms $\lambda x(M_i : S_i \Rightarrow T_i)_{i\in\{1,\dots,n\}}$ and N are both in normal form with respect to Γ, and if

   $$\Gamma \vdash \lambda x(M_i : S_i \Rightarrow T_i)_{i\in\{1,\dots,n\}} : \{S_i \to T_i\}_{i\in\{1,\dots,n\}},$$

   $\Gamma \vdash N : S$ and $S_j = \min_{i\in I}\{S_i \mid S \le S_i\}$, then N is not a closed term and $\{S_i \mid i \in I, S_i \le S_j\} \neq \{S_j\}$.

   If none of these conditions hold, then it is possible to reduce M since one of the cases for reduction applies.

This is a recursive definition on the term structure of M. In order to decide whether M can be reduced or whether it is in normal form, we have to know whether certain subterms of M are in normal form or not. This works since we employ recursion on the built up of M. Hence, when we decide whether M is in normal form, we already know which subterms of M are in normal form and which are not.

If we allowed reductions with open or non-normal arguments also in the cases when we are not sure which branch will be chosen, then the system would not be confluent. The reason is that the type of an open or non-normal argument can be different in different phases of the computation. For example, consider a term

$$\lambda x(\lambda x(P : V \Rightarrow V, M : U \Rightarrow U)x : V \Rightarrow V)N$$

and a typing context Γ implying $x : V$ and $N : U$. Assume $U \le V$. If the inner reduction were performed with the x argument (which is not closed), then the first branch P would be chosen, while if the outer reduction is performed first, then the term becomes $\lambda x(P : V \Rightarrow V, M : U \Rightarrow U)N$ and the second branch M is (correctly) chosen. In short, the argument of an

overloaded application must be closed and in normal form to perform the evaluation since this is the only case where its type can no longer decrease and where its type describes the value as accurately as possible.

The above example also provides a non-trivial term which is in normal form. Assume $U \leq V$ and $V \not\leq U$. If P is in normal form with respect to $x : V$ and M is in normal form with respect to $x : U$, then the term

$$\lambda x(P : V \Rightarrow V, M : U \Rightarrow U)x$$

is in normal form with respect to $x : V$ since x is not closed and

$$V = \min\{X \mid V \leq X \wedge (X = U \vee X = V)\}$$

and

$$\{X \mid X \leq V \wedge (X = V \vee X = U)\} \neq \{V\}.$$

However, $\lambda x(P : V \Rightarrow V, M : U \Rightarrow U)x$ is *not* in normal form with respect to $x : U$ since then we have $U = \min\{X \mid U \leq X \wedge (X = U \vee X = V)\}$ and $\{X \mid X \leq U \wedge (X = V \vee X = U)\} = \{U\}$. That means there is a unique branch to choose although the argument term is not in normal form. This example also illustrates nicely the role of the typing context in the presence of free variables in the argument of an application. If the typing context guarantees that we can choose a unique branch of the overloaded function, then the application gets reduced. If there are several possible branches, then the reduction cannot be performed.

Stratification. In the sequel, we will also consider the *stratified* subsystem $\lambda^{\{\}}_{str}$ of $\lambda^{\{\}}$. This calculus emerges from $\lambda^{\{\}}$ by restricting the subtype relation on the types. First, we introduce the function rank_λ on the pretypes by:

1. $\mathsf{rank}_\lambda(A_i) = 0$ (where A_i is an atomic type),
2. $\mathsf{rank}_\lambda(\{S_i \to T_i\}_{i\in I}) = \max\{\mathsf{rank}_\lambda(S_i), \mathsf{rank}_\lambda(T_i) \mid i \in I\} + 1$.

With this function we define a new subtyping relation $\leq^-$ by adding the condition $\mathsf{rank}_\lambda(\{S_j \to T_j\}_{j\in J}) \leq \mathsf{rank}_\lambda(\{U_i \to V_i\}_{i\in I})$ to the subtyping rule. The type S is called *strict subtype* of T if $S \leq^- T$ holds, see Castagna [21]. We still have $S \leq^- S$ for any type S and if $S \leq^- T$, then $T \leq^- S$ is still possible. However, if we have a term M of type S, a term N of type T, and MN is a well-typed application, then $\mathsf{rank}_\lambda(T) < \mathsf{rank}_\lambda(S)$. That is the rank of the argument type is *strictly smaller* than the rank of the function type.

The stratified calculus $\lambda^{\{\}}_{str}$ is defined by replacing $\leq$ with $\leq^-$ in the typing and reduction rules of $\lambda^{\{\}}$. Furthermore, in the consistency conditions for good type formation we have to add

2′. $S_i \leq^- S_j$ implies $T_i \leq^- T_j$ for all $i, j \in I$.

The calculi $\lambda^{\{\}}$ and $\lambda^{\{\}}_{str}$ both satisfy Church-Rosser and subject reduction. Proofs for these important properties can be found in Castagna [21].

Theorem 1. *The calculi $\lambda^{\{\}}$ and $\lambda^{\{\}}_{str}$ both satisfy that if $\Gamma \vdash M : T$ and $M \triangleright_\Gamma N$, then $\Gamma \vdash N : S \leq T$*

Theorem 2. *The calculi $\lambda^{\{\}}$ and $\lambda^{\{\}}_{str}$ both satisfy that for all Γ the relation $\triangleright_\Gamma$ is Church-Rosser.*

However, $\lambda^{\{\}}$ is not normalizing. We will construct a term w so that ww is well-typed but not normalizing, that is $ww \triangleright ww$. Let S be the type

$$\{\{T \to T\} \to \{T \to T\}, T \to T\}.$$

With the subtyping rule we immediately obtain

$$S \leq \{T \to T\}. \tag{1.1}$$

Let w be the term

$$\lambda x(xx : S \Rightarrow \{T \to T\}, x : \{T \to T\} \Rightarrow \{T \to T\}, x : T \Rightarrow T).$$

We show that ww is well-typed. First, observe

$$x : S \vdash x : S. \tag{1.2}$$

By (1.1) we get

$$\{T \to T\} = \min\{X \mid S \leq X \wedge (X = T \vee X = \{T \to T\})\}. \tag{1.3}$$

By (1.2) and (1.3) we obtain by the typing rule for applications

$$x : S \vdash xx : \{T \to T\}. \tag{1.4}$$

Further we have

$$x : \{T \to T\} \vdash x : \{T \to T\} \text{ and } x : T \vdash x : T. \tag{1.5}$$

By the typing rule for λ abstraction, (1.4) and (1.5) we conclude

$$\vdash w : \{S \to \{T \to T\}, \{T \to T\} \to \{T \to T\}, T \to T\}. \tag{1.6}$$

Let V be the type of w. By the subtyping rule we see $V \leq S$ since V extends S by an additional branch. Therefore, we get

$$S = \min\{X \mid V \leq X \wedge (X = S \vee X = \{T \to T\} \vee X = T)\}. \tag{1.7}$$

By (1.6) and (1.7) we infer with the typing rule for applications

$$\vdash ww : \{T \to T\}.$$

Therefore, ww is well-typed.

Now we are going to show $ww \triangleright ww$. Note that w is closed and in normal form with respect to the empty typing context. Hence by (1.7) we can apply the reduction rule and get

$$\begin{aligned} ww &= \lambda x(xx : S \Rightarrow \{T \to T\}, x : \{T \to T\} \Rightarrow \{T \to T\}, x : T \Rightarrow T)w \\ &\triangleright xx[w/x] = ww. \end{aligned}$$

This example will not work in the stratified subsystem $\lambda^{\{\}}_{str}$. We have

$$\mathsf{rank}_\lambda(S) \not\leq \mathsf{rank}_\lambda(\{T \to T\})$$

and therefore $S \not\leq^- \{T \to T\}$. Hence, we cannot infer $x : S \vdash xx : \{T \to T\}$. In fact, it is provable that $\lambda^{\{\}}_{str}$ is strongly normalizing, again a proof can be found in Castagna [21].

Theorem 3. *The calculus $\lambda^{\{\}}_{str}$ is strongly normalizing.*

1.3 Semantics

According to Castagna [22] the construction of a model for $\lambda^{\{\}}$ poses the following problems: preorder, type dependent computation, late binding and impredicativity.

- *Preorder*: as we have seen, the subtyping relation of $\lambda^{\{\}}$ is a preorder but not an order relation. If $S' \leq S$ holds, then we have both $\{S \to T\} \leq \{S \to T, S' \to T\}$ and $\{S \to T, S' \to T\} \leq \{S \to T\}$. These two types are completely interchangeable from a semantic point of view. Therefore, both types should have the same interpretation, and the subtyping relation has to be modeled by an order relation on the interpretations of the types.

- *Type dependent computation*: the types of the terms determine the result of a computation. For this reason the interpretation of an overloaded function must not only take the interpretation of the value of its argument as input but also the interpretation of its argument type. Therefore, the semantics of an overloaded function type must take into account the interpretations of the argument types of the functions it consists of. Because we work in a calculus with subtyping, all interpretations of subtypes of the argument types have to be regarded equally.

- *Late-binding*: the choice of the branch to be executed of an overloaded function depends on the run-time types of its arguments and not on the types at compile time. Hence, the branch to be executed cannot be chosen at compile time, which means in the translation of the terms. To determine the value of an overloaded application, first the interpretations of its arguments need to be evaluated in order to know the run-time types of the arguments and to select the branch which will be applied.

- *Impredicativity*: the type system of $\lambda^{\{\}}$ is not stratified. This can be seen in the following example: let T be a type of $\lambda^{\{\}}$. We get

$$\{\{T \to T\} \to T, T \to T\} \leq \{T \to T\}. \tag{1.8}$$

  However, $\{T \to T\}$ also is the domain of one the branches of the type $\{\{T \to T\} \to T, T \to T\}$. Let M be a term and Γ a typing context so that

$$\Gamma \vdash M : \{\{T \to T\} \to T, T \to T\}. \tag{1.9}$$

  By (1.8) we get

$$\begin{aligned}\{T \to T\} = \min\{X \mid \{\{T \to T\} \to T, T \to T\} \leq X \wedge \\ (X = T \vee X = \{T \to T\})\}.\end{aligned} \tag{1.10}$$

  The premises of the rule for application are satisfied by (1.9) and (1.10), that is M as function has an overloaded function type and the type of M as argument selects the branch with the domain $\{T \to T\}$. Hence, we conclude by the application typing rule $\Gamma \vdash MM : T$. We see that in $\lambda^{\{\}}$ self-application of M is meaningful, since the overloaded function type $\{\{T \to T\} \to T, T \to T\}$ is a subtype of one of its own domains. The consequence is that it is not possible to give a semantics for the types by induction on the type-structure, as, in order to give the interpretation of an overloaded type, we need to know the

interpretations of the subtypes of its argument types. Therefore, the interpretation of the type $\{\{T \to T\} \to T, T \to T\}$ refers to itself.

Again, as the example of non-normalization, this example will not work if we stratify the type system. It is not possible to derive

$$\{\{T \to T\} \to T, T \to T\} \leq^{-} \{T \to T\}$$

in $\lambda^{\{\}}_{str}$. In this calculus no impredicativity phenomena occur.

The $\lambda\&-$early calculus is introduced in Castagna [21] and Castagna, Ghelli and Longo [23]. This is a λ calculus with overloading and subtyping, but without late-binding. For stratified subsystems of this calculus a category-theoretical semantics (based on partial equivalence relations, a so-called PER model) is presented, which focuses on the problems stemming from the preorder on the types and the type depended computation.

The problem of the preorder on the types is solved by a syntactic construction called type completion. Intuitively, the completion of an overloaded type is formed by adding all subsumed types. Hence, two equivalent types will be transformed by completion to essentially the same type. For example the completion of $\{S \to T\}$ will be something like $\{S \to T, S_1 \to T, S_2 \to T, \dots\}$, where $S_1, S_2, \dots$ are all (infinitely many) subtypes of S.

The problem of type dependent computation is handled by interpreting overloaded types as product types. If A is a type and for every $x \in A$ we know that B_x is a type, then the product type $\Pi_{x \in A} B_x$ consists of all functions f which map an element x of A to an element $f(x)$ of B_x. Now, semantic codes for types are introduced in order to define the interpretation of an overloaded type as an indexed product. The interpretation of the type $\{S_1 \to T_1, S_2 \to T_2\}$ will be the product type consisting of functions f mapping a code d, for a subtype U of S_1 or S_2, to a function $f(d) : S_n \to T_n$ if U selects the n-th branch of $\{S_1 \to T_1, S_2 \to T_2\}$. Hence, an overloaded function $\lambda x(M_1 : S_1 \Rightarrow T_1, M_2 : S_2 \Rightarrow T_2)$ of type $\{S_1 \to T_1, S_2 \to T_2\}$ will be interpreted by a function f which is defined for every code d, for a type U subtype of S_1 or S_2, in the following way:

$$f(d) = \begin{cases} \lambda x.[\![M_1]\!] & \text{if } U \text{ selects the first branch,} \\ \lambda x.[\![M_2]\!] & \text{if } U \text{ selects the second branch.} \end{cases}$$

In $\lambda\&-$early an overloaded application demands an explicit coercion of its arguments. Hence, the types of the arguments of overloaded functions are "frozen", and the same goes for compile-time and run-time. Therefore, the

problems of late-binding are avoided. Further, since only stratified subsystems of $\lambda\&-$early are modeled, there is no problem of impredicativity. No type occurs strictly in itself, hence the definition of the semantics of a type is not self-referential; and the interpretation of the types can be given by induction on the type structure.

1.4 Object Models

In this section we present the object model of Castagna, Ghelli and Longo [21, 24], which is based on overloading and late-binding. In object-oriented programming languages the computation evolves on *objects*. These programming items are grouped by *classes* and the class of an object defines its behavior. An object has an internal state that may be accessed and modified by sending *messages* to the object. When an object receives a message, it invokes the method (i.e. the code) associated with that message. This association between messages and methods is given by the class of the object.

There are two possible ways to understand message-passing. The first is to encapsulate the methods inside the objects. Therefore, when a message is sent to an object, the associated method is retrieved in the receiving object. This approach has been extensively studied and it corresponds to the "objects as records" analogy of Cardelli [16]. According to this analogy, objects are records whose labels are messages and whose fields contain the associated methods. Hence, message-passing corresponds to field selection.

The second view on message-passing is to consider messages as identifiers of overloaded functions and message-passing as their application. In the sequel we will employ this second object model, in which the state of an object is separated from its methods. Only the fields (containing the state) of an object are bundled together as one unit, whereas the methods of an object are *not* encapsulated inside it. Indeed, methods are implemented as branches of global *overloaded* functions, so-called *multi-methods*. If a message is sent to an object, then this message determines a function and this function will be applied to the receiving object. However, messages are not ordinary functions: if the same message is sent to objects of different classes, then different methods may be retrieved, i.e. a different code may be executed. Hence, messages represent overloaded functions: depending on the type of the argument (the object the message is passed to), a different method is chosen. Since this selection of the method is based on the dynamic type of the object, i.e. its type at run-time, we also have to deal with *late-binding*.

If we interpret object-oriented programs using this object model, then we

have to represent classes as types and messages as identifiers of overloaded functions which execute a certain code, depending on the type (class) of their argument (the object the message is being sent to). We see that the classes of an object-oriented program can be modeled by atomic types in the $\lambda^{\{\}}$ calculus. Therefore, all the methods will have the same rank (namely 1). Hence, the stratified subsystem $\lambda^{\{\}}_{str}$ provides enough expressive power to model the usual object-oriented programming languages.

Chapter 2

Explicit Mathematics

> There may, indeed, be other applications of the system than its use as a logic.
>
> Alonzo Church

> Logic is logic.
>
> Isaac Asimov

Explicit Mathematics has been introduced by Feferman [29, 30, 31] for the study of constructive mathematics. In the present thesis, we will not work with Feferman's original formalization of these systems; instead we treat them as *theories of types and names* as developed by Jäger [56]. These theories are built upon Beeson's [8] logic of partial terms, which we will discuss first. All the systems of explicit mathematics that will be used in the subsequent chapters are variants or extensions of the base theory EETJ of explicit elementary types and join presented in the second section.

2.1 The Logic of Partial Terms

We will employ explicit mathematics to reason about programs. Therefore, it is important that in these theories we have the possibility to talk about the termination of these programs. The usual formulations of first order predicate calculus do not permit the formation of terms which do not necessarily denote anything, such as a term which represents a non-terminating computation. The purpose of this section is to introduce Beeson's [8] *logic of partial terms*. This is a version of the predicate calculus that does admit the formation of non-denoting, or undefined, terms.

The logic of partial terms includes a special *definedness predicate* $\downarrow$ so that for any term t the formula $t\downarrow$ is read as "t is defined" or "t has a value", i.e.

t represents a terminating program. Among the main features of the logic of partial terms are its *strictness axioms* stating that if a term has a value, then all its subterms must be defined, too. This corresponds to a call-by-value evaluation strategy, where all arguments of a function must first be fully evaluated before the final result will be computed.

A language $\mathcal{L}$ for the logic of partial terms comprises the following symbols:

1. countably many individual variables $a, b, c, f, g, h, x, y, z, \dots$ (possibly with subscripts);
2. the unary symbol $\downarrow$ for definedness and the binary symbol $=$ for equality;
3. the logical symbols $\neg$ (negation), $\vee$ (disjunction) and $\exists$ (existential quantification);
4. for every natural number n countably many function symbols and relation symbols of arity n.

We have brackets as auxiliary symbols. The 0-ary function symbols of $\mathcal{L}$ are called the *individual constants* of $\mathcal{L}$. From these constants and the variables of $\mathcal{L}$, the *individual terms* $(r, s, t, r_1, s_1, t_1, \dots)$ of $\mathcal{L}$ are inductively generated as usual by means of the other function symbols.

The *atomic formulas* of $\mathcal{L}$ are $t_1\downarrow$, $t_1 = t_2$ and $R(t_1, \dots, t_n)$ provided that $t_1, \dots, t_n$ are $\mathcal{L}$ terms and R is an n-ary relation symbol of $\mathcal{L}$. The *formulas* $(A, B, C, \dots)$ of $\mathcal{L}$ are inductively defined as follows:

1. Every atomic formula of $\mathcal{L}$ is an $\mathcal{L}$ formula.
2. If A and B are fomulas of $\mathcal{L}$, then $\neg A$ and $(A \vee B)$ are $\mathcal{L}$ formulas.
3. If A is an $\mathcal{L}$ formula and x is a variable of $\mathcal{L}$, then $\exists x A$ is a formula of the language $\mathcal{L}$, too.

In this thesis we are going to work within classical logic only. Hence, we can introduce the following abbreviations:

$$\begin{aligned} (A \wedge B) &:= \neg(\neg A \vee \neg B), \\ (A \to B) &:= (\neg A \vee B), \\ (A \leftrightarrow B) &:= ((A \to B) \wedge (B \to A)), \\ \forall x A &:= \neg\exists x \neg A. \end{aligned}$$

In the sequel we will often omit brackets when there is no danger of confusion. We adopt the convention that $\neg$ binds stronger than $\vee$ and $\wedge$ and that these

two connectives bind stronger than $\rightarrow$ and $\leftrightarrow$. Since we are in a context of partiality, we introduce *partial equality* and a strong form of *unequality* by:

$$\begin{aligned} s \simeq t &:= s\downarrow \vee t\downarrow \rightarrow s = t, \\ s \neq t &:= s\downarrow \wedge t\downarrow \wedge \neg(s = t). \end{aligned}$$

Moreover, the vector notation $\vec{\mathcal{Z}}$ is sometimes used to denote finite sequences $\mathcal{Z}_1, \ldots \mathcal{Z}_n$ of expressions. The length of such a sequence $\vec{\mathcal{Z}}$ is either irrelevant or otherwise given by the context. Suppose now that $\vec{a} = a_1, \ldots, a_n$ and $\vec{s} = s_1, \ldots, s_n$. Then $A[\vec{s}/\vec{a}]$ is the formula which is obtained from A by simultaneously replacing all free occurences of the variables $\vec{a}$ by the terms $\vec{s}$; in order to avoid collision of variables, a renaming of bound variables may be necessary. If the $\mathcal{L}$ formula A is written as $B(\vec{a})$, then we often write $B(\vec{s})$ instead of $A[\vec{s}/\vec{a}]$. The substitution of terms for variables in $\mathcal{L}$ terms is treated accordingly.

Now we present the logic of partial terms in form of a Hilbert calculus with the following rules and axioms:

I. Propositional axioms and propositional rules. These comprise the axioms and rules of some sound and complete Hilbert calculus for classical propositional logic.

II. Quantifier axioms and quantifier rules. The axioms for the existential quantifier consist of all $\mathcal{L}$ formulas

$$A(s) \wedge s\downarrow \rightarrow \exists x A(x)$$

where s may be an arbitrary $\mathcal{L}$ term. The rules of inference for the existential quantifier are all figures

$$\frac{A(a) \rightarrow B}{\exists x A(x) \rightarrow B}$$

so that the variable a does not occur in the conclusion.

III. Definedness axioms.

(D1) $r\downarrow$, provided that r is a variable or a constant of $\mathcal{L}$,

(D2) $F(t_1, \ldots, t_n)\downarrow \rightarrow t_1\downarrow \wedge \cdots t_n\downarrow$ for all n-ary function symbols F of $\mathcal{L}$.

(D3) $s = t \rightarrow s\downarrow \wedge t\downarrow$,

(D4) $R(t_1, \ldots, t_n) \rightarrow t_1\downarrow \wedge \cdots t_n\downarrow$ for all n-ary relation symbols R of $\mathcal{L}$.

IV. Equality axioms.

(E1) $x = x$,

(E2) $s = t \to t = s$,

(E3) $r = s \wedge s = t \to r = t$,

(E4) $R(s_1, \ldots, s_n) \wedge s_1 = t_1 \wedge \cdots \wedge s_n = t_n \to R(t_1, \ldots, t_n)$ for all n-ary relation symbols R of $\mathcal{L}$,

(E5) $s_1 = t_1 \wedge \cdots \wedge s_n = t_n \to F(s_1, \ldots, s_n) \simeq F(t_1, \ldots, t_n)$ for all n-ary function symbols F of $\mathcal{L}$.

In our language $\mathcal{L}$ the existential quantifier is taken as a primitive symbol, whereas the universal quantifier is defined. Accordingly, the quantifier axioms and rules are only formulated for $\exists$. However, we have the following expected dual properties:

1. For each $\mathcal{L}$ term s we have

$$\forall x A(x) \wedge s\downarrow \to A(s). \tag{2.1}$$

2. If $A \to B(a)$ is derivable and the variable a does not occur free in A, then we can also derive $A \to \forall x B(x)$.

The axioms (D2), (D3) and (D4) are often called *strictness axioms*. For example, axiom (D2) requires that a compound term has a value only if all its subterms are defined. Thus, the evaluation of a term follows a *call-by-value* strategy. Stärk [88, 89] examines variants of the logic of partial terms which also implement call-by-name evaluation.

Scott [87] has given a logic similar to the logic of partial terms. He writes $E(t)$ instead of $t\downarrow$, which is now read "t exists". This is not merely a variant of notation, but reflects a deeper difference. Scott treats "existence" as a predicate, i.e. as a property of objects. His semantics allows models in which some elements of the model do not satisfy the predicate E. Beeson's logic, by contrast, treats "denoting" as a property of terms. The definedness axiom (D1) specifically rules out "undefined objects". For a further discussion about the different approaches to partiality see Feferman [37] or Troelstra and van Dalen [98].

2.2 The Base Theory EETJ

Our theories of types and names are formulated in the two sorted language $\mathcal{L}_p$ for individuals and types. It comprises *individual variables* $a, b, c, f, g, h, x, y, z, \ldots$ as well as *type variables* $A, B, C, X, Y, Z, \ldots$, both possibly with subscripts. Additionally, $\mathcal{L}_p$ includes the *individual constants* k, s (combinators), $\mathsf{p}, \mathsf{p}_0, \mathsf{p}_1$ (pairing and projections), 0 (zero), s_N (successor), p_N (predecessor) and d_N (definition by numerical cases). There are additional individual constants, called *generators*, which will be used for the uniform representation of types. Namely, we have a constant c_e (elementary comprehension) for every natural number e as well as the constant j (join). There is one binary function symbol $\cdot$ for (partial) application of individuals to individuals. Further, $\mathcal{L}_p$ has unary relation symbols $\downarrow$ (defined) and N (natural numbers) as well as three binary relation symbols $\in$ (membership), $=$ (equality) and $\Re$ (naming, representation).

The *individual terms* $(r, s, t, r_1, s_1, t_1, \ldots)$ of $\mathcal{L}_p$ are built up from individual variables and individual constants by means of the function symbol $\cdot$ for application. In the following, we often abbreviate $(s \cdot t)$ simply as (st) or st and adopt the convention of association to the left so that $s_1 s_2 \ldots s_n$ stands for $(\ldots(s_1 \cdot s_2)\ldots s_n)$. Moreover, we define general n-tupling by induction on $n \geq 2$ as follows: $(s_1, s_2) := \mathsf{p} s_1 s_2$ and $(s_1, \ldots, s_{n+1}) := (s_1, (s_2, \ldots, s_{n+1}))$.

The *atomic formulas* of $\mathcal{L}_p$ are $s\!\downarrow$, $\mathsf{N}(s)$, $s = t$, $s \in U$ and $\Re(s, U)$. Since we work with a logic of partial terms, it is not guaranteed that all terms have values, and $s\!\downarrow$ is read as *s is defined* or *s has a value*. Moreover, $\mathsf{N}(s)$ says that s is a natural number, and the formula $\Re(s, U)$ is used to express that the individual s *represents* the type U or is a *name* of U.

The *formulas* $(F, G, H, \ldots)$ of $\mathcal{L}_p$ are generated from the atomic formulas under closure with respect to the usual propositional connectives as well as quantification in both sorts. A formula is called *elementary* if it contains neither the relation symbol $\Re$ nor bound type variables. The following table contains a list of useful abbreviations, where F is an arbitrary formula of $\mathcal{L}_p$:

$$\begin{array}{rcl}
s \in \mathsf{N} & := & \mathsf{N}(s), \\
(\exists x \in A)F(x) & := & \exists x(x \in A \wedge F(x)), \\
(\forall x \in A)F(x) & := & \forall x(x \in A \to F(x)), \\
f \in (A \to B) & := & (\forall x \in A) fx \in B, \\
A \subset B & := & \forall x(x \in A \to x \in B), \\
A = B & := & A \subset B \wedge A \subset B, \\
f \in (A \rightharpoonup B) & := & \forall x(x \in A \wedge fx\!\downarrow \to fx \in B),
\end{array}$$

$$\begin{array}{rcl} x \in A \cap B & := & x \in A \wedge x \in B, \\ s \mathrel{\dot\in} t & := & \exists X(\Re(t,X) \wedge s \in X), \\ s \mathrel{\dot\subset} t & := & (\forall x \mathrel{\dot\in} s)x \mathrel{\dot\in} t, \\ s \doteq t & := & s \mathrel{\dot\subset} t \wedge t \mathrel{\dot\subset} s, \\ (\exists x \mathrel{\dot\in} s)F(x) & := & \exists x(x \mathrel{\dot\in} s \wedge F(x)), \\ (\forall x \mathrel{\dot\in} s)F(x) & := & \forall x(x \mathrel{\dot\in} s \to F(x)), \\ \Re(s) & := & \exists X \Re(s,X), \\ f \in (\Re \to \Re) & := & \forall x(\Re(x) \to \Re(fx)). \end{array}$$

Again, we will use the vector notation $\vec{\mathcal{Z}}$ to denote finite sequences $\mathcal{Z}_1, \ldots \mathcal{Z}_n$ of expressions. For example, for $\vec{U} = U_1, \ldots, U_n$ and $\vec{s} = s_1, \ldots, s_n$ we write

$$\begin{array}{rcl} \Re(\vec{s}, \vec{U}) & := & \Re(s_1, U_1) \wedge \ldots \wedge \Re(s_n, U_n), \\ \Re(\vec{s}) & := & \Re(s_1) \wedge \ldots \wedge \Re(s_n). \end{array}$$

Now we introduce the theory EETJ which provides a framework for explicit elementary types with join. Its logic is the classical *logic of partial terms* for individuals and classical logic for types. The non-logical axioms of EETJ can be divided into the following groups.

I. Applicative axioms. These axioms formalize that the individuals build a partial combinatory algebra, that we have paring and projections and the usual closure conditions on the natural numbers as well as definition by numerical cases. The theory consisting of the axioms of this group is called *basic theory of operations and numbers* BON, cf. Feferman and Jäger [38].

(1) $\mathsf{k}ab = a$,

(2) $\mathsf{s}ab\downarrow \wedge \mathsf{s}abc \simeq ac(bc)$,

(3) $\mathsf{p}_0 a\downarrow \wedge \mathsf{p}_1 a\downarrow$,

(4) $\mathsf{p}_0(a,b) = a \wedge \mathsf{p}_1(a,b) = b$,

(5) $0 \in \mathsf{N} \wedge (\forall x \in \mathsf{N})(\mathsf{s}_\mathsf{N} x \in \mathsf{N})$,

(6) $(\forall x \in \mathsf{N})(\mathsf{s}_\mathsf{N} x \neq 0 \wedge \mathsf{p}_\mathsf{N}(\mathsf{s}_\mathsf{N} x) = x)$,

(7) $(\forall x \in \mathsf{N})(x \neq 0 \to \mathsf{p}_\mathsf{N} x \in \mathsf{N} \wedge \mathsf{s}_\mathsf{N}(\mathsf{p}_\mathsf{N} x) = x)$,

(8) $a \in \mathsf{N} \wedge b \in \mathsf{N} \wedge a = b \to \mathsf{d}_\mathsf{N} xyab = x$,

(9) $a \in \mathsf{N} \wedge b \in \mathsf{N} \wedge a \neq b \to \mathsf{d}_\mathsf{N} xyab = y$.

II. Explicit representation and extensionality. The following are the usual ontological axioms for systems of explicit mathematics. They state that each type has a name, that there are no homonyms and that $\Re$ respects the extensional equality of types. Note that the representation of types by their names is intensional, while the types themselves are extensional in the usual set-theoretic sense.

(10) $\exists x \Re(x, A)$,

(11) $\Re(s, A) \wedge \Re(s, B) \to A = B$,

(12) $A = B \wedge \Re(s, A) \to \Re(s, B)$.

III. Basic type existence axioms.

In the following we assume that $z1, z2, \ldots$ and $Z1, Z2, \ldots$ are arbitrary but fixed enumerations of the individual and type variables of our language, respectively. If F is an elementary formula with no other individual variables than $z1, \ldots, zm$ and no other type variables than $Z1, \ldots, Zn$ and if $\vec{a} = a_1, \ldots, a_m$ and $\vec{S} = S_1, \ldots, S_n$, then we write $F[\vec{a}, \vec{S}]$ for the formula which results from F by a simultaneous replacement of zi by a_i and Zj by S_j $(1 \leq i \leq m, 1 \leq j \leq n)$.

Elementary Comprehension. Let F be an elementary formula of $\mathcal{L}_i$ with no individual variables other than $z1, \ldots, z(m+1)$ and no type variables other than $Z1, \ldots, Zn$ and let e be the Gödel number of F for any fixed Gödel numbering. Then we have the following axioms for all $\vec{a} = a_1, \ldots, a_m$, $\vec{b} = b_1, \ldots, b_n$ and $\vec{T} = T_1, \ldots, T_n$:

(13) $\Re(\vec{b}) \to \Re(\mathsf{c}_e(\vec{a}, \vec{b}))$,

(14) $\Re(\vec{b}, \vec{T}) \to \forall x(x \mathrel{\dot\in} \mathsf{c}_e(\vec{a}, \vec{b}) \leftrightarrow F[x, \vec{a}, \vec{T}])$.

Join

(15) $\Re(a) \wedge (\forall x \mathrel{\dot\in} a)\Re(fx) \;\to\; \Re(\mathsf{j}(a, f)) \wedge \Sigma(a, f, \mathsf{j}(a, f))$.

In this axiom the formula $\Sigma(a, f, b)$ means that b names the disjoint union of f over a, i.e.

$$\Sigma(a, f, b) \;:=\; \forall x(x \mathrel{\dot\in} b \leftrightarrow \exists y \exists z(x = (y, z) \wedge y \mathrel{\dot\in} a \wedge z \mathrel{\dot\in} fy)).$$

The theory EETJ consist of all the axioms (1)–(15). Later, we will consider extensions of EETJ by two different induction principles on the natural numbers. We are also interested in a *total* version of EETJ, meaning that all terms of $\mathcal{L}_p$ are defined in the sense of $\downarrow$. This can be achieved by the following axiom:

(Tot) $\forall x \forall y (xy\downarrow)$.

The theory $\mathsf{BON} + (\mathsf{Tot})$ is called TON. Jäger and Strahm [62] formalize a term model for TON and thereby, they show that the totality axiom adds nothing to the proof-theoretic strength of various applicative theories.

There are two crucial principles following already from the axioms of a partial combinatory algebra, i.e. the axioms (1) and (2) of BON: λ abstraction and a recursion theorem, cf. e.g. Beeson [8] or Feferman, Jäger and Strahm [40].

Definition 4. We define the $\mathcal{L}_p$ term $(\lambda x.t)$ by induction on the complexity of t as follows:

1. If t is the variable x, then $\lambda x.t$ is skk.
2. If t is a variable different from x or a constant, then $\lambda x.t$ is $\mathsf{k}t$.
3. If t is an application $(t_1 t_2)$, then $\lambda x.t$ is $\mathsf{s}(\lambda x.t_1)(\lambda x.t_2)$.

Next we have the expected theorem about λ abstraction, whose proof is standard. Using our definition of $\lambda x.t$, the first assertion of the theorem below is immediate by an easy inductive argument and by making use of axioms (1) and (2) of BON. The second assertion follows from the first by straightforward reasoning in the logic of partial terms.

Theorem 5 (λ abstraction). *For each $\mathcal{L}_p$ term t and all variables x there exists an $\mathcal{L}_p$ term $(\lambda x.t)$, whose variables are those of t, excluding x, so that*

1. $\mathsf{BON} \vdash \lambda x.t\downarrow \wedge (\lambda x.t)x \simeq t$;
2. $\mathsf{BON} \vdash s\downarrow \to (\lambda x.t)s \simeq t[s/x]$.

The definition of λ abstraction in the context of a *partial* combinatory algebra differs from the well-known definition in the setting of *total* combinatory logic: there one usually defines λ abstraction by:

$$\begin{array}{rcll} \lambda x.x & :\equiv & \mathsf{skk}, & \\ \lambda x.t & :\equiv & \mathsf{k}t, & \text{if } x \text{ is not a free variable of } t, \\ \lambda x.(rs) & :\equiv & \mathsf{s}(\lambda x.r)(\lambda x.s), & \text{otherwise.} \end{array}$$

If, in the sequel, we are working in a total setting, i.e. in a theory with the totality axiom (Tot), then we will implicitly use this definition of λ abstraction. However, it is not suitable in the partial setting since it no longer guarantees that $(\lambda x.t)$ is always defined.

The slightly more complicated definition of $(\lambda x.t)$ in BON has the drawback that it does not commute with substitutions: if x and y are distinct and x does not occur in the term s, then the two terms $(\lambda x.t)[s/y]$ and $(\lambda x.t[s/y])$ are in general not provably equal in BON. For a counterexample let t be the variable y and s the term (zz) for some variable z. Then $(\lambda x.t)[s/y]$ is the term $\mathsf{k}(zz)$ and $(\lambda x.t[s/y])$ is $\mathsf{s}(\mathsf{k}z)(\mathsf{k}z)$. This is not the case in the total theory TON.

However, in BON, a weaker form of the substitution principle for λ terms is provable, which states that the substitution into λ expressions is not problematic if the result of the substitution is immediately applied.

Lemma 6. *For all $\mathcal{L}_p$ terms s and t and different variables x and y of $\mathcal{L}_p$ we have*

$$\mathsf{BON} \vdash (\lambda x.t)[s/y]x \simeq t[s/y].$$

For a more detailed account to the problem of substitution in partial applicative theories see Strahm [91]. As usual, we generalize λ abstraction to several arguments by iterating abstraction for one argument, i.e. $\lambda x_1 \dots x_n.t$ abbreviates $\lambda x_1.(\dots(\lambda x_n.t)\dots)$.

Using λ abstraction we can define a *recursion combinator* in our system of explicit mathematics. Again, we have to distinguish whether we are in a partial or in a total setting.

Theorem 7 (Recursion). *There exist closed $\mathcal{L}_p$ terms rec_p and rec_t so that*

1. $\mathsf{BON} \vdash \mathsf{rec}_p f\downarrow \wedge \mathsf{rec}_p fx \simeq f(\mathsf{rec}_p f)x$,
2. $\mathsf{TON} \vdash \mathsf{rec}_t f = f(\mathsf{rec}_t f)$.

Proof. We let r be the $\mathcal{L}_p$ term $\lambda yx.f(yy)x$ and then set $\mathsf{rec}_p := \lambda f.rr$. Let t be the term $\lambda y.f(yy)$ and $\mathsf{rec}_t := \lambda f.tt$. Now the claims can be verified using the theorem about λ abstraction. 

In general, we cannot prove in the partial context of BON that $\mathsf{rec}_t f\downarrow$. Hence, in order to prove in BON that recursively defined functions have a value, we have to employ the slightly more complicated term rec_p. The price to pay is that we obtain a weaker form of the recursion theorem where the recursion equation holds pointwise only. As for λ abstraction, we will simply write rec and the context will ensure that it is clear whether rec_p or rec_t is meant.

In the following we employ two forms of induction on the natural numbers, type induction and formula induction. Type induction is the axiom

$(\mathsf{T}\text{-}\mathsf{I}_\mathsf{N})\ \ \forall X(0 \in X \wedge (\forall x \in \mathsf{N})(x \in X \to \mathsf{s}_\mathsf{N} x \in X) \ \to\ (\forall x \in \mathsf{N})(x \in X)).$

Formula induction, on the other hand, is the schema

$(\mathcal{L}_p\text{-}\mathsf{I}_\mathsf{N})\ \ F(0) \wedge (\forall x \in \mathsf{N})(F(x) \to F(\mathsf{s}_\mathsf{N} x)) \ \to\ (\forall x \in \mathsf{N}) F(x)$

for each $\mathcal{L}_p$ formula $F(u)$.

Of course, one can consider extensions of EETJ with further type existence principles such as, for example, *universes*. These are types which contain names only and which are closed under elementary comprehension and join, cf. Feferman [32] and Jäger, Kahle and Studer [61]. In the context of the theory of programming languages, universes are closely related to the concept of *predicative polymorphism*, cf. e.g. Nordström, Petersson and Smith [75] as well as Mitchell [72].

Systems of explicit mathematics with the principle of *name induction* have been studied by Kahle and Studer [68]. Name induction states that the names of the types can be only built by the use of the generators. This means that the naming relation $\Re$ is least and that only those types exist which are constructed by some generator. This induction principle provides a step into the impredicative world, and it may be of use for modelling *structural rules*. These rules occur, for example, in type systems dealing with record or object types and they rely on the assumption that the universe of types consists of record or object types only. For details, see Abadi and Cardelli [1]. Since name induction guarantees that only certain types exist, we think that it is a proper principle to study the semantics of structural rules. Another approach to interpret these rules is presented in the discussion of Chapter 4.

There are simple inductive model constructions for systems of explicit mathematics. If we work in a partial context, then the first order part of EETJ can be interpreted in the usual recursion-theoretic way, cf. Beeson [8]. This means that applications $a \cdot b$ in $\mathcal{L}_p$ are translated into $\{a\}(b)$, where $\{n\}$ for $n = 1, 2, 3, \ldots$ is a standard enumeration of the partial recursive functions. If we work with total functions, then we can make use of the formalized total term model of the theory TON provided by Jäger and Strahm [62]. For the type existence axioms, we can inductively generate codes for the types, and simultaneously we can also create a membership relation satisfying elementary comprehension and join. In order to establish a model, we need only a fixed point of this inductive definition, whereas minimality of the fixed point is not necessary, cf. Marzetta [71]. Hence, we obtain that the system $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathcal{L}_p\text{-}\mathsf{I}_\mathsf{N})$ is proof-theoretically equivalent to Martin-Löf's type theory with one universe ML_1 and also to $\widehat{\mathsf{ID}}_1$, the theory of non-iterated

positive arithmetical inductive definitions where only the fixed point property is asserted, cf. Feferman [32]. More on inductive model construction for systems of explicit mathematics (with universes) can be found in Jäger and Studer [63].

Chapter 3

Predicative Overloading

> Mathematics is an experimental science. It matters little that the mathematician experiments with pencil and paper while the chemist uses test-tube and retort, or the biologist stains and the microscope. The only great point of divergence between mathematics and the other sciences lies in the circumstance that experience only whispers 'yes' or 'no' in reply to our questions, while logic shouts.
>
> Norbert Wiener

> Trust me, I am a doctor of logic.
>
> Andrej Bauer

In the sequel we will present a model construction for $\lambda^{\{\}}_{str}$. Our model is not based on category theory; but the construction is performed in a theory of explicit mathematics. To handle late-binding, it is essential that there are first-order values acting for types. It is one of the main features of explicit mathematics that types are represented by names. These are first-order values and hence, we can apply functions to them. This means that computing with types is possible in such systems. Therefore, they are an adequate framework to deal with overloading and late-binding.

3.1 Introduction

As in the semantics for $\lambda\&-$early, we also define semantic codes for types, i.e. every type T of $\lambda^{\{\}}_{str}$ is represented by a natural number T^* in our theory of types and names. T^* is called the *symbol for the type* T. In the language of explicit mathematics, we find a term sub deciding the subtype relation on the type symbols. Using these constructions we can solve the problems mentioned by Castagna in the following way.

- Castagna [21] indicates that the key to model late-binding probably consists of interpreting terms as pairs (symbol for the type, interpretation of the computation). Then the computational part of the interpretation $[\![MN]\!]$ of an application would be something of the form

$$(\Lambda X.(\mathsf{p}_1[\![M]\!])[\![X]\!])(\mathsf{p}_0[\![N]\!])(\mathsf{p}_1[\![N]\!]). \tag{3.1}$$

  This remark is the starting point for our construction. We show that interpreting terms as pairs really does give a semantics for late-binding. When a term, interpreted as such a pair, is used as an argument in an application, its type is explicitly shown and can be used to compute the final result. Hence, this representation enables us to manage late-binding. We investigate how something of the form of (3.1) can be expressed in theories of types and names in order to model overloaded functions. As types are represented by names in explicit mathematics, we do not need a second order quantifier as in (3.1) and we can directly employ the symbol $\mathsf{p}_0[\![N]\!]$ for the type of the argument to select the best matching branch.

- In our model, types of $\lambda^{\{\}}$ will be interpreted as types of EETJ. Using join (disjoint unions) we can perform a kind of completion process on the types, so that the subtype relation can be interpreted by the standard subtype relation. Since types in explicit mathematics are extensional in the usual set-theoretic sense, this relation is an order relation and not just a preorder.

- An overloaded function type is interpreted as the type of functions that map an element of the domain of a branch into the range of that branch, for every branch of the overloaded function type. As the subtype relation is decidable and since an overloaded function consist only of finitely many branches, there exists a function typap such that for two types $\{S_i \to T_i\}_{i\in I}$ and S of $\lambda^{\{\}}_{str}$ with $S_j = \min_{i\in I}\{S_i \mid S \leq^- S_i\}$ we have

$$\mathsf{typap}(\{S_i \to T_i\}^*_{i\in I}, S^*) = \langle S^*_j, T^*_j\rangle.$$

  This means that typap yields symbols for the domain and the range of the branch to be selected. With this term we define the computational part f of the interpretation of a $\lambda^{\{\}}_{str}$ function

$$M := \lambda x(M_1 : S_1 \Rightarrow T_1, M_2 : S_2 \Rightarrow T_2)$$

  such that:

$$f([\![N]\!]) = \begin{cases} [\![M_1[N/x]]\!] & \text{if } \mathsf{typap}(\mathsf{p}_0[\![M]\!], \mathsf{p}_0[\![N]\!]) = \langle S^*_1, T^*_1\rangle, \\ [\![M_2[N/x]]\!] & \text{if } \mathsf{typap}(\mathsf{p}_0[\![M]\!], \mathsf{p}_0[\![N]\!]) = \langle S^*_2, T^*_2\rangle. \end{cases}$$

By means of the remark about late-binding, we know that $\mathsf{p_0}[\![M]\!]$ and $\mathsf{p_0}[\![N]\!]$ are symbols for the types of M and N, respectively. This demonstrates how types will affect the result of computations.

- We consider only the predicative version $\lambda^{\{\}}_{str}$ of $\lambda^{\{\}}$. The stratification of the type system allows us to define the semantics of the types by induction on the type structure.

3.2 Embedding $\lambda^{\{\}}_{str}$ into Explicit Mathematics

In this section we are going to carry out the embedding of $\lambda^{\{\}}_{str}$ into the theory $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathcal{L}_p\text{-}\mathsf{I_N})$ of explicit mathematics. First we represent each pretype T of $\lambda^{\{\}}_{str}$ by a natural number T^*, which will be called the *symbol for the pretype* T. We presume that there exists a term asub deciding the subtype relation on the symbols for atomic pretypes. Using this term we will define terms ptyp, sub and sub^- such that ptyp decides whether a natural number is a symbol for a pretype and sub, sub^- model the subtype relations $\leq$ and $\leq^-$, respectively, on the pretype symbols.

We define the type OTS of all symbols for types of $\lambda^{\{\}}_{str}$ with a well-ordering $\prec$ on it. Since we consider only a stratified type system, this can be done to such an extent that if a represents the type $\{S_i \to T_i\}_{i\in I}$ and b is a symbol for a strict subtype of any S_i or T_i, then $b \prec a$ holds. Therefore, it is possible to define by recursion a term type in such a way that applying this term to the symbol of any $\lambda^{\{\}}_{str}$ type T yields a name for its corresponding type in the system of explicit mathematics. This type will contain all the computational aspects of the interpretations of $\lambda^{\{\}}_{str}$ terms with type T. Then we can define the semantics for a type T of $\lambda^{\{\}}_{str}$ as the disjoint union of all types $\mathsf{type}(S^*)$ for strict subtypes S of T.

The interpretation of a $\lambda^{\{\}}_{str}$ term M is a pair in $\mathcal{L}_p$ consisting of the interpretation of the computational aspect of M and the symbol for its type. Hence, the type information is explicitly shown and can be used to model overloading and late-binding. To do so, we define a term typap which computes out of the symbols a, b for types $\{S_i \to T_i\}_{i\in I}$ and S the term $\langle S^*_j, T^*_j\rangle$ such that $S_j = \min_{i\in I}\{S_i \mid S \leq^- S_i\}$ holds. In other words typap can be employed to select the best matching branch of an overloaded function. Hence, it allows us to give the semantics for overloaded function terms of $\lambda^{\{\}}_{str}$ using definition by cases on natural numbers. We prove the soundness of our interpretation with respect to subtyping, type-checking and reductions.

First we recall that in $\mathsf{EETJ}+(\mathcal{L}_p\text{-}\mathsf{I_N})$ we can represent every primitive recursive function and relation as a closed term of $\mathcal{L}_p$. The term $\mathsf{s_N}0$ is denoted by 1 and we let $<$ denote the usual "less than" relation on the natural numbers. We will need to code finite sequences of natural numbers. Let $\langle x_1, \dots, x_n\rangle$ be the natural number which codes the sequence $x_1, \dots, x_n$ in any fixed coding. $\langle\rangle$ is the empty sequence. We have a length function len that satisfies $\mathsf{len}\langle\rangle = 0$ and $\mathsf{len}\langle x_1, \dots, x_n\rangle = n$. Further, there exists a projection function π so that $\pi i\langle x_1, \dots, x_i, \dots, x_n\rangle = x_i$ for all natural numbers $x_1, \dots, x_i, \dots, x_n$. We suppose that our coding satisfies the following property: if $a_i' < a_i$, then $\langle a_1, \dots, a_i', \dots, a_n\rangle < \langle a_1, \dots, a_i, \dots, a_n\rangle$ holds.

We introduce a translation * from pretypes of $\lambda^{\{\}}_{str}$ to $\mathcal{L}_p$ terms. If T is a pretype, then its *type symbol* T^* is defined as follows: let $A_1, A_2, \dots$ be any fixed enumeration of all atomic types of $\lambda^{\{\}}_{str}$, then we set $A_i^* := \langle 0, 0, i\rangle$ and $\{S_1 \to T_1, \dots, S_n \to T_n\}^*$ is defined as

$$\langle 1, r, \langle S_1^*, T_1^*\rangle, \dots, \langle S_n^*, T_n^*\rangle\rangle,$$

where $r = \max\{\pi 2(S_1^*), \pi 2(T_1^*), \dots, \pi 2(S_n^*), \pi 2(T_n^*)\}+1$. We coded the rank of a pretype at the second position in its symbol. If T is a pretype of $\lambda^{\{\}}_{str}$, then

$$\mathsf{EETJ} + (\mathsf{Tot}) + (\mathcal{L}_p\text{-}\mathsf{I_N}) \vdash \pi 2(T^*) = n \quad \Longleftrightarrow \quad \mathsf{rank}_\lambda(T) = n.$$

We assume that there is a closed individual term asub available, which adequately represents the subtype relation on the atomic type symbols, i.e.

1. $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathcal{L}_p\text{-}\mathsf{I_N})$ proves

$$(\forall x \in \mathsf{N})(\forall y \in \mathsf{N})(\mathsf{asub}(x, y) = 0 \vee \mathsf{asub}(x, y) = 1), \tag{3.2}$$

2. If S and T are pretypes of $\lambda^{\{\}}_{str}$, then we can prove in $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathcal{L}_p\text{-}\mathsf{I_N})$ that

$$\mathsf{asub}(S^*, T^*) = 1 \text{ if and only if } S \leq T \text{ and } S, T \text{ are atomic}, \tag{3.3}$$

3. $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathcal{L}_p\text{-}\mathsf{I_N})$ proves

$$\begin{aligned}(\forall x \in \mathsf{N})(\forall y \in \mathsf{N})(\forall z \in \mathsf{N})\\ (\mathsf{asub}(x, y) = 1 \wedge \mathsf{asub}(y, z) = 1 \ \to \mathsf{asub}(x, z) = 1).\end{aligned} \tag{3.4}$$

We find a closed individual term ptyp which decides whether a natural number n is a symbol for a pretype. If n is of the form $\langle 0,0,i\rangle$, then $\mathsf{ptyp}(n)$ is simply $\mathsf{asub}(n,n)$. Otherwise $\mathsf{ptyp}(n)$ is evaluated using primitive recursion according to the definition of the * translation. We define ptyp so that the following holds.

$$\mathsf{ptyp}(n) = \begin{cases} \mathsf{asub}(n,n) & \text{if } n = \langle 0,0,\pi 3n\rangle, \\ 1 & \text{if } \mathsf{len}n > 2 \wedge \pi 1n = 1 \wedge \\ & \quad (\forall i \in \mathsf{N})(2 < i \leq \mathsf{len}n \rightarrow \\ & \qquad \pi in = \langle \pi 1(\pi in), \pi 2(\pi in)\rangle \wedge \\ & \qquad \mathsf{ptyp}(\pi 1(\pi in)) = 1 \wedge \\ & \qquad \mathsf{ptyp}(\pi 2(\pi in)) = 1) \wedge \\ & \quad (\exists i \in \mathsf{N})(2 < i \leq \mathsf{len}n \wedge \\ & \qquad (\pi 2n = \pi 2(\pi 1(\pi in)) + 1 \vee \\ & \qquad\ \ \pi 2n = \pi 2(\pi 2(\pi in)) + 1)) \wedge \\ & \quad (\forall i \in \mathsf{N})(2 < i \leq \mathsf{len}n \rightarrow \\ & \qquad \pi 2n > \pi 2(\pi 1(\pi in)) \wedge \\ & \qquad \pi 2n > \pi 2(\pi 2(\pi in))), \\ 0 & \text{otherwise.} \end{cases}$$

Similarly, we can define two closed individual terms sub and sub^-, so that these terms properly represent the subtype relations $\leq$ and $\leq^-$, respectively, on the symbols for pretypes. These definitions use the terms asub and ptyp and they are again carried out by primitive recursion so that the following holds.

$$\mathsf{sub}(a,b) = \begin{cases} \mathsf{asub}(a,b) & \text{if } \mathsf{ptyp}(a) \wedge \mathsf{ptyp}(b) \wedge \pi 1a = 0 \wedge \pi 1b = 0, \\ 1 & \text{if } \mathsf{ptyp}(a) \wedge \mathsf{ptyp}(b) \wedge \pi 1a = 1 \wedge \pi 1b = 1 \wedge \\ & \quad (\forall i \in \mathsf{N})(2 < i \leq \mathsf{len}b \rightarrow \\ & \qquad (\exists j \in \mathsf{N})(2 < j < \mathsf{len}a \wedge \\ & \qquad\ \ \mathsf{sub}(\pi 1(\pi ib), \pi 1(\pi ja)) = 1 \wedge \\ & \qquad\ \ \mathsf{sub}(\pi 2(\pi ja), \pi 2(\pi ib)) = 1)), \\ 0 & \text{otherwise.} \end{cases}$$

$$\mathsf{sub}^-(a,b) = \begin{cases} \mathsf{asub}(a,b) & \text{if } \mathsf{ptyp}(a) \wedge \mathsf{ptyp}(b) \wedge \pi 1a = 0 \wedge \pi 1b = 0, \\ 1 & \text{if } \mathsf{ptyp}(a) \wedge \mathsf{ptyp}(b) \wedge \pi 1a = 1 \wedge \pi 1b = 1 \wedge \\ & \quad \pi 2a \leq \pi 2b \wedge \\ & \quad (\forall i \in \mathsf{N})(2 < i \leq \mathsf{len}b \rightarrow \\ & \qquad (\exists j \in \mathsf{N})(2 < j < \mathsf{len}a \wedge \\ & \qquad\ \ \mathsf{sub}^-(\pi 1(\pi ib), \pi 1(\pi ja)) = 1 \wedge \\ & \qquad\ \ \mathsf{sub}^-(\pi 2(\pi ja), \pi 2(\pi ib)) = 1)), \\ 0 & \text{otherwise.} \end{cases}$$

We get the following lemma concerning the terms sub and sub^-.

Lemma 8. *We can prove in* $\mathsf{EETJ}+(\mathsf{Tot})+(\mathcal{L}_p\text{-}\mathsf{I}_\mathsf{N})$ *the following three formulas:*

1. $(\forall x\in\mathsf{N})(\forall y\in\mathsf{N})(\mathsf{sub}(x,y)=0\vee\mathsf{sub}(x,y)=1)$,
2. $(\forall x\in\mathsf{N})(\forall y\in\mathsf{N})(\mathsf{sub}^-(x,y)=0\vee\mathsf{sub}^-(x,y)=1)$,
3. $(\forall x\in\mathsf{N})(\forall y\in\mathsf{N})(\forall z\in\mathsf{N})$
$(\mathsf{sub}^-(x,y)=1\wedge\mathsf{sub}^-(y,z)=1\ \rightarrow\mathsf{sub}^-(x,z)=1)$.

Let S,T *be pretypes of* $\lambda^{\{\}}_{str}$. *We have the following two facts:*

1. $\mathsf{EETJ}+(\mathsf{Tot})+(\mathcal{L}_p\text{-}\mathsf{I}_\mathsf{N})\vdash\mathsf{sub}(S^*,T^*)=1$ *if and only if* $S\leq T$ *is derivable in* $\lambda^{\{\}}_{str}$,
2. $\mathsf{EETJ}+(\mathsf{Tot})+(\mathcal{L}_p\text{-}\mathsf{I}_\mathsf{N})\vdash\mathsf{sub}^-(S^*,T^*)=1$ *if and only if* $S\leq^- T$ *is derivable in* $\lambda^{\{\}}_{str}$.

Proof. The first two formulas follow from the definitions of sub and sub^-, respectively. We employ (3.2) if a and b are symbols for atomic types.
The transitivity of sub^- is proved by induction. Let $F(n)$ be the formula

$$(\forall x\in\mathsf{N})(\forall y\in\mathsf{N})(\forall z\in\mathsf{N})(n=x+y+z\wedge \mathsf{sub}^-(x,y)=1\wedge\mathsf{sub}^-(y,z)=1\ \rightarrow\mathsf{sub}^-(x,z)=1).$$

We show $(\forall n\in\mathsf{N})F(n)$ by induction on n. Assume $n\in\mathsf{N}$ and

$$(\forall m\in\mathsf{N})(m<n\rightarrow F(m)). \tag{3.5}$$

Suppose $n=x+y+z\wedge\mathsf{sub}^-(x,y)=1\wedge\mathsf{sub}^-(y,z)=1$. If x, y and z are symbols for atomic types, then the claim follows by the transitivity of asub, see (3.4). Otherwise we have

$$(\forall i\in\mathsf{N})(2<i\leq\mathsf{len}z\rightarrow(\exists j\in\mathsf{N})(2<j<\mathsf{len}y\wedge \mathsf{sub}^-(\pi1(\pi iz),\pi1(\pi jy))=1\wedge\mathsf{sub}^-(\pi2(\pi jy),\pi2(\pi iz))=1))$$

and

$$(\forall j\in\mathsf{N})(2<j\leq\mathsf{len}y\rightarrow(\exists k\in\mathsf{N})(2<k<\mathsf{len}x\wedge \mathsf{sub}^-(\pi1(\pi jy),\pi1(\pi kx))=1\wedge\mathsf{sub}^-(\pi2(\pi kx),\pi2(\pi jy))=1)).$$

Since we have

$$\pi1(\pi iz)+\pi1(\pi jy)+\pi1(\pi kx)<z+y+x$$

and

$$\pi 2(\pi iz) + \pi 2(\pi jy) + \pi 2(\pi kx) < z + y + x,$$

for $2 < i \leq \mathsf{len}z$, $2 < j \leq \mathsf{len}y$ and $2 < k \leq \mathsf{len}x$, we can apply the induction hypothesis (3.5) to verify our claim.

The "if" part of the last two claims is proved by induction on the length of the derivation of $S \leq T$ and $S \leq^{-} T$, respectively. Again, we have to use the fact that asub is defined properly, see (3.3). The "only if" part is proved by induction on the pretype structure. We have to employ that if S the the overloaded function pretype $\{S_1 \to T_1, \ldots, S_n \to T_n\}$, then S_i and T_i are also pretypes for all $1 \leq i \leq n$ and again, we use (3.3) for the atomic types.

There is an elementary $\mathcal{L}_p$ formula $F(a)$ expressing the fact that the type represented by the symbol a satisfies the consistency conditions on good type formation. Hence, we can define a type OTS consisting of all symbols for $\lambda^{\{\}}_{str}$ types. This is done as follows. Let $\mathsf{min}_{\mathsf{OTS}}(b, z, a)$ be the $\mathcal{L}_p$ formula

$$\begin{array}{l} \mathsf{sub}^{-}(z, b) = 1 \wedge (\exists i \in \mathsf{N})(2 < i \leq \mathsf{len}a \wedge b = \pi 1(\pi ia)) \wedge \\ (\forall i \in \mathsf{N})(2 < i \leq \mathsf{len}a \ \to \\ \qquad (\mathsf{sub}^{-}(z, \pi 1(\pi ia)) = 1 \to \mathsf{sub}^{-}(b, \pi 1(\pi ia)) = 1)). \end{array}$$

That means $\mathsf{min}_{\mathsf{OTS}}(b, z, a)$ holds if and only if b is a minimal element of

$$\{x \mid \mathsf{sub}^{-}(z, x) = 1 \wedge (\exists i \in \mathsf{N})(2 < i \leq \mathsf{len}a \wedge x = \pi 1(\pi ia))\}.$$

We introduce the abbreviation $\mathsf{Cond}(a)$ for

$$\begin{array}{l} (\forall i \in \mathsf{N})(\forall j \in \mathsf{N})(2 < i \leq \mathsf{len}a \wedge 2 < j \leq \mathsf{len}a \ \to \\ \qquad (\mathsf{sub}(\pi 1(\pi ia), \pi 1(\pi ja)) = 1 \to \mathsf{sub}(\pi 2(\pi ia), \pi 2(\pi ja)) = 1) \wedge \\ \qquad (\mathsf{sub}^{-}(\pi 1(\pi ia), \pi 1(\pi ja)) = 1 \to \mathsf{sub}^{-}(\pi 2(\pi ia), \pi 2(\pi ja)) = 1) \wedge \\ \qquad \forall z(\mathsf{ptyp}(z) = 1 \wedge \mathsf{sub}(z, \pi 1(\pi ia)) = 1 \ \to \\ \qquad\qquad (\exists l \in \mathsf{N})(2 < l \leq \mathsf{len}a \wedge \mathsf{min}_{\mathsf{OTS}}(\pi 1(\pi la), z, a) \wedge \\ \qquad\qquad\qquad (\forall k \in \mathsf{N})(2 < k \leq \mathsf{len}a \wedge \mathsf{min}_{\mathsf{OTS}}(\pi 1(\pi ka), z, a) \to k = l)))). \end{array}$$

The formula $\mathsf{Cond}(a)$ states that if a is a symbol for a pretype, then is satisfies the consistency conditions 2 and 3 concerning good type formation. Now we can give the definition of the overloaded type symbols by the following

construction, which again just mimics the syntactic definition of $\lambda^{\{\}}_{str}$ types.

$$\begin{aligned}
\mathsf{A}_{\mathsf{OTS}}(z,f) :=\ & \mathsf{Cond}(f(z)) \wedge \\
& (\forall i \in \mathsf{N})(2 < i \leq \mathsf{len}(f(z)) \ \rightarrow \\
& \quad (\exists r \in \mathsf{N})(\exists s \in \mathsf{N})(r < z \wedge s < z \wedge \\
& \quad\quad \pi 1(\pi i(f(z))) = f(r) \wedge \pi 2(\pi i(f(z))) = f(s) \\
\mathsf{B}_{\mathsf{OTS}}(y,f) :=\ & (\forall z \in \mathsf{N})(z \leq y \ \rightarrow \\
& \quad \mathsf{ptyp}(f(z)) = 1 \wedge (\pi 1(f(z)) = 0 \vee \mathsf{A}_{\mathsf{OTS}}(z,f))) \\
\mathsf{C}_{\mathsf{OTS}}(x,y,f) :=\ & y \in \mathsf{N} \wedge x = f(y) \wedge \mathsf{B}_{\mathsf{OTS}}(y,f)
\end{aligned}$$

The type OTS of the overloaded type symbols is now given by

$$\{x \mid \exists y \exists f \mathsf{C}_{\mathsf{OTS}}(x,y,f)\}.$$

Since elementary comprehension is uniform, there are closed individual $\mathcal{L}_p$ terms domain and range satisfying the following property: if $\{S_i \to T_i\}_{i\in I}$ is an overloaded function type of $\lambda^{\{\}}_{str}$ and a is its symbol, then

1. $\mathsf{domain}(a)$ is a name for the type containing all symbols $x \in \mathsf{OTS}$ for which there is an $i \in I$ such that $\mathsf{sub}^-(x, S^*_i) = 1$ holds,

2. $\mathsf{range}(a)$ represents the type consisting of all symbols $x \in \mathsf{OTS}$ for which there is an $i \in I$ such that $\mathsf{sub}^-(x, T^*_i) = 1$ holds.

That is $x \dot{\in} \mathsf{domain}(a)$ denotes a strict subtype of an S_i and $x \dot{\in} \mathsf{range}(a)$ is a symbol for a strict subtype of a T_i for $i \in I$. This is achieved by the following definitions.

Definition 9. The term $\mathsf{domain}(a)$ represents the type

$$\{x \in \mathsf{OTS} \mid \pi 1a = 1 \wedge (\exists i \in \mathsf{N})(3 \leq i \leq \mathsf{len}a \wedge \mathsf{sub}^-(x, \pi 1(\pi ia)) = 1)\}$$

and $\mathsf{range}(a)$ is a name for

$$\{x \in \mathsf{OTS} \mid \pi 1a = 1 \wedge (\exists i \in \mathsf{N})(3 \leq i \leq \mathsf{len}a \wedge \mathsf{sub}^-(x, \pi 2(\pi ia)) = 1)\}.$$

Now we define the well-ordering $\prec$ on the type symbols so that we can build our types by induction along $\prec$. For this definition remember that the second component $\pi 2m$ of a type symbol m codes its rank. For two types S and T of $\lambda^{\{\}}_{str}$, we will have $S^* \prec T^*$ if and only if $\mathsf{rank}_\lambda(S) < \mathsf{rank}_\lambda(T)$.

Definition 10. The ordering $\prec$ is defined by

$$a \prec b \text{ if and only if } a \in \mathsf{OTS} \wedge b \in \mathsf{OTS} \wedge \pi 2a < \pi 2b.$$

With this definition, we immediately get the following lemma. Keep in mind that the rank of the elements of $\mathsf{domain}(m)$ and $\mathsf{range}(m)$, respectively, is strictly smaller than the rank of m for a type symbol m.

Lemma 11. *We can prove in* $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathcal{L}_p\text{-}\mathsf{I_N})$

1. $a \in \mathsf{OTS} \wedge b \mathrel{\dot\in} \mathsf{domain}(a) \to b \prec a$,
2. $a \in \mathsf{OTS} \wedge b \mathrel{\dot\in} \mathsf{range}(a) \to b \prec a$,
3. $(\forall x \in \mathsf{OTS})((\forall y \in \mathsf{OTS})(y \prec x \to F(y)) \to F(x)) \to$
   $(\forall x \in \mathsf{OTS})F(x)$, *for arbitrary formulas* F *of* $\mathcal{L}_p$.

Proof. To prove the first claim, we assume $a \in \mathsf{OTS} \wedge b \mathrel{\dot\in} \mathsf{domain}(a)$. Hence, $b \in \mathsf{OTS}$ and $(\exists i \in \mathsf{N})(3 \leq i \leq \mathsf{len}a \wedge \mathsf{sub}^-(b, \pi 1(\pi i a)) = 1)$. For this i we have $\pi 2(\pi 1(\pi i a)) < \pi 2 a$ by the definition of the overloaded type symbols since the rank of an overloaded function type is strictly bigger than the rank of its domains and ranges, respectively. For $x \in \mathsf{OTS}$ and $y \in \mathsf{OTS}$ we have $\mathsf{sub}^-(x, y) \to \pi 2 x \leq \pi 2 y$. We obtain $\pi 2 b < \pi 2 a$, which implies $b \prec a$.

The second claim is proved similarly.

Now we prove the third claim. First, we define an auxiliary $\mathcal{L}_p$ term $\mathsf{put_2}$ so that

$$\mathsf{put_2}(n, z) = \langle b_1, n, b_2, \ldots, b_l \rangle$$

if $z = \langle b_1, b_2, \ldots, b_l \rangle$ that is if z is a sequence number with $\mathsf{len}z > 0$. Otherwise we can set $\mathsf{put_2}(n, z) = \langle n \rangle$. Assume

$$(\forall x \in \mathsf{OTS})((\forall y \in \mathsf{OTS})(y \prec x \to F(y)) \to F(x)), \tag{3.6}$$

for an arbitrary $\mathcal{L}_p$ formula F. We have to show $(\forall x \in \mathsf{OTS})F(x)$. We define an auxiliary formula $G(n, z)$ by

$$\mathsf{put_2}(n, z) \in \mathsf{OTS} \to F(\mathsf{put_2}(n, z)). \tag{3.7}$$

We will show

$$(\forall n \in \mathsf{N})((\forall m \in \mathsf{N})(m < n \to (\forall z \in \mathsf{N})G(m, z)) \to (\forall z \in \mathsf{N})G(n, z)). \tag{3.8}$$

Then we obtain by induction on the natural numbers

$$(\forall n \in \mathsf{N})(\forall z \in \mathsf{N})G(n, z),$$

which implies $(\forall x \in \mathsf{OTS})F(x)$ since for all x in OTS we find natural numbers n and z so that $x = \mathsf{put_2}(n, z)$. To show (3.8), we let $n \in \mathsf{N}$ and assume

$$(\forall m \in \mathsf{N})(m < n \to (\forall z \in \mathsf{N})G(m, z)). \tag{3.9}$$

It remains to show $(\forall z \in \mathsf{N})G(n,z)$. Let $z \in \mathsf{N}$ and assume

$$\mathsf{put}_2(n,z) \in \mathsf{OTS}.$$

Further we set $x = \mathsf{put}_2(n,z)$. Hence, we have to show $F(x)$. This follows from (3.6) if we can show

$$(\forall y \in \mathsf{OTS})(y \prec x \to F(y)). \tag{3.10}$$

So let $y \in \mathsf{OTS}$ with $y \prec x$. We find natural numbers p and q so that $p < n$ and $\mathsf{put}_2(p,q) = y$. With (3.9) we obtain $G(p,q)$, which implies $F(y)$ by (3.7). Hence, (3.10) is shown, which finishes this proof. 

Since the subtype relation on the type symbols is decidable, i.e. we have the $\mathcal{L}_p$ term sub^- at our disposal, we find a closed individual term typap of $\mathcal{L}_p$ selecting the best matching branch in an application. Assume $\{S_i \to T_i\}_{i\in I}$ and S are types of $\lambda^{\{\}}_{str}$. Then we have

$$\mathsf{typap}(\{S_i \to T_i\}^*_{i\in I}, S^*) = \langle S^*_j, T^*_j\rangle,$$

if $S_j = \min_{i\in I}\{S_i \mid S \leq^- S_i\}$. We set $\mathsf{typap}(\{S_i \to T_i\}^*_{i\in I}, S^*) = 0$ if such an S_j does not exist. Hence, $\mathsf{typap}(\{S_i \to T_i\}^*_{i\in I}, S^*) = \langle S^*_j, T^*_j\rangle$ means, that if a $\lambda^{\{\}}_{str}$ term M of type $\{S_i \to T_i\}_{i\in I}$ is applied to a $\lambda^{\{\}}_{str}$ term N of type S, then the j^{th} branch of M will be applied to N. As a result of this, S is best approximated by S_j.

We assume that there is term $\mathsf{t_G}$ which assigns to each symbol for an atomic type of $\lambda^{\{\}}_{str}$ the name of its corresponding type in explicit mathematics. If, for example, we have just one atomic type in $\lambda^{\{\}}_{str}$ consisting exactly of the natural numbers, then its symbol is $\langle 0,1\rangle$ and its assigned type is $\{a \mid N(a)\}$. If t is a name for this type, then we can choose $\mathsf{t_G} := \lambda x.t$. If there are two symbols a, b with $\pi 1a = 0$ and $\pi 1b = 0$ and $\mathsf{sub}^-(a,b) = 1$ (e.g. symbols for atomic types S, T of $\lambda^{\{\}}_{str}$ with $S \leq^- T$), then $\mathsf{t_G}$ has to satisfy $\mathsf{t_G}(a) \dot{\subset} \mathsf{t_G}(b)$. This means that $\mathsf{t_G}$ has to respect the subtype hierarchy on the type symbols given by sub. With reference to the recursion theorem we define a closed individual term type of $\mathcal{L}_p$ satisfying

$$\mathsf{type}\ m = \begin{cases} \mathsf{t_G}\ m & \text{if } \pi 1m = 0, \\ \mathsf{t_O}\ \mathsf{type}\ m & \text{if } \pi 1m = 1, \end{cases}$$

where $\mathsf{t_O}\ \mathsf{type}\ m$ is a name for

$$\begin{aligned} \{f \mid\ & (\forall a \dot{\in} \mathsf{domain}(m))(\forall x \dot{\in} \mathsf{type}(a)) \\ & (\mathsf{p_0}(f(a,x)) \dot{\in} \mathsf{range}(m) \wedge \mathsf{p_1}(f(a,x)) \dot{\in} \mathsf{type}(\mathsf{p_0}(f(a,x))) \wedge \\ & \mathsf{sub}^-(\mathsf{p_0}(f(a,x)), \pi 2(\mathsf{typap}(m,a))) = 1)\}. \end{aligned} \tag{3.11}$$

This type depends on the terms type and m. Since in explicit mathematics the representation of types by names is uniform in the parameters of the types, there exists a term $\mathsf{t_O}$ such that $\mathsf{t_O}\ \mathsf{type}\ m$ is a name for the above type.

Now, we first will explain which individuals are contained in $\mathsf{type}\ m$ for a type symbol m. Then we will give the technical definition of the term $\mathsf{t_O}$ in explicit mathematics and show that $\mathsf{type}\ m$ is a name for $m \in \mathsf{OTS}$.

If A is an atomic type, then $\mathsf{type}(A^*)$ is a name for its corresponding type defined by the term $\mathsf{t_G}(A^*)$. Otherwise, if we are given an overloaded function type $\{S_i \to T_i\}_{i\in I}$, then $\mathsf{type}(\{S_i \to T_i\}^*_{i\in I})$ contains all functions f satisfying for every type S of $\lambda^{\{\}}_{str}$, every $i \in I$ and every term x of $\mathcal{L}_p$ with $S \leq^- S_i$ and $x \mathrel{\dot\in} \mathsf{type}(S^*)$ the following:

1. $\mathsf{p_1}(f(S^*, x)) \mathrel{\dot\in} \mathsf{type}(\mathsf{p_0}(f(S^*, x)))$
2. $\mathsf{p_0}(f(S^*, x))$ denotes a strict subtype of T_j, where
$$T^*_j = \pi 2(\mathsf{typap}(\{S_i \to T_i\}^*_{i\in I}, S^*)),$$
i.e. $S_j = \min_{i\in I}\{S_i \mid S \leq^- S_i\}$.

In this definition of type, there is a kind of type completion built in. Assume m is a symbol for an overloaded function type $\{S \to T\}$ and $f \mathrel{\dot\in} \mathsf{type}(m)$. Then $f(a, x)$ is defined for all $a \mathrel{\dot\in} \mathsf{domain}(m)$ and for all $x \mathrel{\dot\in} \mathsf{type}(a)$. Since $\mathsf{domain}(m)$ contains all symbols $S^*_1, S^*_2, \ldots$ for strict subtypes of S, the term $\mathsf{type}(m)$ represents in a sense the type $\{S \to T, S_1 \to T, S_2 \to T, \ldots\}$. In this way, we make use of a form of type completion to handle the problem of the preordering of the types.

Now we are going to present the technical definition of $\mathsf{t_O}$. Let ots be a name for OTS. There are closed individual terms $\mathsf{j_1}$, $\mathsf{j_2}$, $\mathsf{j_3}$ and $\mathsf{j_4}$ of $\mathcal{L}_p$ so that

$$\begin{array}{ll} \mathsf{j_1} := \mathsf{j}(\mathsf{ots}, \mathsf{domain}), & \mathsf{j_2}(m, t) := \mathsf{j}(\mathsf{domain}(m), t), \\ \mathsf{j_3} := \mathsf{j}(\mathsf{ots}, \mathsf{range}), & \mathsf{j_4}(m, t) := \mathsf{j}(\mathsf{range}(m), t). \end{array}$$

Let $\mathsf{G}(f, m, J_{od}, J_{dt}, J_{or}, J_{rt})$ be the $\mathcal{L}_p$ formula

$$\begin{array}{l} \forall a \forall x((m, a) \in J_{od} \wedge (a, x) \in J_{dt} \ \to \\ \qquad (m, \mathsf{p_0}(f(a, x))) \in J_{or} \wedge (\mathsf{p_0}(f(a, x)), \mathsf{p_1}(f(a, x))) \in J_{rt} \wedge \\ \qquad \mathsf{sub}^-(\mathsf{p_0}(f(a, x)), \pi 2(\mathsf{typap}(m, a))) = 1). \end{array}$$

Since elementary comprehension is uniform, we find a closed individual term g of $\mathcal{L}_p$ so that

$$\begin{array}{l} \Re(j_{od}, J_{od}) \wedge \Re(j_{dt}, J_{dt}) \wedge \Re(j_{or}, J_{or}) \wedge \Re(j_{rt}, J_{rt}) \ \to \\ \qquad \Re(\mathsf{g}(m, j_{od}, j_{dt}, j_{or}, j_{rt}), \{f \mid \mathsf{G}(f, m, J_{od}, J_{dt}, J_{or}, J_{rt})\}). \end{array} \tag{3.12}$$

Now we can define the term $\mathsf{t_O}$ by

$$\mathsf{t_O} := \lambda t \lambda m.\mathsf{g}(m, \mathsf{j_1}, \mathsf{j_2}(m,t), \mathsf{j_3}, \mathsf{j_4}(m,t)).$$

The term $\mathsf{t_O}\ \mathsf{type}\ m$ is indeed a name for the type given in (3.11). We observe the following facts about the types represented by $\mathsf{j_1}$, $\mathsf{j_2}(m, \mathsf{type})$, $\mathsf{j_3}$, $\mathsf{j_4}(m, \mathsf{type})$ for an overloaded function type symbol $m \in \mathsf{OTS}$ with $\pi 1 m = 1$. We have

$$\begin{gathered}(m,a) \mathrel{\dot\in} \mathsf{j_1} \leftrightarrow m \in \mathsf{OTS} \wedge a \mathrel{\dot\in} \mathsf{domain}(m),\\ (a,x) \mathrel{\dot\in} \mathsf{j_2}(m,\mathsf{type}) \leftrightarrow a \mathrel{\dot\in} \mathsf{domain}(m) \wedge x \mathrel{\dot\in} \mathsf{type}(a),\\ (m,b) \mathrel{\dot\in} \mathsf{j_3} \leftrightarrow m \in \mathsf{OTS} \wedge b \mathrel{\dot\in} \mathsf{range}(m),\\ (b,z) \mathrel{\dot\in} \mathsf{j_4}(m,\mathsf{type}) \leftrightarrow b \mathrel{\dot\in} \mathsf{range}(m) \wedge z \mathrel{\dot\in} \mathsf{type}(b).\end{gathered}$$

When we plug in these equivalences in the formula G, as it is done in (3.12), we obtain by elementary comprehension the type given in (3.11).

Using Lemma 11 we can prove that for every type symbol $m \in \mathsf{OTS}$ the term $\mathsf{type}(m)$ represents a type in explicit mathematics. We have the following theorem.

Theorem 12. $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathcal{L}_p\text{-}\mathsf{I_N})$ *proves:* $(\forall m \in \mathsf{OTS})\Re(\mathsf{type}(m))$.

Proof. Let m be an arbitrary type symbol in OTS. We show

$$(\forall y \in \mathsf{OTS})(y \prec m \to \Re(\mathsf{type}(y))) \to \Re(\mathsf{type}(m)). \tag{3.13}$$

Then the claim follows by induction on $\prec$ (see Lemma 11). So assume

$$(\forall y \in \mathsf{OTS})(y \prec m \to \Re(\mathsf{type}(y))). \tag{3.14}$$

We have to show $\Re(\mathsf{type}(m))$. Now we distinguish two cases. First, if m is a symbol for an atomic type, that is $\pi 1 m = 0$, then $\mathsf{type}(m) = \mathsf{t_G}(m)$ and $\Re(\mathsf{t_G}(m))$ by the definition of $\mathsf{t_G}$. Second, if m is a symbol for an overloaded function type, that is $\pi 1 m = 1$, then $\mathsf{type}(m) = \mathsf{t_O}\ \mathsf{type}\ m$ and we have to make use of the induction hypothesis (3.14). By Lemma 11 we get

$$p \mathrel{\dot\in} \mathsf{domain}(m) \to p \prec m \text{ and } q \mathrel{\dot\in} \mathsf{range}(m) \to q \prec m.$$

Hence, (3.14) implies

$$p \mathrel{\dot\in} \mathsf{domain}(m) \to \Re(\mathsf{type}(p)) \text{ and } q \mathrel{\dot\in} \mathsf{range}(m) \to \Re(\mathsf{type}(q)).$$

Therefore we obtain $\Re(\mathsf{j_2}(m, \mathsf{type}))$ and $\Re(\mathsf{j_4}(m, \mathsf{type}))$. Moreover, we have $\Re(\mathsf{j_1})$ and $\Re(\mathsf{j_4})$. By (3.12) we conclude $\Re(\mathsf{t_O}\ \mathsf{type}\ m)$, that is $\Re(\mathsf{type}(m))$ and (3.13) is shown. 

These types satisfy the following subtype property.

Lemma 13. $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathcal{L}_p\text{-}\mathsf{I_N})$ *proves:*

$$(\forall a \in \mathsf{OTS})(\forall b \in \mathsf{OTS})(\mathsf{sub}^-(a,b) = 1 \to \mathsf{type}(a) \dot{\subset} \mathsf{type}(b)).$$

Proof. By the definition of the term sub^- we have either $\pi 1a = 0 \wedge \pi 1b = 0$ or $\pi 1a = 1 \wedge \pi 1b = 1$. That is either a and b are both symbols for atomic types or both symbols represent an overloaded function type. Let us distinguish these two cases. If $\pi 1a = 0 \wedge \pi 1b = 0$, then the claim follows by the definition of $\mathsf{t_G}$. This definition requires that $\mathsf{t_G}$ has to respect the subtype hierarchy on the symbols for pretypes given by sub^-. If we are in the second case, then assume $f \dot{\in} \mathsf{type}(a)$. We have to show $f \dot{\in} \mathsf{type}(b)$. So suppose $p \dot{\in} \mathsf{domain}(b)$ and $x \dot{\in} \mathsf{type}(p)$. We get $p \dot{\in} \mathsf{domain}(a)$ by $\mathsf{sub}^-(a,b) = 1$ and the definitions of domain and sub^-. Hence, by $f \dot{\in} \mathsf{type}(a)$ we find $\mathsf{p_0}(f(p,x)) \dot{\in} \mathsf{range}(a)$,

$$\mathsf{p_1}(f(p,x)) \dot{\in} \mathsf{type}(\mathsf{p_0}(f(p,x)) \tag{3.15}$$

and

$$\mathsf{sub}^-(\mathsf{p_0}(f(p,x)), \pi 2(\mathsf{typap}(a,p))) = 1. \tag{3.16}$$

By $\mathsf{sub}^-(a,b) = 1$ we obtain that there exists $2 < j \leq \mathsf{len}a$ so that

$$\mathsf{sub}^-(\pi 1(\mathsf{typap}(b,p)), \pi 1(\pi ja)) = 1$$

and

$$\mathsf{sub}^-(\pi 2(\pi ja), \pi 2(\mathsf{typap}(b,p))) = 1. \tag{3.17}$$

By the definition of typap we get $\mathsf{sub}^-(\pi 1(\mathsf{typap}(a,p)), \pi 1(\pi ja)) = 1$. Therefore, we have

$$\mathsf{sub}^-(\pi 2(\mathsf{typap}(a,p)), \pi 2(\pi ja)) = 1 \tag{3.18}$$

by the consistency conditions concerning good type formation. Hence, we get by (3.16), (3.18), (3.17) and the transitivity of sub^-, see Lemma 8 that

$$\mathsf{sub}^-(\mathsf{p_0}(f(p,x)), \pi 2(\mathsf{typap}(b,p))) = 1. \tag{3.19}$$

This also implies

$$\mathsf{p_0}(f(p,x)) \dot{\in} \mathsf{range}(b). \tag{3.20}$$

Finally, we conclude by (3.20), (3.15) and (3.19) that $f \in \mathsf{type}(b)$. 

Now we define a closed individual term type_2 of $\mathcal{L}_p$, so that for all $a \in \mathsf{OTS}$ we have $\Re(\mathsf{type}_2(a))$ and

$$\forall m \forall f((m, f) \dot{\in} \mathsf{type}_2(a) \leftrightarrow m \in \mathsf{OTS} \wedge \mathsf{sub}^-(m, a) = 1 \wedge f \dot{\in} \mathsf{type}(m)).$$

This is achieved with the following definition. Let h be a closed $\mathcal{L}_p$ term satisfying $\Re(h(a), \{m \mid m \in \mathsf{OTS} \wedge \mathsf{sub}^-(m, a) = 1\})$. Now, we can define $\mathsf{type}_2 := \lambda x.\mathsf{j}(h(x), \mathsf{type})$. Since sub^- is transitive, see Lemma 8, the following lemma about subtyping is just a corollary of the definition of the $\mathcal{L}_p$ term type_2.

Lemma 14. $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathcal{L}_p\text{-}\mathsf{I_N})$ *proves:*

$$(\forall a \in \mathsf{OTS})(\forall b \in \mathsf{OTS})(\mathsf{sub}^-(a, b) = 1 \rightarrow \mathsf{type}_2(a) \dot{\subset} \mathsf{type}_2(b)).$$

With the $\mathcal{L}_p$ term type_2 we define the interpretation of $\lambda^{\{\}}_{str}$ types as follows.

Definition 15. Interpretation $[\![T]\!]$ of a $\lambda^{\{\}}_{str}$ type T
If T is a type of $\lambda^{\{\}}_{str}$, then $[\![T]\!]$ is the type represented by $\mathsf{type}_2(T^*)$.

As an immediate consequence of this definition and the previous lemmas about subtyping, we obtain the soundness of our interpretation with respect to subtyping.

Theorem 16. *Let S, T be types of $\lambda^{\{\}}_{str}$ with $S \leq^- T$, then*

$$\mathsf{EETJ} + (\mathsf{Tot}) + (\mathcal{L}_p\text{-}\mathsf{I_N}) \vdash [\![S]\!] \subset [\![T]\!].$$

Terms of $\lambda^{\{\}}_{str}$ will be interpreted as ordered pairs, where the first component is a symbol for the type of the term and the second component models the computational aspect of the term. To define the semantics for $\lambda^{\{\}}_{str}$ terms we need an injective translation $\hat{}$ from the variables of $\lambda^{\{\}}_{str}$ to the individual variables of $\mathcal{L}_p$. Then the computational part of a $\lambda^{\{\}}_{str}$ term $\lambda x.(M_i : S_i \Rightarrow T_i)_{i \in I}$ can be interpreted by a function f as such that if $\mathsf{typap}(\{S_i \rightarrow T_i\}^*_{i \in I}, \mathsf{p_0} y) = \langle S^*_j, T^*_j \rangle$ holds, then f satisfies

$$f(y) = (\lambda \hat{x}.[\![M_j]\!])y.$$

Such a function exists, because an overloaded function is composed only of finitely many branches and we have the $\mathcal{L}_p$ term typap available, which selects the best matching branch. An application of two $\mathcal{L}_p$ terms MN is simply modeled by applying the function $\mathsf{p_1}[\![M]\!]$ to $[\![N]\!]$.

Definition 17. Interpretation $[\![M]\!]$ of a $\lambda^{\{\}}_{str}$ term M
We define $[\![M]\!]$ by induction on the term structure:

1. $M \equiv x$: $[\![M]\!] := \hat{x}$.
2. $M \equiv \lambda x.(M_i : S_i \Rightarrow T_i)_{i\in\{1,\dots,n\}}$: $[\![M]\!] := (\{S_i \to T_i\}^*_{i\in\{1,\dots,n\}}, f)$, where f is defined as follows:

$$f(y) := \begin{cases} (\lambda\hat{x}.[\![M_1]\!])y, & \mathsf{typap}(\{S_i \to T_i\}^*_{i\in\{1,\dots,n\}}, \mathsf{p}_0 y) = \langle S_1^*, T_1^*\rangle, \\ \vdots & \\ (\lambda\hat{x}.[\![M_n]\!])y, & \mathsf{typap}(\{S_i \to T_i\}^*_{i\in\{1,\dots,n\}}, \mathsf{p}_0 y) = \langle S_n^*, T_n^*\rangle. \end{cases}$$

3. $M \equiv M_1 M_2$: $[\![M]\!]$ is defined as $\mathsf{p}_1[\![M_1]\!][\![M_2]\!]$.

Employing definition by cases on natural numbers we can combine the interpretations of the branches of a $\lambda^{\{\}}_{str}$ term M defined by λ abstraction to one overloaded function which serves as the interpretation of M. This definition by cases is evaluated using the typap function and the type information which is shown in M for each branch.

Before we can prove two of our main results, soundness of our interpretation with respect to type-checking and to reductions, we have to mention the following preparatory lemma.

Lemma 18. *If M and N are terms of $\lambda^{\{\}}_{str}$, then $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathcal{L}_p\text{-}\mathsf{I}_\mathsf{N})$ proves:*

$$[\![M]\!][[\![N]\!]/\hat{x}] = [\![M[N/x]]\!].$$

Proof. The proof proceeds by induction on the term structure of M. The only critical case is when M is defined by λ abstraction. There, totality in our system of explicit mathematics is essential since it guarantees that substitution is compatible with λ abstraction. 

We define the interpretation $[\![\Gamma]\!]$ of a context $x_1 : T_1, \dots, x_n : T_n$ as

$$[\![x_1]\!] \in [\![T_1]\!] \wedge \dots \wedge [\![x_n]\!] \in [\![T_n]\!].$$

Our interpretation is sound with respect to type checking.

Theorem 19. *If $\Gamma \vdash M : T$ holds in $\lambda^{\{\}}_{str}$, then in $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathcal{L}_p\text{-}\mathsf{I}_\mathsf{N})$ one can prove:*

$$[\![\Gamma]\!] \to [\![M]\!] \in [\![T]\!].$$

Proof. The proof proceeds by induction on $\Gamma \vdash M : T$.

1. $M \equiv x$: trivial.

2. $M \equiv \lambda x.(M_i : S_i \Rightarrow T_i)_{i\in I}$: let $f := \mathsf{p}_1[\![M]\!]$. The type T is of the form $\{S_i \to T_i\}_{i\in I}$. Therefore, we have to show $(T^*, f) \mathrel{\dot\in} \mathsf{type}_2(T^*)$. That is $f \mathrel{\dot\in} \mathsf{type}(T^*)$, i.e.

$$\begin{aligned}(\forall a \mathrel{\dot\in} \mathsf{domain}(T^*))(\forall y \mathrel{\dot\in} \mathsf{type}(a))\\ (\mathsf{p}_0(f(a,y)) \mathrel{\dot\in} \mathsf{range}(T^*) \wedge \mathsf{p}_1(f(a,y)) \mathrel{\dot\in} \mathsf{type}(\mathsf{p}_0(f(a,y))) \wedge\\ \mathsf{sub}^-(\mathsf{p}_0(f(a,y)), \pi 2(\mathsf{typap}(T^*, a))) = 1).\end{aligned} \tag{3.21}$$

   Choose $a \mathrel{\dot\in} \mathsf{domain}(T^*)$, $y \mathrel{\dot\in} \mathsf{type}(a)$ and let the natural number j be such that $\mathsf{typap}(T^*, a) = \langle S_j^*, T_j^*\rangle$, then we obtain

$$f(a,y) = (\lambda \hat{x}.[\![M_j]\!])(a,y)$$

   by the definition of f. With the induction hypothesis we get

$$[\![\Gamma]\!] \wedge\ [\![x]\!] \in [\![S_j]\!] \to [\![M_j]\!] \in [\![V_j]\!],$$

   for a type $V_j \leq^- T_j$. From Lemma 14 we infer $\mathsf{type}_2(V_j^*) \mathrel{\dot\subset} \mathsf{type}_2(T_j^*)$. Our choice of a, y and j yields $(a,y) \in [\![S_j]\!]$ and we conclude that $f(a,y) \mathrel{\dot\in} \mathsf{type}_2(T_j^*)$. That is $\mathsf{p}_1(f(a,y)) \mathrel{\dot\in} \mathsf{type}(\mathsf{p}_0(f(a,y)))$ as well as $\mathsf{p}_0(f(a,y)) \in \mathsf{OTS}$ and $\mathsf{sub}^-(\mathsf{p}_0(f(a,y)), T_j^*) = 1$. Therefore, we conclude that (3.21) holds.

3. $M \equiv M_1 M_2$: in this case there are types $\{S_i \to T_i\}_{i\in I}$ and S of $\lambda_{str}^{\{\}}$ and $j \in I$, such that in $\lambda_{str}^{\{\}}$ one can derive $\Gamma \vdash M_1 : \{S_i \to T_i\}_{i\in I}$ and $\Gamma \vdash M_2 : S$, where $S_j = \min_{i\in I}\{S_i \mid S \leq^- S_i\}$ and $T = T_j$. By the induction hypothesis we know

$$[\![\Gamma]\!] \to [\![M_1]\!] \in [\![\{S_i \to T_i\}_{i\in I}]\!], \tag{3.22}$$

$$[\![\Gamma]\!] \to [\![M_2]\!] \in [\![S]\!]. \tag{3.23}$$

   From (3.23) we infer $\mathsf{p}_1[\![M_2]\!] \mathrel{\dot\in} \mathsf{type}(\mathsf{p}_0[\![M_2]\!])$ as well as

$$\mathsf{sub}^-(\mathsf{p}_0[\![M_2]\!], S^*) = 1. \tag{3.24}$$

   Let k be such that $\mathsf{typap}(\{S_i \to T_i\}_{i\in I}^*, \mathsf{p}_0[\![M_2]\!]) = \langle S_k^*, T_k^*\rangle$. Using (3.22) we get

$$\mathsf{p}_1(\mathsf{p}_1[\![M_1]\!][\![M_2]\!]) \mathrel{\dot\in} \mathsf{type}(\mathsf{p}_0(\mathsf{p}_1[\![M_1]\!][\![M_2]\!]))$$

and $\mathsf{sub}^-(\mathsf{p_0}(\mathsf{p_1}[\![M_1]\!][\![M_2]\!]), T_k^*) = 1$. From (3.24) and the consistency conditions on good type formation we obtain $\mathsf{sub}^-(T_k^*, T_j^*)$. Therefore, we conclude by Lemma 14 that $\mathsf{p_1}[\![M_1]\!][\![M_2]\!] \in [\![T_j]\!]$ holds. 

In the sequel we will prove the soundness of our model construction with respect to reductions. In contrast to the semantics for $\lambda\&-$early presented in Castagna, Ghelli and Longo [23], our interpretation of a term does not change, when the term is reduced. We can show that if a term M reduces in $\lambda_{str}^{\{\}}$ to a term N, then the interpretations of M and N are equal.

Theorem 20. *If P, Q are well-typed $\lambda_{str}^{\{\}}$ terms and $P \triangleright_\Gamma Q$, then the following is provable in* $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathcal{L}_p\text{-}\mathsf{I_N})$*:*

$$[\![\Gamma]\!] \to [\![P]\!] = [\![Q]\!].$$

Proof. By induction on $\triangleright_\Gamma$. The critical case is:
$P :\equiv M \cdot N$, where we have $M \equiv \lambda x(M_i : S_i \Rightarrow T_i)_{i \in I}$, $\Gamma \vdash N : S$ as well as $S_j = \min_{i \in I}\{S_i \mid S \leq^- S_i\}$. Furthermore, $\{S_i \mid i \in I, S_i \leq^- S_j\} = \{S_j\}$ or N is in closed normal form. Assuming that $[\![\Gamma]\!]$ holds, then we obtain in both cases

$$\mathsf{typap}(\{S_i \to T_i\}_{i \in I}^*, \mathsf{p_0}[\![N]\!]) = \langle S_j^*, T_j^* \rangle.$$

Therefore, we conclude

$$[\![P]\!] = \mathsf{p_1}[\![M]\!][\![N]\!] = (\lambda \hat{x}.[\![M_j]\!])[\![N]\!] = [\![M_j]\!][[\![N]\!]/\hat{x}] = [\![M_j[N/x]]\!] = [\![Q]\!].$$

3.3 Loss of Information

Castagna [21] indicated a relationship between modeling late-binding and the problem of loss of information. This is a problem in type-theoretic research on object-oriented programming introduced by Cardelli [16]. It can be described as follows: assume that we are given the $\lambda_{str}^{\{\}}$ function $\lambda x(x : T \Rightarrow T)$ of type $\{T \to T\}$, i.e. the identity function on the type T. If we apply this function to a term N of type S, where S is a strict subtype of T, then we can only infer that $\lambda x(x : T \Rightarrow T)N$ has type T (rather than S). Thus, in the application, we have lost some information: we no longer know that N is of type S after having applied the identity function to it.

Usually, the solution to this problem is to use a second order calculus, which was originally proposed in Cardelli and Wegner [19]. The identity function

is no longer considered to take an argument of a type smaller than or equal to T and to return a result of type T. Instead, it is a function which takes any argument smaller than or equal to T and returns a result of the same type as that of the argument, i.e. it takes an argument of type $X \leq^{-} T$ and returns a result of type X. In a second order calculus we can write the type of this function as

$$(\forall X \leq^{-} T)(X \to X). \tag{3.25}$$

Recalling Castagna's proposal how a semantics for late-binding might work, we note the second order quantifier in the expression (3.1). This shows the connection between late-binding and the problem of loss of information. In a semantics for late-binding, we have to deal with functions which take types as arguments. The same is the case in order to solve the problem of loss of information.

This interplay of late-binding and solving the problem of loss of information also appears in our semantics. Let M be the $\lambda^{\{\}}_{str}$ term $\lambda x(x : T \Rightarrow T)$ of type $\{T \to T\}$ and N be a term of type $S \leq^{-} T$. Then we still can prove in $\mathsf{EETJ}{+}(\mathsf{Tot}){+}(\mathcal{L}_p\text{-}\mathsf{I_N})$ that $[\![MN]\!] \in [\![S]\!]$. Thus, there is no loss of information in our interpretation of $\lambda^{\{\}}_{str}$. After having applied the identity function M to N, we still can prove that the interpretation of the result is an element of the interpretation of the type S.

We have no loss of information in our semantics because the types of the terms are explicitly shown. Hence, in an application the types of the arguments can be employed to derive the type of the result. First the type information of the argument types is used to select the best matching branch. Then, in $\lambda^{\{\}}_{str}$, the type of the result is fixed by the type of this branch. The information of the argument types is lost. Whereas in our model, once the branch to be executed is chosen, the result type is computed from the types of the arguments by the computational aspect of the function. Accordingly, the type of the function will not be used in the computational process except for selecting the branch to be executed. Therefore, all the information is still available.

Of course, we do not have appropriate types in our model to express that a certain function has no loss of information. This is not possible since we only have elementary comprehension. Hence, quantification over types is not available to express something like (3.25). However, there are impredicative systems of explicit mathematics featuring stronger type existence principles, which allow to define types involving bound type variables or quantification over certain names; and it is easy to embed higher order λ calculi in them, cf. Feferman[33] or the next chapter.

Castagna [21] presents a higher order type system for late-bound overloading in order to model functions without loss of information. Our work also is a step towards a better understanding of such calculi combining parametric polymorphism and type dependent computations. Since there is a strong connection between loss of information and parametric polymorphism and since we have obtained a solution to the problem of loss of information for free in our model, we think that explicit mathematics is also an appropriate framework to further explore parametric polymorphism in the context of late-bound overloading.

More on the problem of loss of information can be found in the discussion of Chapter 4. There, we will show its connection to the problem of sound record update and hint at a solution for this second problem.

Chapter 4

Impredicative Overloading

> Der Verdacht gegen die "Logik", als deren folgerichtige Ausartung die Logistik gelten darf, entspringt dem Wissen von jenem Denken, das in der Erfahrung der Wahrheit des Seins, nicht aber in der Betrachtung der Gegenständlichkeit des Seienden, seine Quelle findet. Niemals ist das exakte Denken das strengste Denken, wenn anders die Strenge ihr Wesen aus der Art der Anstrengung empfängt, mit der jeweils das Wissen den Bezug zum Wesenhaften des Seienden innehält. Das exakte Denken bindet sich lediglich in das Rechnen mit dem Seienden und dient ausschliesslich diesem.
>
> Martin Heidegger

As we have seen before, overloading and late-binding in their full generality demand for a highly impredicative type structure which seems to be very different from known impredicative constructions. As Castagna [21] puts it 'We are in the presence of a new form of impredicativity.' This chapter is concerned with the mathematical meaning of these impredicativity phenomena.

4.1 Power Types in Explicit Mathematics

In this chapter we will not work with the base theory EETJ, but we will introduce a system of explicit mathematics based on elementary separation, join, product and weak power types.

This theory is formulated in the two-sorted language $\mathcal{L}_i$, which comprises *individual variables* a, b, c, f, g, x, y, z as well as *type variables* S, T, U, V, X, Y

(both possibly with subscripts). The language $\mathcal{L}_i$ includes the individual constants k, s (combinators), $\mathsf{p}, \mathsf{p}_0, \mathsf{p}_1$ (pairing and projections), 0 (zero), s_N (successor), p_N (predecessor), d_N (definition by numerical cases) and the individual constant sub (subset decidability). There are additional individual constants, called *generators*, which will be used for the uniform naming of types, namely nat (natural numbers), for every natural number e a generator sep_e (elementary separation), un (union), j (join), prod (product) and pow (power type).

Every individual variable and every individual constant is an individual term of $\mathcal{L}_i$ and the individual terms are closed under the binary function symbol $\cdot$ for (partial) application.

The formulas of $\mathcal{L}_i$ are built up similarly to the formulas of $\mathcal{L}_p$ and we will also use the same abbreviations. We introduce the theory OTN of overloading, types and names which will provide a framework for the study of impredicative overloading. Its logic is the classical *logic of partial terms* for individuals and classical logic for types. The non-logical axioms of OTN can be divided into the following five groups.

I. Applicative axioms. As usual, these are the axioms (1)–(9) of BON.

II. Explicit representation and extensionality. These are the axioms (10)–(12) of the theory EETJ.

III. Basic type existence axioms. These include axioms for the natural numbers, elementary separation, unions, joins and product types.

Natural number type

(16) $\Re(\mathsf{nat}) \wedge \forall x(x \mathrel{\dot\in} \mathsf{nat} \leftrightarrow N(x))$.

Elementary separation. Let F be an elementary formula of $\mathcal{L}_i$ with no individual variables other than $z1, \ldots, z(m+1)$ and no type variables other than $Z1, \ldots, Zn$ and let e be the Gödel number of F for any fixed Gödel numbering. Then we have the following axioms for all $\vec{a} = a_1, \ldots, a_m$, $\vec{b} = b_1, \ldots, b_n$ and $\vec{T} = T_1, \ldots, T_n$:

(17) $\Re(c, \vec{b}) \rightarrow \Re(\mathsf{sep}_e(\vec{a}, c, \vec{b}))$,

(18) $\Re(c, \vec{b}, S, \vec{T}) \rightarrow \forall x(x \mathrel{\dot\in} \mathsf{sep}_e(\vec{a}, c, \vec{b}) \leftrightarrow x \in S \wedge F[x, \vec{a}, \vec{T}])$.

Unions

(19) $\Re(a) \wedge \Re(b) \rightarrow \Re(\mathsf{un}(a, b)) \wedge \forall x(x \mathrel{\dot\in} \mathsf{un}(a, b) \leftrightarrow x \mathrel{\dot\in} a \vee x \mathrel{\dot\in} b)$.

We will employ $S \cup T$ and $S \cap T$ to denote the union of S and T, and the intersection of S and T, respectively. The same notation will also be used for names.

Join. This is the usual join axiom given by (15).

Product

(20) $\Re(a) \wedge (\forall x \mathrel{\dot\in} a)\Re(fx) \rightarrow \Re(\mathsf{prod}(a,f)) \wedge \Pi(a,f,\mathsf{prod}(a,f))$.

Here the formula $\Pi(a,f,b)$ says that b represents the product of all $f(x)$ such that $x \mathrel{\dot\in} a$, i.e.

$$\Pi(a,f,b) := \forall g(g \mathrel{\dot\in} b \leftrightarrow (\forall x \mathrel{\dot\in} a)gx \mathrel{\dot\in} fx).$$

The axioms (1)–(12) and (15)–(20) plus formula induction on the natural numbers correspond to Feferman's theory $\mathsf{S_0}$ (introduced in [31]) without inductive generation. We are going to extend this system by axioms for weak power types and for subset decidability operations on these power types.

IV. Power type axioms. These axioms state that weak power types exist and that the generator pow is a monotone operation on names.

(21) $\Re(a) \rightarrow \Re(\mathsf{pow}(a))$,

(22) $\Re(a) \rightarrow \forall x(x \in \mathsf{pow}(a) \rightarrow x \mathrel{\dot\subset} a)$,

(23) $\Re(a) \rightarrow \forall x \exists y(x \mathrel{\dot\subset} a \rightarrow y \doteq x \wedge y \mathrel{\dot\in} \mathsf{pow}(a))$,

(24) $a \mathrel{\dot\subset} b \rightarrow \mathsf{pow}(a) \mathrel{\dot\subset} \mathsf{pow}(b)$.

V. Subset decidability on power types. We can decide the subtype relation between elements of a power type.

(25) $\Re(c) \wedge a \mathrel{\dot\in} \mathsf{pow}(c) \wedge b \mathrel{\dot\in} \mathsf{pow}(c) \rightarrow (\mathsf{sub}ab = 0 \vee \mathsf{sub}ab = 1)$,

(26) $\Re(c) \wedge a \mathrel{\dot\in} \mathsf{pow}(c) \wedge b \mathrel{\dot\in} \mathsf{pow}(c) \rightarrow (\mathsf{sub}ab = 1 \leftrightarrow a \mathrel{\dot\subset} b)$,

(27) $\mathsf{sub}ab = 1 \rightarrow a \mathrel{\dot\in} \mathsf{pow}(b)$.

We will consider the extensions of OTN by complete induction on the natural numbers. We introduce the schema of formula induction (full induction).

Formula induction on the natural numbers

($\mathcal{L}_i$-$\mathsf{I_N}$) $F(0) \wedge (\forall x \in \mathsf{N})(F(x) \rightarrow F(\mathsf{s_N}x)) \rightarrow (\forall x \in \mathsf{N})F(x)$

for all formulas F of $\mathcal{L}_i$.

In the following we will discuss the axioms of OTN. Usually, theories of explicit mathematics are based on elementary comprehension rather than separation. Hence, there is a universal type available, which can be used, together with join, to prove the product axiom. This is not possible with elementary separation and therefore we have to add an axiom for product types (see Feferman [31]).

The weak power type axiom states that for every type A there exists a type B containing at least one name of every subtype of A and each element of B is a name for a subtype of A. By a simple diagonalization argument we can see that in the presence of join this axiom is inconsistent with the existence of a universal type. Therefore, we dispense with the universal type and include only elementary separation in our list of axioms.

Glass [48] provides a proof-theoretic analysis of the weak power type axiom and shows that adding it to many systems of explicit mathematics without join does not increase their proof-theoretic strength. In our system the presence of join makes the weak power type axiom much more effective. The question of the proof-theoretic strength of systems of explicit mathematics with join and a weak power type axiom was first asked by Feferman [31] and still remains open.

Of course one can imagine also a strong power type axiom saying that for every type A there is a type consisting of every name of every subtype of A. Jäger [59] proved that this principle is inconsistent with uniform elementary comprehension, but his argument does not work if we have only separation at our disposal. Recently Cantini [15] presented a model for non-uniform comprehension and a strong power type axiom using a variant of Quine's New Foundations.

We demand that our power type generator pow is a monotone operation on names. With a strong power type axiom monotonicity can be proved, but this is not the case if we have only weak power types. Moreover, the constructions of Glass [48] do not respect our monotonicity axiom. Hence, it is also an open question whether the monotonicity axiom for pow affects the proof-theoretic strength of the underlying system of types and names.

Subset decidability on power types will turn out to be crucial for the development of overloaded functions in OTN. The term sub may also be regarded as very special quantification operations, since for $a, b \in \mathsf{pow}(c)$ we have $\mathsf{sub}ab = 0 \leftrightarrow (\exists x \mathrel{\dot\in} a)x \mathrel{\dot\notin} b$. This point of view is supported by our model construction in the next section, where the term sub are dealt with already in the first-order part.

4.2 A Set-Theoretic Model for $\mathsf{OTN}+(\mathcal{L}_i\text{-}\mathsf{I_N})$

Starting with any model $\mathcal{M}$ of set theory we generate a model $\mathsf{Gen}(\mathcal{M})$ of the applicative axioms so that the natural numbers are interpreted by ω and every partial set-theoretic function F of $\mathcal{M}$ is represented in $\mathsf{Gen}(\mathcal{M})$. To obtain $\mathsf{Gen}(\mathcal{M})$ we simply use pairing in $\mathcal{M}$ to build codes for the constants $\mathsf{k}, \mathsf{s}, \mathsf{p}, \mathsf{p_0}, \mathsf{p_1}, \mathsf{s_N}, \mathsf{p_N}, \mathsf{d_N}$, and we define a code $\hat{F}$ for each partial set-theoretic function F of $\mathcal{M}$. We choose $\hat{F}$ so that it is a set-theoretic pair whose first component is the natural number 0. Further, we introduce a code $\underline{\mathsf{sub}}$ for the subset decidability operation and auxiliary codes $\underline{\mathsf{sub}}^F$ for every function F and for the empty set $\emptyset$. Now we regard the applicative axioms as closure conditions on the inductively generated relation $\mathsf{App}(x, y, z)$. This is done so that pairing is interpreted by set-theoretic pairing and the natural numbers are modeled by ω. Additionally, $\mathsf{App}(x, y, z)$ is closed under a condition expressing that the code $\hat{F}$ represents the function F as well as under two clauses for the subset decidability operations. We confine ourselves to showing only these closure conditions for the subset decidability operations.

- If F is a partial function of $\mathcal{M}$ so that its range is a subset of $\{1\}$, and if $x = \underline{\mathsf{sub}}$ and $y = \hat{F}$ as well as $z = \underline{\mathsf{sub}}^F$, then $\mathsf{App}(x, y, z)$ holds.

- If F and G are partial functions of $\mathcal{M}$ so that their respective range is a subset of $\{1\}$ and $x = \underline{\mathsf{sub}}^F$ and $y = \hat{G}$ as well as

$$\exists a \neg (F(a) = 1 \to G(a) = 1),$$

  then $\mathsf{App}(x, y, 0)$ holds.

- If F and G partial functions of $\mathcal{M}$ so that their respective range is a subset of $\{1\}$ and $x = \underline{\mathsf{sub}}^F$ and $y = \hat{G}$ as well as

$$\forall a (F(a) = 1 \to G(a) = 1),$$

  then $\mathsf{App}(x, y, 1)$ holds.

Hence, $\mathsf{App}(x, y, z)$ can be employed to interpret $xy \simeq z$. This is how the first order part of $\mathsf{OTN}+(\mathcal{L}_i\text{-}\mathsf{I_N})$ is modeled by $\mathsf{Gen}(\mathcal{M})$. This standard procedure is carried through in greater detail for example in Feferman [31] or in Jäger [58].

Feferman [31] presents a set-theoretic model for $\mathsf{S_0}$ plus a weak power type axiom over $\mathsf{Gen}(\mathcal{M})$. We will modify this construction so that it satisfies all

axioms of $\mathsf{OTN} + (\mathcal{L}_i\text{-}\mathsf{I_N})$. First we assign to each generator of $\mathcal{L}_i$ a code in $\mathsf{Gen}(\mathcal{M})$, for example:

$$\underline{\mathsf{nat}} := (1,0), \qquad \underline{\mathsf{sep}}_e(\vec{a}) := (2,e,(\vec{a})), \qquad \underline{\mathsf{un}}(a,b) := (3,a,b),$$
$$\underline{\mathsf{j}}(a,f) := (4,a,f), \qquad \underline{\mathsf{prod}}(a,f) := (5,a,f), \qquad \underline{\mathsf{pow}}(a) := (6,a).$$

These codes are chosen so that no conflicts arise with the representations $\hat{F}$ of set-theoretic functions F. We define the relations Cl_α and $\in_\alpha$ by transfinite induction on α. At stage α one has a structure $(\mathsf{Gen}(\mathcal{M}), \mathsf{Cl}_\alpha, \in_\alpha)$ in which the formulas of $\mathcal{L}_i$ are interpreted by taking $\in_\alpha$ for $\in$ and letting the names range over Cl_α. We will not give the complete definition of Cl_α and $\in_\alpha$, but we will show only the cases relevant for the power type axioms. All other cases are standard, see Feferman [31].

- For each partial function F in $\mathcal{M}$ so that the range of F is a subset of $\{1\}$ we have $\hat{F} \in \mathsf{Cl}_0$ and $x \in_0 \hat{F}$ if and only if $F(x) = 1$.
- For each $a \in \mathsf{Cl}_\alpha$ we have $\underline{\mathsf{pow}}(a) \in \mathsf{Cl}_{\alpha+1}$ and $x \in_{\alpha+1} \underline{\mathsf{pow}}(a)$ if and only if x is an $\hat{F}$ such that the domain of F is a subset of $\{y \mid y \in_\alpha a\}$.

The first condition ensures that the code of a partial functions whose range is a subset of $\{1\}$ is interpreted as a name. The code $\hat{F}$ represents the type of all x with $F(x) = 1$, that is $\hat{F}$ represents its domain. This condition guarantees that all elements of power types will indeed by names.

The second clause states that the power type of any type A is interpreted by the collection of all partial functions F from A to $\{1\}$ in $\mathcal{M}$. In order to make this work, we need the fact that the interpretation of a type of our system of explicit mathematics is a set in $\mathcal{M}$. This would not be the case if we allow elementary comprehension since this implies the existence of the universal type. Hence, we have to restrict our system to elementary separation.

As mentioned above, the first order part of $\mathsf{OTN} + (\mathcal{L}_i\text{-}\mathsf{I_N})$ is interpreted over $\mathsf{Gen}(\mathcal{M})$. In order to give the semantics of the type structure of $\mathsf{OTN}+(\mathcal{L}_i\text{-}\mathsf{I_N})$ we define the relations $\mathsf{Cl} := \bigcup_\alpha \mathsf{Cl}_\alpha$ and $\in := \bigcup_\alpha \in_\alpha$. For $a \in \mathsf{Cl}$ the set of all $x \in a$ is denoted by $\mathsf{ext}(a)$. Hence, the second order quantifiers of $\mathcal{L}_i$ will range over all $\mathsf{ext}(a)$ for $a \in \mathsf{Cl}$ and the naming relation $\Re$ will be interpreted by the collection of all pairs $(a, \mathsf{ext}(a))$ for $a \in \mathsf{Cl}$. We get the following soundness result.

Theorem 21. *For every formula F of $\mathcal{L}_i$ we have*

$$\mathsf{OTN} + (\mathcal{L}_i\text{-}\mathsf{I_N}) \vdash F \implies (\mathsf{Gen}(\mathcal{M}), \mathsf{Cl}, \in) \models F.$$

Proof. The only critical axioms are the power type axioms and the axioms for the subset decidability operations. All other axioms can be verified in a straightforward way, see Feferman's model for $\mathsf{S_0}$ [31]. Observe that $\mathsf{ext}(a)$ is a set in $\mathcal{M}$ for all $a \in \mathsf{Cl}$. Moreover, we have for all α, all $a \in \mathsf{Cl}_\alpha$ and all x

$$x \in \mathsf{ext}(a) \leftrightarrow x \in_\alpha a.$$

That is if a represents a type at stage α, then this type is completely defined at that stage. Later, no new elements will be included to it. Now we can check the power type axioms.

(21) If $a \in \mathsf{Cl}$, then we immediately obtain $\mathsf{pow}(a) \in \mathsf{Cl}$.

(22) If $x \in \mathsf{ext}(\underline{\mathsf{pow}}(a))$ and $y \in \mathsf{ext}(x)$, then $\mathsf{App}(x, y, 1)$ holds, and x represents a set-theoretic function whose domain is a subset of $\mathsf{ext}(a)$. Therefore, we conclude $y \in \mathsf{ext}(a)$.

(23) Let a, x be elements of Cl such that $\forall z(z \in \mathsf{ext}(x) \to z \in \mathsf{ext}(a))$. Since $\mathsf{ext}(x)$ is a set in $\mathcal{M}$, there exists a function $F := \{(z, 1) \mid z \in \mathsf{ext}(x)\}$ in $\mathcal{M}$. Then $\hat{F} \in \mathsf{ext}(\underline{\mathsf{pow}}(a))$. Moreover, $\hat{F} \in \mathsf{Cl}$ holds and $z \in \mathsf{ext}(\hat{F})$ if and only if $\mathsf{App}(\hat{F}, z, 1)$. By the construction of F this is the case if and only if $z \in \mathsf{ext}(x)$.

(24) Let a, b be in Cl such that $\forall x(x \in \mathsf{ext}(a) \to x \in \mathsf{ext}(b))$ as well as $y \in \mathsf{ext}(\underline{\mathsf{pow}}(a))$. Hence, y represents a set-theoretic function whose domain is a subset of $\mathsf{ext}(a)$. But then its domain is also a subset of $\mathsf{ext}(b)$ and we conclude $y \in \mathsf{ext}(\underline{\mathsf{pow}}(b))$.

Now we turn to the axioms for the subset decidability operations. Assume $c \in \mathsf{Cl}$ and $a \in \mathsf{ext}(\underline{\mathsf{pow}}(c))$ and $b \in \mathsf{ext}(\underline{\mathsf{pow}}(c))$. Hence, a and b are representations of partial set-theoretic functions from $\mathsf{ext}(c)$ to $\{1\}$. By the definition of $\in_0$ and App we obtain $x \in \mathsf{ext}(a) \leftrightarrow \mathsf{App}(a, x, 1)$ and $x \in \mathsf{ext}(b) \leftrightarrow \mathsf{App}(b, x, 1)$ for all x. This yields that $\mathsf{ext}(a) \subset \mathsf{ext}(b)$ holds if and only if we have $\forall x(\mathsf{App}(a, x, 1) \to \mathsf{App}(b, x, 1))$. Therefore, by the definition of App and our interpretation of sub the first two axioms for subset decidability are satisfied. In order to verify the last axiom about subset decidability assume $b \in \mathsf{ext}(\underline{\mathsf{pow}}(c))$ and $\mathsf{App}(\underline{\mathsf{sub}}^F, b, 1)$. In order to define App, one takes the least fixed point of the inductively defined application. Therefore, $\mathsf{App}(\underline{\mathsf{sub}}^F, b, 1)$ implies that b is the code of a partial set-theoretic function G whose range is a subset of $\{1\}$ and F is also such a function. We get $b \in \mathsf{Cl}$ with $\forall x(x \in \mathsf{ext}(b) \leftrightarrow G(x) = 1)$. Moreover, we have $\forall x(F(x) = 1 \to G(x))$. Hence, the domain of F is a subset of $\mathsf{ext}(b)$ and therefore, we get $\hat{F} \in \underline{\mathsf{pow}}(b)$. 

4.3 Impredicative Overloading in OTN

In this section we are going to develop a theory of impredicative overloading and late-binding in OTN. Together with the model construction of the previous section this yields a denotational semantics for these concepts of object-oriented programming.

To implement overloading and late-binding it is necessary that terms carry their run-time type information. This can be achieved if terms are ordered pairs, where the first component shows the type information and the second component is the computational aspect of the term. This way the run-time type of a term is explicitly displayed and can be used to evaluate expressions in the context of late-bound overloading, see Castagna [21] or the previous chapter.

If t is a name in OTN, then we define t^* to be the name $\mathsf{j}(\mathsf{pow}(t), \lambda x.x)$, and if T is a type with name t, then T^* is the type represented by t^*, i.e. T^* is the disjoint union of all subtypes of T. We have the following property of the type named t^* for any name t. If $x \mathrel{\dot\in} t^*$, then $\mathsf{p_0}x \mathrel{\dot\subset} t$ and $\mathsf{p_1}x \mathrel{\dot\in} \mathsf{p_0}x$. Therefore, $\mathsf{p_0}x$ can be viewed as the run-time type of x and $\mathsf{p_1}x$ as the computational aspect of the term x.

Now we will define the overloaded function types. Let $S_1, T_1, \ldots, S_n, T_n$ be types of OTN. Then we write $\{S_1 \to T_1, \ldots, S_n \to T_n\}$ for the overloaded function type

$$\{f \mid (\forall x \in S_1^*) fx \in T_1^* \wedge \ldots \wedge (\forall x \in S_n^*) fx \in T_n^*\}.$$

This type exists by the product axiom and elementary separation. It can be named uniformly in the names of $S_1, T_1, \ldots, S_n, T_n$. As for $\lambda^{\{\}}$, we will often employ a notation with index sets and write $\{S_i \to T_i\}_{i \in I}$ for the type $\{S_1 \to T_1, \ldots, S_n \to T_n\}$ where I is the set $\{1, \ldots, n\}$.

Castagna, Ghelli and Longo [24] remark that overloaded types are strongly related to intersection types: an intersection type $T \cap U$ is a type whose elements can play both the role of an element of T and of an element of U. This also holds for overloaded types. In the case of intersection types a coherence condition is additionally imposed, which basically means that a value can freely choose any of these roles without affecting the final result of a computation. This is not the case with overloaded functions: there is no such condition and it is essential that the type of an argument can affect the result of a computation. The overloaded type $\{S_1 \to T_1, \ldots, S_n \to T_n\}$ is simply defined as the intersection

$$\{S_1^* \to T_1^*\} \cap \ldots \cap \{S_n^* \to T_n^*\}$$

without any condition being imposed. If we consider this type not as an overloaded function type, but as an intersection type, then the additional coherence condition states the following: let f be an element of this intersection type and let x be either an element of S_i or S_j $(1 \leq i, j \leq n)$, then the result of fx must not depend on the type of x. If we drop this condition, then the computation may depend on the argument type, which is the essential feature of overloaded function types. Hence, our model of overloading in explicit mathematics reflects the relationship between intersection types and overloaded function types.

We obtain the following result about subtyping which corresponds to the subtyping rule of $\lambda^{\{\}}$.

Theorem 22. *Let $\{U_j \to V_j\}_{j\in J}$ and $\{S_i \to T_i\}_{i\in I}$ be overloaded function types. If for all $i \in I$ there exists $j \in J$ so that both $S_i \subset U_j$ and $V_j \subset T_i$ hold, then in* OTN *the following can be proven:*

$$\{U_j \to V_j\}_{j\in J} \subset \{S_i \to T_i\}_{i\in I}.$$

Proof. By the monotonicity of the power type generator pow we know that the operation * is monotone, i.e.

$$S \subset T \;\to\; S^* \subset T^*. \tag{4.1}$$

Assume now that $f \in \{U_j \to V_j\}_{j\in J}$ and let $x \in S_i^*$ for an $i \in I$. Then there exists $j \in J$ so that $S_i \subset U_j$ and $V_j \subset T_i$ hold. Hence, by (4.1) we get $x \in U_j^*$ and therefore $fx \in V_j^*$. Again by (4.1) we conclude $fx \in T_i^*$ and this finishes the proof of our theorem.

Next we investigate overloaded function terms. In order to construct them, we need a term that serves at selecting the best matching branch. Assume we are given types $S_1, \ldots, S_n$. Then there are names $s_1, \ldots, s_n$ of $S_1, \ldots, S_n$, respectively, so that $s_1, \ldots, s_n \in \mathsf{pow}(s_1 \cup \cdots \cup s_n)$. Let t be the name $\mathsf{pow}(s_1 \cup \cdots \cup s_n)$. Then sub satisfies $\mathsf{sub}ss_i = 0 \vee \mathsf{sub}ss_i = 1$ as well as $\mathsf{sub}ss_i = 1 \leftrightarrow s \mathrel{\dot\subset} s_i$ for all $s \mathrel{\dot\in} t$ and every s_i. Using sub we build for each $j \leq n$ a term $\mathsf{Min}^j_{s_1,\ldots,s_n}$ of $\mathcal{L}_i$ so that for all names $s \mathrel{\dot\in} t$ we have $\mathsf{Min}^j_{s_1,\ldots,s_n}(s) = 0 \vee \mathsf{Min}^j_{s_1,\ldots,s_n}(s) = 1$ and $\mathsf{Min}^j_{s_1,\ldots,s_n}(s) = 1$ if and only if

$$\mathsf{sub}ss_j = 1 \bigwedge_{\substack{1\leq l\leq n\\ l\neq j}} (\mathsf{sub}ss_l = 1 \to \mathsf{sub}s_l s_j = 0).$$

If the name s_j is a minimal element of the set $\{s_i \mid s \mathrel{\dot\subset} s_i \text{ for } 1 \leq i \leq n\}$, then $\mathsf{Min}^j_{s_1,\ldots,s_n}(s) = 1$ holds, otherwise $\mathsf{Min}^j_{s_1,\ldots,s_n}(s) = 0$.

Assume now that we are given two function terms f_1 and f_2 as well as names s_1, t_1, s_2, t_2 so that $f_1 \dot{\in} (s_1^* \to t_1^*)$ and $f_2 \dot{\in} (s_2^* \to t_2^*)$. Using definition by cases on natural numbers we can combine f_1 and f_2 to an overloaded function f so that

$$fx \simeq \begin{cases} f_1 x & \text{if } \mathsf{Min}^1_{s_1,s_2}(\mathsf{p}_0 x) = 1, \\ f_2 x & \text{if } \mathsf{Min}^2_{s_1,s_2}(\mathsf{p}_0 x) = 1 \wedge \mathsf{Min}^1_{s_1,s_2}(\mathsf{p}_0 x) \neq 1. \end{cases}$$

Of course it is also possible to combine more than two functions. Let us introduce the following notation for overloaded function terms. Assume we have functions $f_1 \in (S_1^* \to T_1^*), \ldots, f_n \in (S_n^* \to T_n^*)$, then the term $\mathsf{over}_{S_1,\ldots,S_n}(f_1 \ldots, f_n)$ denotes the overloaded function built up from the branches $f_1, \ldots, f_n$ as above. Overloaded functions $\mathsf{over}_{S_1,\ldots,S_n}(f_1 \ldots, f_n)$ behave according to the reduction rule of $\lambda^{\{\}}$, where the run-time type of the argument selects the best matching branch.

In $\lambda^{\{\}}$ consistency conditions concerning good type formation have been introduced, so that the static typing of a term can assure that the computation will be type-error free although it is based on run-time types. Now we introduce corresponding conditions for our system OTN of types and names. An overloaded function type $\{S_i \to T_i\}_{i \in I}$ is called *well-formed* if and only if it satisfies the following consistency conditions for all $i, j \in I$:

(1) $S_i \subset S_j \to T_i \subset T_j$,

(2) if the intersection of S_i and S_j is nonempty, then there exists a unique $k \in I$ so that $S_k = S_i \cap S_j$.

Theorem 23. *Let $\{S_1 \to T_1, \ldots, S_n \to T_n\}$ be a well-formed overloaded function type, i.e. it satisfies the consistency conditions (1) and (2), and let $f_1, \ldots, f_n$ be $\mathcal{L}_i$ terms so that $f_1 \in (S_1^* \to T_1^*), \ldots, f_n \in (S_n^* \to T_n^*)$. Then in OTN the following can be proven:*

$$\mathsf{over}_{S_1,\ldots,S_n}(f_1, \ldots, f_n) \in \{S_1 \to T_1, \ldots, S_n \to T_n\}.$$

Proof. First, let $s_1, \ldots, s_n$ be names of $S_1, \ldots, S_n$, respectively, so that $s_1, \ldots, s_n \in \mathsf{pow}(s_1 \cup \cdots \cup s_n)$. Assume we are given an $x \in S_i^*$ for $1 \leq i \leq n$. Then $x = (\mathsf{p}_0 x, \mathsf{p}_1 x)$, $\mathsf{p}_1 x \dot{\in} \mathsf{p}_0 x$ and $\mathsf{p}_0 x$ is an element of the power type of S_i. By Condition (2) there is a unique $j \in \{1, \ldots, n\}$ so that

$$\mathsf{Min}^j_{s_1,\ldots,s_n}(\mathsf{p}_0 x) = 1. \tag{4.2}$$

The uniqueness of j can be seen as follows: assume that $\mathsf{Min}^j_{s_1,\ldots,s_n}(\mathsf{p}_0 x) = 1$ and $\mathsf{Min}^k_{s_1,\ldots,s_n}(\mathsf{p}_0 x) = 1$ hold. Then we have $\mathsf{p}_0 x \dot{\subset} s_j$ and $\mathsf{p}_0 x \dot{\subset} s_k$. Since

$\mathsf{p}_1 x \mathrel{\dot\in} \mathsf{p}_0 x$ holds we know that $s_j \cap s_k$ is non empty. Therefore by Consistency Condition (2) there is a unique l so that $s_l \doteq s_j \cap s_k$. By the minimality of s_j and s_k we obtain $j = l = k$. Hence, there is a unique j so that (4.2) holds. Therefore we get $\mathsf{over}_{S_1,\dots,S_n}(f_1, \dots, f_n)x = f_j x$. By (4.2) and the axioms for subset decidability we find that $\mathsf{p}_0 x$ is an element of the power type of S_j and therefore $x \in S_j^*$. By our premise for f_j we conclude

$$\mathsf{over}_{S_1,\dots,S_n}(f_1, \dots, f_n)x \in T_j^*. \tag{4.3}$$

From $x \in S_i^*$ and $\mathsf{Min}^j_{s_1,\dots,s_n}(\mathsf{p}_0 x) = 1$ we conclude as above by Consistency Condition (2) that there exists a unique k so that $S_k = S_i \cap S_j$. The name $\mathsf{p}_0 x$ represents a subset of S_k. Suppose $S_j \not\subset S_i$. This implies $S_k \subsetneq S_j$ which contradicts $\mathsf{Min}^j_{s_1,\dots,s_n}(\mathsf{p}_0 x) = 1$. Therefore we have $S_j \subset S_i$ and applying Consistency Condition (1) yields $T_j \subset T_i$. By (4.3) and the monotonicity of * we finally get

$$\mathsf{over}_{S_1,\dots,S_n}(f_1, \dots, f_n)x \in T_i^*.$$

4.4 Discussion and Remarks

First we will show that both consistency conditions are necessary premises in Theorem 23. Then we are going to discuss an important difference between our model of late-bound overloading in explicit mathematics and models for second order λ calculi, which may be a good starting point for developing a natural denotational semantics for operations on records.

Let s_1, s_2, t_1, t_2 be names for the types $\{1\}, \{1,2\}, \{1\}$ and $\{2\}$, respectively, so that $s_1 \mathrel{\dot\in} \mathsf{pow}(s_2)$, $t_1 \mathrel{\dot\in} \mathsf{pow}(t_1)$ and $t_2 \mathrel{\dot\in} \mathsf{pow}(t_2)$. Therefore, we have $s_1 \mathrel{\dot\subset} s_2$ but not $t_1 \mathrel{\dot\subset} t_2$, i.e. Condition (1) is not satisfied. Further, let $f_1 := \lambda x.(t_1, 1) \mathrel{\dot\in} (s_1^* \to t_1^*)$ and $f_2 := \lambda x.(t_2, 2) \mathrel{\dot\in} (s_2^* \to t_2^*)$. However, we find $(s_1, 1) \mathrel{\dot\in} s_2^*$ but $\mathsf{over}_{s_1,s_2}(f_1, f_2)(s_1, 1)$ yields $(t_1, 1)$ which is not an element of t_2^*. Hence, $\mathsf{over}_{s_1,s_2}(f_1, f_2)$ does not belong to $\{s_1 \to t_1, s_2 \to t_2\}$.

Let s_1, s_2, s_3, t_2, t_3 be names for $\{1\}, \{1,2\}, \{1,3\}, \{2\}$ and $\{3\}$, respectively, so that $s_1 \mathrel{\dot\in} \mathsf{pow}(s_1)$, $t_2 \mathrel{\dot\in} \mathsf{pow}(t_2)$ and $t_3 \mathrel{\dot\in} \mathsf{pow}(t_3)$. By the monotonicity of the generator pow we get $s_1 \mathrel{\dot\in} \mathsf{pow}(s_2)$ and $s_1 \mathrel{\dot\in} \mathsf{pow}(s_3)$. Hence, the type $\{s_2 \to t_2, s_3 \to t_3\}$ does not satisfy Consistency Condition (2). We find that $f_2 := \lambda x.(t_2, 2) \mathrel{\dot\in} (s_2^* \to t_2^*)$ as well as $f_3 := \lambda x.(t_3, 3) \mathrel{\dot\in} (s_3^* \to t_3^*)$ hold. We get $(s_1, 1) \mathrel{\dot\in} s_3^*$, but $\mathsf{over}_{s_2,s_3}(f_2, f_3)(s_1, 1)$ yields $(t_2, 2)$ which is not in t_3^*. Hence, the type $\{s_2 \to t_2, s_3 \to t_3\}$ does not contain the function $\mathsf{over}_{s_2,s_3}(f_2, f_3)$.

In the previous chapter, we have seen that in order to solve the problem of loss of information, one usually employs second order λ calculi. However, in many natural models for such calculi we face the problem of "too many subtypes": the only closed term f of type $(\forall X \subset \mathsf{N})(X \to X)$ is the identity function on the natural numbers, cf. Bruce and Longo [14]. This is for the following reason: consider the type $\{n\}$ for each natural number n. Of course $\{n\} \subset \mathsf{N}$ holds (hence the name of the problem) and therefore $f : \{n\} \to \{n\}$ for each n. Since in these models the type of the argument affects only the type of the result, but not its value, we obtain that the term f must be the identity function.

This is not the case if we look at overloaded functions in explicit mathematics. For example, choose names $s_1, s_2, s_3, n \in \mathsf{pow}(\mathsf{nat})$ representing the types $\{1\}, \{2\}, \{1, 2\}$ and N, respectively. Now consider the term

$$t := \mathsf{over}_{s_1,s_2,s_3,n}(\lambda x.x, \lambda x.x, \lambda x.(s_3, 1), \lambda x.x).$$

This term is not the identity function since it maps $(s_3, 2)$ to $(s_3, 1)$. Nevertheless, the term t satisfies

$$(\forall X \subset \mathsf{N})t \in (X^* \to X^*). \tag{4.4}$$

If we restrict the universe of types to subtypes of the natural numbers, i.e. to elements of the power type of N, and if we let function types contain overloaded functions, then OTN provides a natural model for second order λ calculi. As shown before, the identity function is not the only function satisfying (4.4) in this model; but there are also many other functions of this type. We think that this gives further evidence that it is very promising to study the combination of overloading and parametric polymorphism from the point of view of explicit mathematics.

For example, this approach may help to solve the problem of polymorphic record update: typing rules for record update are only sound if the subtype relation is very restricted, cf. Cardelli [17] and Cardelli and Mitchell [18]. If one tries to build a denotational semantics of polymorphic update operations, one has to use models with properties like "all subtypes of a record type are record types". This kind of invariant is easy to realize in operational semantics, where there is no problem at all. However, standard domains used in denotational semantics of subtyping (such as PER models) do not have this property: there are always strange subsets that do not correspond to subtypes. One can force these properties, but the techniques have a very syntactic flavor; and up to now, there are no natural denotational semantics known for polymorphic record update. Since this problem is strongly related

to the problem of the elements of the type $\forall X(X \to X)$, an alternative to restricting the subtype relation may be to explore models for second order overloading in theories of types and names.

Chapter 5

Formalizing Non-termination of Recursive Programs

> I wish to God these calculations had been executed by steam.
>
> Charles Babbage

A recursively defined program is given by a recursion equation

$$f(\vec{x}) = t(f, \vec{x}),$$

where the program f can be called in the body of its definition. Every higher programming language offers a syntactical construction to define programs recursively. In general, there are several different solutions to such a recursive definition, i.e. there are several functions satisfying the recursion equation. In every introduction to the semantics of programming languages one finds that the intended semantics is given by the *least fixed point* of the recursion equation (with respect to the definedness order), see for instance Manna [70], Schmidt [85] or Jones [64]. Hence, we need a least fixed point operator in order to represent recursive programs. In this chapter we will present an applicative theory which allows us to define such an operator.

5.1 Introduction

Applicative theories are based on a type free combinatory logic. Hence, we have the recursion theorem at our disposal, which provides a term rec (or Y) to solve recursive equations. However, it is not provable in BON that a solution obtained by rec is minimal. In order to tackle this problem, we will make use of the fact that applicative theories are formulated within a

partial logic. This means that we have an additional predicate expressing the definedness of a term; and quantifiers and variables are ranging over defined objects only. Still, the term language is not restricted, i.e. there may be undefined terms. Using the definedness predicate $\downarrow$, we can introduce a definedness ordering on the terms, which allows us to talk about *monotonic* functionals. Only these functionals will have a least fixed point. Moreover, we will introduce the concept of *classes*, which are similar to types in a typed setting, in order to prove that our least fixed point belongs to a certain function space.

Since in general there are total term models for applicative theories, we often cannot prove that there exist undefined terms or equivalently that the corresponding programs loop forever. For this reason, we strengthen the basic theory by so-called *computability* axioms and we restrict the universe to natural numbers. These additional axioms represent the *recursion-theoretic* view of computations. They are motivated by Kleene's T predicate, which is a ternary primitive recursive relation on the natural numbers so that $\{a\}(\vec{m}) \simeq n$ holds if and only if there exists a computation sequence u with $\mathsf{T}(a, \langle \vec{m} \rangle, u)$ and $(u)_0 = n$. The usage of these computability axioms for the definition of a least fixed point operator can be seen as a marriage of convenience of the recursion-theoretic semantics and the least fixed point semantics for computer programs, see Jones [64].

Using the computability axioms we will define the least fixed point combinator as a combinator iterating the functional that is associated with a given recursive equation starting from the function which never terminates. To get the desired properties, we have to ensure that the functional operator is monotone with respect to the definedness order. For this reason, we will need the notion of monotonicity mentioned above.

The given theory still has a standard recursion-theoretic model; and with respect to the proof-theoretic strength we will not exceed Peano arithmetic. There exists a standard theory to formalize least fixed points, namely the theory ID_1 of non-iterated positive arithmetical inductive definitions, cf. Moschovakis [73] for an introduction to inductive definability or Kahle and Studer [68] for a corresponding theory in the context of explicit mathematics. However, our work essentially concerns Σ^0_1 monotone inductive definitions whereas ID_1 deals with arbitrary arithmetically definable positive operators. Hence ID_1 belongs to a rather different "world" and with respect to its proof-theoretic strength it is much stronger than Peano arithmetic, see Pohlers [79].

5.2 Applicative Theories

In this section we extend the basic theory of operations and numbers BON with axioms about computability and the statement that everything is a natural number. These two additional principles make the definition of a least fixed point operator possible.

The applicative theory of this chapter is formulated in the language $\mathcal{L}_c$ which contains the individual variables $a, b, c, f, g, h, m, n, x, y, z, \ldots$. The language $\mathcal{L}_c$ comprises the constants k, s (combinators), $\mathsf{p}, \mathsf{p}_0, \mathsf{p}_1$ (pairing and projections), 0 (zero), s_N (successor), p_N (predecessor) and d_N (definition by numerical cases). Further, we have the constant c (computation).

The *terms* $(r, s, t, \ldots)$ of $\mathcal{L}_c$ are built up from the variables and constants by means of the function symbol $\cdot$ for (partial) application. The *atomic formulas* of $\mathcal{L}_c$ are $\mathsf{N}(s)$, $s{\downarrow}$ and $s = t$. The formulas of $\mathcal{L}_c$ are generated from the atomic formulas by closing against the usual propositional connectives and quantifiers. We use the same abbreviations as for $\mathcal{L}_p$.

In the sequel we will employ the theory BON formulated in the language $\mathcal{L}_c$ extended with the schema of $\mathcal{L}_c$ formula induction on the natural numbers. In this theory all the primitive recursive functions and relations are available. Particularly, we will use addition $+$ and multiplication $*$ of natural numbers (both also in infix notation) as well as the usual "less than" $<$ and "less than or equal" $\leq$ relations. Further, we can define a least number operator μ so that the following holds, see Beeson [8].

Lemma 24. $\mathsf{BON} + (\mathcal{L}_c\text{-}\mathsf{I}_\mathsf{N})$ *proves:*

1. $f \in (\mathsf{N} \to \mathsf{N}) \to (\mu f \in \mathsf{N} \leftrightarrow (\exists n \in \mathsf{N}) fn = 0)$,
2. $f \in (\mathsf{N} \to \mathsf{N}) \wedge \mu f \in \mathsf{N} \to f(\mu f) = 0$.

This least number operator μ can be defined from the recursion theorem. Note that it is not the same as the non-constructive quantification operator, also called μ, which is studied by Feferman and Jäger in [38, 39].

Now we introduce non-strict definition by cases (cf. Beeson [8] or Kahle [66]). Observe that if $\mathsf{d}_\mathsf{N} rsuv{\downarrow}$, then $r{\downarrow}$ and $s{\downarrow}$ hold by strictness. However, we often want to define a function by cases so that it is defined if one case holds, even if the value that would have been computed in the other case is undefined. Hence, we let $\mathsf{d}_\mathsf{s} rsuv$ stand for the term $\mathsf{d}_\mathsf{N}(\lambda z.r)(\lambda z.s)uv0$ where the variable z does not occur in the terms r and s. From now on, non-strict definition by cases is denoted by the following notation:

$$\mathsf{d}_\mathsf{s} rsuv \simeq \begin{cases} r & \text{if } u = v, \\ s & \text{otherwise.} \end{cases}$$

Note that it already anticipates the axiom $\forall x \mathsf{N}(x)$, otherwise we should add $\mathsf{N}(u) \wedge \mathsf{N}(v)$ as a premise; and of course, strictness still holds with respect to u and v. We have $\mathsf{d_s} rsuv\downarrow \to u\downarrow \wedge v\downarrow$. If u or v is undefined, then $\mathsf{d_s} rsuv$ is also undefined. However, if r is a defined term and u and v are defined natural numbers that are equal, then $\mathsf{d_s} rsuv = r$ holds even if s is not defined.

We are interested in the extension of BON with axioms about computability (Comp) and the assertion that everything is a number.

Computability. These axioms are intended to capture the idea that convergent computations should converge in finitely many steps. In the formal statement of the axioms the expression $\mathsf{c}(f, x, n) = 0$ can be read as "the computation fx converges in n steps." The idea of these axioms is due to Friedman (unpublished) and discussed in Beeson [8]. Note that these axioms are satisfied in the usual recursion-theoretic model. The constant c can be interpreted by the characteristic function of Kleene's T predicate.

($\mathsf{Comp.1}$) $\forall f \forall x (\forall n \in \mathsf{N})(\mathsf{c}(f, x, n) = 0 \vee \mathsf{c}(f, x, n) = 1)$,

($\mathsf{Comp.2}$) $\forall f \forall x (fx\downarrow \leftrightarrow (\exists n \in \mathsf{N})\mathsf{c}(f, x, n) = 0)$.

In addition we will restrict the universe to natural numbers. Of course, this axiom is absolutely in the spirit of a recursion-theoretic interpretation.

Everything is a number. Formally, this is given by the statement $\forall x \mathsf{N}(x)$.

Now we define the applicative theory for least fixed points LFP as the union of all these axioms:

$$\mathsf{LFP} := \mathsf{BON} + (\mathsf{Comp}) + \forall x \mathsf{N}(x) + (\mathcal{L}_c\text{-}\mathsf{I_N}).$$

Before we can go on and define the least fixed point operator we have to introduce some auxiliary terms. With some coding provided by pairing and projection, we can easily define a term c^3 which behaves for ternary functions like c does for unary functions, i.e.

1. $\forall f \forall x \forall y \forall z (\forall n \in \mathsf{N})(\mathsf{c}^3(f, x, y, z, n) = 0 \vee \mathsf{c}^3(f, x, y, z, n) = 1)$,
2. $\forall f \forall x \forall y \forall z (fxyz\downarrow \leftrightarrow (\exists n \in \mathsf{N})\mathsf{c}^3(f, x, y, z, n) = 0)$.

The following lemma shows that there exists a function b which is never defined. Later, we will define an order relation on our functions and there b will play the role of the bottom element. Hence, we will be able the define least fixed points of monotonic functionals by recursion starting from b.

Lemma 25. *There exists a closed $\mathcal{L}_c$ term b so that LFP proves $\forall x(\neg \mathsf{b}x\downarrow)$.*

Proof. We can define $\mathsf{not_N} := \mathsf{rec}\,(\lambda f, x.\mathsf{d_N}\,1\,0\,(f\,x)\,0)\,0$. So it follows that $\neg\mathsf{N}(\mathsf{not_N})$ holds. Since we included $\forall x \mathsf{N}(x)$ to our list of axioms, we get $\neg(\mathsf{not_N}\downarrow)$. Thus, we can set $\mathsf{b} := \lambda x.\mathsf{not_N}$. 

5.3 Least Fixed Point Operator

In this section we will show how to define a least fixed point operator I in the theory LFP. As usual, in order to find the least fixed point of a monotonic functional g, the operator I will iterate g starting from the bottom element b. The stages of this inductive process are given by the term h, which will be defined first.

Definition 26. We define the term h so that

$$\mathsf{h}gn \simeq \begin{cases} \mathsf{b} & \text{if } n = 0, \\ g(\mathsf{h}g(\mathsf{p}_{\mathsf{N}}n)) & \text{else.} \end{cases}$$

Let t be such that

$$tgxn \simeq \begin{cases} 0 & \text{if } \mathsf{h}g(\mathsf{p}_0 n)x = \mathsf{p}_1 n, \\ \mathsf{not}_{\mathsf{N}} & \text{else.} \end{cases}$$

Then the term I is defined by

$$\mathsf{I} \;:=\; \lambda g\lambda x.\mathsf{p}_1(\mathsf{p}_0(\mu(\lambda y.\mathsf{c}^3(t, g, x, \mathsf{p}_0(y), \mathsf{p}_1(y))))).$$

The idea of this definition can be explained roughly as follows. We would like to have that $\mathsf{I}gx = z$ implies that there exists a finite computation of z by iterating the operator g starting from b. Formally, this is expressed by $\exists n(\mathsf{h}gnx = z)$, cf. the third claim of the following lemma. The definition of I is somewhat clumsy because of the several codings. Let g and x be given, then the μ operator is looking for an n so that

$$\mathsf{c}^3(t, g, x, \mathsf{p}_0(n), \mathsf{p}_1(n)) = 0. \tag{5.1}$$

If there is no such natural number n, then $\mu(\lambda y.\mathsf{c}^3(t, g, x, \mathsf{p}_0(y), \mathsf{p}_1(y)))$ will be undefined. Assume we have found an n so that (5.1) holds. This means that $tgx(\mathsf{p}_0(n))$ is terminating in $\mathsf{p}_1(n)$ steps. By the definition of t, we obtain that $tgx(\mathsf{p}_0(n))\!\downarrow$ implies $\mathsf{h}g(\mathsf{p}_0(\mathsf{p}_0(n)))x = \mathsf{p}_1(\mathsf{p}_0(n))$. Finally, the outer projections are used to extract this value $\mathsf{p}_1(\mathsf{p}_0(n))$.

The behavior of I can be also gathered from (the proof of) the following lemma. Moreover, it also shows why we had to include the axiom $\forall x\mathsf{N}(x)$ to LFP. Without this axiom the sophisticated interplay between the least number operator μ, the computability term c^3, and the coding machinery provided by $\mathsf{p}_0, \mathsf{p}_1$ and p would hardly work. This proof also makes use of the fact that the projection functions are total, see axiom (3) of BON.

Lemma 27. LFP *proves:*

1. $\mathsf{l}g\downarrow$,
2. $\mathsf{l}gx\downarrow \leftrightarrow \exists n(\mathsf{h}gnx\downarrow)$,
3. $\mathsf{l}gx = z \rightarrow \exists n(\mathsf{h}gnx = z)$.

Proof. The first claim is a consequence of the theorem about λ abstraction. For the second claim we have:

$$\begin{aligned}
\mathsf{l}gx\downarrow &\leftrightarrow \mathsf{p}_1(\mathsf{p}_0(\mu(\lambda y.\mathsf{c}^3(t,g,x,\mathsf{p}_0(y),\mathsf{p}_1(y)))))\downarrow \\
&\leftrightarrow \mu(\lambda y.\mathsf{c}^3(t,g,x,\mathsf{p}_0(y),\mathsf{p}_1(y)))\downarrow \\
&\leftrightarrow \exists n(\mathsf{c}^3(t,g,x,\mathsf{p}_0(n),\mathsf{p}_1(n)) = 0) \\
&\leftrightarrow \exists n(tgxn\downarrow) \\
&\leftrightarrow \exists n(\mathsf{d_s}0\mathsf{not_N}(\mathsf{h}g(\mathsf{p}_0 n)x)(\mathsf{p}_1 n)\downarrow) \\
&\leftrightarrow \exists n(\mathsf{h}gnx\downarrow)
\end{aligned}$$

The third claim follows by:

$$\begin{aligned}
\mathsf{l}gx = z &\rightarrow \mathsf{p}_1(\mathsf{p}_0(\mu(\lambda y.\mathsf{c}^3(t,g,x,\mathsf{p}_0(y),\mathsf{p}_1(y))))) = z \\
&\rightarrow \exists n(\mathsf{p}_1(\mathsf{p}_0 n) = z \wedge \mu(\lambda y.\mathsf{c}^3(t,g,x,\mathsf{p}_0(y),\mathsf{p}_1(y))) = n) \\
&\rightarrow \exists n(\mathsf{p}_1(\mathsf{p}_0 n) = z \wedge \mathsf{c}^3(t,g,x,\mathsf{p}_0(n),\mathsf{p}_1(n)) = 0) \\
&\rightarrow \exists n(\mathsf{p}_1(\mathsf{p}_0 n) = z \wedge tgx(\mathsf{p}_0 n)\downarrow) \\
&\rightarrow \exists n(\mathsf{p}_1(\mathsf{p}_0 n) = z \wedge \mathsf{d_s}0\mathsf{not_N}(\mathsf{h}g(\mathsf{p}_0(\mathsf{p}_0 n))x)(\mathsf{p}_1(\mathsf{p}_0 n))\downarrow) \\
&\rightarrow \exists n(\mathsf{p}_1(\mathsf{p}_0 n) = z \wedge \mathsf{h}g(\mathsf{p}_0(\mathsf{p}_0 n))x = \mathsf{p}_1(\mathsf{p}_0 n)) \\
&\rightarrow \exists n(\mathsf{h}g(\mathsf{p}_0(\mathsf{p}_0 n))x = z) \\
&\rightarrow \exists n(\mathsf{h}gnx = z)
\end{aligned}$$

We define the closed term a which will later serve at showing that we can replace the term $g(\mathsf{l}g)$ by a "finite approximation" $g(\mathsf{h}gn)$ (cf. Lemma 34, Claim 7).

Definition 28. Let t be such that

$$tfx \simeq \begin{cases} \mathsf{c}(\lambda x.xx, f, x) & \text{if } x = 0, \\ tf(\mathsf{p_N}x) * \mathsf{c}(\lambda x.xx, f, x) & \text{otherwise.} \end{cases}$$

We define the term a using λ abstraction so that

$$\mathsf{a}gfx \simeq \begin{cases} \mathsf{not_N} & \text{if } tfx = 0, \\ gx & \text{otherwise.} \end{cases}$$

Lemma 29. LFP *proves:*

1. $\forall g \forall f(\neg f f\downarrow \to \forall n(\mathsf{a} g f n \simeq g n))$,
2. $\forall g \forall f(f f\downarrow \to \exists m \forall n(\mathsf{a} g f n\downarrow \to \mathsf{a} g f n = g n \wedge n < m))$.

Proof. From the axioms about computability we obtain by induction:

$$\forall f \forall n(t f n = 0 \vee t f n = 1)$$

and

$$\forall f \forall n \forall m(m \leq n \wedge t f m = 0 \to t f n = 0).$$

Now, we get the first claim by:

$$\begin{array}{rcl} \neg f f\downarrow & \to & \neg(\lambda x.xx) f\downarrow \\ & \to & \forall n(\mathsf{c}(\lambda x.xx, f, n) = 1) \\ & \to & \forall n(t f n = 1) \\ & \to & \forall n(\mathsf{d_s not_N}(g n)(t f n)0 \simeq g n) \\ & \to & \forall n(\mathsf{a} g f n \simeq g n) \end{array}$$

The second claim follows with:

$$\begin{array}{rcl} f f\downarrow & \to & (\lambda x.xx) f\downarrow \\ & \to & \exists m(\mathsf{c}(\lambda x.xx, f, m) = 0) \\ & \to & \exists m(t f m = 0) \\ & \to & \exists m \forall n(m \leq n \to t f n = 0) \\ & \to & \exists m \forall n(t f n \neq 0 \to n < m) \\ & \to & \exists m \forall n(\mathsf{a} g f n\downarrow \to \mathsf{a} g f n = g n \wedge n < m) \end{array}$$

Since there is not a least fixed point for every recursion equation, cf. Example 32 below, we can only expect a meaningful solution for functionals satisfying an additional property, namely *monotonicity*. To define this notion, we will first introduce the concept of classes.

An $\mathcal{L}_c$ formula A containing exactly x as free variable will be called a *class*. Let A and B be classes and let F be an arbitrary formula of $\mathcal{L}_c$. We will

employ the following abbreviations:

$$\begin{aligned} t \in A &:= t{\downarrow} \wedge A[t/x], \\ A \to B &:= \forall y(y \in A \to xy \in B), \\ A \curvearrowright B &:= \forall y(y \in A \wedge xy{\downarrow} \to xy \in B), \\ A \cap B &:= x \in A \wedge x \in B, \\ (\forall x \in A)F(x) &:= \forall x(x \in A \to F(x)). \end{aligned}$$

Note that $t \in A$ has a strictness property built in. We have $t \in A \to t{\downarrow}$. Next we are going to introduce the definedness ordering $\sqsubseteq_T$ with respect to a class T. The meaning of $r \sqsubseteq s$ is that if r has a value, then r equals s; and $f \sqsubseteq_{A \curvearrowright B} g$ says that for every $x \in A$ if the computation fx terminates, then gx also terminates and both computations yield the same result.

Definition 30. Let $A_1, \ldots, A_n, B_1, \ldots, B_n$ be classes. Further, let $\mathcal{T}$ be the class $(A_1 \curvearrowright B_1) \cap \cdots \cap (A_n \curvearrowright B_n)$. Then $\mathcal{T}$ is called an *arrow class*. We define:

$$\begin{aligned} r \sqsubseteq s &:= r{\downarrow} \to r = s, \\ f \sqsubseteq_{\mathcal{T}} g &:= \bigwedge_{1 \le i \le n} (\forall x \in A_i) fx \sqsubseteq gx), \\ f \cong_{\mathcal{T}} g &:= f \sqsubseteq_{\mathcal{T}} g \wedge g \sqsubseteq_{\mathcal{T}} f. \end{aligned}$$

The formula $r \sqsubseteq s \wedge s \sqsubseteq r$ is equivalent to the standard partial equality relation $r \simeq s$. Hence, our definedness ordering $\sqsubseteq$ is in accordance with the notion of partiality of our applicative theory. In Chapter 7 we are going to employ a least fixed point operator to get a denotational semantics for Featherweight Java. There, it will be important that arrow classes are defined as the intersection of several function spaces.

The relations $\sqsubseteq$ and $\sqsubseteq_{\mathcal{T}}$ are transitive.

Lemma 31. *Let $\mathcal{T}$ be an arrow class as given in Definition 30. Then we can prove in* LFP*:*

1. $r \sqsubseteq s \wedge s \sqsubseteq t \to r \sqsubseteq t$,
2. $f \sqsubseteq_{\mathcal{T}} g \wedge g \sqsubseteq_{\mathcal{T}} h \to f \sqsubseteq_{\mathcal{T}} h$.

Proof. We have $r{\downarrow} \to r = s$ as well as $s{\downarrow} \to s = t$. Obviously we get $r{\downarrow} \to r = t$ proving Claim 1. Now we show the second claim. Assume $x \in A_i$ for some i. We have $fx \sqsubseteq gx$ and $gx \sqsubseteq hx$. Therefore, we conclude $fx \sqsubseteq hx$ by the first claim. ∎

Using the rec term we will find a fixed point for every operation g. But as mentioned before we cannot prove that this is a least fixed point; and of course, there are terms g that do not have a least fixed point.

Example 32. Let f_1 and f_2 be closed terms so that

$$f_1 x \simeq \begin{cases} 1 & \text{if } x = 1, \\ \mathsf{not_N} & \text{else} \end{cases} \qquad \text{and} \qquad f_2 x \simeq \begin{cases} \mathsf{not_N} & \text{if } x = 1, \\ 1 & \text{else.} \end{cases}$$

Now we let g be the operation

$$gx \simeq \begin{cases} f_1 & \text{if } x = f_1, \\ f_2 & \text{else.} \end{cases}$$

Let V be the universal class $x = x$. Then we know $g \in ((\mathsf{V} \curvearrowright \mathsf{V}) \to (\mathsf{V} \curvearrowright \mathsf{V}))$, and if f is a fixed point of g then we have either $f = f_1$ or $\forall x(fx \simeq f_2 x)$. However, g does not have a least fixed point in the sense of $\sqsubseteq_{(\mathsf{V} \curvearrowright \mathsf{V})}$, for we find $\neg f_1 \sqsubseteq_{(\mathsf{V} \curvearrowright \mathsf{V})} f_2 \wedge \neg f_2 \sqsubseteq_{(\mathsf{V} \curvearrowright \mathsf{V})} f_1$. That is f_1 is not comparable with any other fixed point of g and therefore, we do not have a least fixed point.

Only for *monotonic* $g \in (\mathcal{T} \to \mathcal{T})$ we can show that $\mathsf{l}g$ is the least fixed point of g.

Definition 33. Let $\mathcal{T}$ be an arrow class as given in Definition 30. A function $f \in (\mathcal{T} \to \mathcal{T})$ is called $\mathcal{T}$ *monotonic, if*

$$(\forall g \in \mathcal{T})(\forall h \in \mathcal{T})(g \sqsubseteq_{\mathcal{T}} h \to fg \sqsubseteq_{\mathcal{T}} fh).$$

Claims 1–5 of the following lemma correspond to the corollary in the appendix of Feferman [36]. Furthermore, in order to show that l yields a fixed point we need the compactness property stated in the last claim of our lemma. Compare this with the proof of the Myhill-Shepherdson Theorem, for example in Amadio and Curien [4], Rogers [84] or Odifreddi [76].

Lemma 34. *Let $\mathcal{T}$ be the arrow class $(A_1 \curvearrowright B_1) \cap \cdots \cap (A_k \curvearrowright B_k)$. We can prove in LFP that if $g \in (\mathcal{T} \to \mathcal{T})$ is $\mathcal{T}$ monotonic, then the following claims hold for all $i \leq k$:*

1. $\forall n(\mathsf{h}gn \in \mathcal{T})$,
2. $\forall n(\mathsf{h}gn \sqsubseteq_{\mathcal{T}} \mathsf{h}g(n+1))$,
3. $\mathsf{l}g \in \mathcal{T}$,

4. $\forall n(\mathsf{h}gn \sqsubseteq_{\mathcal{T}} \mathsf{l}g)$,

5. $\mathsf{l}g \sqsubseteq_{\mathcal{T}} g(\mathsf{l}g)$,

6. $\forall m \exists n(\forall x \in A_i)(x \leq m \to \mathsf{l}gx \sqsubseteq \mathsf{h}gnx)$,

7. $(\forall x \in A_i)\exists n(g(\mathsf{l}g)x \sqsubseteq g(\mathsf{h}gn)x)$.

Proof. 1. Proof by induction on the natural numbers. For $n = 0$ we have $\mathsf{h}g0 = \mathsf{b}$. Since $\forall x(\neg \mathsf{b}x\downarrow)$ we obviously get $\mathsf{h}g0 \in \mathcal{T}$. Assume $\mathsf{h}gn \in \mathcal{T}$. Then we have $g(\mathsf{h}gn) \in \mathcal{T}$ and this yields $\mathsf{h}g(n+1) \in \mathcal{T}$.

2. We proceed by induction on the natural numbers. As above we get $\forall x(\neg \mathsf{h}g0x\downarrow)$. Hence we have $(\forall x \in A_i)(\mathsf{h}g0x \sqsubseteq \mathsf{h}g1x)$ for any i. For the induction step assume $\mathsf{h}gn \sqsubseteq_{\mathcal{T}} \mathsf{h}g(n+1)$. Since g is $\mathcal{T}$ monotonic and by the previous claim $(\forall n \in \mathsf{N})\mathsf{h}gn\downarrow$ holds, we get

$$g(\mathsf{h}gn) \sqsubseteq_{\mathcal{T}} g(\mathsf{h}g(n+1)).$$

This yields $\mathsf{h}g(n+1) \sqsubseteq_{\mathcal{T}} \mathsf{h}g(n+2)$.

3. By Lemma 27 we find $(\forall x \in A_i)(\mathsf{l}gx\downarrow \to \exists n(\mathsf{h}gnx = \mathsf{l}gx))$ for any i. Then, by Claim 1 we get $(\forall x \in A_i)(\mathsf{l}gx\downarrow \to \mathsf{l}gx \in B_i)$. Hence $\mathsf{l}g \in \mathcal{T}$.

4. We have to show $(\forall x \in A_i)(\mathsf{h}gnx \sqsubseteq \mathsf{l}gx)$ for all i. So assume $x \in A_i$ and $\mathsf{h}gnx\downarrow$. We conclude $\mathsf{l}gx\downarrow$ by Lemma 27. Hence there exists a natural number m with

$$\mathsf{h}gmx = \mathsf{l}gx. \tag{5.2}$$

From Claim 2 we get by induction

$$\forall n \forall m(\forall x \in A_i)(\mathsf{h}gnx\downarrow \wedge \mathsf{h}gmx\downarrow \to hgnx = hgmx).$$

By $x \in A_i$ and (5.2) we therefore finally obtain $\mathsf{h}gnx \sqsubseteq \mathsf{l}gx$.

5. We have to show $(\forall x \in A_i)\mathsf{l}gx \sqsubseteq g(\mathsf{l}g)x$ for all i. So let $x \in A_i$ and $\mathsf{l}gx\downarrow$. Then by Lemma 27 we get $\exists n(\mathsf{l}gx = \mathsf{h}gnx)$. By the definition of h we see $\forall x(\neg \mathsf{h}g0x\downarrow)$. Hence $\exists n(\mathsf{l}gx = \mathsf{h}g(n+1)x)$. This is

$$\exists n(\mathsf{l}gx = g(\mathsf{h}gn)x). \tag{5.3}$$

For this natural number n we have by Claim 4 that $\mathsf{h}gn \sqsubseteq_{\mathcal{T}} \mathsf{l}g$. Because g is $\mathcal{T}$ monotonic we obtain $g(\mathsf{h}gn) \sqsubseteq_{\mathcal{T}} g(\mathsf{l}g)$ and since $x \in A_i$ this implies $g(\mathsf{h}gn)x \sqsubseteq g(\mathsf{l}g)x$. Finally, we conclude by (5.3) that $\mathsf{l}gx \sqsubseteq g(\mathsf{l}g)x$.

6. Proof by induction on m. For $m = 0$ the claim follows from Lemma 27. For the induction step assume

$$\exists n_1(\forall x \in A_i)(x \leq m \to \mathsf{l}gx \sqsubseteq \mathsf{h}gn_1x).$$

Employing Lemma 27 we find

$$m+1 \in A_i \wedge \mathsf{l}g(m+1)\downarrow \to \exists n_2(\mathsf{l}g(m+1) = \mathsf{h}gn_2(m+1)).$$

Taken together this yields

$$\begin{array}{l}\exists n_1 \exists n_2(\forall x \in A_i) \\ \quad (x \leq m+1 \wedge \mathsf{l}gx\downarrow \to (\mathsf{l}gx = \mathsf{h}gn_1x \vee \mathsf{l}gx = \mathsf{h}gn_2x)).\end{array}$$

By Claim 2 and Lemma 31 we get

$$\begin{array}{l}\exists n_1 \exists n_2(\forall x \in A_i) \\ \quad (x \leq m+1 \wedge \mathsf{l}gx\downarrow \to \mathsf{l}gx = \mathsf{h}g(n_1+n_2)x).\end{array}$$

We finally conclude

$$\exists n(\forall x \in A_i)(x \leq m+1 \to \mathsf{l}gx \sqsubseteq \mathsf{h}gnx).$$

7. Proof by contrapositive: suppose there exists an $x \in A_i$ so that

$$\forall n \neg(g(\mathsf{l}g)x \sqsubseteq g(\mathsf{h}gn)x). \tag{5.4}$$

With this $x \in A_i$ we define a term k by

$$k := \lambda f.\mathsf{d_s}0\mathsf{not_N}(g(\mathsf{a}(\mathsf{l}g)f)x)(g(\mathsf{l}g)x).$$

For the so defined k we will show that either assumption $\neg kk\downarrow$ or $kk\downarrow$ leads to a contradiction. As consequence we conclude that there cannot exist an $x \in A_i$ satisfying (5.4) and hence this claim is proved.

Now suppose $\neg kk\downarrow$. As a direct consequence of Lemma 29 we obtain for any j

$$\forall f(\neg ff\downarrow \to (\forall y \in A_j)\mathsf{a}(\mathsf{l}g)fy \simeq \mathsf{l}gy).$$

Therefore, we get

$$(\forall y \in A_j)\mathsf{a}(\mathsf{l}g)ky \simeq \mathsf{l}gy$$

for any j. The term a is defined by λ abstraction. Hence by Theorem 5 and Claim 3 we find $\mathsf{a}(\mathsf{l}g)k \in \mathcal{T}$. Therefore we obtain by the $\mathcal{T}$

monotonicity of g and $x \in A_i$ that $g(\mathsf{a}(\mathsf{l}g)k)x \simeq g(\mathsf{l}g)x$. By (5.4) it is the case that $g(\mathsf{l}g)x\downarrow$. Hence $g(\mathsf{a}(\mathsf{l}g)k)x = g(\mathsf{l}g)x$. This implies

$$\mathsf{d_s0not_N}(g(\mathsf{a}(\mathsf{l}g)k)x)(g(\mathsf{l}g)x)\downarrow,$$

i.e. $(\lambda f.\mathsf{d_s0not_N}(g(\mathsf{a}(\mathsf{l}g)f)x)(g(\mathsf{l}g)x))k\downarrow$ and $kk\downarrow$. Contradiction.

Suppose $kk\downarrow$. Hence $\mathsf{d_s0not_N}(g(\mathsf{a}(\mathsf{l}g)k)x)(g(\mathsf{l}g)x)\downarrow$ and

$$g(\mathsf{a}(\mathsf{l}g)k)x = g(\mathsf{l}g)x. \tag{5.5}$$

By Lemma 29 $kk\downarrow$ implies for any j

$$\exists m(\forall y \in A_j)(\mathsf{a}(\mathsf{l}g)ky\downarrow \to \mathsf{a}(\mathsf{l}g)ky = \mathsf{l}gy \wedge y < m). \tag{5.6}$$

Using Claim 6 we get $\exists n(\forall y \in A_j)(\mathsf{a}(\mathsf{l}g)ky \sqsubseteq \mathsf{h}gny)$. for any j. Hence $\exists n(\mathsf{a}(\mathsf{l}g)k \sqsubseteq_{\mathcal{T}} \mathsf{h}gn)$. Claim 3 together with (5.6) yields $\mathsf{a}(\mathsf{l}g)k \in \mathcal{T}$. Since g is $\mathcal{T}$ monotonic we therefore have

$$\exists n(g(\mathsf{a}(\mathsf{l}g)k) \sqsubseteq_{\mathcal{T}} g(\mathsf{h}gn)).$$

Our assumption $x \in A_i$ yields

$$\exists n(g(\mathsf{a}(\mathsf{l}g)k)x \sqsubseteq g(\mathsf{h}gn)x). \tag{5.7}$$

From (5.4) we know $\forall n\neg(g(\mathsf{l}g)x \sqsubseteq g(\mathsf{h}gn)x)$. Using (5.7) we conclude

$$\neg(g(\mathsf{l}g)x = g(\mathsf{a}(\mathsf{l}g)k)x)$$

which contradicts (5.5). ∎

The following theorem states that l indeed yields a fixed point of a monotonic operation g.

Theorem 35. *We can prove in* LFP *that if $g \in (\mathcal{T} \to \mathcal{T})$ is $\mathcal{T}$ monotonic for $\mathcal{T}$ given as in Definition 30, then*

$$\mathsf{l}g \cong_{\mathcal{T}} g(\mathsf{l}g).$$

Proof. By the previous lemma $\mathsf{l}g \sqsubseteq_{\mathcal{T}} g(\mathsf{l}g)$ holds. In order to show the other direction let $x \in A_i$. By the last claim of the previous lemma we obtain

$$\exists n(g(\mathsf{l}g)x \sqsubseteq g(\mathsf{h}gn)x).$$

By the definition of h we get $\exists n(g(\mathsf{l}g)x \sqsubseteq \mathsf{h}g(n+1)x)$. Using Claim 4 of the previous lemma we find $\forall n(\mathsf{h}g(n+1)x \sqsubseteq \mathsf{l}gx)$. Hence, by Lemma 31 we have $g(\mathsf{l}g)x \sqsubseteq \mathsf{l}gx$. Finally, we conclude $g(\mathsf{l}g) \sqsubseteq_{\mathcal{T}} \mathsf{l}g$. ∎

The next theorem states that $\mathsf{l}g$ is the *least* fixed point of g.

Theorem 36. *We can prove in* LFP *that if* $g \in (\mathcal{T} \to \mathcal{T})$ *is* $\mathcal{T}$ *monotonic for* $\mathcal{T}$ *given as in Definition 30, then*

$$f \in \mathcal{T} \wedge gf \cong_{\mathcal{T}} f \to \mathsf{l}g \sqsubseteq_{\mathcal{T}} f.$$

Proof. Let f be such that $gf \cong_{\mathcal{T}} f$. First, we show by induction on N that

$$\forall n(\mathsf{h}gn \sqsubseteq_{\mathcal{T}} f). \tag{5.8}$$

We obviously have $\mathsf{h}g0 \sqsubseteq_{\mathcal{T}} f$. Suppose $\mathsf{h}gn \sqsubseteq_{\mathcal{T}} f$ for a natural number n. By the $\mathcal{T}$ monotonicity of g we get $\mathsf{h}g(n+1) = g(\mathsf{h}gn) \sqsubseteq_{\mathcal{T}} gf \cong_{\mathcal{T}} f$. Therefore, by Lemma 31 we obtain $\mathsf{h}g(n+1) \sqsubseteq_{\mathcal{T}} f$ and (5.8) is shown. By $g(\mathsf{h}gn) = \mathsf{h}g(n+1)$ this implies

$$\forall n(g(\mathsf{h}gn) \sqsubseteq_{\mathcal{T}} f). \tag{5.9}$$

It remains to show $(\forall x \in A_i)(\mathsf{l}gx \sqsubseteq fx)$ for each i. So let $x \in A_i$. By Claim 5 of Lemma 34 we get $\mathsf{l}gx \sqsubseteq g(\mathsf{l}g)x$. By Claim 7 of the same lemma we obtain $\exists n(g(\mathsf{l}g)x \sqsubseteq g(\mathsf{h}gn)x)$. Therefore with (5.9) and Lemma 31 we conclude $\mathsf{l}gx \sqsubseteq fx$.

5.4 Conclusion

For the conclusion let us look at the following recursively defined method written in a Java like language.

```
A m (B x) {
  return m(x);
}
```

Of course, any program calling `m` with some argument s is non-terminating. The semantics of the method `m` is usually given as the least fixed point of the functional $\lambda f \lambda x.fx$. If we model this fixed point by $\mathsf{rec}(\lambda f \lambda x.fx)$, then we cannot prove in BON that $\neg(\mathsf{rec}(\lambda f \lambda x.fx)s\downarrow)$ for any argument s. This is simply because one can build total term models for the theory BON, in which every term has a value.

On the other hand, defining the semantics of the method `m` employing our least fixed point operator l enables us to prove non-termination in LFP. Let V be the universal class $x = x$ and $\emptyset$ be the empty class $x \neq x$. Then the functional $\lambda f \lambda x.fx$ is an element of $(\mathsf{V} \rightharpoonup \emptyset) \to (\mathsf{V} \rightharpoonup \emptyset)$ and it is of course

$\mathsf{V} \curvearrowright \emptyset$ monotonic. Therefore, by Lemma 34 we have $\mathsf{I}(\lambda f \lambda x.fx) \in (\mathsf{V} \curvearrowright \emptyset)$ and this implies $\forall y(\neg \mathsf{I}(\lambda f \lambda x.fx)y\downarrow)$. Hence, we have proved in LFP that the method `m` loops forever.

Like BON, the theory LFP can also be interpreted in the usual recursion-theoretic way, cf. Beeson [8] or Kahle [65]. In fact, the computability axioms are inspired by Kleene's T predicate, which therefore can be used to verify the axioms. So we can reduce LFP to Peano arithmetic. This will be important for obtaining an expressively strong but proof-theoretically weak system for the study of Featherweight Java.

The investigation of a least fixed point operator in [65] was motivated by defining an applicative theory with the proof-theoretic strength of Peano arithmetic for studying the interactive proof system LAMBDA [42, 45]. This proof system was designed for proving properties of ML programs. Up to Release 3.2 it was based on a partial logic and it was generating minimality rules for recursive function definitions. The theory defined in [65] was capturing a large part of this proof system, in particular, the minimality rules were modeled using the least fixed point operator.

We will finish this chapter by addressing two related approaches. First, Feferman [36] develops a form of generalized recursion theory in which computational procedures on domains that are contained in the natural numbers reduce to ordinary computations. There he shows how to obtain a uniform index for the least fixed point operator in the intensional recursion-theoretic model of computation. The construction of our least fixed point combinator is inspired by this approach.

Secondly, Stärk [89] introduces a *typed* logic of partial terms which incorporates a least fixed point operator and a schema for computational induction. One may question why we do not axiomatize our fixed point operator in a similar way as a primitive operator instead of using the computability axioms. The reason is that formulating Theorem 36 as an axiom would require to introduce the notions of monotonicity and classes before. In our opinion, this would be a rather inelegant approach since then the axioms would already depend on complex abbreviated notions. A further difference between Stärk's theory and our LFP is that he gives a *domain-theoretic* interpretation whereas LFP has a *recursion-theoretic* model.

Chapter 6

Featherweight Java

> Because we do not understand the brain very well we are constantly tempted to use the latest technology as a model for trying to understand it. In my childhood we were always assured that the brain was a telephone switchboard. ('What else could it be?') I was amused to see that Sherrington, the great British neuroscientist, thought that the brain worked like a telegraph system. Freud often compared the brain to hydraulic and electro-magnetic systems. Leibniz compared it to a mill, and I am told some of the ancient Greeks thought the brain functions like a catapult. At present, obviously, the metaphor is the digital computer.
>
> John R. Searle

Featherweight Java, called FJ, is a minimal core calculus for Java proposed by Igarashi, Pierce and Wadler [54] for the formal study of an extension of Java with parameterized classes. Further, Igarashi and Pierce [53] employed Featherweight Java also to obtain a precise understanding of inner classes. FJ is a minimal core calculus in the sense that as many features of Java as possible are omitted while maintaining the essential flavor of the language and its type system. Nonetheless, this fragment is large enough to include many useful programs. In particular, most of the examples in Felleisen and Friedman's text [41] are written in the purely functional style of Featherweight Java. In the next section we will present the formulation of Featherweight Java given in [53].

6.1 The Definition of Featherweight Java

Syntax. The abstract syntax of FJ class declarations, constructor declarations, method declarations and expressions is given by:

$$
\begin{array}{rcl}
\texttt{CL} & ::= & \texttt{class C extends C } \{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \\
\texttt{K} & ::= & \texttt{C } (\overline{\texttt{C}}\ \overline{\texttt{f}})\ \{\texttt{super}(\overline{\texttt{f}});\ \texttt{this.}\overline{\texttt{f}}\ =\ \overline{\texttt{f}};\} \\
\texttt{M} & ::= & \texttt{C m}(\overline{\texttt{C}}\ \overline{\texttt{x}})\ \{\texttt{return e};\} \\
\texttt{e} & ::= & \texttt{x} \\
 & | & \texttt{e.f} \\
 & | & \texttt{e.m}(\overline{\texttt{e}}) \\
 & | & \texttt{new C}(\overline{\texttt{e}}) \\
 & | & \texttt{(C)e}
\end{array}
$$

The meta-variables `A,B,C,D,E` range over class names, `f` and `g` range over field names, `m` ranges over method names, `x` ranges over variable names and `d,e` range over expressions (all possibly with subscripts). `CL` ranges over class declarations, `K` ranges over constructor declarations and `M` ranges over method declarations. We assume that the set of variables includes the special variable `this`, but that `this` is never used as the name of an argument to a method.

We write $\overline{\texttt{f}}$ as shorthand for $\texttt{f}_1, \ldots, \texttt{f}_n$ (and similarly for $\overline{\texttt{C}}, \overline{\texttt{x}}, \overline{\texttt{e}}$, etc.) and we use $\overline{\texttt{M}}$ for $\texttt{M}_1 \ldots \texttt{M}_n$ (without commas). The empty sequence is written as $\bullet$ and $\sharp(\overline{\texttt{x}})$ denotes the length of the sequence $\overline{\texttt{x}}$. Operations on pairs of sequences are abbreviated in the obvious way, e.g. "$\overline{\texttt{C}}\ \overline{\texttt{f}}$" stands for "$\texttt{C}_1\ \texttt{f}_1, \ldots, \texttt{C}_n\ \texttt{f}_n$" and "$\overline{\texttt{C}}\ \overline{\texttt{f}};$" is a shorthand for "$\texttt{C}_1\ \texttt{f}_1; \ldots; \texttt{C}_n\ \texttt{f}_n;$" and similarly "$\texttt{this.}\overline{\texttt{f}} = \overline{\texttt{f}};$" abbreviates "$\texttt{this.f}_1 = \texttt{f}_1; \ldots; \texttt{this.f}_n = \texttt{f}_n;$". We assume that sequences of field declarations, parameter names and method declarations contain no duplicate names.

A class table CT is a mapping from class names `C` to class declarations `CL`. A program is a pair $(CT, \texttt{e})$ of a class table and an expression. In the following we always assume that we have a *fixed* class table CT which satisfies the following conditions:

1. $CT(\texttt{C}) = \texttt{class C}\ldots$ for every `C` in the domain of CT, i.e. the class name `C` is mapped to the declaration of the class `C`,

2. `Object` is not an element of the domain of CT,

3. every class `C` (except `Object`) appearing anywhere in CT belongs to the domain of CT,

4. there are no cycles in the subtype relation induced by CT, i.e. the $<:$ relation is antisymmetric.

Subtyping. The following rules define the subtyping relation $<:$ which is induced by the class table. Note that every class defined in the class table has a super class, declared with `extends`.

$$\mathtt{C} <: \mathtt{C}$$

$$\frac{\mathtt{C} <: \mathtt{D} \qquad \mathtt{D} <: \mathtt{E}}{\mathtt{C} <: \mathtt{E}}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ldots\}}{\mathtt{C} <: \mathtt{D}}$$

Computation. These rules define the reduction relation $\longrightarrow$ which models field accesses, method calls and casts. In order to look up fields and method declarations in the class table we use some auxiliary functions that will be defined later on. We write $\mathtt{e}_0[\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{e}/\mathtt{this}]$ for the result of simultaneously replacing $\mathtt{x}_1$ by $\mathtt{d}_1, \ldots, \mathtt{x}_n$ by $\mathtt{d}_n$ and `this` by `e` in the expression $\mathtt{e}_0$.

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\mathtt{new\ C}(\overline{\mathtt{e}}).\mathtt{f_i} \longrightarrow \mathtt{e_i}}$$

$$\frac{\mathit{mbody}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e}_0)}{\mathtt{new\ C}(\overline{\mathtt{e}}).\mathtt{m}(\overline{\mathtt{d}}) \longrightarrow \mathtt{e_o}[\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ C}(\overline{\mathtt{e}})/\mathtt{this}]}$$

$$\frac{\mathtt{C} <: \mathtt{D}}{(\mathtt{D})\mathtt{new\ C}(\overline{\mathtt{e}}) \longrightarrow \mathtt{new\ C}(\overline{\mathtt{e}})}$$

We say that an expression `e` is in *normal form* if there is no expression `d` so that $\mathtt{e} \longrightarrow \mathtt{d}$.

Now we present the typing rules for expressions, method declarations and class declarations. An environment Γ is a finite mapping from variables to class names, written $\overline{\mathtt{x}} : \overline{\mathtt{C}}$. Again, we employ some auxiliary functions which will be given later. There are three rules for type casts: in an *upcast* the subject is a subclass of the target, in a *downcast* the target is a subclass of the subject, and in a *stupid cast* the target is unrelated to the subject. Stupid casts are included only for technical reasons, see Igarashi, Pierce and Wadler

[54]. The Java compiler will reject expressions containing stupid casts as ill typed. This is expressed by the hypothesis *stupid warning* in the typing rule for stupid casts.

Expression typing.

$$\Gamma \vdash \mathtt{x} \in \Gamma(\mathtt{x})$$

$$\frac{\Gamma \vdash \mathtt{e}_0 \in \mathtt{C}_0 \qquad \mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\Gamma \vdash \mathtt{e}_0.\mathtt{f}_\mathtt{i} \in \mathtt{C}_\mathtt{i}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e}_\mathtt{o} \in \mathtt{C}_0 \\ \mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{D}} \to \mathtt{C} \\ \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{C}} \qquad \overline{\mathtt{C}} <: \overline{\mathtt{D}}\end{array}}{\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) \in \mathtt{C}}$$

$$\frac{\begin{array}{c}\mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}} \\ \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{C}} \qquad \overline{\mathtt{C}} <: \overline{\mathtt{D}}\end{array}}{\Gamma \vdash \mathtt{new\ C}(\overline{\mathtt{e}}) \in \mathtt{C}}$$

$$\frac{\Gamma \vdash \mathtt{e}_0 \in \mathtt{D} \qquad \mathtt{D} <: \mathtt{C}}{\Gamma \vdash (\mathtt{C})\mathtt{e}_0 \in \mathtt{C}}$$

$$\frac{\Gamma \vdash \mathtt{e}_0 \in \mathtt{D} \qquad \mathtt{C} <: \mathtt{D} \qquad \mathtt{C} \neq \mathtt{D}}{\Gamma \vdash (\mathtt{C})\mathtt{e}_0 \in \mathtt{C}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e}_0 \in \mathtt{D} \qquad \mathtt{C} \not<: \mathtt{D} \qquad \mathtt{D} \not<: \mathtt{C} \\ \text{stupid warning}\end{array}}{\Gamma \vdash (\mathtt{C})\mathtt{e}_0 \in \mathtt{C}}$$

Method typing.

$$\frac{\begin{array}{c}\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e}_0 \in \mathtt{E}_0 \qquad \mathtt{E}_0 <: \mathtt{C}_0 \\ \mathit{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ldots\} \\ \text{if } \mathit{mtype}(\mathtt{m},\mathtt{D}) = \overline{\mathtt{D}} \to \mathtt{D}_0, \text{ then } \overline{\mathtt{C}} = \overline{\mathtt{D}} \text{ and } \mathtt{C}_0 = \mathtt{D}_0\end{array}}{\mathtt{C}_0\ \mathtt{m}\ (\overline{\mathtt{C}}\ \overline{\mathtt{x}})\ \{\mathtt{return\ e}_0;\}\ \text{OK in C}}$$

Class typing.

$$\frac{\begin{array}{c}\mathtt{K = C}(\overline{\mathtt{D}}\ \overline{\mathtt{g}},\ \overline{\mathtt{C}}\ \overline{\mathtt{f}})\ \{\mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\} \\ \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}} \qquad \overline{\mathtt{M}}\ \text{OK in C}\end{array}}{\mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\ \text{OK}}$$

We define the auxiliary functions which are used in the rules for computation and expression typing.

Field lookup.

$$\mathit{fields}(\texttt{Object}) = \bullet$$

$$\frac{CT(\texttt{C}) = \texttt{class C extends D } \{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \qquad \mathit{fields}(\texttt{D}) = \overline{\texttt{D}}\ \overline{\texttt{g}}}{\mathit{fields}(\texttt{C}) = \overline{\texttt{D}}\ \overline{\texttt{g}},\ \overline{\texttt{C}}\ \overline{\texttt{f}}}$$

Method type lookup.

$$\frac{CT(\texttt{C}) = \texttt{class C extends D } \{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \quad \texttt{B m } (\overline{\texttt{B}}\ \overline{\texttt{x}})\ \{\texttt{return e;}\} \text{ belongs to } \overline{\texttt{M}}}{\mathit{mtype}(\texttt{m}, \texttt{C}) = \overline{\texttt{B}} \to \texttt{B}}$$

$$\frac{CT(\texttt{C}) = \texttt{class C extends D } \{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \quad \texttt{m} \text{ is not defined in } \overline{\texttt{M}}}{\mathit{mtype}(\texttt{m}, \texttt{C}) = \mathit{mtype}(\texttt{m}, \texttt{D})}$$

Method body lookup.

$$\frac{CT(\texttt{C}) = \texttt{class C extends D } \{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \quad \texttt{B m } (\overline{\texttt{B}}\ \overline{\texttt{x}})\ \{\texttt{return e;}\} \text{ belongs to } \overline{\texttt{M}}}{\mathit{mbody}(\texttt{m}, \texttt{C}) = (\overline{\texttt{x}}, \texttt{e})}$$

$$\frac{CT(\texttt{C}) = \texttt{class C extends D } \{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \qquad \texttt{m} \text{ is not defined in } \overline{\texttt{M}}}{\mathit{mbody}(\texttt{m}, \texttt{C}) = \mathit{mbody}(\texttt{m}, \texttt{D})}$$

We call a Featherweight Java expression e *well-typed* if $\Gamma \vdash \texttt{e} \in \texttt{C}$ can be derived for some environment Γ and some class C.

Igarashi, Pierce and Wadler [54] prove that if an FJ program is well-typed, then the only way it can get stuck is if it reaches a point where it cannot perform a downcast. This is stated in the following theorem about progress, which is proved in [54].

Theorem 37. *Suppose* e *is a well-typed expression.*

1. *If* e *is of the form* $\texttt{new C}_0(\overline{\texttt{e}}).\texttt{f}$ *or contains such a subexpression, then* $\mathit{fields}(\texttt{C}_0) = \overline{\texttt{D}}\ \overline{\texttt{f}}$ *and* $\texttt{f} \in \overline{\texttt{f}}$.

2. *If* e *is of the form* $\texttt{new C}_0(\overline{\texttt{e}}).\texttt{m}(\overline{\texttt{d}})$ *or contains such a subexpression, then* $\mathit{mbody}(\texttt{m}, \texttt{C}_0) = (\overline{\texttt{x}}, \texttt{e}_0)$ *and* $\sharp(\overline{\texttt{x}}) = \sharp(\overline{\texttt{d}})$.

6.2 Evaluation Strategy and Typing

Now we study some examples written in Featherweight Java which will motivate our semantics for FJ as presented in the next chapter. We will focus on Java's evaluation strategy and on typing issues. In the last example the interplay of free variables, static types and late-binding is investigated.

Java features a call-by-value evaluation strategy, cf. the Java language specification by Gosling, Joy and Steele [49]. This strategy corresponds to the strictness axioms of the logic of partial terms upon which explicit mathematics is built. These axioms imply that an application only has a value, i.e. is terminating if all its arguments have a value.

In Java we not only have non-terminating programs we also have run-time exceptions, for example when an illegal down cast should be performed. With respect to these features, Featherweight Java is much more coarse grained. There is no possibility to state that a program terminates and exceptions are completely ignored. For good reasons, as we should say, since FJ is intended as a minimal core calculus for modeling Java's type system. However, this lack of expressiveness has some important consequences, which will be studied in the sequel. Let us first look at the following example.

Example 38.

```
class A extends Object {
  A () { super(); }

  C m() {
    return this.m();
  }
}

class C extends Object {
    int x;
    A y;
    C (int a,A b) {
        super();
        this.x = a;
        this.y = b;
    }
}
```

Of course, `new A().m()` is a non-terminating loop. Although, if it is evaluated on an actual Java implementation, then we get after a short while

a `java.lang.StackOverflowError` because of too many recursive method calls. In Featherweight Java `new A().m()` has no normal form, which reflects the fact that it loops forever.

Now let `e` be the expression `new C(5,new A().m()).x`. Due to Java's call-by-value evaluation strategy, the computation of this expression will not terminate either since `new A().m()` is a subexpression of `e` and has therefore to be evaluated first.

Featherweight Java's operational semantics uses a non-deterministic small-step reduction relation which does not enforce a call-by-value evaluation strategy. Hence, we have two different possibilities for reduction paths starting from `e`. If we adopt a call-by-value strategy, then we have to evaluate `new A().m()` first and we obtain an infinite reduction path starting from `e`. Since FJ's reduction relation is non-deterministic, we also have the possibility to apply the computation rule for field access. If we decide to do so, then `e` reduces to 5, which is in normal form.

In theories of types and names we have the possibility to state that a computation terminates. The formula $t\!\downarrow$ expresses that t has a value, meaning the computation represented by t is terminating. Let $[\![\texttt{e}]\!]$ be the interpretation of `e`. In our mathematical model Java's call-by-value strategy is implemented by the strictness axioms and $\neg[\![\texttt{e}]\!]\!\downarrow$ will be provable. Since 5 surely has a value we obtain $[\![\texttt{e}]\!] \not\simeq 5$ although 5 is the normal form of `e`. That means that in our interpretation we cannot model the non-deterministic reduction relation of Featherweight Java.

Non-terminating programs are not the only problem in modeling computations of Java. A second problem is the lack of a notion of run-time exception in Featherweight Java. For example, if a term is in normal form, then we cannot tell whether this is the case because the computation has finished properly or because an illegal down-cast should be performed. It may even be the case that the final expression does not contain any down-casts at all, but earlier during the computation an exception should have been thrown. Let us illustrate this fact with the following example, where the class `C` is as in Example 38.

Example 39.

```
class main extends Object{
  public static void main (String arg[]) {
      System.out.println(new C(5,(A)(new Object())).x);
  }
}
```

If we run this `main` method, then Java throws the following exception:

```
java.lang.ClassCastException: java.lang.Object
        at main.main(main.java:4)
```

Whereas in Featherweight Java the expression

```
new C(5,(A)(new Object())).x
```

reduces to 5. This is due to the fact that the term (A)(new Object()), which causes the exception in Java, is treated as final value in Featherweight Java and therefore, it can be used as argument in further method calls.

In our model we will introduce a special value ex to denote the result of a computation which throws an exception. An illegal down cast produces $(\mathsf{ex}, 0)$ as result and we can check every time an expression is used as argument in a method or constructor call whether its value is $(\mathsf{ex}, 0)$ or not. If it is not, then the computation can continue; but if an argument value represents an exception, then the result of the computation is this exception value. Therefore, in our model we can distinguish whether an exception has occurred or not. For example, the above expression evaluates to $(\mathsf{ex}, 0)$. We will have

$$\begin{aligned} [\![\texttt{(A)(new Object())}]\!] &= \mathsf{cast}\,\texttt{A}^*\,[\![\texttt{new Object()}]\!] \\ &= \mathsf{cast}\,\texttt{A}^*\,(\texttt{Object}^*, 0) \\ &= (\mathsf{ex}, 0) \end{aligned}$$

since $\mathsf{sub}(\texttt{Object}^*, \texttt{A}^*) \neq 1$, i.e. Object is not a subclass of A. By the definition of the terms new and proj, modeling object creation and field access, respectively, the term $(\mathsf{ex}, 0)$ will be propagated through the remaining computation and $(\mathsf{ex}, 0)$ will also be the final result in our semantics.

From these considerations it follows that we cannot prove soundness of our model construction with respect to reductions as formalized in Featherweight Java. However, we are going to equip FJ with a restricted reduction relation $\longrightarrow'$ which enforces a call-by-value evaluation strategy as it is used in the Java language and which also respects illegal down casts. With respect to this new notion of reduction, we will be able to prove that our semantics adequately models FJ computations.

In Featherweight Java we cannot talk about termination of programs. As usual in type systems for programming languages the statement "expression e has type T" has to be read as "if the computation of e terminates, then its result is of type T". Let A be the class of Example 38. Then in FJ new A().m() $\in$ C is derivable although the expression new A().m() denotes a non-terminating loop. Hence, in our model we will have to interpret e $\in$ C as $[\![\texttt{e}]\!]\downarrow \rightarrow [\![\texttt{e}]\!] \in [\![\texttt{C}]\!]$.

As we have seen before, the computation of `e` may result in an exception. In this case we have $[\![\mathtt{e}]\!] = (\mathsf{ex}, 0)$, which is a defined value. Hence, by our interpretation of the typing relation, we have to include $(\mathsf{ex}, 0)$ to the interpretation of every type. Alternatively, interpreting $\mathtt{e} \in \mathtt{C}$ as

$$[\![\mathtt{e}]\!]\downarrow \wedge [\![\mathtt{e}]\!] \neq (\mathsf{ex}, 0) \rightarrow [\![\mathtt{e}]\!] \in [\![\mathtt{C}]\!]$$

would also be possible, but then the soundness proofs would be more complicated.

In the following we consider a class `B` which is the same as `A` except that the result type of the method `m` is changed to `D`.

```
class B extends Object {
  B () { super(); }

  D m() {
    return this.m();
  }
}
```

Since the method bodies for `m` are the same in both classes `A` and `B`, we can assume that the interpretations of `new A().m()` and `new B().m()` will be the same, i.e. $[\![\texttt{new A().m()}]\!] \simeq [\![\texttt{new B().m()}]\!]$. In this example the classes `C` and `D` may be chosen arbitrarily. In particular, they may be disjoint, i.e. perhaps there is no object belonging to both of them. Hence, if our modeling of the typing relation is sound, it follows that we are in the position to prove that the computation of `new B().m()` is non-terminating, i.e. $\neg[\![\texttt{new B().m()}]\!]\downarrow$.

Usually, in lambda calculi such recursive functions are modeled using a fixed point combinator. In continuous λ-models, such as $\mathsf{P}\omega$ or D_∞, these fixed point combinators are interpreted by *least* fixed point operators and hence, one can *semantically* show that certain functions do not terminate. In applicative theories on the other hand, recursive equations are solved with the rec term provided by the recursion theorem. Unfortunately, one cannot *prove* that this operator yields a least fixed point. Therefore, we have to employ the special term I to define the semantics of FJ expressions. Since this term provides a least solution to certain fixed point equations, it will be possible to show $\neg[\![\texttt{new B().m()}]\!]\downarrow$, which is needed in order to prove soundness of our interpretation with respect to typing.

Now we are going to examine the role of free variables in the context of static types and late-binding. In the following example let `C` be an arbitrary class with no fields.

Example 40.

```
class A extends Object{
  A () { super(); }

  C m() {
    return this.m();
  }
}

class B extends A{
  B () { super(); }

  C m() {
    return new C();
  }
}
```

As in Example 38 class `A` defines the method `m`, which does not terminate. Class `B` extends `A`, hence it is a subclass of `A` and it overrides the method `m`. Here `m` creates a new object of type `C` and returns it to the calling object.

Let `x` be a free variable with (static) type `A`. The rules for method typing guarantee that the return type of `m` cannot be changed by the overriding method; and by the typing rules of FJ we can derive $\texttt{x:A} \vdash \texttt{x.m()} \in \texttt{C}$. As we have seen before this means "if `x.m()` yields a result, then it belongs to `C`." Indeed, as a consequence of Java's late-binding evaluation strategy, knowing only the static type `A` of `x` we cannot tell whether in `x.m()` the method `m` defined in class `A` or the one of class `B` will be executed. Hence, we do not know whether this computation terminates or not. Only if we know the object which is referenced by `x`, we can look at its dynamic type and then decide by the rules of method body lookup which method actually gets called.

This behavior has the consequence that there are FJ expressions in normal form whose interpretation will not have a value. For example, `x.m()` is in normal form, but maybe `x` references an object of type `A` and in this case the interpretation of `x.m()` will not have a value. Therefore, only if we have a *closed* term in normal form , we can be sure that its interpretation is defined.

Chapter 7

A Semantics for Featherweight Java

> Und ausserhalb der Logik ist alles Zufall.
>
> Ludwig Wittgenstein

> There is nothing more practical than a good theory.
>
> Albert Einstein

In this chapter we present a recursion-theoretic denotational semantics for Featherweight Java. Our interpretation is based on a formalization of the object model of Castagna, Ghelli and Longo in a predicative theory of types and names. Although this theory is proof-theoretically weak, it allows us to prove many properties of programs written in Featherweight Java. This underpins Feferman's thesis that impredicative assumptions are not needed for computational practice. Moreover, the present work is also a contribution to the ongoing research on unifying functional and object-oriented programming. It shows that these two paradigms fit well together and that their combination has a sound mathematical model.

7.1 Fixed Point Types

We will add the principle of dependent choice to the base theory EETJ and show that this extension is sufficient to prove the existence of certain fixed points. The theory of types and names which we will consider in the sequel is formulated in the two sorted language $\mathcal{L}_j$ about individuals and types. It comprises *individual variables* $a, b, c, f, g, h, x, y, z, \ldots$ as well as *type variables* $A, B, C, X, Y, Z, \ldots$ (both possibly with subscripts).

The language $\mathcal{L}_j$ includes the *individual constants* k, s (combinators), $\mathsf{p}, \mathsf{p}_0, \mathsf{p}_1$ (pairing and projections), 0 (zero), s_N (successor), p_N (predecessor), d_N (definition by numerical cases) and the constant c (computation). There are additional individual constants, called *generators*, which will be used for the uniform representation of types. Namely, we have a constant c_e (elementary comprehension) for every natural number e, as well as the constants j (join) and dc (dependent choice).

The *individual terms* $(r, s, t, r_1, s_1, t_1, \dots)$ of $\mathcal{L}_j$ are built up from the variables and constants by means of the function symbol $\cdot$ for (partial) application. Again, the $\mathcal{L}_j$ formulas are built up like the formulas of $\mathcal{L}_p$ and we will make use of the same abbreviations. In the following we will consider extensions of the theory $\mathsf{EETJ} + (\mathsf{T\text{-}I}_\mathsf{N})$ formulated in the language $\mathcal{L}_j$.

The classes of Java will be modeled by types in explicit mathematics. Since the Java classes may be defined by mutual recursion, i.e. class `A` may contain an attribute of class `B` and vice versa, their interpretations have to be given as a fixed point type in our theory of types and names. From a recursion-theoretic perspective, we see that a fixed point of a Σ_1 positive operator form is needed in order to model these classes and such a fixed point can be obtained by iterating the operator form ω many times, cf. Hinman [52]. *Dependent choice* (dc) is a principle which allows us to perform this construction in explicit mathematics. These axioms have been proposed by Jäger and their proof-theoretic analysis has been carried out by Probst [80].

Dependent choice.

($\mathsf{dc}.1$) $\Re(a) \wedge f \in (\Re \to \Re) \to \mathsf{dc}(a, f) \in (\mathsf{N} \to \Re)$,

($\mathsf{dc}.2$) $\Re(a) \wedge f \in (\Re \to \Re) \to$
$\mathsf{dc}(a, f)0 \simeq a \wedge (\forall n \in \mathsf{N})\mathsf{dc}(a, f)(n+1) \simeq f(\mathsf{dc}(a, f)n)$.

First, let us introduce some notation. By primitive recursion we can define in $\mathsf{EETJ} + (\mathsf{T\text{-}I}_\mathsf{N})$ the usual relations $<$ and $\leq$ on the natural numbers. The i^{th} section of U is defined by $(U)_i := \{y \mid (i, y) \in U\}$. If s is a name for U, then $(s)_i$ represents the type $(U)_i$. By abuse of notation, we let the formula $(s)_i \in U$ stand for $\mathsf{p}_0 s = i \wedge \mathsf{p}_1 s \in U$. The context will ensure that it is clear how to read $(s)_i$. Product types are defined according to the definition of n-tupling by $S_1 \times S_2 := \{(x, y) \mid x \in S_1 \wedge y \in S_2\}$ and

$$S_1 \times S_2 \times \cdots \times S_{n+1} := S_1 \times (S_2 \times \cdots \times S_{n+1}).$$

We define projection functions π_i^k for $k \geq 2$ and $1 \leq i \leq k$ so that

$$\pi_i^k(s_1, \dots, s_k) \simeq s_i.$$

A *fixed point specification* is a system of formulas of the form

$$\begin{aligned}(X)_1 &= Y_{11} \times \cdots \times Y_{1m_1}\\ &\vdots\\ (X)_n &= Y_{n1} \times \cdots \times Y_{nm_n}\end{aligned}$$

where each Y_{ij} may be any type variable other than X or of the form

$$\{x \in X \mid \mathsf{p}_0 x = k_1\} \cup \cdots \cup \{x \in X \mid \mathsf{p}_0 x = k_l\} \cup \{c\}$$

for $k_i \leq n$ and an arbitrary $\mathcal{L}_j$ term c. Those Y_{ij} which are just a type variable other than X are called *parameters* of the specification.

Our aim is to show that for every fixed point specification there exists a fixed point satisfying it and this fixed point can be named uniformly in the parameters of its specification.

Assume we are given a fixed point specification as above with parameters $\vec{Y}$. Then we find by elementary comprehension that there exists a closed individual term t of $\mathcal{L}_j$ such that EETJ proves for all $\vec{a}$ whose length is equal to the number of parameters of the specification:

1. $\Re(\vec{a}) \wedge \Re(b) \rightarrow \Re(t(\vec{a}, b))$,
2. $\Re(\vec{a}, \vec{Y}) \wedge \Re(b, X) \rightarrow \forall x(x \mathrel{\dot\in} t(\vec{a}, b) \leftrightarrow (x)_1 \in Y_{11} \times \cdots \times Y_{1m_1} \vee \ldots \vee (x)_n \in Y_{n1} \times \cdots \times Y_{nm_n})$.

In the following we assume $\Re(\vec{a})$ and let a_{ij} denote that element of $\vec{a}$ which represents Y_{ij}. The term $\lambda x.t(\vec{a}, x)$ is an operator form mapping names to names. Note that it is monotonic, i.e.

$$b \mathrel{\dot\subset} c \rightarrow t(\vec{a}, b) \mathrel{\dot\subset} t(\vec{a}, c). \tag{7.1}$$

Starting from the empty type, represented by $\emptyset$, this operation can be iterated in order to define the stages of the inductive definition of our fixed point. To do so, we define a function f by:

$$f(\vec{a}, n) \simeq \mathsf{dc}(\emptyset, \lambda x.t(\vec{a}, x))n.$$

As a direct consequence of ($\mathsf{dc}.1$) we find $(\forall n \in \mathsf{N})\Re(f(\vec{a}, n))$. Hence, we let J be the type represented by $\mathsf{j}(\mathsf{nat}, \lambda x.f(\vec{a}, x))$. Making use of ($\mathsf{T\text{-}I_N}$) we can prove

$$(\forall n \in \mathsf{N})\forall x((n, x) \in J \rightarrow (n + 1, x) \in J)$$

and therefore

$$(\forall m \in \mathsf{N})(\forall n \in \mathsf{N})(m \leq n \rightarrow f(\vec{a}, m) \mathrel{\dot{\subset}} f(\vec{a}, n)). \tag{7.2}$$

We define the fixed point $\mathsf{FP} := \{x \mid (\exists n \in \mathsf{N})(n, x) \in J\}$. By the uniformity of elementary comprehension and join there exists a closed individual term fp so that $\mathsf{fp}(\vec{a})$ is a name for FP, i.e. the fixed point can be represented uniformly in its parameters. A trivial corollary of this definition is

$$(\exists n \in \mathsf{N})(x \mathrel{\dot{\in}} f(\vec{a}, n)) \leftrightarrow x \mathrel{\dot{\in}} \mathsf{fp}(\vec{a}). \tag{7.3}$$

The following theorem states that FP is indeed a fixed point of t. We employ $(s)_{ij} \in U$ as abbreviation for $\mathsf{p}_0 s = i \wedge \pi_j^{m_i}(\mathsf{p}_1 s) \in U$.

Theorem 41. *It is provable in* $\mathsf{EETJ} + (\mathsf{dc}) + (\mathsf{T\text{-}I_N})$ *that* FP *is a fixed point satisfying the fixed point specification, i.e.*

$$\Re(\vec{a}) \rightarrow \forall x(x \mathrel{\dot{\in}} \mathsf{fp}(\vec{a}) \leftrightarrow x \mathrel{\dot{\in}} t(\vec{a}, \mathsf{fp}(\vec{a}))).$$

Proof. Assume $x \mathrel{\dot{\in}} \mathsf{fp}(\vec{a})$. By (7.3) there exists a natural number n so that $x \mathrel{\dot{\in}} f(\vec{a}, n)$. By (7.2) we find $x \in f(\vec{a}, n+1)$ and by the definition of f we get $f(\vec{a}, n+1) = t(\vec{a}, f(\vec{a}, n))$. By (7.3) we obtain $f(\vec{a}, n) \mathrel{\dot{\subset}} \mathsf{fp}(\vec{a})$ and with (7.1) we conclude $x \in t(\vec{a}, \mathsf{fp}(\vec{a}))$.
Next, we show $\forall x(x \mathrel{\dot{\in}} t(\vec{a}, \mathsf{fp}(\vec{a})) \rightarrow x \mathrel{\dot{\in}} \mathsf{fp}(\vec{a}))$. Let $x \mathrel{\dot{\in}} t(\vec{a}, \mathsf{fp}(\vec{a}))$, i.e. we have for all i, j with $1 \leq i \leq n$ and $1 \leq j \leq m_i$ either $(x)_{ij} \mathrel{\dot{\in}} a_{ij}$ or by (7.3)

$$\begin{aligned}(x)_{ij} \in \quad & \{y \mid (\exists n \in \mathsf{N}) y \mathrel{\dot{\in}} f(\vec{a}, n) \wedge \mathsf{p}_0 y = k_1\} \cup \cdots \cup \\ & \{y \mid (\exists n \in \mathsf{N}) y \mathrel{\dot{\in}} f(\vec{a}, n) \wedge \mathsf{p}_0 y = k_l\} \cup \{c\}\end{aligned}$$

depending on the specification. Since f is monotonic there exists a natural number n so that for all i, j with $1 \leq i \leq n$ and $1 \leq j \leq m_i$ we have either $(x)_{ij} \mathrel{\dot{\in}} a_{ij}$ or

$$(x)_{ij} \in \{y \mid y \mathrel{\dot{\in}} f(\vec{a}, n) \wedge \mathsf{p}_0 y = k_1\} \cup \cdots \cup \{y \mid y \mathrel{\dot{\in}} f(\vec{a}, n) \wedge \mathsf{p}_0 y = k_l\} \cup \{c\}$$

depending on the specification and this implies $x \mathrel{\dot{\in}} f(\vec{a}, n+1)$. Hence we conclude by (7.3) that $x \mathrel{\dot{\in}} \mathsf{fp}(\vec{a})$ holds. 

As we have seen in the previous chapter, we will need a least fixed point operator in order to model Featherweight Java programs. Hence, we include the computability axioms and the statement that everything is a natural number to our list of axioms. The theory PTN about programming with types and names is defined as the union of all these axioms:

$$\mathsf{PTN} := \mathsf{EETJ} + (\mathsf{dc}) + (\mathsf{Comp}) + \forall x \mathsf{N}(x) + (\mathsf{T\text{-}I_N}).$$

We adapt the definition of a monotonic operation to the typed context of PTN. First, we are going to define order relations $\sqsubseteq_T$ for certain types T. As before, the meaning of $s \sqsubseteq_T t$ is that s is smaller than t with respect to the usual pointwise ordering of functions, e.g. we have

$$f \sqsubseteq_{A \frown B} g \to (\forall x \in A)(fx\downarrow \to fx = gx).$$

Definition 42. Let $A_1, \ldots, A_n, B_1, \ldots, B_n$ be types. Further, let $\mathcal{T}$ be the type $(A_1 \frown B_1) \cap \cdots \cap (A_n \frown B_n)$. We define:

$$\begin{array}{rcl} f \sqsubseteq g & := & f\downarrow \to f = g, \\ f \sqsubseteq_{\mathcal{T}} g & := & \bigwedge\limits_{1 \le i \le n} (\forall x \in A_i) fx \sqsubseteq gx, \\ f \cong_{\mathcal{T}} g & := & f \sqsubseteq_{\mathcal{T}} g \wedge g \sqsubseteq_{\mathcal{T}} f. \end{array}$$

Definition 43. Let $\mathcal{T}$ be as given in Definition 42. A function $f \in (\mathcal{T} \to \mathcal{T})$ is called $\mathcal{T}$ *monotonic, if*

$$(\forall g \in \mathcal{T})(\forall h \in \mathcal{T})(g \sqsubseteq_{\mathcal{T}} h \to fg \sqsubseteq_{\mathcal{T}} fh).$$

The following theorem summarizes the results of Chapter 5 in the context of the theory PTN.

Theorem 44. *There exists a closed individual term* I *of* $\mathcal{L}_j$ *such that we can prove in* PTN *that if* $g \in (\mathcal{T} \to \mathcal{T})$ *is* $\mathcal{T}$ *monotonic for* $\mathcal{T}$ *given as in Definition 42, then*

1. $\mathsf{I}g \in \mathcal{T}$,
2. $\mathsf{I}g \cong_{\mathcal{T}} g(\mathsf{I}g)$,
3. $f \in \mathcal{T} \wedge gf \cong_{\mathcal{T}} f \to \mathsf{I}g \sqsubseteq_{\mathcal{T}} f$.

Probst [80] presents a recursion-theoretic model for the system $\mathsf{EETJ}+(\mathsf{dc})+(\mathsf{T}\text{-}\mathsf{I}_\mathsf{N})$ and shows that the proof-theoretic ordinal of this theory is $\varphi\omega 0$, i.e. it is slightly stronger than Peano arithmetic but weaker than Martin-Löf type theory with one universe ML_1 or the system $\mathsf{EETJ} + (\mathcal{L}_p\text{-}\mathsf{I}_\mathsf{N})$ of explicit mathematics with elementary comprehension, join and full induction on the natural numbers.

It is no problem to combine this construction with the recursion-theoretic interpretation of the computability axioms. Therefore, we get a model for PTN such that computations in PTN are modeled by ordinary recursion-theoretic functions. Moreover, this construction shows that PTN is proof-theoretically

still equivalent to $\mathsf{EETJ} + (\mathsf{dc}) + (\mathsf{T\text{-}I_N})$. Hence, PTN is a predicative theory, which is proof-theoretically much weaker than all the systems that are usually used to talk about object-oriented programming, for most of these calculi are extensions of system F, which already contains full analysis, cf. e.g. Bruce, Cardelli and Pierce [13]. Nevertheless, PTN is sufficiently strong to model Featherweight Java and to prove many properties of the represented programs. In PTN we can also prove soundness of our interpretation with respect to subtyping, typing and reductions.

7.2 Interpreting Featherweight Java

Assume we are given a program written in Featherweight Java. This consists of a fixed class table CT and an expression `e`. In this section we will show how to translate such a program into the language of types and names. This allows us to state and prove properties of FJ programs in the theory PTN of explicit mathematics.

We generally assume that all classes and methods occurring in our fixed class table CT are well-typed. This means for every class `C` of CT we can derive `class C...OK` by the rules for class typing and for every method `m` defined in this class we can derive `...m...OK IN C` by the rules for method typing.

The *basic types* of Java such as `boolean`, `int`, ... are not included in FJ. However, EETJ provides a rich type structure which is well-suited to model these basic data types, see Feferman [33] and Jäger [57]. Hence, we will include them in our modeling of Featherweight Java.

Let $*$ be an injective mapping from all the names for classes, basic types, fields and methods occurring in the class table CT into the numerals of $\mathcal{L}_j$. This mapping will be employed to handle the run-time type information of FJ terms as well as to model field access and method selection.

First, we show how objects will be encoded as sequences in our theory of types and names. Let `C` be a class of our class table CT with $fields(\mathtt{C}) =$ $\mathtt{D_1}\ \mathtt{g_1}, \ldots, \mathtt{D_n}\ \mathtt{g_n}$ and let $m_{\mathtt{C}}$ be the least natural number such that for all field names $\mathtt{g_j}$ occurring in $fields(\mathtt{C})$ we have ${\mathtt{g_j}}^* < m_{\mathtt{C}}$. An object of type `C` will be interpreted by a sequence $(\mathtt{C}^*, (s_1, \ldots, s_{m_{\mathtt{C}}}))$, where s_i is the interpretation of the field $\mathtt{g_j}$ if $i = {\mathtt{g_j}}^*$ and $s_i = 0$ if there is no corresponding field. In particular, we always have $s_{m_{\mathtt{C}}} = 0$. Note that in this model the type of an object is encoded in the interpretation of the object.

In order to deal with the subtype hierarchy of FJ, we define a term sub such that for all $a, b \in \mathsf{N}$ we have:

1. If a or b codes a basic type, i.e. $a = \mathtt{A}^*$ for a basic type A, and $a = b$, then $\mathsf{sub}(a, b) = 1$.
2. If $\mathtt{C} <: \mathtt{D}$ can be derived for two classes C and D, and $\mathtt{C}^* = a$ as well as $\mathtt{D}^* = b$ hold, then $\mathsf{sub}(a, b) = 1$.
3. Otherwise we set $\mathsf{sub}(a, b) = 0$.

Since CT is finite, sub can be defined using definition by cases on the natural numbers; recursion is not needed.

In the following, we will define a semantics for expression of Featherweight Java. In a first step, this semantics will be given only relative to a term invk which is used to model method invocations. Then we can define invk in a second step as the least fixed point of a recursive equation involving all methods occurring in our fixed class table.

We have to find a way for dealing with invalid down-casts. What should be the value of (A) `new Object()` in our model, when A is a class different from `Object`? In FJ the computation simply gets stuck, no more reductions will be performed. In our model we choose a natural number ex which is not in the range of * and set the interpretation of illegal down casts to $(\mathsf{ex}, 0)$. This allows us to distinguish them from other expressions using definition by cases on the natural numbers. Hence, every time an expression gets evaluated we can check whether one of its arguments is the result of an illegal cast. If this is the case, then $(\mathsf{ex}, 0)$ will also be the value of the whole expression.

This is the reason why we will have to add run-time type information to elements of basic types. Let us look, for example, at the constant `17` of Java which is of type `int`. If it is simply modeled by the numeral 17 of $\mathcal{L}_j$, then it might happen that $17 = (\mathsf{ex}, 0)$ and we could not decide whether this $\mathcal{L}_j$ term indicates that an illegal down cast has occurred or whether it simply denotes the constant `17` of Java. On the other hand, if the Java constant `17` is modeled by $(\mathtt{int}^*, 17)$, i.e. with run-time type information, then it is provably different from $(\mathsf{ex}, 0)$.

Of course, if at some stage of a computation an invalid down cast occurs and we obtain $(\mathsf{ex}, 0)$ as intermediate result, then we have to propagate it to the end of the computation. Therefore, all of the following terms are defined by distinguishing two cases: if none of the arguments equals $(\mathsf{ex}, 0)$, then the application will be evaluated; if one of the arguments is $(\mathsf{ex}, 0)$, then the result is also $(\mathsf{ex}, 0)$.

In order to model field access, we define a term proj so that

$$\mathsf{proj}\, i\, x \simeq \begin{cases} x & \mathsf{p_0} x = \mathsf{ex}, \\ \mathsf{p_0}(\mathsf{tail}\, i\, (\mathsf{p_1} x)) & \text{otherwise}, \end{cases}$$

where tail is defined by primitive recursion such that

$$\mathsf{tail}\, 1\, s \simeq s \qquad \mathsf{tail}\, (n+1)\, s \simeq \mathsf{p_1}(\mathsf{tail}\, n\, s)$$

for all natural numbers $n \geq 1$. Hence for $i \leq n$ and $t \neq \mathsf{ex}$ we have

$$\mathsf{proj}\, i\, (t, (s_1, \ldots, s_n, s_{n+1})) \simeq s_i.$$

Next, we show how to define the interpretation of the keyword `new`. For every class $\mathtt{C}$ of our class table CT with $fields(\mathtt{C}) = \mathtt{D_1\ g_1}, \ldots, \mathtt{D_n\ g_n}$ we find a closed $\mathcal{L}_j$ term $t_{\mathtt{C}}$ such that:

1. If $a_i{\downarrow}$ holds for all a_i $(i \leq n)$ and if there is a natural number j such that $\mathsf{p_0} a_j = \mathsf{ex}$, then we get $t_{\mathtt{C}}(a_1, \ldots, a_n) = a_j$ where j is the least number satisfying $\mathsf{p_0} a_j = \mathsf{ex}$.

2. Else we find $t_{\mathtt{C}}(a_1, \ldots, a_n) \simeq (\mathtt{C}^*, (b_1, \ldots, b_{m_{\mathtt{C}}}))$, where $b_i \simeq a_j$ if there exists $j \leq n$ with $i = \mathtt{g_j}^*$ and $b_i = 0$ otherwise.

Using definition by cases on the natural numbers we can build a term new so that $\mathsf{new}\, \mathtt{C}^*\, (\vec{s}) \simeq t_{\mathtt{C}}(\vec{s})$ for every class $\mathtt{C}$ in CT.

Again, using definition by cases we build a term cast satisfying

$$\mathsf{cast}\, a\, b \simeq \begin{cases} (\mathsf{ex}, 0) & \mathsf{sub}(\mathsf{p_0} b, a) = 0, \\ b & \text{otherwise}. \end{cases}$$

Now we give the translation $[\![\mathtt{e}]\!]_h$ of a Featherweight Java expression $\mathtt{e}$ into an $\mathcal{L}_j$ term relative to a term h for method invocations. For a sequence $\overline{\mathtt{e}} = \mathtt{e_1}, \ldots, \mathtt{e_n}$ we write $[\![\overline{\mathtt{e}}]\!]_h$ for $[\![\mathtt{e_1}]\!]_h, \ldots, [\![\mathtt{e_n}]\!]_h$. We assume that for every variable $\mathtt{x}$ of Featherweight Java there exists a corresponding variable x of $\mathcal{L}_j$ such that two different variables of FJ are mapped to different variables of $\mathcal{L}_j$. In particular, we suppose that our language $\mathcal{L}_j$ of types and names includes a variable $this$ so that $[\![\mathtt{this}]\!]_h = this$.

$$\begin{aligned} [\![\mathtt{x}]\!]_h &:= x \\ [\![\mathtt{e.f}]\!]_h &:= \mathsf{proj}\, \mathtt{f}^*\, [\![\mathtt{e}]\!]_h \\ [\![\mathtt{e.m}(\overline{\mathtt{f}})]\!]_h &:= h(\mathtt{m}^*, [\![\mathtt{e}]\!]_h, [\![\overline{\mathtt{f}}]\!]_h) \\ [\![\mathtt{new\ C}(\overline{\mathtt{e}})]\!]_h &:= \mathsf{new}\, \mathtt{C}^*([\![\overline{\mathtt{e}}]\!]_h) \\ [\![(\mathtt{C})\mathtt{e}]\!]_h &:= \mathsf{cast}\, \mathtt{C}^*\, [\![\mathtt{e}]\!]_h \end{aligned}$$

In the following we are going to define the term invk which models method invocations. To this aim, we have to deal with overloading and late-binding in explicit mathematics as in the previous chapters. Assume we are given n natural numbers $s_1, \ldots, s_n$. Using sub we build for each $j \leq n$ a term $\mathsf{min}^j_{s_1,\ldots,s_n}$ such that for all $s \in \mathsf{N}$ we have $\mathsf{min}^j_{s_1,\ldots,s_n}(s) = 0 \vee \mathsf{min}^j_{s_1,\ldots,s_n}(s) = 1$ and $\mathsf{min}^j_{s_1,\ldots,s_n}(s) = 1$ if and only if

$$\mathsf{sub}(s, s_j) = 1 \wedge \bigwedge_{\substack{1 \leq l \leq n \\ l \neq j}} (\mathsf{sub}(s, s_l) = 1 \rightarrow \mathsf{sub}(s_l, s_j) = 0).$$

Hence, $\mathsf{min}^j_{s_1,\ldots,s_n}(s) = 1$ holds if s_j is a minimal element (with respect to sub) of the set $\{s_i \mid \mathsf{sub}(s, s_i) = 1 \wedge 1 \leq i \leq n\}$; and otherwise we have $\mathsf{min}^j_{s_1,\ldots,s_n}(s) = 0$.

We can define a term $\mathsf{over}_{s_1,\ldots,s_n}$ which combines several functions $f_1, \ldots, f_n$ to one overloaded function $\mathsf{over}_{s_1,\ldots,s_n}(f_1, \ldots, f_n)$ such that

$$\mathsf{over}_{s_1,\ldots,s_n}(f_1, \ldots, f_n)\,(x, \vec{y}) \simeq \begin{cases} f_1(x, \vec{y}) & \mathsf{min}^1_{s_1,\ldots,s_n}(\mathsf{p}_0 x) = 1, \\ \vdots & \\ f_n(x, \vec{y}) & \mathsf{min}^n_{s_1,\ldots,s_n}(\mathsf{p}_0 x) = 1 \\ & \bigwedge_{i<n} \mathsf{min}^i_{s_1,\ldots,s_n}(\mathsf{p}_0 x) \neq 1, \\ x & \mathsf{p}_0 x = \mathsf{ex}. \end{cases}$$

Next, we define the term r which gives the recursive equation which will be solved by invk. Assume the method $\mathtt{m}$ is defined exactly in the classes $\mathtt{C_1}, \ldots, \mathtt{C_n}$ and $mbody(\mathtt{m}, \mathtt{C_i}) = (\overline{\mathtt{x}}_\mathtt{i}, \mathsf{e_i})$ for all $i \leq n$. Assume further that $\overline{\mathtt{x}}_i$ is $\mathtt{x}_1, \ldots, \mathtt{x}_z$, then we can define a term $g^h_{\mathsf{e}_i}$ of $\mathcal{L}_j$ so that we can prove in PTN for $\vec{b} = b_1, \ldots, b_z$:

1. If $a\!\downarrow$ and $\vec{b}\!\downarrow$ hold and there exists a natural number j so that $\mathsf{p}_0 b_j = \mathsf{ex}$, then we get $g^h_{\mathsf{e}_i}(a, \vec{b}) = b_j$ where j is the least natural number satisfying $\mathsf{p}_0 b_j = \mathsf{ex}$.

2. Else we find $g^h_{\mathsf{e}_i}(a, \vec{b}) \simeq (\lambda this.\lambda[\![\overline{\mathtt{x}}_i]\!]_h.[\![\mathsf{e}_i]\!]_h) a \vec{b}$.

We see that the terms $g^h_{\mathsf{e}_i}$ depend on h. Now we let the $\mathcal{L}_j$ term r be such that for every method $\mathtt{m}$ in our class table CT we have

$$\mathsf{r}\, h(\mathtt{m}^*, x, \vec{y}) \simeq \mathsf{over}_{\mathtt{C}^*_1,\ldots,\mathtt{C}^*_\mathtt{n}}(g^h_{\mathsf{e}_1}, \ldots, g^h_{\mathsf{e}_n})(x, \vec{y}). \tag{7.4}$$

We define invk to be the least fixed point of r, i.e. we set $\mathsf{invk} := \mathsf{l} \cdot \mathsf{r}$. In the following the interpretation of an FJ expression e will be its translation

$[\![\mathtt{e}]\!]_{\mathsf{r}\cdot\mathsf{invk}}$ and we will only write $[\![\mathtt{e}]\!]$ for this translation of an expression $\mathtt{e}$ relative to the term $\mathsf{r}\cdot\mathsf{invk}$.

It remains to give the interpretation of Featherweight Java classes. Let us begin with the basic types. The example of the type `boolean` will show how we can use the type structure of explicit mathematics to model the basic types of Java. If we let 0 and 1 denote "false" and "true", then the interpretation $[\![\mathtt{boolean}]\!]$ of the basic type `boolean` is given by

$$\{(\mathtt{boolean}^*, b) \mid b = 0 \vee b = 1\}.$$

Here we see that an element $x \in [\![\mathtt{boolean}]\!]$ is a pair whose first component carries the run-time type information of x, namely $\mathtt{boolean}^*$, and whose second component is the actual truth value. For the Java expressions `false` and `true` we can set

$$[\![\mathtt{false}]\!] = (\mathtt{boolean}^*, 0) \qquad [\![\mathtt{true}]\!] = (\mathtt{boolean}^*, 1).$$

We will interpret the classes of FJ as fixed point types in explicit mathematics satisfying the following fixed point specification. If the class table CT contains a class named $\mathtt{C}$ with $\mathtt{C}^* = i$, then the following formula is included in our specification:

$$(X)_i = Y_{i1} \times \cdots \times Y_{im_{\mathtt{C}}},$$

where $m_{\mathtt{C}}$ is again the least natural number such that for all field names $\mathtt{f}$ occurring in *fields*($\mathtt{C}$) we have $\mathtt{f}^* < m_{\mathtt{C}}$. Y_{ij} is defined according to the following three clauses:

1. If there is a basic type $\mathtt{D}$ and a field name $\mathtt{f}$ such that $\mathtt{D}\ \mathtt{f}$ belongs to *fields*($\mathtt{C}$), then $Y_{i\mathtt{f}^*}$ is equal to the interpretation of $\mathtt{D}$.

2. If there is a class $\mathtt{D}$ and a field name $\mathtt{f}$ such that $\mathtt{D}\ \mathtt{f}$ belongs to *fields*($\mathtt{C}$) and $\mathtt{E}_1, \ldots, \mathtt{E}_n$ is the list of all classes $\mathtt{E}_j$ in CT for which $\mathtt{E}_j <: \mathtt{D}$ is derivable, then

$$Y_{i\mathtt{f}^*} = \{(\mathtt{E}_1^*, x) \mid x \in (X)_{\mathtt{E}_1^*}\} \cup \cdots \cup \{(\mathtt{E}_n^*, x) \mid x \in (X)_{\mathtt{E}_n^*}\} \cup \{(\mathsf{ex}, 0)\}.$$

3. If there is no field name $\mathtt{f}$ occurring in *fields*($\mathtt{C}$) such that $\mathtt{f}^* = j$, then Y_{ij} is the universal type V, in particular we get $Y_{im_{\mathtt{C}}} = \mathsf{V}$.

As we have shown before, in PTN there provably exists a fixed point FP satisfying the above specification. Since our fixed class table CT contains only finitely many classes, we can set up the following definition for the

interpretation $[\![\mathtt{C}]\!]$ of a class $\mathtt{C}$: if $\mathtt{E}_1, \ldots, \mathtt{E}_n$ is the list of all classes $\mathtt{E}_i$ in CT for which $\mathtt{E}_i <: \mathtt{C}$ is derivable, then

$$[\![\mathtt{C}]\!] = \{(\mathtt{E}_1^*, x) \mid x \in (\mathsf{FP})_{\mathtt{E}_1^*}\} \cup \cdots \cup \{(\mathtt{E}_n^*, x) \mid x \in (\mathsf{FP})_{\mathtt{E}_n^*}\} \cup \{(\mathsf{ex}, 0)\}.$$

We include the value $(\mathsf{ex}, 0)$ to the interpretation of all classes because this simplifies the presentation of the proofs about soundness with respect to typing. Of course, we could exclude $(\mathsf{ex}, 0)$ from the above types, which would be more natural, but then we would have to treat it as a special case in all of the proofs.

7.3 Soundness results

In this section we will prove that our model for Featherweight Java is sound with respect to subtyping, typing and reductions. We start with a theorem about soundness with respect to subtyping, which is an immediate consequence of the interpretation of classes.

Theorem 45. *For all classes* $\mathtt{C}$ *and* $\mathtt{D}$ *of the class table with* $\mathtt{C} <: \mathtt{D}$ *it is provable in* PTN *that* $[\![\mathtt{C}]\!] \subset [\![\mathtt{D}]\!]$.

The soundness of the semantics with respect to the subtype relation does not depend on the fact that a type is interpreted as the union of all its subtypes. As the next theorem states, our model is sound with respect to subtyping if we allow to coerce the run-time type of an object into a super type. Coercions are operations which change the type of an object. In models of object-oriented programming, these constructs can be used to give a semantics for early-bound overloading, cf. Castagna, Ghelli and Longo [21, 23] or Chapter 1 of this thesis. This is achieved by coercing the type of an object into its static type before selecting the best matching branch. In Java for example, we find that if there are several methods with the same name but different signatures defined in one class, then the selection of the method to be invoked is based on the static types of the arguments, i.e. on early-binding. Note that in Featherweight Java this form of overloading is not included.

Theorem 46. *For all classes* $\mathtt{C}$ *and* $\mathtt{D}$ *with* $\mathtt{C} <: \mathtt{D}$ *the following is provable in* PTN*:*

$$x \in [\![\mathtt{C}]\!] \wedge \mathsf{p}_0 x \neq \mathsf{ex} \rightarrow (\mathtt{D}^*, \mathsf{p}_1 x) \in [\![\mathtt{D}]\!].$$

Proof. By induction on the length of the derivation of $\mathtt{C} <: \mathtt{D}$ we show that $(\mathsf{FP})_{\mathtt{C}^*} \subset (\mathsf{FP})_{\mathtt{D}^*}$. The only non-trivial case is when the following rule has

been applied

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ldots\}}{\mathtt{C} <: \mathtt{D}}.$$

So we assume $CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ldots\}$. By our definition of FP we obtain

$$(\mathsf{FP})_{\mathtt{C}^*} = Y_{\mathtt{C}^*1} \times \cdots \times Y_{\mathtt{C}^* m_{\mathtt{C}}}$$

and $(\mathsf{FP})_{\mathtt{D}^*}$ is of the form $Y_{\mathtt{D}^*1} \times \cdots \times Y_{\mathtt{D}^* m_{\mathtt{D}}}$. By the rules for field lookup we know that if *fields*(D) contains E g, then E g also belongs to *fields*(C). Therefore, we have $m_{\mathtt{D}} \leq m_{\mathtt{C}}$ and for all $i < m_{\mathtt{D}}$ we get $Y_{\mathtt{C}^* i} \subset Y_{\mathtt{D}^* i}$ by the fixed point specification for FP and our general assumption that class typing is OK. Moreover, we obviously have

$$Y_{\mathtt{C}^* m_{\mathtt{D}}} \times \cdots \times Y_{\mathtt{C}^* m_{\mathtt{C}}} \subset Y_{\mathtt{D}^* m_{\mathtt{D}}} = \mathsf{V}.$$

Therefore, we conclude that the claim holds. 

Before proving soundness with respect to typing, we have to show some preparatory lemmas.

Definition 47. If $\overline{\mathtt{D}}$ is the list $\mathtt{D}_1, \ldots, \mathtt{D}_n$, then $[\![\overline{\mathtt{D}}]\!]$ stands for $[\![\mathtt{D}_1]\!] \times \cdots \times [\![\mathtt{D}_n]\!]$; and if $\overline{\mathtt{e}} = \mathtt{e}_1, \ldots \mathtt{e}_n$, then $[\![\overline{\mathtt{e}}]\!]_h \in [\![\overline{\mathtt{D}}]\!]$ means

$$([\![\mathtt{e}_1]\!]_h, \ldots, [\![\mathtt{e}_n]\!]_h) \in [\![\mathtt{D}_1]\!] \times \cdots \times [\![\mathtt{D}_n]\!].$$

For $\Gamma = \mathtt{x}_1 : \mathtt{D}_1, \ldots, \mathtt{x}_n : \mathtt{D}_n$ we set

$$[\![\Gamma]\!]_h := [\![\mathtt{x}_1]\!]_h \in [\![\mathtt{D}_1]\!] \wedge \cdots \wedge [\![\mathtt{x}_n]\!]_h \in [\![\mathtt{D}_n]\!].$$

Definition 48. We define the type $\mathcal{T}$ to be the intersection of all the types

$$(\{\mathtt{m}^*\} \times [\![\mathtt{C}]\!] \times [\![\overline{\mathtt{D}}]\!]) \curvearrowright [\![\mathtt{B}]\!]$$

for all methods m and for all classes C occurring in CT with $mtype(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{D}} \to \mathtt{B}$.

Lemma 49. *If* $\Gamma \vdash \mathtt{e} \in \mathtt{C}$ *is derivable in FJ, then we can prove in* PTN *that* $h \in \mathcal{T}$ *implies*

$$[\![\Gamma]\!]_h \wedge [\![\mathtt{e}]\!]_h\!\downarrow \to [\![\mathtt{e}]\!]_h \in [\![\mathtt{C}]\!].$$

Proof. Proof by induction on the length of the derivation of $\Gamma \vdash \mathtt{e} \in \mathtt{C}$. We assume $[\![\Gamma]\!]_h \wedge [\![\mathtt{e}]\!]_h\!\downarrow$ and distinguish the different cases for the last rule in the derivation of $\Gamma \vdash \mathtt{e} \in \mathtt{C}$:

1. $\Gamma \vdash \mathtt{x} \in \Gamma(\mathtt{x})$: trivial.

2. $\Gamma \vdash \mathtt{e.f}_i \in \mathtt{C}_i$: $[\![\mathtt{e.f}_i]\!]_h\!\downarrow$ implies $[\![\mathtt{e}]\!]_h\!\downarrow$ by strictness. Hence, we get by the induction hypothesis $[\![\mathtt{e}]\!]_h \in [\![\mathtt{C}_0]\!]$ and $\mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$. By the definition of $[\![\mathtt{C}_0]\!]$ this yields $\mathsf{proj}\,\mathtt{f}_i^*\,[\![\mathtt{e}]\!]_h \in [\![\mathtt{C}_i]\!]$. Finally we conclude by $\mathsf{proj}\,\mathtt{f}_i^*\,[\![\mathtt{e}]\!]_h \simeq [\![\mathtt{e.f}_i]\!]_h$ that the claim holds.

3. $\Gamma \vdash \mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{C}$: by the induction hypothesis and Theorem 45 we obtain $[\![\overline{\mathtt{e}}]\!]_h \in [\![\overline{\mathtt{C}}]\!] \subset [\![\overline{\mathtt{D}}]\!]$ and $[\![\mathtt{e}_0]\!]_h \in [\![\mathtt{C}_0]\!]$. Moreover, we have

$$\mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{D}} \to \mathtt{C}.$$

   Hence, we conclude by $h \in \mathcal{T}$ and $[\![\mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)}]\!]_h \simeq h(\mathtt{m}^*, [\![\mathtt{e}_0]\!]_h, [\![\overline{\mathtt{e}}]\!]_h)$ that the claim holds.

4. $\Gamma \vdash \mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{C}$: by the induction hypothesis and Theorem 45 we have $[\![\overline{\mathtt{e}}]\!]_h \in [\![\overline{\mathtt{C}}]\!] \subset [\![\overline{\mathtt{D}}]\!]$. Further we know $\mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}}$. Therefore, the claim holds by the definition of new.

5. If the last rule was an upcast, then the claim follows immediately from the induction hypothesis, the definition of the term cast and Theorem 45.

6. Assume the last rule was a downcast or a stupid cast. By the induction hypothesis we get $[\![\mathtt{e}]\!]_h \in [\![\mathtt{D}]\!]$. Then $\mathtt{D} \not<: \mathtt{C}$ implies $\mathsf{sub}(\mathsf{p}_0[\![\mathtt{e}]\!]_h, \mathtt{C}^*) = 0$ and by the definition of cast we get $[\![\mathtt{(C)\,e}]\!]_h \simeq (\mathsf{ex}, 0)$. Hence, the claim holds. 

Lemma 50. *If* $\Gamma \vdash \mathtt{e} \in \mathtt{C}$ *is derivable in FJ, then we can prove in* PTN *that* $g, h \in \mathcal{T}$ *and* $g \sqsubseteq_{\mathcal{T}} h$ *imply* $[\![\Gamma]\!]_g \wedge [\![\mathtt{e}]\!]_g\!\downarrow \to [\![\mathtt{e}]\!]_g = [\![\mathtt{e}]\!]_h$.

Proof. Proof by induction on the length of the derivation of $\Gamma \vdash \mathtt{e} \in \mathtt{C}$. Assume $[\![\Gamma]\!]_g$ and $[\![\mathtt{e}]\!]_g\!\downarrow$ hold. We distinguish the following cases:

1. $\Gamma \vdash \mathtt{x} \in \Gamma(\mathtt{x})$: trivial.

2. $\Gamma \vdash \mathtt{e}_0\mathtt{.f}_i \in \mathtt{C}_i$: we know $\Gamma \vdash \mathtt{e}_0 \in \mathtt{C}_0$. Hence, we get by the induction hypothesis $[\![\mathtt{e}_0]\!]_g = [\![\mathtt{e}_0]\!]_h$ and therefore the claim holds.

3. $\Gamma \vdash \mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{C}$: we get $\Gamma \vdash \mathtt{e}_0 \in \mathtt{C}_0$, $\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{C}}$ and

$$\mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{D}} \to \mathtt{C} \quad \text{as well as} \quad \overline{\mathtt{C}} <: \overline{\mathtt{D}}. \tag{7.5}$$

   Because of $[\![\mathtt{e}]\!]_g\!\downarrow$ we get $[\![\mathtt{e}_0]\!]_g\!\downarrow$ and $[\![\overline{\mathtt{e}}]\!]_g\!\downarrow$. Hence, the induction hypothesis yields $[\![\mathtt{e}_0]\!]_g = [\![\mathtt{e}_0]\!]_h$ as well as $[\![\overline{\mathtt{e}}]\!]_g = [\![\overline{\mathtt{e}}]\!]_h$. By Lemma 49 we get

$[\![\Gamma]\!]_g \vdash [\![\mathtt{e}_0]\!]_g \in [\![\mathtt{C}_0]\!]$, $[\![\Gamma]\!]_g \vdash [\![\overline{\mathtt{e}}]\!]_g \in [\![\overline{\mathtt{C}}]\!]$ as well as $[\![\Gamma]\!]_g \vdash [\![\mathtt{e}]\!]_g \in [\![\mathtt{C}]\!]$. With (7.5), $g \in \mathcal{T}$, $h \in \mathcal{T}$ and $g \sqsubseteq_{\mathcal{T}} h$ we conclude

$$\begin{aligned} [\![\mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}})]\!]_g &= g(\mathtt{m}^*, [\![\mathtt{e}_0]\!]_g, [\![\overline{\mathtt{e}}]\!]_g) \\ &= h(\mathtt{m}^*, [\![\mathtt{e}_0]\!]_h, [\![\overline{\mathtt{e}}]\!]_h) \\ &= [\![\mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}})]\!]_h \end{aligned}$$

4. $\Gamma \vdash \mathtt{new\ C}(\overline{\mathtt{e}}) \in \mathtt{C}$: as in the second case, the claim follows immediately from the induction hypothesis.

5. If the last rule was a cast, then again the claim is a direct consequence of the induction hypothesis.

Lemma 51. *In* PTN *it is provable that* $\mathsf{r} \in (\mathcal{T} \to \mathcal{T})$.

Proof. Assume $h \in \mathcal{T}$ and let $(\mathtt{m}^*, c, \vec{d}) \in (\{\mathtt{m}^*\} \times [\![\mathtt{C}_0]\!] \times [\![\overline{\mathtt{D}}]\!])$ for a method $\mathtt{m}$ and classes $\mathtt{C}_0, \overline{\mathtt{D}}, \mathtt{C}$ with $mtype(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{D}} \to \mathtt{C}$. We have to show

$$\mathsf{r}h(\mathtt{m}^*, c, \vec{d})\downarrow \to \mathsf{r}h(\mathtt{m}^*, c, \vec{d}) \in [\![\mathtt{C}]\!].$$

So assume $\mathsf{r}h(\mathtt{m}^*, c, \vec{d})\downarrow$. By (7.4) we find

$$\mathsf{r}h(\mathtt{m}^*, c, \vec{d}) = \mathsf{over}_{\mathtt{C}_1^*,\dots,\mathtt{C}_n^*}(g^h_{\mathtt{e}_1}, \dots, g^h_{\mathtt{e}_n})(c, \vec{d}).$$

Hence, if $c = (\mathsf{ex}, 0)$ then we obtain $\mathsf{r}h(\mathtt{m}^*, c, \vec{d}) = (\mathsf{ex}, 0)$ and the claim holds. If $c \neq (\mathsf{ex}, 0)$ then we get $\mathsf{sub}(\mathsf{p}_0 c, \mathtt{C}_0) = 1$. By our interpretation of classes, there exists a class $\mathtt{B}$ such that $\mathtt{B} <: \mathtt{C}_0$, $\mathsf{p}_0 c = \mathtt{B}^*$ as well as $c \in [\![\mathtt{B}]\!]$. Let $mbody(\mathtt{m}, \mathtt{B}) = (\overline{\mathtt{x}}_i, \mathtt{e}_i)$. Hence we have

$$\mathsf{r}\, h\, (\mathtt{m}^*, c, \vec{d}) = g^h_{\mathtt{e}_i}(c, \vec{d}) = [\![\mathtt{e}_i]\!]_h[c/this, \vec{d}/\vec{x}_i]. \tag{7.6}$$

By the rules for method body lookup there is a class $\mathtt{A}$ such that $\mathtt{B} <: \mathtt{A} <: \mathtt{C}_0$ and the method $\mathtt{m}$ is defined in $\mathtt{A}$ by the expression $\mathtt{e}_i$. By our general assumption that method typing is OK we obtain

$$\overline{\mathtt{x}} : \overline{\mathtt{D}}, \mathtt{this} : \mathtt{A} \vdash \mathtt{e}_i \in \mathtt{E}_0 \qquad \mathtt{E}_0 <: \mathtt{C}. \tag{7.7}$$

By Theorem 45 we get $c \in [\![A]\!]$ and therefore, we conclude by $h \in \mathcal{T}$, (7.6), Lemma 49 and (2.1) that $\mathsf{r}h(\mathtt{m}^*, c, \vec{d}) \in [\![\mathtt{C}]\!]$ holds.

Lemma 52. *In* PTN *it is provable that* $\mathsf{invk} \in \mathcal{T}$.

Proof. We have defined invk as $\mathsf{l}\cdot\mathsf{r}$. Lemma 51 states $\mathsf{r} \in (\mathcal{T} \to \mathcal{T})$. Therefore it remains to show that r is $\mathcal{T}$ monotonic. Let $g, h \in \mathcal{T}$ such that $g \sqsubseteq_{\mathcal{T}} h$. We have to show $\mathsf{r}\,g \sqsubseteq_{\mathcal{T}} \mathsf{r}\,h$. That is we show for all methods $\mathtt{m}$ and classes $\mathtt{C}_0$, $\overline{\mathtt{D}}$ and $\mathtt{C}$ with $\mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{D}} \to \mathtt{C}$

$$(\forall x \in \{\mathtt{m}^*\} \times [\![\mathtt{C}_0]\!] \times [\![\overline{\mathtt{D}}]\!])\mathsf{r}\,g\,x \sqsubseteq \mathsf{r}\,h\,x.$$

Let $(\mathtt{m}^*, c, \vec{d}) \in \{\mathtt{m}^*\} \times [\![\mathtt{C}_0]\!] \times [\![\overline{\mathtt{D}}]\!]$ and $\mathsf{r}\,g\,(\mathtt{m}^*, c, \vec{d})\!\downarrow$. It remains to show

$$\mathsf{r}\,g(\mathtt{m}^*, c, \vec{d}) = \mathsf{r}\,h\,(\mathtt{m}^*, c, \vec{d}). \tag{7.8}$$

As in Lemma 51 we find $\mathsf{r}\,g\,(\mathtt{m}^*, c, \vec{d}) = [\![\mathsf{e}_i]\!]_g[c/this, \vec{d}/\vec{x}_i]$ for some i and we have to show that this is equal to $[\![\mathsf{e}_i]\!]_h[c/this, \vec{d}/\vec{x}_i]$. By (7.7), which was a consequence of our general assumption that method typing is OK and Lemma 50 we find $x \in [\![\overline{\mathtt{D}}]\!] \wedge this \in [\![\mathtt{C}_0]\!] \to [\![\mathsf{e}_i]\!]_g = [\![\mathsf{e}_i]\!]_h$. Hence, by (2.1) we obtain

$$[\![\mathsf{e}_i]\!]_g[c/this, \vec{d}/\vec{x}_i] = [\![\mathsf{e}_i]\!]_h[c/this, \vec{d}/\vec{x}_i]$$

and we finally conclude that (7.8) holds. 

The next theorem states that our model is sound with respect to typing.

Theorem 53. *If* $\Gamma \vdash \mathsf{e} \in \mathtt{C}$ *is derivable in FJ, then in* PTN *it is provable that*

$$[\![\Gamma]\!] \wedge [\![\mathsf{e}]\!]\!\downarrow \to [\![\mathsf{e}]\!] \in [\![\mathtt{C}]\!].$$

Proof. By the previous lemma we obtain $\mathsf{invk} \in \mathcal{T}$ and therefore $\mathsf{r}\cdot\mathsf{invk} \in \mathcal{T}$ by Lemma 51. Then we apply Lemma 49 in order to verify our claim.

As we have seen in Example 38 we cannot prove soundness with respect to reductions of our model construction for the original notion of reduction of Featherweight Java. The reason is that FJ does not enforce a call-by-value evaluation strategy whereas theories of types and names adopt call-by-value evaluation via their strictness axioms. Moreover, Examples 39 and 40 show that we also have to take care of exceptions and the role of late-binding. Let $\longrightarrow'$ be the variant of the reduction relation $\longrightarrow$ with a call-by-value evaluation strategy which respects exceptions.

Definition 54. Let $\mathtt{a}$ and $\mathtt{b}$ be two FJ expressions. We define the reduction relation $\longrightarrow'$ by induction on the structure of $\mathtt{a}$: $\mathtt{a} \longrightarrow' \mathtt{b}$ if and only if $\mathtt{a} \longrightarrow \mathtt{b}$, where all subexpressions of $\mathtt{a}$ are in closed normal form with respect to $\longrightarrow'$ and $\mathtt{a}$ does not contain subexpressions like $(\mathtt{D})\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})$ with $\mathtt{C} \not<: \mathtt{D}$.

As shown in Example 40 the following lemma can only be established for *closed* expressions in normal form.

Lemma 55. *Let* $\mathtt{e}$ *be a well-typed Featherweight Java expression in closed normal form with respect to* $\longrightarrow'$*. Then in* PTN *it is provable that* $[\![\mathtt{e}]\!]\downarrow$*. Moreover, if* $\mathtt{e}$ *is not of the form* $(\mathtt{D})\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})$ *with* $\mathtt{C} \not<: \mathtt{D}$ *and does not contain subexpressions of this form, then it is provable in* PTN *that* $\mathsf{p}_0[\![\mathtt{e}]\!] \neq \mathsf{ex}$*.*

Proof. Let $\mathtt{e}$ be a well-typed closed FJ expression in normal form. First, we prove by induction on the structure of $\mathtt{e}$ that one of the following holds: $\mathtt{e}$ itself is of the form $(\mathtt{D})\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})$ with $\mathtt{C} \not<: \mathtt{D}$ or it contains a subexpression of this form or $\mathtt{e}$ does not contain such subexpressions and $\mathtt{e}$ is $\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})$ for a class $\mathtt{C}$ and expressions $\overline{\mathtt{e}}$. We distinguish the five cases for the structure of $\mathtt{e}$ as given by the syntax for expressions.

1. $\mathtt{x}$. This is not possible since $\mathtt{e}$ is a closed term.

2. $\mathtt{e}_0.\mathtt{f}$. The induction hypothesis applies to $\mathtt{e}_0$. In the first two cases we obtain that $\mathtt{e}$ contains a subexpression of the form $(\mathtt{D})\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})$ with $\mathtt{C} \not<: \mathtt{D}$. In the last case we get by Theorem 37 that $\mathtt{e}$ cannot be in normal form.

3. $\mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}})$. Similar to the previous case.

4. $\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})$. The induction hypothesis applies to $\overline{\mathtt{e}}$. Again, we obtain in the first two cases that $\mathtt{e}$ contains a subexpression of the form $(\mathtt{D})\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})$ with $\mathtt{C} \not<: \mathtt{D}$. In the last case we also see that $\mathtt{e}$ fulfils the conditions of the last case.

5. $(\mathtt{C})\mathtt{e}_0$. We apply the induction hypothesis and infer that $\mathtt{e}$ must satisfy condition one or two since it is in normal form.

Now, we know that $\mathtt{e}$ satisfies one of the three conditions above. In the first two cases we obtain by induction on the structure of $\mathtt{e}$ that $[\![\mathtt{e}]\!] = (\mathsf{ex}, 0)$. If the first two cases do not apply, then $\mathtt{e}$ is only built up of $\mathtt{new}$ expressions and we can prove by induction on the structure of $\mathtt{e}$ that $[\![\mathtt{e}]\!]\downarrow$ and $\mathsf{p}_0[\![\mathtt{e}]\!] \neq \mathsf{ex}$. 

Lemma 56. *For all FJ expressions* $\mathtt{e}, \overline{\mathtt{d}}$ *and variables* $\overline{\mathtt{x}}$ *it is provable in* PTN *that* $[\![\mathtt{e}]\!][[\![\overline{\mathtt{d}}]\!]/[\![\overline{\mathtt{x}}]\!]] \simeq [\![\mathtt{e}[\overline{\mathtt{d}}/\overline{\mathtt{x}}]]\!]$*.*

Proof. We proceed by induction on the term structure of $\mathtt{e}$. The following cases have to be distinguished.

1. $\mathtt{e}$ is a variable, then the claim obviously holds.

2. $\mathtt{e}$ is of the form $\mathtt{e}_0.\mathtt{g}$. We have $[\![\mathtt{e}_0.\mathtt{g}]\!][[\![\overline{\mathtt{d}}]\!]/[\![\overline{\mathtt{x}}]\!]] \simeq (\mathsf{proj}\,\mathtt{g}^*\,[\![\mathtt{e}_0]\!])[[\![\overline{\mathtt{d}}]\!]/[\![\overline{\mathtt{x}}]\!]]$. Since none of the variables of $[\![\overline{\mathtt{x}}]\!]$ occur freely in proj or $\mathtt{g}^*$ we get $\mathsf{proj}\,\mathtt{g}^*\,([\![\mathtt{e}_0]\!][[\![\overline{\mathtt{d}}]\!]/[\![\overline{\mathtt{x}}]\!]])$. This is equal to $\mathsf{proj}\,\mathtt{g}^*\,([\![\mathtt{e}_0[\overline{\mathtt{d}}/\overline{\mathtt{x}}]]\!])$ by the induction hypothesis and we finally obtain $[\![\mathtt{e}_0.\mathtt{g}[\overline{\mathtt{d}}/\overline{\mathtt{x}}]]\!]$.

3. $\mathtt{e}$ is of the form $\mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}})$. We have

$$[\![\mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}})]\!][[\![\overline{\mathtt{d}}]\!]/[\![\overline{\mathtt{x}}]\!]] \simeq (\mathsf{r\,invk}\,(\mathtt{m}^*, [\![\mathtt{e}_0]\!], [\![\overline{\mathtt{e}}]\!]))[[\![\overline{\mathtt{d}}]\!]/[\![\overline{\mathtt{x}}]\!]].$$

   Again, since none of the variables of $[\![\overline{\mathtt{x}}]\!]$ occur freely in $\mathsf{r\,invk}$ or in $\mathtt{m}^*$ this is equal to $\mathsf{r\,invk}\,(\mathtt{m}^*, [\![\mathtt{e}_0]\!][[\![\overline{\mathtt{d}}]\!]/[\![\overline{\mathtt{x}}]\!]], [\![\overline{\mathtt{e}}]\!][[\![\overline{\mathtt{d}}]\!]/[\![\overline{\mathtt{x}}]\!]])$. By the induction hypothesis we obtain $\mathsf{r\,invk}\,(\mathtt{m}^*, [\![\mathtt{e}_0[\overline{\mathtt{d}}/\overline{\mathtt{x}}]]\!], [\![\overline{\mathtt{e}}[\overline{\mathtt{d}}/\overline{\mathtt{x}}]]\!])$, and finally we get $[\![\mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}})[\overline{\mathtt{d}}/\overline{\mathtt{x}}]]\!]$.

4. $\mathtt{e}$ is of the form $\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})$. We have

$$[\![\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})]\!][[\![\overline{\mathtt{d}}]\!]/[\![\overline{\mathtt{x}}]\!]] \simeq (\mathsf{new}\,\mathtt{C}^*([\![\overline{\mathtt{e}}]\!]))[[\![\overline{\mathtt{d}}]\!]/[\![\overline{\mathtt{x}}]\!]].$$

   Again this is equal to $\mathsf{new}\,\mathtt{C}^*([\![\overline{\mathtt{e}}]\!][[\![\overline{\mathtt{d}}]\!]/[\![\overline{\mathtt{x}}]\!]])$ which is by the induction hypothesis $\mathsf{new}\,\mathtt{C}^*([\![\overline{\mathtt{e}}[\overline{\mathtt{d}}/\overline{\mathtt{x}}]]\!])$. Finally we obtain $[\![\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})[\overline{\mathtt{d}}/\overline{\mathtt{x}}]]\!]$.

5. $\mathtt{e}$ is of the form $(\mathtt{C})\mathtt{e}_0$. We obtain

$$[\![(\mathtt{C})\mathtt{e}_0]\!][[\![\overline{\mathtt{d}}]\!]/[\![\overline{\mathtt{x}}]\!]] \simeq \mathsf{cast}\,\mathtt{C}^*([\![\mathtt{e}_0]\!][[\![\overline{\mathtt{d}}]\!]/[\![\overline{\mathtt{x}}]\!]]).$$

   By the induction hypothesis this is equal to

$$\mathsf{cast}\,\mathtt{C}^*\,[\![\mathtt{e}_0[\overline{\mathtt{d}}/\overline{\mathtt{x}}]]\!] \simeq [\![(\mathtt{C})\mathtt{e}_0[\overline{\mathtt{d}}/\overline{\mathtt{x}}]]\!].$$

Now we prove soundness with respect to call-by-value reductions.

Theorem 57. *Let* $\mathtt{g}, \mathtt{h}$ *be two FJ expressions so that* $\mathtt{g}$ *is well-typed and* $\mathtt{g} \longrightarrow' \mathtt{h}$ *is derivable in FJ, then in* PTN *it is provable that* $[\![\mathtt{g}]\!] \simeq [\![\mathtt{h}]\!]$.

Proof. We distinguish the three different rules for computations.

1. $\mathtt{g}$ is of the form $(\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})).\mathtt{f_i}$ and $\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$. We obtain

$$[\![(\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})).\mathtt{f_i}]\!] \simeq \mathsf{proj}\,\mathtt{f}_i^*\,[\![\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})]\!].$$

   By the definition of new this is equal to $\mathsf{proj}\,\mathtt{f}_i^*\,(t_{\mathtt{C}}[\![\overline{\mathtt{e}}]\!])$. For $\mathtt{g} \longrightarrow' \mathtt{h}$, we know that all subexpressions of $\mathtt{g}$ are closed, fully evaluated and not of the form $(\mathtt{D})\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})$ with $\mathtt{C} \not<: \mathtt{D}$. Hence we obtain by the Lemma 55 that $\mathsf{p}_0[\![\overline{\mathtt{e}}]\!] \neq \mathsf{ex}$ holds and therefore $\mathsf{proj}\,\mathtt{f}_i^*\,(t_{\mathtt{C}}[\![\overline{\mathtt{e}}]\!]) \simeq [\![\mathtt{e}_i]\!]$.

2. $\mathtt{g}$ is of the form $(\mathtt{new\ C}(\overline{\mathtt{e}})).\mathtt{m}(\overline{\mathtt{d}})$ and $mbody(\mathtt{m},\mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e}_0)$. Assume that the method $\mathtt{m}$ is defined in the classes $\mathtt{C}_1, \ldots, \mathtt{C}_n$ and not defined in any other class. Now we show by induction on the length of the derivation of $mbody(\mathtt{m},\mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e}_0)$ that there exists a k such that $1 \leq k \leq n$,

$$\mathsf{min}^k_{\mathtt{C}_1^*,\ldots,\mathtt{C}_n^*}(\mathtt{C}^*) = 1 \bigwedge_{l<k} \mathsf{min}^l_{\mathtt{C}_1^*,\ldots,\mathtt{C}_n^*}(\mathtt{C}^*) \neq 1 \tag{7.9}$$

and

$$mbody(\mathtt{m},\mathtt{C}) = mbody(\mathtt{m},\mathtt{C_k}). \tag{7.10}$$

If $\mathtt{m}$ is defined in $\mathtt{C}$, then there exists a k in $1,\ldots,n$ so that $\mathtt{C} = \mathtt{C_k}$. Hence (7.9) and (7.10) trivially hold. If $\mathtt{m}$ is not defined in $\mathtt{C}$, then $\mathtt{C}$ extends a class $\mathtt{B}$ with $mbody(\mathtt{m},\mathtt{B}) = (\overline{\mathtt{x}}, \mathtt{e}_0)$. In this case we have $mbody(\mathtt{m},\mathtt{C}) = mbody(\mathtt{m},\mathtt{B})$. Therefore (7.9) and (7.10) follow by the induction hypothesis.

We have assumed that $(\mathtt{new\ C}(\overline{\mathtt{e}})).\mathtt{m}(\overline{\mathtt{d}})$ is well-typed. Therefore we get that $\mathtt{new\ C}(\overline{\mathtt{e}})$ and $\overline{\mathtt{d}}$ are well-typed. Let $\mathtt{B}$ be the type satisfying $mbody(\mathtt{m},\mathtt{C}) = mbody(\mathtt{m},\mathtt{B})$ so that $\mathtt{m}$ is defined in $\mathtt{B}$. We find $\mathtt{C} <: \mathtt{B}$. Furthermore, let the types $\overline{\mathtt{D}}$ be such that $\overline{\mathtt{d}} \in \overline{\mathtt{D}}$. The expressions $\overline{\mathtt{e}}, \overline{\mathtt{d}}$ are in closed normal form and hence we obtain by Lemma 55, Theorem 53 and Theorem 45

$$[\![\mathtt{new\ C}(\overline{\mathtt{e}})]\!]_{\mathsf{r\,invk}} \in [\![\mathtt{B}]\!] \qquad [\![\overline{\mathtt{d}}]\!]_{\mathsf{r\,invk}} \in [\![\overline{\mathtt{D}}]\!]. \tag{7.11}$$

By our general assumption that method typing is OK we obtain

$$\overline{\mathtt{x}} : \overline{\mathtt{D}}, \mathtt{this} : \mathtt{B} \vdash \mathtt{e}_0 \in \mathtt{E}_0.$$

Hence applying Lemma 50 with $\mathsf{r\,invk} \cong_{\mathcal{T}} \mathsf{invk}$ yields

$$[\![\overline{\mathtt{x}}]\!]_{\mathsf{invk}} \in [\![\overline{\mathtt{D}}]\!] \wedge [\![\mathtt{this}]\!]_{\mathsf{invk}} \in [\![\mathtt{B}]\!] \rightarrow [\![\mathtt{e}_0]\!]_{\mathsf{invk}} \simeq [\![\mathtt{e}_0]\!]_{\mathsf{r\,invk}}. \tag{7.12}$$

Summing up, we get by (7.11), (7.12) and (2.1)

$$\begin{aligned} [\![(\mathtt{new\ C}(\overline{\mathtt{e}})).\mathtt{m}(\overline{\mathtt{d}})]\!]_{\mathsf{r\,invk}} &\simeq \mathsf{r\,invk}\,(\mathtt{m}^*, [\![\mathtt{new\ C}(\overline{\mathtt{e}})]\!]_{\mathsf{r\,invk}}, [\![\overline{\mathtt{d}}]\!]_{\mathsf{r\,invk}}) \\ &\simeq [\![\mathtt{e}_0]\!]_{\mathsf{invk}}[[\![\mathtt{new\ C}(\overline{\mathtt{e}})]\!]_{\mathsf{r\,invk}}/this, [\![\overline{\mathtt{d}}]\!]_{\mathsf{r\,invk}}/[\![\overline{\mathtt{x}}]\!]] \\ &\simeq [\![\mathtt{e}_0]\!]_{\mathsf{r\,invk}}[[\![\mathtt{new\ C}(\overline{\mathtt{e}})]\!]_{\mathsf{r\,invk}}/this, [\![\overline{\mathtt{d}}]\!]_{\mathsf{r\,invk}}/[\![\overline{\mathtt{x}}]\!]]. \end{aligned}$$

In view of Lemma 56 this is partially equal to

$$[\![\mathtt{e}_0[\mathtt{new\ C}(\overline{\mathtt{e}})/\mathtt{this}, \overline{\mathtt{d}}/\overline{\mathtt{x}}]]\!]_{\mathsf{r\,invk}}.$$

3. $\mathtt{g}$ is of the form $(\mathtt{D})(\mathtt{new\ C}(\overline{\mathtt{e}}))$ and $\mathtt{C} <: \mathtt{D}$. We have $\mathsf{sub}(\mathtt{C}^*, \mathtt{D}^*) = 1$ and therefore

$$[\![(\mathtt{D})(\mathtt{new\ C}(\overline{\mathtt{e}}))]\!] \simeq [\![\mathtt{new\ C}(\overline{\mathtt{e}})]\!].$$

7.4 Discussion and Remarks

Usually, denotational semantics are given in domain-theoretic notions. In such a semantics one has to include to each type an element $\perp$ which denotes the result of a non-terminating computation of this type, see for instance Alves-Foss and Lam [3]; whereas our recursion-theoretic model has the advantage that computations are interpreted as ordinary computations. This means we work with *partial* functions, which possibly do not yield a result for certain arguments, i.e. computations may really not terminate. In our opinion this model is very natural and captures well our intuition about non-termination.

In this section we have presented a predicative model of Featherweight Java. To be more precise, we have interpreted FJ in the theory PTN of explicit mathematics. As mentioned before, PTN is only slightly stronger than Peano arithmetic. However, we have completely left open the question whether we really need its full power. A careful analysis of our interpretation might reveal that this is not the case. In our construction we employ type-induction to prove Theorem 44 about the least fixed point operator as well as to prove that there are fixed points satisfying our specifications, cf. Theorem 41. Hence, comprehension is used in these induction arguments; and moreover it is needed to model the basic types of Java.

There are two interesting questions in this context: how much induction really is necessary for our construction and is it essential that we have full elementary comprehension and dependent choice available? Maybe one could rebuild our model with a restricted form of elementary comprehension, for example with some kind of Σ^+ comprehension. This could lead to an interpretation of FJ in a system with the proof-theoretic strength of Peano arithmetic or even primitive recursive arithmetic.

As already pointed out by Castagna, Ghelli and Longo [24] the dynamic definition of new classes is one of the main problems when overloaded functions are used to define methods. Indeed, in our semantics we assumed a fixed class table, i.e. the classes are given from the beginning and they will not change. An important goal would be to investigate an overloading based semantics for object-oriented programs with dynamic class definitions.

One approach to solve this problem is to consider an object model with *encapsulated multi-methods*, cf. Bruce, Cardelli, Castagna, the Hopkins Objects Group, Leavens and Pierce [12]. In this model, multi-methods are not defined as global functions, but the idea is to use them to define the bodies of some methods in a class definition, cf. Castagna [20]. Hence, this model can

be seen as a marriage of the "object as records" analogy and an overloading based object model: an object is modeled by a record which encapsulates all the methods defined for this object. However, these methods are not modeled by ordinary functions, but by overloaded ones. Hence, overriding a method in a subclass is basically achieved by adding a new branch to the corresponding multi-method (which is encapsulated in the subclass). Unfortunately, it seems that this object model has not been further analyzed, although we think that it offers a very elegant and promising way to deal with covariant method specialization.

Epilogue

> It would appear that we have reached the limits of what is possible to achieve with computer technology although one should be careful with such statements, as they tend to sound pretty silly in 5 years.
>
> John von Neuman

> Grau, teurer Freund, ist alle Theorie,
> Und grün des Lebens goldner Baum.
>
> Mephistopheles in Goethe's Faust

At the start of this thesis, we had a new object model which was based on *overloading* and *late-binding* instead of *encapsulation*. The question of its mathematical meaning is rather subtle and it was not known how a semantics for late-bound overloading looks like. We have investigated these questions using *systems of explicit mathematics*. Although originally designed for the study of constructive mathematics, these systems turned out to be a powerful and highly expressive tool for analyzing programming language concepts.

Castagna [21] proposed to interpret terms as pairs (type symbol, computation) in order to get a denotational semantics for overloading and late-binding. However, it was not known how to actually construct such a model. We have solved this problem by means of theories of types and names. In such systems, the types are represented by names, which are first order values. Hence, they can be used as arguments to functions and therefore, theories of types and names have type dependent computations naturally built in. In their most general form, overloading and late-binding imply a new form of *impredicativity*. We have analyzed these impredicativity phenomena using power types in explicit mathematics. Additionally, this provides a first example of an application of power types in systems of explicit mathematics.

Our investigation of overloading and late-binding has illuminated some delicate issues. One of those issues is the relationship between late-binding and the problem of *loss of information*. We have found that our explicit modeling of type-dependent computations yields a solution to this problem for free.

Moreover, we have seen that overloading in explicit mathematics gives rise to new models for second order λ calculi. They solve the problem of "too many subtypes" not by restricting the universe of types but by including overloaded functions. This indicates that systems of types and names are an appropriate framework for a further study of parametric polymorphism in the context of type-dependent computations.

We have applied our mathematical work on overloading and late-binding also to get a semantics for Featherweight Java. Usually, the research on Java's semantics takes an *operational* approach; and if a denotational semantics for object-oriented principles is presented, then it is often given in *domain-theoretic* notions. In contrast to that work, we have investigated a *denotational* semantics for Featherweight Java which is based on *recursion-theoretic* concepts. Hence, we look at models for Java, and object-oriented programming languages in general, from a new and more mathematical point of view.

To this aim, an applicative theory which allows to define a *least fixed point operator* had to be developed. We have obtained a proof-theoretically weak but highly expressive theory for representing object-oriented programs and for stating and proving many properties of them. Our theory is similar to the systems studied by Feferman [33, 34, 35] and Turner [101, 102] for functional programming. However, due to the fact that a least fixed point operator is definable in our theory, it is provable that certain recursive programs will not terminate. This is not the case in the systems of Feferman and Turner.

We have formalized our semantics in a *predicative* theory of types and names, thereby providing constructive foundations for Featherweight Java. This gives further evidence for Feferman's claim that impredicative assumptions are not needed for computational practice, a claim which has, up to now, only been verified for polymorphic functional programs. Our work shows that Feferman's claim also holds in the context of object-oriented programming.

Since Featherweight Java is the functional core of the Java language, this thesis also contributes to the ongoing research on extensions of Java combining aspects of functional and object-oriented programming. The concept of mixins or parametric heir classes, for example, originates from Common Lisp. Variants of Java featuring mixins have been recently studied by Ancona, Lagorio and Zucca [5] as well as by Flatt, Krishnamurthi and Felleisen [44]. We think that it is worthwhile to study the mathematical meaning of such new combinations of programming language concepts from the point of view of explicit mathematics.

Taking this all together, we see that Mephistopheles (see page 115) was com-

pletely misled in his statement about theory. We believe that our work is a good starting point for further investigations on the semantics of programming language concepts, and that the use of theories of types and names will remain a fruitful approach to understanding object-oriented programming.

Bibliography

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

[2] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Wolfram Menzel, and Peter H. Schmitt. The KeY approach: integrating object oriented design and formal verification. In Gerhard Brewka and Luís Moniz Pereira, editors, *Proc. 8th European Workshop on Logics in AI (JELIA)*, Lecture Notes in Artificial Intelligence. Springer, 2000.

[3] Jim Alves-Foss and Fong Shing Lam. Dynamic denotational semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 201–240. Springer, 1999.

[4] Roberto M. Amadio and Pierre-Louis Curien. *Domains and Lambda-Calculi*. Cambrigde University Press, 1998.

[5] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam: A smooth extension of Java with mixins. In E. Bertino, editor, *ECOOP 2000 - 14th European Conference on Object-Oriented Programming*, volume 1850 of *Lecture Notes in Computer Science*. Springer, 2000.

[6] Davide Ancona and Elena Zucca. A module calculus for Featherweight Java. Submitted.

[7] Hendrik Barendregt. *The Lambda Calculus*. North-Holland, revised edition, 1984.

[8] Michael J. Beeson. *Foundations of Constructive Mathematics: Metamathematical Studies*. Springer, 1985.

[9] Michael J. Beeson. Proving programs and programming proofs. In R. Barcan Marcus, G.J.W. Dorn, and P. Weingartner, editors, *Logic, Methodology and Philosophy of Science VII*, pages 51–82. North-Holland, 1986.

[10] Egon Börger, Joachim Schmid, Wolfram Schulte, and Robert Stärk. *Java and the Java Virtual Machine*. Lecture Notes in Computer Science. Springer, 2000. to appear.

[11] Kim B. Bruce. A paradigmatic object-oriented programming language: design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, 1994.

[12] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, the Hopkins Objects Group, Gary T. Leavens, and Benjamin C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

[13] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155:108–133, 1999.

[14] Kim B. Bruce and Giuseppe Longo. A modest model of records, inheritance, and bounded quantification. In C. Gunter and J. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, pages 151–195. MIT Press, 1994. First appeared in *Information and Compuation*, 87:196–240, 1990.

[15] Andrea Cantini. Relating Quine's NF to Feferman's EM. *Studia Logica*, 62:141–163, 1999.

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

[17] Luca Cardelli. Operationally sound update, 1995. Slides to the HOOTS '95 talk, available via `http://www.luca.demon.co.uk/Slides.html`.

[18] Luca Cardelli and John C. Mitchell. Operations on records. In C. Gunter and J. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, pages 295–350. MIT Press, 1994. First appeared in *Mathematical Structures in Computer Science*, 1:3–48, 1991.

[19] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[20] Giuseppe Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995.

[21] Giuseppe Castagna. *Object-Oriented Programming: A Unified Foundation*. Birkhäuser, 1997.

[22] Giuseppe Castagna. Unifying overloading and λ-abstraction: $\lambda^{\{\}}$. *Theoretical Computer Science*, 176:337–345, 1997.

[23] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A semantics for $\lambda\&$-early: a calculus with overloading and early binding. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 107–123. Springer, 1993.

[24] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995.

[25] Pietro Cenciarelli, Alexander Knapp, Bernhard Reus, and Martin Wirsing. An event-based structural operational semantics of multi-threaded Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 157–200. Springer, 1999.

[26] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proc. 17th ACM Symp. Principles of Programming Languages*. ACM Press, New York, 1990.

[27] Haskell Curry, James Hindley, and Jonathan Seldin. *Combinatory Logic*, volume II. North-Holland, 1972.

[28] Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the Java type system sound? *Theory and practice of object systems*, 5(1):3–24, 1999.

[29] Solomon Feferman. A language and axioms for explicit mathematics. In J.N. Crossley, editor, *Algebra and Logic*, volume 450 of *Lecture Notes in Mathematics*, pages 87–139. Springer, 1975.

[30] Solomon Feferman. Recursion theory and set theory: a marriage of convenience. In J. E. Fenstad, R. O. Gandy, and G. E. Sacks, editors, *Generalized Recursion Theory II, Oslo 1977*, pages 55–98. North Holland, 1978.

[31] Solomon Feferman. Constructive theories of functions and classes. In M. Boffa, D. van Dalen, and K. McAloon, editors, *Logic Colloquium '78*, pages 159–224. North Holland, 1979.

[32] Solomon Feferman. Iterated inductive fixed-point theories: application to Hancock's conjecture. In G. Metakides, editor, *Patras Logic Symposion*, pages 171–196. North Holland, 1982.

[33] Solomon Feferman. Polymorphic typed lambda-calculi in a type-free axiomatic framework. In W. Sieg, editor, *Logic and Computation*, volume 106 of *Contemporary Mathematics*, pages 101–136. American Mathematical Society, 1990.

[34] Solomon Feferman. Logics for termination and correctness of functional programs. In Y. N. Moschovakis, editor, *Logic from Computer Science*, volume 21 of *MSRI Publications*, pages 95–127. Springer, 1991.

[35] Solomon Feferman. Logics for termination and correctness of functional programs II: Logics of strength PRA. In P. Aczel, H. Simmons, and S. S. Wainer, editors, *Proof Theory*, pages 195–225. Cambridge University Press, 1992.

[36] Solomon Feferman. A new approach to abstract data types II: computation on ADTs as ordinary computations. In E. Börger, G. Jäger, H. Kleine Büning, and M. M. Richter, editors, *Computer Science Logic '91*, volume 626 of *Lecture Notes in Computer Science*, pages 79–95. Springer, 1992.

[37] Solomon Feferman. Definedness. *Erkenntnis*, 43:295–320, 1995.

[38] Solomon Feferman and Gerhard Jäger. Systems of explicit mathematics with non-constructive μ-operator. Part I. *Annals of Pure and Applied Logic*, 65(3):243–263, 1993.

[39] Solomon Feferman and Gerhard Jäger. Systems of explicit mathematics with non-constructive μ-operator. Part II. *Annals of Pure and Applied Logic*, 79(1):37–52, 1996.

[40] Solomon Feferman, Gerhard Jäger, and Thomas Strahm. Explicit Mathematics. In preparation.

[41] Matthias Felleisen and Daniel P. Friedman. *A Little Java, A Few Patterns*. MIT Press, 1998.

[42] Simon Finn and Michael P. Fourman. *Logic Manual for the LAMBDA System 3.2*. Abstract Hardware Ltd., November 1990.

[43] Marcello Fiore, Achim Jung, Eugenio Moggi, Peter O'Hearn, Jon Riecke, Giuseppe Rosolini, and Ian Stark. Domains and denotational semantics: history, accomplishments and open problems. *Bulletin of the European Association for Theoretical Computer Science*, 59:227–256, 1996.

[44] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. Technical report, Rice University, 1999. Corrected Version, original in J. Alvess-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 241–269, Springer, 1999.

[45] Mick Francis, Simon Finn, and Ellie Mayger. *Reference Manual for the LAMBDA System 3.2.* Abstract Hardware Ltd., 1990.

[46] Giorgio Ghelli. A static type system for late binding overloading. In A. Paepcke, editor, *Proc. of the Sixth International ACM Conference on Object-Oriented Programming Systems and Applications*, pages 129–145. Addison-Wesley, 1991.

[47] Jean-Yves Girard. *Interpretation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Thèse de doctorad d'état, Université de Paris VII, 1972.

[48] Thomas Glass. On power set in explicit mathematics. *The Journal of Symbolic Logic*, 61(2):468–489, 1996.

[49] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison Wesley, 1996. Also available via `http://java.sun.com/docs/`.

[50] Carl A. Gunter and John C. Mitchell. *Theoretical Aspects of Object-Oriented Programming.* MIT Press, 1994.

[51] James Hindley and Jonathan Seldin. *Introduction to Combinators and λ-calculus.* Cambridge University Press, 1986.

[52] Peter G. Hinman. *Recursion-Theoretic Hierarchies.* Springer, 1978.

[53] Atsushi Igarashi and Benjamin Pierce. On inner classes. In *Informal Proceedings of the Seventh Internatianal Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2000.

[54] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Object Oriented Programming: Systems, Languages and Applications (OOPSLA '99)*, volume 34 of *ACM SIGPLAN Notices*, pages 132–146, 1999.

[55] Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Berkum, Ulrich Hensel, and Hendrik Tews. Reasoning about Java classes (preliminary report). In *Object Oriented Programming: Systems, Languages and Applications (OOPSLA '98)*, volume 33 of *ACM SIGPLAN Notices*, pages 329–340, 1998.

[56] Gerhard Jäger. Induction in the elementary theory of types and names. In E. Börger, H. Kleine Büning, and M.M. Richter, editors, *Computer Science Logic '87*, volume 329 of *Lecture Notes in Computer Science*, pages 118–128. Springer, 1988.

[57] Gerhard Jäger. Type theory and explicit mathematics. In H.-D. Ebbinghaus, J. Fernandez-Prida, M. Garrido, M. Lascar, and M. Rodriguez Artalejo, editors, *Logic Colloquium '87*, pages 117–135. North-Holland, 1989.

[58] Gerhard Jäger. Applikative Theorien und explizite Mathematik. Technical Report IAM 97-001, Institut für Informatik und angewandte Mathematik, Universität Bern, 1997.

[59] Gerhard Jäger. Power types in explicit mathematics? *The Journal of Symbolic Logic*, 62(4):1142–1146, 1997.

[60] Gerhard Jäger, Reinhard Kahle, and Thomas Strahm. On applicative theories. In A. Cantini, E. Casari, and P. Minari, editors, *Logic and Foundations of Mathematics*, pages 83–92. Kluwer, 1999.

[61] Gerhard Jäger, Reinhard Kahle, and Thomas Studer. Universes in explicit mathematics. To appear in *Annals of Pure and Applied Logic*.

[62] Gerhard Jäger and Thomas Strahm. Totality in applicative theories. *Annals of Pure and Applied Logic*, 74:105–120, 1995.

[63] Gerhard Jäger and Thomas Studer. Extending the system T_0 of explicit mathematics: the limit and Mahlo axioms. Submitted.

[64] Neil Jones. *Computability and Complexity*. MIT Press, 1997.

[65] Reinhard Kahle. Einbettung des Beweissystems Lambda in eine Theorie von Operationen und Zahlen. Diploma thesis, Mathematisches Institut der Universität München, 1992.

[66] Reinhard Kahle. *Applikative Theorien und Frege-Strukturen*. PhD thesis, Institut für Informatik und angewandte Mathematik, Universität Bern, 1997.

[67] Reinhard Kahle and Thomas Studer. Formalizing non-termination of recursive programs. Submitted.

[68] Reinhard Kahle and Thomas Studer. A theory of explicit mathematics equivalent to ID_1. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic CSL 2000*, volume 1862 of *Lecture Notes in Computer Science*, pages 356–370. Springer, 2000.

[69] Samuel N. Kamin and Uday 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 463–496. MIT Press, 1994.

[70] Zohar Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.

[71] Markus Marzetta. *Predicative Theories of Types and Names*. PhD thesis, Institut für Informatik und angewandte Mathematik, Universität Bern, 1993.

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

[73] Yiannis Moschovakis. *Elementary Induction on Abstract Structures*. North-Holland, 1974.

[74] Tobias Nipkow and David von Oheimb. $\text{Java}_{\ell ight}$ is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*. ACM Press, New York, 1998.

[75] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990.

[76] Piergiorgio Odifreddi. *Classical Recursion Theory*. North-Holland, 1989.

[77] David von Oheimb. Axiomatic semantics for Java$_{light}$. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*. Fernuniversität Hagen, 2000.

[78] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, 1994.

[79] Wolfram Pohlers. *Proof Theory*, volume 1407 of *Lecture Notes in Mathematics*. Springer, 1989.

[80] Dieter Probst. Dependent choice in explicit mathematics. Diploma thesis, Institut für Informatik und angewandte Mathematik, Universität Bern, 1999.

[81] Dieter Probst and Thomas Studer. How to normalize the jay. To appear in *Theoretical Computer Science*.

[82] Chris Reade. *Elements of Functional Programming*. Addison-Wesley, 1989.

[83] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium: Proc. of Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer, 1974.

[84] Hartley Rogers, Jr. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, 1967.

[85] David Schmidt. *Denotational Semantics*. Allyn and Bacon, 1986.

[86] Dana S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. Manuscript, 1969. Later published in *Theoretical Computer Science*, 121:411–440,1993.

[87] Dana S. Scott. Identity and existence in intuitionistic logic. In M. Fourman, C. Mulvey, and D. Scott, editors, *Applications of Sheaves*, volume 753 of *Lecture Notes in Mathematics*, pages 660–696. Springer, 1979.

[88] Robert Stärk. Call-by-value, call-by-name and the logic of values. In D. van Dalen and M. Bezem, editors, *Computer Science Logic '96*, volume 1258 of *Lecture Notes in Computer Science*, pages 431–445. Springer, 1997.

[89] Robert Stärk. Why the constant 'undefined'? Logics of partial terms for strict and non-strict functional programming languages. *Journal of Functional Programming*, 8(2):97–129, 1998.

[90] Christopher Strachey. Fundamental concepts in programming languages, 1967. Notes for the International Summer School in Computer Programming, Copenhagen.

[91] Thomas Strahm. Partial applicative theories and explicit substitutions. *Journal of Logic and Computation*, 6(1):55–77, 1996.

[92] Thomas Studer. A semantics for $\lambda_{str}^{\{\}}$: a calculus with overloading and late-binding. To appear in *Journal of Logic and Computation*.

[93] Thomas Studer. Impredicative overloading in explicit mathematics. Submitted.

[94] Thomas Studer. Constructive Foundations for Featherweight Java. Submitted.

[95] Don Syme. Proving Java type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 83–118. Springer, 1999.

[96] Makoto Tatsuta. Realizability for constructive theory of functions and classes and its application to program synthesis. In *Proceedings of Thirteenth Annual IEEE Symposium on Logic in Computer Science, LICS '98*, pages 358–367, 1998.

[97] Anne Sjerp Troelstra. *Metamathematical Investigations of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics*. Springer, 1973.

[98] Anne Sjerp Troelstra and Dirk van Dalen. *Constructivism in Mathematics, vol I*. North Holland, 1988.

[99] Hideki Tsuiki. *A Record Calculus with a Merge Operator*. PhD thesis, Keio University, 1992.

[100] Hideki Tsuiki. A computationally adequate model for overloading via domain-valued functors. *Mathematical Structures in Computer Science*, 8:321–349, 1998.

[101] Raymond Turner. *Constructive Foundations for Functional Languages*. McGraw Hill, 1991.

[102] Raymond Turner. Weak theories of operations and types. *Journal of Logic and Computation*, 6(1):5–31, 1996.

Index