Explicit Mathematics: Power Types and Overloading

Thomas Studer *

Abstract

Systems of explicit mathematics provide an axiomatic framework to represent programs and to prove properties of them. We introduce such a system with a new form of power types using a *monotone* power type generator. These power types allow us to model impredicative overloading. This is a very general form of type dependent computation which occurs in the study of object-oriented programming languages. We also present a set-theoretic interpretation for monotone power types. Thus establishing the concistency our system of explicit mathematics.

Keywords: explicit mathematics, proof theory, power types, overloading.

1 Introduction

Systems of explicit mathematics, or theories of types and names, were introduced by Feferman [7, 9] in the seventies. Beyond its original aim to provide a basis for Bishop style constructive mathematics, the explicit framework gained considerable importance in proof theory and in the study of various forms of abstract computation. More recently, these systems have been employed 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 [10, 11, 12], Studer [29] and Turner [36, 37]. A discussion of different evaluation strategies, such as call-by-value and call-by-name, in the applicative setting of theories of types and names is given by Stärk [25, 26].

*This paper is a part of the author's dissertation thesis [29] which has been supported by the Swiss National Science Foundation.

Beeson [2] and Tatsuta [31] make use of realizability interpretations for systems of explicit mathematics to prove theorems about program extraction. Most recently, our paper [28] provides an interpretation of Featherweight Java in a theory of types and names.

Let us now give a short explanation of the two basic kinds of objects which are present in explicit mathematics. These are *operations* or *rules* and *classes* or *types*. The former may be thought as mechanical rules of computation, which can freely be applied to each other: self-application is meaningful, though not necessary total. The basic axioms concerning operations are those of a partial combinatory algebra, thus giving immediate rise to explicit definitions (λ abstraction) and a form of the recursion theorem. The standard interpretation of the operations is the domain of the partial recursive functions. See Jäger, Kahle and Strahm [21] for a survey on the applicative axioms of explicit mathematics. Classifications or types, on the other hand, are collections of operations and must be thought of as being generated successively from preceding ones. In contrast to the simple character of operations, types can have quite complicated defining properties. What is essential in the whole explicit mathematics approach is the fact that types are again represented by operations or, as we will call them in this case, *names*, see Jäger [19]. Thus each type U is named or represented by a name u. In general, a type U may have many different names or representations. It is exactly this interplay between operations and types on the level of names which makes explicit mathematics extremely powerful, and, in fact, witnesses its explicit character.

Weak power type axioms assert that for every type A, there exists a type P containing at least one name of every subtype of A. Feferman [9] introduced power type axioms in the context of explicit mathematics and asked the question of their proof-theoretic strength. A partial answer was provided by Glass [16] who showed that adding weak power types 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 powerful. However, Feferman's question about the proof-theoretic strength of systems of explicit mathematics with join and a weak power type axiom is still 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 [20] proved that this principle is inconsistent with uniform elementary comprehension, but his argument does not work if only separation is available. Cantini [4] presented a model for non-uniform comprehension and a strong power type axiom using a variant of Quine's New Foundations.

In addition to the usual weak power type axioms, 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 construction of Glass [16] does not respect our monotonicity axiom. It is this new monotonicity property that allows us to make use of power types to model impredicative overloaded functions.

Overloading is an important concept in object-oriented programming. For example, it occurs when a method is redefined in a subclass or when a class provides several methods with the same name but with different argument types. 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 [15] first defined typed calculi with overloading and late-binding to model object-oriented programming. This approach was further developed in Castagna, Ghelli and Longo [6] where they introduce $\lambda\&$, a calculus for overloaded functions with subtyping.

One of the problems in the construction of a semantics for calculi with overloading is impredicativity. Consider any term M of the type

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

This term M can be applied to arguments of type $\{S \to T\}$ as well as to arguments of the type S and yields a result of the type T. Hence, the term M also belongs to the type $\{S \to T\}$. This implies that the term M can be applied to itself, that is the type $\{\{S \to T\} \to T, S \to T\}$ is a part of its own domain. Therefore, the interpretation of that type refers to itself, whence the impredicativity. This has the consequence that the interpretation of the types cannot be given by induction on the type structure.

In his Ph.D. thesis, Tsuiki [34] 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.

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 [35] 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 says that it would be difficult to extend his model so that it could deal with such a subtype relation.

In this article we introduce the system OTN of explicit mathematics based on elementary separation, product, join and monotone weak power types. We present a set-theoretic model for OTN, and we develop in OTN a theory of *impredicative* overloading. This continues our work in [30] where we treated *predicative* overloading in explicit mathematics which was much simpler since we had only to deal with a subsystem of $\lambda\&$ without self-application. Hence we could do with a weaker system of explicit mathematics which did not feature power types. Moreover, in the construction in [30] we could not employ names directly to model type dependent computations but we had to introduce so-called *codes* for types which had to be mapped to names by a recursively defined function. The approach in the present paper is much more natural since now we work directly with names.

In order to model type dependent computations, it is essential that there are first-order values acting for types. In explicit mathematics, each type is represented by a name on the first-order level and hence, the types can participate in the computations via their names. Therefore theories of types and names naturally contain type dependent computations. Still we do not get an interpretation of $\lambda\&$ since the application operation in λ calculi, like $\lambda\&$, is total whereas in our system OTN we work with a partial application. Unfortunately, the definition of λ abstraction in partial applicative theories does not commute with substitutions, see Example 3 below or, for a detailed account on this topic, the paper by Strahm [27]. Therefore, we cannot prove that reduction in the sense of $\lambda\&$ preserves the interpretation of terms in OTN. However, a weaker version of the substitution principle still holds (see Lemma 4), which will suffice for most applications in computational practice.

2 Explicit Mathematics

In this section we introduce a system of explicit mathematics based on elementary separation, join, product and weak power types. We will not use Feferman's original formalization of explicit mathematics but treat it as a theory of types and names as developed by Jäger [19].

We work with a two-sorted language $\mathbb{L}$ 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).

Further, $\mathbb{L}$ 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* $(r, s, t, \ldots)$ of $\mathbb{L}$ and the individual terms are closed under the binary function symbol $\cdot$ for (partial) application.

In the following we often abbreviate $(s \cdot t)$ simply as (st), st or sometimes also as $s(t)$; the context will always assure that no confusion arises. We further adopt the convention of association to the left so that $s_1 s_2 \ldots s_n$ stands for $(\ldots(s_1 \cdot s_2)\ldots s_n)$. Finally, we define general n tupling by induction on $n \geq 2$ as follows: $(s_1, s_2) := \mathsf{p} s_1 s_2$ and $(s_1, \ldots, s_{n+1}) := ((s_1, \ldots, s_n), s_{n+1})$.

The *atomic formulas* of $\mathbb{L}$ are the formulas $s{\downarrow}$, $\mathsf{N}(s)$, $s = t$, $s \in U$, $U = V$ 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* of $\mathbb{L}$ $(\phi, \psi, \ldots)$ are generated from the atomic formulas by closing against the usual propositional connectives as well as quantification in both sorts. A formula ϕ of $\mathbb{L}$ is called *elementary* if it contains neither the relation symbol $\Re$ nor bound type variables. The following table contains a useful list of abbreviations:

$$\begin{aligned} s \simeq t &:= s{\downarrow} \vee t{\downarrow} \to s = t, \\ s \in \mathsf{N} &:= \mathsf{N}(s), \\ (\exists x \in S)\phi(x) &:= \exists x(x \in S \wedge \phi(x)), \\ (\forall x \in S)\phi(x) &:= \forall x(x \in S \to \phi(x)), \end{aligned}$$

$$
\begin{array}{rcl}
U \subset V & := & \forall x(x \in U \to x \in V), \\
(\forall X \subset \mathsf{N})\phi(X) & := & \forall X(X \subset \mathsf{N} \to \phi(X)), \\
f \in (S \to T) & := & (\forall x \in S)fx \in T, \\
s \mathrel{\dot\in} t & := & \exists X(\Re(t, X) \wedge s \in X), \\
(\exists x \mathrel{\dot\in} s)\phi(x) & := & \exists x(x \mathrel{\dot\in} s \wedge \phi(x)), \\
(\forall x \mathrel{\dot\in} s)\phi(x) & := & \forall x(x \mathrel{\dot\in} s \to \phi(x)), \\
s \doteq t & := & \exists X(\Re(s, X) \wedge \Re(t, X)), \\
s \mathrel{\dot\subset} t & := & \exists X \exists Y(\Re(s, X) \wedge \Re(t, Y) \wedge X \subset Y), \\
f \mathrel{\dot\in} (s \to t) & := & (\forall x \mathrel{\dot\in} s)fx \mathrel{\dot\in} t, \\
\Re(s) & := & \exists X \Re(s, X).
\end{array}
$$

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 then either given by the context or irrelevant. 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 OTN of overloading, types and names which will provide a framework for the study of impredicative overloading. It's logic is Beeson's classical *logic of partial terms* (cf. Beeson [1] or Troelstra and van Dalen [32]) for individuals and classical logic with equality for types. The non-logical axioms of OTN can be divided into the following five groups.

I. Applicative axioms. These axioms formalize that the individuals form 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, see Feferman and Jäger [13].

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

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

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

(4) $0 \in \mathsf{N} \wedge (\forall x \in \mathsf{N})(\mathsf{s}_\mathsf{N} x \in \mathsf{N})$,

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

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

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

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

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 equality of types is extensional.

(9) $\exists x \Re(x, U)$,

(10) $\Re(a, U) \wedge \Re(a, V) \to U = V$,

(11) $\forall x(x \in U \leftrightarrow x \in V) \to U = V$.

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

Natural numbers

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

Elementary separation. Let $\phi(x, \vec{y}, \vec{Z})$ be an elementary formula of $\mathbb{L}$ with at most the indicated free variables and with Gödelnumber e for any fixed Gödelnumbering, then we have the following axioms:

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

(14) $\Re(c, \vec{b}, S, \vec{T}) \to \forall x(x \mathrel{\dot\in} \mathsf{sep}_e(\vec{a}, c, \vec{b}) \leftrightarrow x \in S \wedge \phi(x, \vec{a}, \vec{T}))$.

Union

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

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

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

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

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

Product

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

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

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

The axioms (1) – (17) plus formula induction on the natural numbers correspond to Feferman's theory S_0 (introduced in [9]) without inductive generation. We are going to extend this system by axioms for weak power types and for the subset decidability operation 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.

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

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

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

(21) $a \,\dot{\subset}\, b \rightarrow \mathsf{pow}(a) \,\dot{\subset}\, \mathsf{pow}(b)$.

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

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

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

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

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

Formula induction on the natural numbers

($\mathbb{L}$-I_N) $\phi(0) \wedge (\forall x \in \mathsf{N})(\phi(x) \rightarrow \phi(\mathsf{s}_\mathsf{N}x)) \rightarrow (\forall x \in \mathsf{N})\phi(x)$

for all formulas ϕ of $\mathbb{L}$.

In the following we are going to discuss the axioms of OTN. There are two crucial principles following already from the axioms of a partial combinatory algebra, that is the axioms (1) and (2) of BON: λ abstraction and a recursion theorem, cf. e.g. Beeson [1] or Feferman [7].

Definition 1. We define the $\mathbb{L}$ 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 2 (λ abstraction). *For each $\mathbb{L}$ term t and all variables x there exists an $\mathbb{L}$ 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 has the drawback that it does not commute with substitutions.

Example 3. If x and y are distinct variables 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 $\mathsf{s}(\mathsf{k}z)(\mathsf{k}z)$.

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 4. *For all $\mathbb{L}$ terms s and t and different variables x and y of $\mathbb{L}$ we have*

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

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.

Theorem 5 (Recursion). *There exists a closed $\mathbb{L}$ term rec so that*

$$\mathsf{BON} \vdash \mathsf{rec}f\!\downarrow \wedge\ \mathsf{rec}fx \simeq f(\mathsf{rec}f)x.$$

Now we come to the type existence axioms. Usually, theories of explicit mathematics are based on elementary comprehension rather than separation. In this case, there is a universal type available which contains everything. Together with join, this universal type can be used to prove the product axiom. Since separation does not prove the existence of a universal type, we have to add an axiom for product types (see Feferman [9]).

Note that for elementary comprehension, there is a finite axiomatization available, see Feferman and Jäger [14]. Jäger and Studer [23] present an inductive model construction for this finite axiomatization which exhibits nicely the constructive character of theories of types and names. For elementary separation no such axiomatization is known in the literature.

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. Assume that there is a weak power type C of the universal type, so

$$\forall x(x \in C \to \Re(x)) \wedge \forall X \exists y(y \in C \wedge \Re(y, X)). \tag{1}$$

Let c be a name for C and let J be the join represented by $\mathsf{j}(c, \lambda x.x)$. Hence $(a, x) \in J \leftrightarrow x \mathrel{\dot\in} a$. Let $A := \{x \mid x \in C \wedge (x, x) \notin J\}$. By (1), this type has a name a so that $a \in C$. We get

$$a \mathrel{\dot\in} a \leftrightarrow a \in A \leftrightarrow a \in C \wedge (a, a) \notin J \leftrightarrow a \in C \wedge a \mathrel{\dot\notin} a \leftrightarrow a \mathrel{\dot\notin} a,$$

which gives a contradiction. Therefore we dispense with the universal type and include only elementary separation in our list of axioms.

Subset decidability on power types will turn out to be crucial for the development of overloaded functions in OTN. These axioms state that if we have two names a and b such that both of them are elements of a given power type, then the term sub decides whether the type represented by a is a subtype of the type represented by b. Moreover, axiom (24) states that if two names a and b are in the subtype relation with respect to sub, then a must be an element of the power type of b. That is the $\mathbb{L}$ term sub is compatible with pow. This subset decidability operator will be employed to model applications of overloaded functions. There, one has to select the best matching branch, which requires that the subtype relation is decidable. This will be the task of the term sub. This term may also be regarded as very special quantification operation since for elements $a, b \in \mathsf{pow}(c)$ 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 is dealt with in the first-order part. The proof theory of several quantification functionals in explicit mathematics has been studied in various papers by Feferman, Jäger, Glass, Strahm and Kahle [13, 14, 18, 22, 24].

3 A Set-Theoretic Model for $\mathsf{OTN} + (\mathbb{L}\text{-}\mathsf{I}_{\mathsf{N}})$

There is a general set-theoretic model construction for systems of explicit mathematics. We are going to extend it so that our axioms for weak power types and subset decidability will be satisfied. Starting with any model $\mathcal{M}$ of set theory we generate a model $\mathsf{Gen}(\mathcal{M})$ of the applicative axioms such that the natural numbers are interpreted by ω and every set-theoretic function F of $\mathcal{M}$ is represented in $\mathsf{Gen}(\mathcal{M})$. Over $\mathsf{Gen}(\mathcal{M})$, there is a natural model construction for systems of explicit mathematics. We can inductively generate codes for the types and simultaneously we can also create a membership relation satisfying the type existence axioms. This iterative process, where the interpretation of a type is given in terms of previously defined types, reflects the constructive content of explicit mathematics.

These standard set-theoretic model constructions for applicative theories and systems of explicit mathematics are carried out at various places, for instance in Feferman [7, 8, 9], Beeson [1], Glass, Rathjen and Schlüter [17] as well as Troelstra and van Dalen [33]. Since this procedure for generating models of explicit mathematics is already well described in the literature, we do not present the full details. Only the cases concerning power types and the subset decidability operation will be emphasized.

For simplicity, we will first give the construction for a model $\mathsf{Gen}(\mathcal{M})$ satisfying only the axioms of BON. Let $\mathcal{M}$ be any model of set theory; let P, P_0, P_1 be the pairing and projection functions of $\mathcal{M}$ and let 0 be the natural number 0 in $\mathcal{M}$. Choose codes $\underline{\mathsf{k}}, \underline{\mathsf{s}}, \underline{\mathsf{p}}, \underline{\mathsf{p}_0}, \underline{\mathsf{p}_1}, \underline{\mathsf{s}_{\mathsf{N}}}, \underline{\mathsf{p}_{\mathsf{N}}}, \underline{\mathsf{d}_{\mathsf{N}}}, \underline{\mathsf{k}}_x, \underline{\mathsf{s}}_x, \underline{\mathsf{s}}_{xy}, \underline{\mathsf{d}_{\mathsf{N}}}_a$, $\underline{\mathsf{d}_{\mathsf{N}}}_{ab}$, $\underline{\mathsf{d}_{\mathsf{N}}}_{abx}$, $\underline{\mathsf{p}}_x$ and $\underline{F}$ for every set-theoretic function F of $\mathcal{M}$ and all $a, b, x, y \in \mathcal{M}$. These codes are all distinct from the natural number 0 and from each other for all $a, b, x, y \in \mathcal{M}$. Then we take App to be the least ternary relation satisfying:

- $\mathsf{App}(\underline{\mathsf{k}}, x, \underline{\mathsf{k}}_x)$ and $\mathsf{App}(\underline{\mathsf{k}}_x, y, x)$,
- $\mathsf{App}(\underline{\mathsf{s}}, x, \underline{\mathsf{s}}_x)$, $\mathsf{App}(\underline{\mathsf{s}}_x, y, \underline{\mathsf{s}}_{xy})$ and if $\mathsf{App}(x, z, w)$, $\mathsf{App}(y, z, v)$ as well as $\mathsf{App}(w, v, u)$ hold, then also $\mathsf{App}(\underline{\mathsf{s}}_{xy}, z, u)$,
- $\mathsf{App}(\underline{\mathsf{p}}, x, \underline{\mathsf{p}}_x)$, $\mathsf{App}(\underline{\mathsf{p}}_x, y, P(x, y))$, $\mathsf{App}(\underline{\mathsf{p}_0}, x, P_0(x))$, $\mathsf{App}(\underline{\mathsf{p}_1}, x, P_1(x))$,
- $\mathsf{App}(\underline{\mathsf{s}_{\mathsf{N}}}, x, x+1)$ and $\mathsf{App}(\underline{\mathsf{p}_{\mathsf{N}}}, x+1, x)$ for $x \in \omega$,

- $\mathsf{App}(\underline{\mathsf{d}_{\mathsf{N}}}, a, \underline{\mathsf{d}_{\mathsf{N}}}_a)$, $\mathsf{App}(\underline{\mathsf{d}_{\mathsf{N}}}_a, b, \underline{\mathsf{d}_{\mathsf{N}}}_{ab})$, $\mathsf{App}(\underline{\mathsf{d}_{\mathsf{N}}}_{ab}, x, \underline{\mathsf{d}_{\mathsf{N}}}_{abx})$ and if $x = y$, then $\mathsf{App}(\underline{\mathsf{d}_{\mathsf{N}}}_{abx}, y, a)$ as well as if $x \neq y$, then $\mathsf{App}(\underline{\mathsf{d}_{\mathsf{N}}}_{abx}, y, b)$,
- $\mathsf{App}(\underline{F}, x, F(x))$ for each set-theoretic function F in $\mathcal{M}$.

That is 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, which are the natural numbers in $\mathcal{M}$. Additionally, $\mathsf{App}(x, y, z)$ is closed under a condition expressing that the code $\underline{F}$ represents the function F. The resulting structure $\mathsf{Gen}(\mathcal{M})$ is an applicative model and $\mathsf{App}(x, y, z)$ can be employed to interpret $xy \simeq z$.

Now we will adapt this construction so that it will be compatible with the axioms for power types and subset decidability. First, we have to require that each code $\underline{F}$ is a set-theoretic pair whose first component is the natural number 0; otherwise we would have a conflict in the model with the type existence axioms. These codes $\underline{F}$ will possibly be names for types and we have to ensure that they are different from the names constructed by the type existence axioms. Moreover, 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 $\emptyset$. Then we need three more closure conditions for the relation App so that we will get a model for the subset decidability operation. These conditions are:

- 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 $y = \underline{F}$ 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 $y = \underline{G}$ as well as

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

  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 $y = \underline{G}$ as well as

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

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

This relation $\mathsf{App}(x, y, z)$ will interpret the partial application $xy \simeq z$. In this way the first order part of $\mathsf{OTN} + (\mathbb{L}\text{-}\mathsf{I}_{\mathsf{N}})$ is modeled by $\mathsf{Gen}(\mathcal{M})$.

Note that in applicative theories, sets are usually encoded as *total* functions from the natural numbers to $\{0,1\}$. In our definition of the relation App with the terms $\underline{\mathsf{sub}}_F$ we consider *partial* functions as sets. We will have that $x \in F$ if $F(x) = 1$ and $x \notin F$ if $F(x)$ is undefined. This is important for the construction of the model for the type structure. There we will interpret these partial functions as *names* for types. This is done in the sequel.

Feferman [9] 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 $\mathsf{OTN} + (\mathbb{L}\text{-}\mathsf{I_N})$. To build the set-theoretic model, one defines two relations

$$\mathsf{Cl} := \bigcup_\alpha \mathsf{Cl}_\alpha \text{ and } \in := \bigcup_\alpha \in_\alpha$$

by transfinite induction on the ordinal α. The predicate $\mathsf{Cl}(a)$ means that a is a name for a type in the model and the relation $x \in a$ states that in the model, x is an element of the type represented by a. We begin with assigning to each generator of $\mathbb{L}$ a code in $\mathsf{Gen}(\mathcal{M})$, e.g. such that:

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

These codes will be the names for the corresponding types in the model. They are chosen so that no conflicts arise with the representations $\underline{F}$ of set-theoretic functions F, i.e. none of these sequences starts with the natural number 0. 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 $\mathbb{L}$ are interpreted by taking $\in_\alpha$ for $\in$ and letting the names range over Cl_α. For example, at stage 0 we have

$$\mathsf{Cl}_0(\underline{\mathsf{nat}}) \text{ and } x \in_0 \underline{\mathsf{nat}} \text{ if } x \text{ is a natural number in } \mathcal{M}.$$

At later stages, we can define the interpretation of a type based on the interpretation of previously constructed types. The simplest case are union types. If we have $\mathsf{Cl}_\beta(a)$ and $\mathsf{Cl}_\beta(b)$ for some $\beta < \alpha$, then we get

$$\mathsf{Cl}_\alpha(\underline{\mathsf{un}}(a,b)) \text{ and } x \in_\alpha \underline{\mathsf{un}}(a,b) \text{ if } x \in_\beta a \vee x \in_\beta b.$$

Similarly, one can define the interpretation of elementary separation, join and product types. Details can be found in the above mentioned literature. The following two cases deal with the power type axioms.

- For each partial function F in $\mathcal{M}$ so that the range of F is a subset of $\{1\}$ we have $\mathsf{Cl}_0(\underline{F})$ and $x \in_0 \underline{F}$ if and only if $F(x) = 1$.

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

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 $\underline{F}$ represents the type of all x with $F(x) = 1$, that is $\underline{F}$ 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 $\mathsf{OTN} + (\mathbb{L}\text{-}\mathsf{I}_\mathsf{N})$ is interpreted over $\mathsf{Gen}(\mathcal{M})$. In order to give the semantics of the type structure of $\mathsf{OTN} + (\mathbb{L}\text{-}\mathsf{I}_\mathsf{N})$ we have defined 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 $\mathbb{L}$ 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 6. *For every formula ϕ of $\mathbb{L}$ we have*

$$\mathsf{OTN} + (\mathbb{L}\text{-}\mathsf{I}_\mathsf{N}) \vdash \phi \implies (\mathsf{Gen}(\mathcal{M}), \mathsf{Cl}, \in) \models \phi.$$

Proof. The only critical axioms are the power type axioms and the axioms for the subset decidability operation. All other axioms can be verified in a straightforward way, see Feferman's model for S_0 [9]. 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.

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

(19) 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 $y \in \mathsf{ext}(a)$.

(20) 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 $\underline{F} \in \mathsf{ext}(\underline{\mathsf{pow}}(a))$. Moreover, $\underline{F} \in \mathsf{Cl}$ holds and $z \in \mathsf{ext}(\underline{F})$ if and only if $\mathsf{App}(\underline{F}, z, 1)$. By the construction of F this is the case if and only if $z \in \mathsf{ext}(x)$.

(21) 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 operation. 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 the equivalences $x \in \mathsf{ext}(a) \leftrightarrow \mathsf{App}(a, x, 1)$ 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 that our model satisfies $\mathsf{sub}ab = 1$. In order to define App, one takes the least fixed point of the inductively defined application. This implies that we have two set-theoretic functions F and G whose ranges are a subsets of $\{1\}$. The terms a and b are then interpreted by the codes $\underline{F}$ and $\underline{G}$, respectively. We have $\mathsf{App}(\underline{\mathsf{sub}}, \underline{F}, \underline{\mathsf{sub}}_F)$ and $\mathsf{App}(\underline{\mathsf{sub}}_F, \underline{G}, 1)$. By the construction of the set-theroretic model for the type structure we get $\underline{G} \in \mathsf{Cl}$ with $\forall x(x \in \mathsf{ext}(\underline{G}) \leftrightarrow G(x) = 1)$. Moreover, we have $\forall x(F(x) = 1 \to G(x) = 1)$. Hence, the domain of F is a subset of $\mathsf{ext}(\underline{G})$ and therefore, we get $\underline{F} \in \underline{\mathsf{pow}}(\underline{G})$ which is the interpretation of $a \mathrel{\dot\in} \mathsf{pow}(b)$. □

4 Impredicative Overloading in OTN

In this section we are going to develop a theory of impredicative overloading and late-binding in OTN. This examines the relationship of power types and impredicative overloaded function types. It shows that the monotonicity axiom for power types is a key principle to extend the range of possible applications of power types.

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. So 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 [5] or Studer [30]).

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 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

$$\{f \mid (\forall x \in S_1^*) fx \in T_1^* \wedge \ldots \wedge (\forall x \in S_n^*) fx \in T_n^*\}. \tag{2}$$

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

The general intuition of this type construction is the following. The type S^* is the disjoint union of all subtypes of S. In order the build this disjoint union, we have to know all subtypes of a given type. Therefore, we need the power type axioms, which are the essential impredicative feature of OTN. The elements of S^* are pairs (a, x) so that a represents a subtype of S and x is an element of this subtype. Hence we also have $x \in S$ and a may be regarded as the run-time type of x. Hence, the formula

$$(\forall x \in S^*) fx \in T^* \tag{3}$$

states that the function f maps any subtype of S to some subtype of T so that it respects the run-time type information which is explicitly shown in the elements of S^* and T^*. If f is an overloaded function, then it has to satisfy several formulas of the form of (3). This is the meaning of (2).

Castagna, Ghelli and Longo [6] 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 in $\lambda\&$: 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 considered this type not as an overloaded function type, but as an intersection type, then the additional coherence condition would state 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 an essential feature of overloaded function types in $\lambda\&$. Our model of overloading in explicit mathematics reflects this relationship between intersection types and overloaded function types.

We obtain the following result about subtyping which corresponds to the subtyping rule of $\lambda\&$ in Castagna, Ghelli and Longo [6].

Theorem 7. *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 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.

$$S \subset T \;\to\; S^* \subset T^*. \tag{4}$$

Assume now $f \in \{U_j \to V_j\}_{j\in J}$ 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) we get $x \in U_j^*$ and therefore $fx \in V_j^*$. Again by (4) we conclude $fx \in T_i^*$. □

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 $s_1, \ldots, s_n \mathrel{\dot\in} \mathsf{pow}(s_1 \cup \cdots \cup s_n)$. 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 $\mathbb{L}$ 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 and otherwise $\mathsf{Min}^j_{s_1,\ldots,s_n}(s) = 0$.

Assume now 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 $f_1 \mathrel{\dot\in} (s_1^* \to t_1^*)$ and $f_2 \mathrel{\dot\in} (s_2^* \to t_2^*)$. Using definition by

cases on natural numbers we can combine f_1 and f_2 to an overloaded function f so that

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

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.

Castagna, Ghelli and Longo [6] introduce two consistency conditions concerning good type formation for overloaded function types, 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. An overloaded function type $\{S_i \to T_i\}_{i \in I}$ is called *well-formed* if and only if it satisfies the following 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 $S_k = S_i \cap S_j$.

The condition (2) is very much adapted to our explicit mathematics framework. In $\lambda\&$ it has to be formulated as

(2′) if there exists $i \in I$ and a (pre)type S so that $S \leq S_i$, then there exists a unique $z \in I$ such that S_z is a minimal element of $\{S_j \,|\, S \leq S_j \wedge j \in I\}$,

see for instance Studer [30]. This condition is much closer to "real" object-oriented programming. However, since we have arbitrary intersection types in OTN, the conditions (2) and (2′) are equivalent in OTN and we think that (2) corresponds more to the set-theoretic spirit of explicit mathematics.

With the notion of a well-formed overloaded function type, we can prove the following theorem about typing of overloaded functions.

Theorem 8. *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 $\mathbb{L}$ terms so that $f_1 \in (S_1^* \to T_1^*), \ldots, f_n \in (S_n^* \to T_n^*)$. Then 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 $s_1, \ldots, s_n \mathrel{\dot\in} \mathsf{pow}(s_1 \cup \cdots \cup s_n)$. Assume we are given an $x \in S_i^*$ for $1 \leq i \leq n$. Then $x = (\mathsf{p}_0 x, \mathsf{p}_1 x)$, $\mathsf{p}_1 x \mathrel{\dot\in} \mathsf{p}_0 x$ 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

$$\mathsf{Min}^j_{s_1,\ldots,s_n}(\mathsf{p}_0 x) = 1. \tag{5}$$

The uniqueness of j can be seen as follows: assume that $\mathsf{Min}^j_{s_1,\ldots,s_n}(\mathsf{p}_0 x) = 1$ as well as $\mathsf{Min}^k_{s_1,\ldots,s_n}(\mathsf{p}_0 x) = 1$ hold. Then we have $\mathsf{p}_0 x \mathrel{\dot\subset} s_j$ and $\mathsf{p}_0 x \mathrel{\dot\subset} s_k$. Since $\mathsf{p}_1 x \mathrel{\dot\in} \mathsf{p}_0 x$ holds we know that $s_j \cap s_k$ is nonempty. 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 (5) holds. Therefore we get $\mathsf{over}_{S_1,\ldots,S_n}(f_1, \ldots, f_n)x = f_j x$. By (5) 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

$$\mathsf{over}_{S_1,\ldots,S_n}(f_1, \ldots, f_n)x \in T_j^*. \tag{6}$$

From $x \in S_i^*$ and $\mathsf{Min}^j_{s_1,\ldots,s_n}(\mathsf{p}_0 x) = 1$ 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 $\mathsf{Min}^j_{s_1,\ldots,s_n}(\mathsf{p}_0 x) = 1$. Therefore we have $S_j \subset S_i$ and applying consistency condition (1) yields $T_j \subset T_i$. By (6) and the monotonicity of * we finally get

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

□

Remark 9. Both consistency conditions are necessary premises in the above theorem. Let s_1, s_2, t_1, t_2 be names for the classes $\{1\}, \{1, 2\}, \{1\}$ and $\{2\}$, respectively, so that $s_1 \mathrel{\dot\in} \mathsf{pow}(s_2)$, $t_1 \mathrel{\dot\in} \mathsf{pow}(t_1)$ and $t_2 \mathrel{\dot\in} \mathsf{pow}(t_2)$. We find that $s_1 \mathrel{\dot\subset} s_2$ holds but not $t_1 \mathrel{\dot\subset} t_2$, that is the type $\{s_1 \to t_1, s_2 \to t_2\}$ does not satisfy the first consistency condition. We see that

$$f_1 := \lambda x.(t_1, 1) \mathrel{\dot\in} (s_1^* \to t_1^*) \text{ and } f_2 := \lambda x.(t_2, 2) \mathrel{\dot\in} (s_2^* \to t_2^*)$$

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 the type $\{s_1 \to t_1, s_2 \to t_2\}$.

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 $s_1 \mathrel{\dot\in} \mathsf{pow}(s_3)$. Note that the type $\{s_2 \to t_2, s_3 \to t_3\}$ does not satisfy the condition (2). We find

that $f_2 := \lambda x.(t_2, 2) \dot{\in} (s_2^* \to t_2^*)$ as well as $f_3 := \lambda x.(t_3, 3) \dot{\in} (s_3^* \to t_3^*)$ hold. We get $(s_1, 1) \dot{\in} s_3^*$, 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 $\mathsf{over}_{s_2,s_3}(f_2, f_3)$.

Remark 10. Now we will explain why we do not give an interpretation of $\lambda\&$ in OTN. Basically, there are two problems. The first is, as already mentioned, that OTN is formalized in a partial setting and therefore, the definition of λ abstraction does not commute with substitutions, see Example 3. This has the consequence that reduction would not preserve a straightforward interpretation of $\lambda\&$ terms in OTN.

The second problem is more subtle. As we have seen in the previous remark, if an overloaded function type has two branches with respective domains $\{1, 2\}$ and $\{1, 3\}$, then it must have also a third branch with the domain $\{1\}$ in order to be well-formed. Assume that we have in $\lambda\&$ two atomic types A and B, which are modeled by the types $\{1, 2\}$ and $\{1, 3\}$, respectively. Assume further, that A and B do not have a common subtype in $\lambda\&$. Then an overloaded function type consisting of two branches with the domains A and B would be well-formed in $\lambda\&$ but its interpretation in OTN would *not* be well-formed since in OTN the type $\{1\}$ exists.

Of course, one could require that two atomic types of $\lambda\&$ that do not have a common subtype have to be modeled by disjoint types in OTN. Still, it would remain to show by induction that the same also holds for overloaded function types. Probably, it would only be possible to prove that if two types A and B of $\lambda\&$ do not have a common subtype, then there is no type C of $\lambda\&$ so that the interpretation of C is contained in both the interpretation of A and the interpretation of B. If this is the case, then one would have to find a weaker version of the consistency condition (2) which takes into account only the interpretations of $\lambda\&$ types but not all types of OTN. Then Theorem 8 might be reformulated as a soundness theorem with respect to typing. However, this construction would be non-trivial and a lot of details would have to be checked.

Remark 11. In many natural models for second order λ calculi one faces the problem of "too many subtypes": the only closed term f of the type $(\forall X \subset \mathsf{N})(X \to X)$ is the identity function on the natural numbers, see Bruce and Longo [3]. 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 mathe-

matics. 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

$$(\forall X \subset \mathsf{N})t \in (X^* \to X^*). \tag{7}$$

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 a (partial) second order λ calculus. As shown before, the identity function is not the only function satisfying (7) in this model; but there are also many other functions of this type.

5 Conclusion

We have introduced the system $\mathsf{OTN}+(\mathbb{L}\text{-}\mathsf{I}_\mathsf{N})$ of types and names based on elementary separation, product, join and power types; and we have presented a set-theoretic model for this theory. Then we have shown how to define in OTN late-bound overloaded functions with a corresponding impredicative type structure à la $\lambda\&$. The key ingredient to make this work was the existence of *monotone* power types in explicit mathematics.

Due to the naming relation, systems of explicit mathematics naturally contain type-dependent computations. However, up to now, this feature has not been used to model programming language concepts. In this paper we try to clarify the relation between the explicit naming of types in explicit mathematics and calculi with late-bound overloaded functions like $\lambda\&$. We cannot provide an actual interpretation of $\lambda\&$ in OTN since systems of explicit mathematics are based on *partial* applicative theories. Nevertheless, our construction provides a set-theoretic model for partial impredicative overloading without restricting itself to early-binding or coherent overloading. We have proven two theorems that exhibit the strong relationship between our construction and $\lambda\&$. Theorem 7 about subtyping of overloaded function types in OTN is an exact copy of the subtyping rule of $\lambda\&$. Theorem 8 about typing corresponds to the typing rule of $\lambda\&$. It shows why there are some consistency conditions needed for the soundness of this rule.

One direction of future work is to continue the study of type systems for modern programming languages in the framework of explicit mathematics.

For example, it would be interesting to explore the combination of late-bound overloading with parametric polymorphism. Another direction of research lies in the proof-theoretic analysis of the power type axiom. As Feferman [9] noticed, join makes the power type axiom much more effective. Hence, it is natural to ask what is the proof-theoretic strength of the power type axiom in the context of uniform elementary separation and join. Also, it is an open question whether the monotonicity axiom for pow affects the proof-theoretic strength of the underlying system of explicit mathematics.

Acknowledgments. We would like to thank Gerhard Jäger and Thomas Strahm for many helpful comments on an earlier version of this paper.

References

[1] Michael J. Beeson. *Foundations of Constructive Mathematics: Metamathematical Studies*. Springer, 1985.

[2] 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.

[3] 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.

[4] Andrea Cantini. Relating Quine's NF to Feferman's EM. *Studia Logica*, 62:141–163, 1999.

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

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

[7] 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.

[8] 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.

[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. 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] Thomas Glass. On power set in explicit mathematics. *The Journal of Symbolic Logic*, 61(2):468–489, 1996.

[17] Thomas Glass, Michael Rathjen, and Andreas Schlüter. On the proof-theoretic strenght of monotone induction in explicit mathematics. *Annals of Pure and Applied Logic*, 85:1–46, 1997.

[18] Thomas Glass and Thomas Strahm. Systems of explicit mathematics with non-constructive μ-operator and join. *Annals of Pure and Applied Logic*, 82:193–219, 1996.

[19] 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.

[20] Gerhard Jäger. Power types in explicit mathematics? *The Journal of Symbolic Logic*, 62(4):1142–1146, 1997.

[21] 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.

[22] Gerhard Jäger and Thomas Strahm. The proof-theoretic analysis of the Suslin operator in applicative theories. In W. Sieg, R. Sommer, and C. Talcott, editors, *Reflections on the Foundations of Mathematics: Essays in Honor of Solomon Feferman*. A K Peters, 2002.

[23] Gerhard Jäger and Thomas Studer. Extending the system T_0 of explicit mathematics: the limit and Mahlo axioms. *Annals of Pure and Applied Logic*, 114:79–101, 2002.

[24] Reinhard Kahle. N-strictness in applicative theories. *Archive for Mathematical Logic*, 39(2):125–144, 2000.

[25] 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.

[26] 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.

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

[28] Thomas Studer. Constructive Foundations for Featherweight Java. In R. Kahle, P. Schröder-Heister, and R. Stärk, editors, *Proof Theory in Computer Science*, volume 2183 of *Lecture Notes in Computer Science*, pages 202–238. Springer, 2001.

[29] Thomas Studer. *Object-Oriented Programming in Explicit Mathematics: Towards the Mathematics of Objects*. PhD thesis, Institut für Informatik und angewandte Mathematik, Universität Bern, 2001.

[30] Thomas Studer. A semantics for $\lambda^{\{\}}_{str}$: a calculus with overloading and late-binding. *Journal of Logic and Computation*, 11(4):527–544, 2001.

[31] 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.

[32] Anne Sjerp Troelstra and Dirk van Dalen. *Constructivism in Mathematics, vol I*. North Holland, 1988.

[33] Anne Sjerp Troelstra and Dirk van Dalen. *Constructivism in Mathematics, vol II*. North Holland, 1988.

[34] Hideki Tsuiki. *A Record Calculus with a Merge Operator*. PhD thesis, Keio University, 1992.

[35] Hideki Tsuiki. A computationally adequate model for overloading via domain-valued functors. *Mathematical Structures in Computer Science*, 8:321–349, 1998.

[36] Raymond Turner. *Constructive Foundations for Functional Languages*. McGraw Hill, 1991.

[37] Raymond Turner. Weak theories of operations and types. *Journal of Logic and Computation*, 6(1):5–31, 1996.

Address

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