A semantics for $\lambda_{\mathbf{str}}^{\{\}}$: a calculus with overloading and late-binding

Thomas Studer

IAM-99-004

November 1999

A semantics for $\lambda_{str}^{\{\}}$: a calculus with overloading and late-binding

Thomas Studer *

Institut für Informatik und angewandte Mathematik, Universität Bern
Neubrückstrasse 10, CH–3012 Bern, Switzerland
tstuder@iam.unibe.ch

November 9, 1999

Abstract

Up to now there was no interpretation available for λ-calculi featuring overloading and late-binding, although these are two of the main principles of any object-oriented programming language. In this paper we provide a new semantics for a stratified version of Castagna's $\lambda^{\{\}}$, a λ-calculus combining overloading with late-binding. The model-construction is carried out in $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathsf{F\text{-}I_N})$, a system of explicit mathematics. We will prove the soundness of our model with respect to subtyping, type-checking and reductions. Furthermore, we show that our semantics yields a solution to the problem of loss of information in the context of type dependent computations.

Keywords: Explicit mathematics, typed λ-calculus, overloading, late-binding, loss of information.

1 Introduction

Polymorphism is one of the concepts to which the object-oriented paradigm owes its power. 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

*Research supported by the Swiss National Science Foundation.

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

"Ad hoc" polymorphism denotes the possibility that two objects of different classes can respond differently to the same message. Castagna, Ghelli and Longo [6] 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 behaviour 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 of 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 must possibly affect the final result of the computation. Selecting the branch to be executed of an overloaded function 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, which can affect the final result of a computation, if we base the selection of the branch of an overloaded function on the types at a given moment of the execution. 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.

Usually higher order lambda calculi are used 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 these calculi. Hence parametricity allows us to define functions that work on many different types, but always in the same way. Overloading on the other hand characterises 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 [15] who introduced his system F for a consistency proof of analysis. For this reason, system F is highly impredicative. Independently, Reynolds [22] rediscovered it later and used it for applications in programming languages. Feferman [11] 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 explicit mathematics.

Until now there are only a few systems available featuring "ad hoc" polymorphism. Ghelli [14] first defined typed calculi with overloading and late-binding to model object-oriented programming. This approach was further studied in Castagna, Ghelli and Longo [7]. In our paper we will use $\lambda^{\{\}}$ presented in Castagna [4, 5]. 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 have only overloaded functions and consider ordinary functions as overloaded with only one branch defined.

Castagna, Ghelli and Longo [6] 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 the type system is stratified in order to avoid certain problems of impredicativity in the definition of the semantics. 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 $\mathsf{EETJ}+(\mathsf{Tot})+(\mathsf{F}\text{-}\mathsf{I_N})$, a predicative theory of explicit mathematics.

Systems of explicit mathematics have been introduced by Feferman [8, 9] as a framework for Bishop style constructive mathematics. More recently, Feferman's systems were used to develop 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 [11, 12, 13] and Jäger [16, 17]. In systems of explicit mathematics types are represented by names, and those are first order values. Hence they can be used in computations and, as we have seen above, this allows us to model overloading and late-binding. Furthermore, we will show that our interpretation yields a solution to the problem of loss of information in the context of type dependent computations.

This problem introduced in Cardelli [2] 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 subtype of T, then we can only derive that $(\lambda x.x)a$ has type T. The information that a was of type S, is lost, although this will not be the case in our model. At this point we have to mention that Castagna [4] is developing a second order calculus with overloading and late-binding in order to deal with the problem of loss of information and parametric polymorphism. Our work is also a first step towards a better understanding of that system and the integration of parametric and "ad hoc" polymorphism.

2 The $\lambda^{\{\}}$-calculus

In this section we introduce Castagna's $\lambda^{\{\}}$-calculus. This minimal system, implementing overloading and late-binding, has been first presented in [4, 5]. 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 (λ-abstraction) 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. Now the pretypes are inductively defined by:

1. Every atomic type is a pretype.
2. If $S_1, T_1, \ldots, S_n, T_n$ are pretypes, then the finite set $\{S_1 \to T_1, \ldots, S_n \to T_n\}$ is a pretype.

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.

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 *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, T_i are types for all $i, j \in I$,
2. $S_i \leq S_j$ implies $T_i \leq T_j$ for all $i, j \in I$ and
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 $\{S_j \mid S \leq S_j \wedge j \in I\}$.

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 condition (3) 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 an operator of abstraction and one of 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

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

where $S_j = \min_{i \in I}\{S_i \mid S \leq S_i\}$ holds.

Reduction: when we apply an overloaded function to an argument, the argument type selects the branch of the overloaded function which will be executed. This has to be formally expressed by the reduction rules of the system. Since the argument of an application may be an open term, reduction will depend on a typing context Γ. With induction on the term structure we simultaneously define the *notion of reduction* and the terms in *normal form*.

1. We have the following *notion of reduction*:
   (ζ) Let $\Gamma \vdash \lambda x(M_i : S_i \Rightarrow T_i)_{i \in \{1,\ldots,n\}} : \{S_i \to T_i\}_{i \in \{1,\ldots,n\}}$ and $\Gamma \vdash N : S$, where $S_j = \min_{i \in I}\{S_i \mid S \leq S_i\}$. If N is in closed normal form with respect to Γ or $\{S_i \mid i \in I, S_i \leq S_j\} = \{S_j\}$, then
   $$\lambda x(M_i : S_i \Rightarrow T_i)_{i \in I} N \rhd_\Gamma M_j[x := N],$$
   where $M_j[x := N]$ denotes the substitution of x in M_j by N. Then there are rules for the context closure: let $\Gamma \vdash M : \{S_i \to T_i\}_{i \in I}$, $\Gamma \vdash N : S$ and if there exists an $i \in I$ with $S \leq S_i$, then
   $$\frac{M \rhd_\Gamma M'}{MN \rhd_\Gamma M'N} \qquad \frac{N \rhd_\Gamma N'}{MN \rhd_\Gamma MN'}$$
   Let $\Gamma \vdash \lambda x(M_i : S_i \Rightarrow T_i)_{i \in I} : \{S_i \to T_i\}_{i \in I}$, 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 $M \rhd_\Gamma N$.

In the sequel, we will consider only 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 $\mathbf{rank}_\lambda$ on the pretypes by:

1. $\mathbf{rank}_\lambda(A_i) = 0$,

2. $\mathbf{rank}_\lambda(\{S_i \to T_i\}_{i \in I}) = \max\{\mathbf{rank}_\lambda(S_i) + 1, \mathbf{rank}_\lambda(T_i) \mid i \in I\}$.

With this function we define a stricter subtyping relation $\leq^-$ by adding the condition $\mathbf{rank}_\lambda(\{S_j \to T_j\}_{j \in J}) \leq \mathbf{rank}_\lambda(\{U_i \to V_i\}_{i \in I})$ to the subtyping rule. Now, $\lambda^{\{\}}_{str}$ 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 $i, j \in I$.

3 Semantics

According to Castagna [5] 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 $\{S \to T\} \leq \{S \to T, S' \to T\}$ as well as $\{S \to T, S' \to T\} \leq \{S \to T\}$. 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, equally all the interpretations of subtypes of the argument types have to be regarded.

- *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, meaning in the translation of the terms. To determine the value of an overloaded application, first the interpretation of its argument needs to be evaluated in order to know its run-time type.

- *Impredicativity*: the type system of $\lambda^{\{\}}$ is not stratified. This can be seen in the following example: we know that $\{\{T \to T\} \to T, T \to T\}$ is a subtype of $\{T \to T\}$, but $\{T \to T\}$ is a strict occurrence of $\{\{T \to T\} \to T, T \to T\}$. Hence a term M of the type $\{\{T \to T\} \to T, T \to T\}$ may be applied to any term which is of a subtype of $\{T \to T\}$; and therefore M may be applied to itself, as M is of a subtype of $\{T \to T\}$. 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.

In Castagna [4] and Castagna, Ghelli and Longo [6] the calculus $\lambda\&-$early is introduced. It is a λ-calculus with overloading and subtyping, but without late-binding.

For stratified subsystems of this calculus a category-theoretical semantics 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 $\{S \to T, S_1 \to T, S_2 \to T, \ldots\}$, where $S_1, S_2, \ldots$ are all (infinitely many) subtypes of S.
The problem of type depended 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 to define the interpretation of an overloaded type as 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. Since only stratified subsystems of $\lambda\& -$early are modeled, there is no problem of impredicativity, either. 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 defined by induction on the type structure.

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. Such systems have first been presented by Feferman in [8] and [9]. Systems of explicit mathematics deal with functions and classes where functions are given by rules for mechanical computing and classes or types are successively defined or generated from previous ones. 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. In this sense, computing with types is possible in such systems and therefore they are an adequate framework to deal with overloading and late-binding.

In our model 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 *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 [4] indicates that the key to model late-binding probably consists of interpreting terms as pairs (interpretation of the computation, symbol for the type). Then the computational part of the interpretation $[\![MN]\!]$ of an application would be something of the form

$$(\Lambda X.(\mathbf{p_0}[\![M]\!])[\![X]\!])(\mathbf{p_1}[\![N]\!])(\mathbf{p_0}[\![N]\!]). \tag{1}$$

  This remark is the starting point for our construction. We show that interpreting terms as pairs really 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 (1) can be expressed in theories for 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 (1) and we can directly employ the symbol $\mathbf{p_1}[\![N]\!]$ for the type of the argument to select the best matching branch.

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

- An overloaded function type is interpreted as the class 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

$$\mathbf{typap}(\{S_i \to T_i\}_{i\in I}^*, S^*) = \langle S_j^*, T_j^*\rangle.$$

  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:

$$f(\llbracket N \rrbracket) = \begin{cases} \llbracket M_1[x := N] \rrbracket & \text{if } \textsf{\textbf{typap}}(\textsf{\textbf{p}}_\textbf{1}\llbracket M \rrbracket, \textsf{\textbf{p}}_\textbf{1}\llbracket N \rrbracket) = \langle S_1^*, T_1^* \rangle, \\ \llbracket M_2[x := N] \rrbracket & \text{if } \textsf{\textbf{typap}}(\textsf{\textbf{p}}_\textbf{1}\llbracket M \rrbracket, \textsf{\textbf{p}}_\textbf{1}\llbracket N \rrbracket) = \langle S_2^*, T_2^* \rangle. \end{cases}$$

By means of the remark about late-binding, we know that $\textsf{\textbf{p}}_\textbf{1}\llbracket M \rrbracket$ and $\textsf{\textbf{p}}_\textbf{1}\llbracket N \rrbracket$ are symbols for the types of M, N respectively. This demonstrates how types will affect the result of computations.

- We are not able to deal with the problem of impredicativity. For that reason we consider only the stratified version $\lambda^{\{\}}_{str}$ of $\lambda^{\{\}}$. The stratification of the type systems allows us to define the semantics of the types by induction on the type structure.

4 The theory $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathsf{F\text{-}I_N})$

In this section we present the theory $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathsf{F\text{-}I_N})$ of explicit mathematics with elementary comprehension and join as basic type existence axioms. Furthermore, in this system, term application is total and we have full induction on the natural numbers.

We will not employ Feferman's original formalization of $\mathsf{EETJ}+(\mathsf{Tot})+(\mathsf{F\text{-}I_N})$; but we will treat it as a theory of types and names as developed in Jäger [16]. The language $\mathcal{L}_t$ is two-sorted with *individual variables* $a, b, c, f, g, m, x, y, z, \ldots$ and *type variables* $A, B, C, X, Y, Z, \ldots$. Additionally, $\mathcal{L}_t$ includes the following *individual constants*: $\textsf{\textbf{k}}, \textsf{\textbf{s}}$ (combinators), $\textsf{\textbf{p}}, \textsf{\textbf{p}}_\textbf{0}, \textsf{\textbf{p}}_\textbf{1}$ (pairing), 0 (zero), $\textsf{\textbf{s}}_{\boldsymbol{N}}$ (successor), $\textsf{\textbf{p}}_{\boldsymbol{N}}$ (predecessor), $\textsf{\textbf{d}}_{\boldsymbol{N}}$ (definition by cases on natural numbers), for every natural number e a constant $\textsf{\textbf{c}}_e$ (elementary comprehension) and $\textsf{\textbf{j}}$ (join). $\mathcal{L}_t$ has the binary function symbol $\cdot$ for term application. In $\mathcal{L}_t$ we have the unary relation symbol N (natural number) as well as binary relation symbols $=, \in$ and $\Re$ (naming).

The *individual terms* $r, s, t, \ldots$ of $\mathcal{L}_t$ are inductively defined by closing individual variables and constants against application. We will drop $\cdot$ and only write (st) or st instead of $(s \cdot t)$ and we usually omit parenthesis with the implicit assumption that $\cdot$ associates to the left, i.e. rst stands for $((r \cdot s) \cdot t)$. We use (t_0, t_1) for $\textsf{\textbf{p}}t_0t_1$. The atomic formulas of $\mathcal{L}_t$ are the formulas $N(s)$, $(s = t)$, $(s \in A)$, $(A = B)$ and $\Re(s, A)$, where $N(s)$ says that s is a natural number and the formula $\Re(s, A)$ is used to express that the individual s *represents* the type A or is a *name* of A. The *formulas* $F, G, H, \ldots$ of $\mathcal{L}_t$ are generated from the atomic formulas by closing under negations, disjunctions, conjunctions and quantification in both sorts. A formula F of $\mathcal{L}_t$ is called *elementary*, if the relation symbol $\Re$ does not occur in F and F does not contain bound type variables. The following table contains a useful list of abbreviations:

$$
\begin{array}{rcl}
t \in N & :\equiv & N(t), \\
\exists x \in A.F & :\equiv & \exists x.(x \in A \wedge F), \\
\forall x \in A.F & :\equiv & \forall x.(x \in A \rightarrow F), \\
(f : A \rightarrow B) & :\equiv & \forall x \in A. fx \in B, \\
A \subset B & :\equiv & \forall x \in A. x \in B, \\
s \dot{\in} t & :\equiv & \exists Y.(\Re(t, Y) \wedge s \in Y), \\
s \dot{\subset} t & :\equiv & \exists X, Y.(\Re(s, X) \wedge \Re(t, Y) \wedge X \subset Y), \\
\Re(s) & :\equiv & \exists X. \Re(s, X).
\end{array}
$$

Now we are ready to state the axioms of the theory $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathsf{F\text{-}I_N})$. The underlying logic is classical first order logic with equality. Hence, the remaining logical connectives are defined as usual. The non-logical axioms can be divided into the following four groups.

I. Applicative axioms. These axioms formalize that the individuals form a combinatory algebra, that we have pairing and projection and the usual closure conditions on the natural numbers as well as definition by numerical cases. This first order part corresponds to the theory TON of Jäger and Strahm [19].

Combinatory algebra

(1) $\mathbf{k}xy = x$

(2) $\mathbf{s}xyz = xz(yz)$

Pairing and projection

(3) $\mathbf{p_0}(x, y) = x \wedge \mathbf{p_1}(x, y) = y$

Natural Numbers

(4) $0 \in N \wedge \forall x \in N. \mathbf{s_N} x \in N$

(5) $\forall x \in N.(\mathbf{s_N} x \neq 0 \wedge \mathbf{p_N}(\mathbf{s_N} x) = x)$

(6) $\forall x \in N.(x \neq 0 \rightarrow \mathbf{p_N}(\mathbf{s_N} x) \in N \wedge \mathbf{s_N}(\mathbf{p_N} x) = x)$

Definition by cases on natural numbers

(7) $u \in N \wedge v \in N \wedge u = v \rightarrow \mathbf{d_N} xyuv = x$

(8) $u \in N \wedge v \in N \wedge u \neq v \rightarrow \mathbf{d_N} xyuv = y$

It is standard work in combinatory logic that with the axioms (1) and (2) lambda abstraction can be defined and a recursion theorem can be proven (cf. e.g. Barendregt [1], Feferman [8] or Jäger [18]).

Definition 1 *We define λ abstraction by:*

$$
\begin{array}{rcll}
\lambda x.x & :\equiv & \mathsf{skk}, & \\
\lambda x.t & :\equiv & \mathsf{k}t, & \textit{if } x \notin \mathsf{FV}(t), \\
\lambda x.(rs) & :\equiv & \mathsf{s}(\lambda x.r)(\lambda x.s), & \textit{otherwise.}
\end{array}
$$

This definition of λ-abstraction is compatible with substitution, but the totality of the application is needed to make it work. In a partial setting a more complex definition of λ abstraction would be required and it would behave very badly as far as substitution in λ expressions is concerned (cf. Strahm [23]).

Theorem 2 *(Recursion theorem) There is a closed term* rec *of* $\mathcal{L}_t$ *such that:*

$$\forall f.\mathsf{rec} f = f(\mathsf{rec} f).$$

II. Explicit representation and extensionality. The relation $\Re$ acts as a naming relation between objects and types, i.e. $\Re(s, A)$ says that s is a name of the type A. While the representation of types by their names is intensional, the types themselves are extensional in the usual set-theoretical sense.

Extensionality

(EXT) $\quad \forall x.(x \in A \leftrightarrow x \in B) \to A = B$

The axioms about explicit representation state that every type has a name (E.1) and that there are no homonyms (E.2).

Explicit representation

(E.1) $\quad \exists x.\Re(x, A)$

(E.2) $\quad \Re(a, B) \wedge \Re(a, C) \to B = C$

III. Basic type existence axioms. In order to build types, there exists the principle of elementary comprehension. Let $F[x, \vec{y}, \vec{Z}]$ be an elementary formula of $\mathcal{L}_t$ with Gödelnumber e for any fixed Gödelnumbering, then we have the following axioms:

Elementary comprehension

(ECA.1) $\quad \exists X.\forall x.(x \in X \leftrightarrow F[x, \vec{a}, \vec{B}])$

(ECA.2) $\quad \Re(\vec{b}, \vec{B}) \wedge \forall x.(x \in A \leftrightarrow F[x, \vec{a}, \vec{B}]) \to \Re(\mathsf{c}_e(\vec{a}, \vec{b}), A)$

Besides elementary comprehension, we can also make use of the type building axiom for join. Let us write $A = \Sigma(B, f)$ for the statement

$$\forall x.(x \in A \leftrightarrow x = (\mathbf{p_0}x, \mathbf{p_1}x) \wedge \mathbf{p_0}x \in B \wedge \exists X.(\Re(f(\mathbf{p_0}x), X) \wedge \mathbf{p_1}x \in X)),$$

i.e. A is the disjoint sum over all $x \in B$ of the types named by fx. Now the (uniform) axiom of join has the form

Join

(J) $\Re(a, A) \wedge \forall x \in A.\exists Y.\Re(fx, Y) \rightarrow \exists Z.(\Re(\mathbf{j}(a, f), Z) \wedge Z = \Sigma(A, f))$.

IV. Formula induction on the natural numbers. Our theory enjoys full induction on the natural numbers:

Formula induction on N

($\mathsf{F\text{-}I_N}$) $F(0) \wedge \forall x \in N.(F(x) \rightarrow F(\mathbf{s_N}x)) \rightarrow \forall x \in N.F(x)$,

where F is an arbitrary formula of $\mathcal{L}_t$.

This induction principle allows us to represent every primitive recursive function and relation as a closed term of $\mathcal{L}_t$. 1 stands for the term $\mathbf{s_N}0$ and we let $<$ denote the usual "less than" relation on the natural numbers. We will need to code finite sequences of natural numbers. Let $\langle x_1, \ldots, x_n \rangle$ be the natural number which codes the sequence $x_1, \ldots, x_n$ in any fixed coding. $\langle\rangle$ is the empty sequence. There exists a projection function $\boldsymbol{\pi}$ so that $\boldsymbol{\pi}i\langle x_1, \ldots, x_i, \ldots, x_n \rangle = x_i$ for all natural numbers $x_1, \ldots, x_i, \ldots, x_n$. We suppose that our coding satisfies the following property: if $a'_i < a_i$, then $\langle a_1, \ldots, a'_i, \ldots, a_n \rangle < \langle a_1, \ldots, a_i, \ldots, a_n \rangle$ holds.

Since the totality of the application adds nothing to the proof-theoretic strength of our system, $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathsf{F\text{-}I_N})$ is proof-theoretically equivalent to Martin-Löf's type theory with one universe $\mathsf{ML_1}$ and $\widehat{\mathsf{ID}}_1$, the theory of non-iterated positive arithmetical inductive definitions where only the fixed-point property is asserted (cf. Feferman [10]). There are simple models of $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathsf{F\text{-}I_N})$. The applicative part can be interpreted in a standard way by a formalized total term model of TON (cf. Jäger and Strahm [19]). For the basic type existence axioms, codes for classes can be inductively generated and simultaneously a membership relation can be generated to satisfy elementary comprehension and join (actually one generates the element relation and its complement simultaneously). Only the fixed point property of this inductive definition is needed to establish a model, whereas minimality is not necessary (cf. Marzetta [21]). Formalizing this procedure in $\widehat{\mathsf{ID}}_1$ yields a model for $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathsf{F\text{-}I_N})$. More on inductive model constructions for systems of explicit mathematics can be found in Jäger and Studer [20].

5 Embedding $\lambda^{\{\}}_{str}$ into explicit mathematics

In this section we are going to carry out the embedding of $\lambda^{\{\}}_{str}$ into the theory $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathsf{F\text{-}I_N})$ of explicit mathematics. First, we represent each pretype T of $\lambda^{\{\}}_{str}$ by a natural number T^*, which will be called *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 can define terms **ptyp**, **sub** and **sub**$^-$ as 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 class **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 classes **type**(S^*) for strict subtypes S of T.
The interpretation of a $\lambda^{\{\}}_{str}$ term M is a pair in $\mathcal{L}_t$ 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 introduce a translation * from pretypes of $\lambda^{\{\}}_{str}$ to $\mathcal{L}_t$ 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 $A_i^* := \langle 0, i\rangle$ and

$$\{S_1 \to T_1, \dots, S_n \to T_n\}^* := \langle 1, \langle S_1^*, T_1^*\rangle, \dots, \langle S_n^*, T_n^*\rangle\rangle.$$

There exists a closed individual term **rank** of $\mathcal{L}_t$, so that if T is a pretype of $\lambda^{\{\}}_{str}$, then

$$\mathsf{EETJ} + (\mathsf{Tot}) + (\mathsf{F\text{-}I_N}) \vdash \mathbf{rank}(T^*) = n \quad\Longleftrightarrow\quad \mathbf{rank}_{\boldsymbol{\lambda}}(T) = n.$$

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. $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathsf{F\text{-}I_N}) \vdash \forall x, y \in N.(\mathbf{asub}(x, y) = 0 \vee \mathbf{asub}(x, y) = 1)$,
2. If S, T are pretypes of $\lambda^{\{\}}_{str}$, then $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathsf{F\text{-}I_N}) \vdash \mathbf{asub}(S^*, T^*) = 1$ if and only if $S \leq T$ and S, T are atomic,
3. $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathsf{F\text{-}I_N}) \vdash \forall x, y, z \in N.(\mathbf{asub}(x, y) = 1 \wedge \mathbf{asub}(y, z) = 1 \rightarrow \mathbf{asub}(x, z) = 1)$.

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

Using the terms $\mathbf{asub}, \mathbf{ptyp}$ and $\mathbf{rank}$ we define by primitive recursion two closed individual terms $\mathbf{sub}$ and $\mathbf{sub}^-$, so that these terms properly represent the subtype relations $\leq$ and $\leq^-$, respectively, on the symbols for pretypes.

Lemma 3 *Let S, T be pretypes of $\lambda^{\{\}}_{str}$.*

1. $\forall x, y \in N.(\mathbf{sub}(x, y) = 0 \vee \mathbf{sub}(x, y) = 1)$,
2. $\forall x, y \in N.(\mathbf{sub}^-(x, y) = 0 \vee \mathbf{sub}^-(x, y) = 1)$,
3. $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathsf{F\text{-}I_N}) \vdash \mathbf{sub}(S^*, T^*) = 1$ *if and only if $S \leq T$ is derivable in $\lambda^{\{\}}_{str}$,*
4. $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathsf{F\text{-}I_N}) \vdash \mathbf{sub}^-(S^*, T^*) = 1$ *if and only if $S \leq^- T$ is derivable in $\lambda^{\{\}}_{str}$.*

There is an elementary $\mathcal{L}_t$ 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 class $\mathbf{OTS}$ consisting of all symbols for $\lambda^{\{\}}_{str}$ types. Since elementary comprehension is uniform, there are closed individual $\mathcal{L}_t$ terms $\mathbf{domain}$ and $\mathbf{range}$ satisfying the following property: assume $\{S_i \rightarrow T_i\}_{i \in I}$ is an overloaded function type of $\lambda^{\{\}}_{str}$ and a is its symbol. Then $\mathbf{domain}(a)$ is a name for the class containing all symbols $x \in \mathbf{OTS}$ for which there is an $i \in I$ such that $\mathbf{sub}^-(x, S_i^*) = 1$ holds. The term $\mathbf{range}(a)$ represents the class consisting of the symbols $x \in \mathbf{OTS}$ for types $V_i \leq^- T_i$ for $i \in I$. That is $x \,\dot{\in}\, \mathbf{domain}(a)$ denotes a strict subtype of an S_i and $x \,\dot{\in}\, \mathbf{range}(a)$ is a symbol for a strict subtype of a T_i for $i \in I$. Using the $\mathbf{rank}$ function, we find a primitive recursive well-ordering $\prec$ on the type symbols so that the next lemma holds.

Lemma 4 $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathsf{F\text{-}I_N})$ *proves:*

1. $a \in \mathbf{OTS} \wedge b \,\dot{\in}\, \mathbf{domain}(a) \rightarrow b \prec a$,

2. $a \in \textsf{\textbf{OTS}} \wedge b \,\dot{\in}\, \textsf{\textbf{range}}(a) \rightarrow b \prec a$,

3. $\forall x \in \textsf{\textbf{OTS}}.(\forall y \in \textsf{\textbf{OTS}}.(y \prec x \rightarrow F(y)) \rightarrow F(x)) \rightarrow \forall x \in \textsf{\textbf{OTS}}.F(x)$, *for arbitrary formulas F of $\mathcal{L}_t$.*

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

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

if $S_j = \min_{i\in I}\{S_i \,|\, S \leq^- S_i\}$. We set $\textsf{\textbf{typap}}(\{S_i \rightarrow T_i\}^*_{i\in I}, S^*) = 0$, if such an S_j does not exist. Hence, $\textsf{\textbf{typap}}(\{S_i \rightarrow T_i\}^*_{i\in I}, S^*) = \langle S^*_j, T^*_j\rangle$ means, that if a $\lambda^{\{\}}_{str}$ term M of type $\{S_i \rightarrow T_i\}_{i\in I}$ 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 $\textsf{\textbf{t}}_{\textsf{\textbf{G}}}$ 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 $\textsf{\textbf{t}}_{\textsf{\textbf{G}}} := \lambda x.t$. If there are two symbols a, b with $\boldsymbol{\pi}1a = 0$ and $\boldsymbol{\pi}1b = 0$ and $\textsf{\textbf{sub}}(a, b) = 1$ (e.g. symbols for atomic types S, T of $\lambda^{\{\}}_{str}$ with $S \leq T$), then $\textsf{\textbf{t}}_{\textsf{\textbf{G}}}$ has to satisfy $\textsf{\textbf{t}}_{\textsf{\textbf{G}}}(a) \,\dot{\subset}\, \textsf{\textbf{t}}_{\textsf{\textbf{G}}}(b)$. This means that $\textsf{\textbf{t}}_{\textsf{\textbf{G}}}$ has to respect the subtype hierarchy on the type symbols given by $\textsf{\textbf{sub}}$. With reference to the recursion theorem we define a closed individual term $\textsf{\textbf{type}}$ of $\mathcal{L}_t$ satisfying

$$\textsf{\textbf{type}}\ m = \begin{cases} \textsf{\textbf{t}}_{\textsf{\textbf{G}}}\ m, & \text{if } \boldsymbol{\pi}1m = 0, \\ \textsf{\textbf{t}}_{\textsf{\textbf{O}}}\ \textsf{\textbf{type}}\ m, & \text{if } \boldsymbol{\pi}1m = 1, \end{cases}$$

where $\textsf{\textbf{t}}_{\textsf{\textbf{O}}}\ \textsf{\textbf{type}}\ m$ is a name for

$$\begin{aligned}\{f \mid \forall a \,\dot{\in}\, \textsf{\textbf{domain}}(m).\forall x \,\dot{\in}\, \textsf{\textbf{type}}(a).\\ &(\textsf{\textbf{p}}_{\textsf{\textbf{1}}}(f(x, a)) \,\dot{\in}\, \textsf{\textbf{range}}(m) \wedge \textsf{\textbf{p}}_{\textsf{\textbf{0}}}(f(x, a)) \,\dot{\in}\, \textsf{\textbf{p}}_{\textsf{\textbf{1}}}(f(x, a)) \wedge \\ &\textsf{\textbf{sub}}^-(\textsf{\textbf{p}}_{\textsf{\textbf{1}}}(f(x, a)), \boldsymbol{\pi}2(\textsf{\textbf{typap}}(m, a))) = 1)\}.\end{aligned}$$

Hence, if A is an atomic type, then $\textsf{\textbf{type}}(A^*)$ is a name for its corresponding type defined by $\textsf{\textbf{t}}_{\textsf{\textbf{G}}}$. Otherwise, if we are given an overloaded type $\{S_i \rightarrow T_i\}_{i\in I}$, then $\textsf{\textbf{type}}(\{S_i \rightarrow T_i\}^*_{i\in I})$ contains all functions f to such an extent that for every type S of $\lambda^{\{\}}_{str}$, every $i \in I$ and every term x of $\mathcal{L}_t$ with $S \leq^- S_i$ and $x \,\dot{\in}\, \textsf{\textbf{type}}(S^*)$ we obtain $\textsf{\textbf{p}}_{\textsf{\textbf{0}}}(f(x, S^*)) \,\dot{\in}\, \textsf{\textbf{p}}_{\textsf{\textbf{1}}}(f(x, S^*))$ and $\textsf{\textbf{p}}_{\textsf{\textbf{1}}}(f(x, S^*))$ denotes a strict subtype of T_j, where $T^*_j = \boldsymbol{\pi}2(\textsf{\textbf{typap}}(\{S_i \rightarrow T_i\}^*_{i\in I}, S^*))$, i.e. $S_j = \min_{i\in I}\{S_i \mid S \leq^- S_i\}$.

In this definition of $\textsf{\textbf{type}}$, there is a kind of type completion built in. Assume m is a symbol for an overloaded function type $\{S \rightarrow T\}$ and $f \,\dot{\in}\, \textsf{\textbf{type}}(m)$. Then $f(x, a)$ is defined for all $a \dot{\in} \textsf{\textbf{domain}}(m)$ and for all $x \dot{\in} \textsf{\textbf{type}}(a)$. Since $\textsf{\textbf{domain}}(m)$ contains all symbols $S_1^*, S_2^*, \ldots$ for strict subtypes of S, the term $\textsf{\textbf{type}}(m)$ represents in a sense 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 preorder on the types.

Using Lemma 4 we can prove that for every type symbol $m \in \textsf{\textbf{OTS}}$ the term $\textsf{\textbf{type}}(m)$ is a name for a type in explicit mathematics, meaning $\forall m \in \textsf{\textbf{OTS}}.\Re(\textsf{\textbf{type}}(m))$. These types satisfy the following subtype property.

Lemma 5 $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathsf{F\text{-}I_N})$ *proves:*

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

Now, we define a closed individual term $\textsf{\textbf{type}}_\mathbf{2}$ of $\mathcal{L}_t$, so that for all $a \in \textsf{\textbf{OTS}}$ we have $\Re(\textsf{\textbf{type}}_\mathbf{2}(a))$ and

$$(f, m) \dot{\in} \textsf{\textbf{type}}_\mathbf{2}(a) \leftrightarrow m \in \textsf{\textbf{OTS}} \wedge \textsf{\textbf{sub}}^-(m, a) = 1 \wedge f \dot{\in} \textsf{\textbf{type}}(m).$$

The following lemma about subtyping is just a corollary of the definition of the $\mathcal{L}_t$ term $\textsf{\textbf{type}}_\mathbf{2}$.

Lemma 6 $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathsf{F\text{-}I_N})$ *proves:*

$$a, b \in \textsf{\textbf{OTS}} \wedge \textsf{\textbf{sub}}^-(a, b) = 1 \to \textsf{\textbf{type}}_\mathbf{2}(a) \dot{\subset} \textsf{\textbf{type}}_\mathbf{2}(b).$$

With the $\mathcal{L}_t$ term $\textsf{\textbf{type}}_\mathbf{2}$ we define the interpretation of $\lambda^{\{\}}_{str}$ types as follows.

Definition 7 Interpretation $[\![T]\!]$ of a $\lambda^{\{\}}_{str}$ type T
If T is a type of $\lambda^{\{\}}_{str}$, then $[\![T]\!]$ is the type represented by $\textsf{\textbf{type}}_\mathbf{2}(T^*)$.

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

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

Terms of $\lambda^{\{\}}_{str}$ will be interpreted as ordered pairs, where the first component models the computational aspect of the term and the second component is a symbol for the type 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 $\mathcal{L}_t$. 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 $\textsf{\textbf{typap}}(\{S_i \to T_i\}^*_{i \in I}, \mathbf{p_1} y) = \langle S_j^*, T_j^* \rangle$ 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 $\mathcal{L}_t$ term $\textsf{\textbf{typap}}$ available, which selects the best matching branch. An application of two $\mathcal{L}_t$ terms MN is simply modeled by applying the function $\mathsf{p_0}[\![M]\!]$ to $[\![N]\!]$.

Definition 9 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. *$M \equiv \lambda x.(M_i : S_i \Rightarrow T_i)_{i\in\{1,\dots,n\}}$: $[\![M]\!] := (f, \{S_i \to T_i\}^*_{i\in\{1,\dots,n\}})$, where f is defined as follows:*

$$f(y) := \begin{cases} (\lambda\hat{x}.[\![M_1]\!])y, & \textsf{\textbf{typap}}(\{S_i \to T_i\}^*_{i\in\{1,\dots,n\}}, \mathsf{p_1}y) = \langle S_1^*, T_1^*\rangle, \\ \vdots \\ (\lambda\hat{x}.[\![M_n]\!])y, & \textsf{\textbf{typap}}(\{S_i \to T_i\}^*_{i\in\{1,\dots,n\}}, \mathsf{p_1}y) = \langle S_n^*, T_n^*\rangle. \end{cases}$$

3. *$M \equiv M_1M_2$: $[\![M]\!]$ is defined as $\mathsf{p_0}[\![M_1]\!][\![M_2]\!]$.*

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

Lemma 10 *If M, N are terms of $\lambda^{\{\}}_{str}$ and $P[x := Q]$ denotes the substitution of x in P by Q for both $\lambda^{\{\}}_{str}$ terms and $\mathcal{L}_t$ terms, then*

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

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 11 *If $\Gamma \vdash M : T$ holds in $\lambda^{\{\}}_{str}$, then in $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathsf{F\text{-}I_N})$ 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 $f := \mathbf{p_0}[\![M]\!]$. T is of the form $\{S_i \to T_i\}_{i \in I}$. Therefore, we have to show $(f, T^*) \mathrel{\dot\in} \mathbf{type_2}(T^*)$. That is $f \mathrel{\dot\in} \mathbf{type}(T^*)$, i.e.

$$\begin{aligned} &\forall a \mathrel{\dot\in} \mathbf{domain}(T^*).\forall y \mathrel{\dot\in} \mathbf{type}(a). \\ &\quad (\mathbf{p_1}(f(y,a)) \mathrel{\dot\in} \mathbf{range}(T^*) \wedge \mathbf{p_0}(f(y,a)) \mathrel{\dot\in} \mathbf{p_1}(f(y,a)) \wedge \\ &\quad \mathbf{sub}^-(\mathbf{p_1}(f(y,a)), \boldsymbol{\pi}2(\mathbf{typap}(T^*,a))) = 1). \end{aligned} \tag{2}$$

Choose $a \mathrel{\dot\in} \mathbf{domain}(T^*)$, $y \mathrel{\dot\in} \mathbf{type}(a)$ and let the natural number j be such that $\mathbf{typap}(T^*, a) = \langle S_j^*, T_j^* \rangle$, then we obtain $f(y,a) = (\lambda \hat{x}.[\![M_j]\!])(y,a)$ 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 6 we infer $\mathbf{type_2}(V_j^*) \mathrel{\dot\subset} \mathbf{type_2}(T_j^*)$. Our choice of a, y and j yields $(y,a) \in [\![S_j]\!]$ and we conclude $f(y,a) \mathrel{\dot\in} \mathbf{type_2}(T_j^*)$. That is $\mathbf{p_0}(f(y,a)) \mathrel{\dot\in} \mathbf{type}(\mathbf{p_1}(f(y,a)))$ and $\mathbf{p_1}(f(y,a)) \in \mathbf{OTS}$ as well as $\mathbf{sub}^-(\mathbf{p_1}(f(y,a)), T_j^*) = 1$. Therefore, we conclude that (2) 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

$$[\![\Gamma]\!] \to [\![M_1]\!] \in [\![\{S_i \to T_i\}_{i \in I}]\!], \tag{3}$$

$$[\![\Gamma]\!] \to [\![M_2]\!] \in [\![S]\!]. \tag{4}$$

From (4) we infer $\mathbf{p_0}[\![M_2]\!] \mathrel{\dot\in} \mathbf{type}(\mathbf{p_1}[\![M_2]\!])$ as well as

$$\mathbf{sub}^-(\mathbf{p_1}[\![M_2]\!], S^*) = 1. \tag{5}$$

Let k be such that $\mathbf{typap}(T^*, \mathbf{p_1}[\![M_2]\!]) = \langle S_k^*, T_k^* \rangle$. Using (3) we get

$$\mathbf{p_0}(\mathbf{p_0}[\![M_1]\!][\![M_2]\!]) \mathrel{\dot\in} \mathbf{p_1}(\mathbf{p_0}[\![M_1]\!][\![M_2]\!])$$

and $\mathbf{sub}^-(\mathbf{p_1}(\mathbf{p_0}[\![M_1]\!][\![M_2]\!]), T_k^*) = 1$. From (5) and the consistency conditions on good type formation we obtain $\mathbf{sub}^-(T_k^*, T_j^*)$. Therefore, we conclude by Lemma 6 that $\mathbf{p_0}[\![M_1]\!][\![M_2]\!] \in [\![T_j]\!]$ 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 [6], 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 12 *If P, Q are well-typed $\lambda^{\{\}}_{str}$ terms and $P \rhd_\Gamma Q$, then the following is provable in* $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathsf{F}\text{-}\mathsf{I_N})$*:*

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

Proof By induction on $\rhd_\Gamma$. 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

$$\mathbf{typap}(\{S_i \rightarrow T_i\}^*_{i\in I}, \mathbf{p_1}[\![N]\!]) = \langle S^*_j, T^*_j\rangle.$$

Therefore, we conclude

$$[\![P]\!] = \mathbf{p_0}[\![M]\!][\![N]\!] = (\lambda\hat{x}.[\![M_j]\!])[\![N]\!] = [\![M_j]\!][\hat{x} := [\![N]\!]] = [\![M_j[x := P]]\!] = [\![Q]\!].$$

□

6 Loss of information

Castagna [4] 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 in Cardelli [2]. It can be described in the following way. We assume we are given a $\lambda^{\{\}}_{str}$ function $\lambda x(x : T \Rightarrow T)$ of type $\{T \rightarrow T\}$, 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. For this reason, 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 [3]. The identity function is no longer considered to take an argument of 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

$$\forall X \leq^- T.(X \rightarrow X).$$

Recalling Castagna's proposal how a semantics for late-binding might work, we note the second order quantifier in the expression (1). This shows the connection between late-binding and the problem of loss of information. In 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 the 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 $\mathsf{EETJ} + (\mathsf{Tot}) + (\mathsf{F\text{-}I_N})$ 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 depends only on the types of the arguments and 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. Castagna [4] presents a type system for late-bound overloading in which this can be expressed. Our work also is a step towards a better understanding of calculi combining parametric polymorphism and type dependent computations

7 Conclusion

Overloading, subtyping and late-binding are important features of many object-oriented programming languages. Still, there was no interpretation available for λ calculi including these principles. In this paper we presented a semantics for $\lambda_{str}^{\{\}}$, a calculus combining these three features. Castagna [4] proposed to interpret terms as pairs (computation, type-symbol) in order to handle late-binding, but he did not give an actual model. Our construction is based on this idea and provides an interpretation for overloading, subtyping and late-binding.

The model construction is carried out in a system of explicit mathematics. Types are represented by names in these theories and names are first order values. Hence, they can be used in computations, which is the key to model overloading and late-binding. This paper provides a first study on how this feature of explicit mathematics can be used to investigate principles of object-oriented programming.

Our work shows that theories of types and names are well suited to examine principles that involve computations with types such as overloading and late-binding. We propose to develop a system of explicit mathematics which directly supports overloading. This theory should be proof-theoretically weak, but should also have strong expressive power, cf. Feferman [12, 13]. As noticed in the previous section, there is a strong connection between loss of information and parametric polymorphism. Since we got 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 explore parametric polymorphism in the context of late-bound overloading.

Acknowledgements. We would like to thank Giorgio Ghelli, Gerhard Jäger and Thomas Strahm for many helpful comments on earlier versions of this paper.

References

[1] Hendrik Barendregt. *The Lambda Calculus.* North-Holland, revised edition, 1985.

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

[3] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[4] Giuseppe Castagna. *Object-Oriented Programming: A Unified Foundation.* Birkhäuser, 1997.

[5] Giuseppe Castagna. Unifying overloading and λ-abstraction: $\lambda^{\{\}}$. *Theoretical Computer Science*, 176:337–345, 1997.

[6] 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.

[7] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995.

[8] 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, Berlin, 1975.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] Gerhard Jäger. Applikative Theorien und explizite Mathematik. Technical Report IAM 97-001, Universität Bern, 1997.

[19] Gerhard Jäger and Thomas Strahm. Totality in applicative theories. *Annals of Pure and Applied Logic*, 74:105–120, 1995.

[20] Gerhard Jäger and Thomas Studer. Extending the system T_0 of explicit mathematics: the limit and Mahlo axioms. Submitted.

[21] Markus Marzetta. *Predicative Theories of Types and Names*. PhD thesis, Institut für Informatik und angewandte Mathematik, Universität Bern, 1994.

[22] 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.

[23] Thomas Strahm. Partial applicative theories and explicit substitutions. *Journal of Logic and Computation*, 6(1):55–77, 1996.