entropy

MDPI

Article

Computational Abstraction

Raymond Turner

School of Computer Science and Electronic Engineering, University of Essex, Colchester CO4 3SQ, UK; turnr@essex.ac.uk

Abstract: Representation and abstraction are two of the fundamental concepts of computer science. Together they enable "high-level" programming: without abstraction programming would be tied to machine code; without a machine representation, it would be a pure mathematical exercise. Representation begins with an abstract structure and seeks to find a more concrete one. Abstraction does the reverse: it starts with concrete structures and abstracts away. While formal accounts of representation are easy to find, abstraction is a different matter. In this paper, we provide an analysis of data abstraction based upon some contemporary work in the philosophy of mathematics. The paper contains a mathematical account of how Frege's approach to abstraction may be interpreted, modified, extended and imported into type theory. We argue that representation and abstraction, while mathematical siblings, are philosophically quite different. A case of special interest concerns the abstract/physical interface which houses both the physical representation of abstract structures and the abstraction of physical systems.

Keywords: abstraction; computation; representation; specification; Frege

Citation: Turner, R. Computational Abstraction. *Entropy* **2021**, *23*, 213. http://doi.org/10.3390/e23020213

Academic Editor: Neal G. Anderson
Received: 11 January 2021
Accepted: 3 February 2021
Published: 10 February 2021

1. Introduction

This paper is intended to be a contribution to the "philosophy of computer science" [1]. This is a new discipline and uncovering its core philosophical concerns is still in process [2]. However, it appears to raise subject specific ontological, methodological and epistemological questions. What kinds of things are programs? Are they abstract or concrete? There are conceptual questions that arise in connection with the multiple languages of the discipline. What is a "good" language design? What is the function of semantic theory; does it raise parallel issues to those of the philosophy of language? There are methodological and epistemological questions pertaining to the construction of "correct" programs and software. What is the nature of program correctness? Is it a mathematical or empirical affair—or both? Can we guarantee correctness?

The nature of computational abstraction invokes several of these traditional philosophical categories, and is a common theme in the informal arena of computational discussion [3] . It is said to be one of the mechanisms behind the design of programming and specification languages: it is the mechanism that supports the move to "high-level" from machine oriented languages. It is also at the centre of program and software specification and design in the guise of " data abstraction" [4–7], and it is this area where we focus attention.

Finding a representation for a given specification is the essence of programming: some goal is specified, and the programming task is to locate a suite of programs that satisfy it. At first gloss, specifications employ abstract concepts that involve little or no process information: specifications say "what has to be done" without saying "how to do it". Presumably, the latter is the job of actual programs. But this is an over simplification. In practice, "programming" and "specification" are relative terms. In particular, one level of programs can serve as the specification of more concrete ones. For example, Pascal [8] or Miranda [9] may be employed as the specification medium for programs that are eventually implemented in some form of machine code. In these cases, what is the specification and

what is the program is a matter of intention: what is taken to provide the correctness conditions for what [10]. What is generally true is that the vehicles of specification, the concepts employed, are at a more "abstract level" than those of the representation or program. Moreover, these different levels of abstraction are enshrined in different notions of data type that are built into contemporary programming and specification languages [4,8,11–16]. For instance, Haskell [14] employs recursive data types whereas Fortran [11] uses arrays and iteration. Some data types are "abstractions" of more "concrete" ones in the sense that some information is neglected, hidden or ignored in order to arrive at a more abstract structure. But what is it to neglect, hide or ignore information? What precisely is "computational abstraction"? Is there an exact mechanism for such? Addressing these questions is a crucial part of the present task.

In Sections 2–5, we introduce our formulation of abstract types, and provide some working examples to illustrate representation and abstraction. Sections 6 and 7 introduce "Fregean abstraction" in a type-theoretic setting, and Section 8 brings representation into the picture. Sections 9–12 consider levels of abstraction, abstraction over families and physical abstraction. Sections 13–15 discuss the mathematical and philosophical differences between abstraction and representation, and the last section reflects a little on the appropriate foundational framework.

1.1. Methodology

As we said at the outset, the present paper is a contribution to this general area. This work is theoretical and foundational: there are philosophical and mathematical aspects to the research. But there is no empirical component; there are no experimental results.

1.2. Previous Work

While there is a good amount of contemporary philosophical work aimed at providing a philosophical foundations for abstraction in mathematics [17–21], there is no substantial parallel study aimed at abstraction in computer science. The present paper attempts to fill this gap by applying the insights of this philosophical work aimed at mathematics to computer science. This work has its origins in [2] where the present approach is sketched.

1.3. Results

We provide a conceptual and mathematical analysis of "data abstraction" and "representation" and their relationship. In particular, we provide a type-theoretic approach to Fregean abstraction: abstraction generates new abstract types. We demonstrate that abstraction and representation are related to each other as "congruence" and "homomorphism". However, there is a philosophical twist in that our treatment of abstraction is not based on the normal construction of "Quotient Types", but on a method of abstraction that has its origins in Frege [22], and underpins contemporary abstractionism in the philosophy of mathematics [17,23,24]. One particular instance of the abstraction/representation pairing sits on the abstract/physical interface, and its analysis provides some insight into physical abstraction/representation, and how this impacts upon any related notion of computation.

2. Abstract Data Types

Representation and abstraction come together in the notion of an "abstract data type" [4,7,25], the original formulation of which emphasizes the idea that it is the operations of the type that play the characterizing role.

> An abstract data type defines a class of abstract objects which is completely characterized by the operations available on those objects. This means that an abstract data type can be defined by defining the characterizing operations for that type. [7]

According to this, an abstract type is somehow constituted by its collection of operations. There are various interpretations of this [6], but a minimal one has it that an abstract type has the following form. (There is a distinction between "state-based" and "functional"

abstract types. We shall concentrate on the latter—which will be sufficient to illustrate the broad idea of abstraction. See [26] for an account of abstraction in state based systems.)

$$\boldsymbol{A} =< A, \Omega >$$

Here, A is the carrier type and Ω is the set of functions that operate on A. (One can think of these as set-theoretic structures where A is a set and the functions are set theoretic functions. We shall later discuss this assumption in more detail).

But what does it mean to say that a type is completely characterized by "defining" its characterizing functions? There are two kinds of definitions that seem appropriate. An "explicit" definition of a function would be one that defines it in terms of something else, presumably in terms of functions from some other type. For example, we might specify the type of queues by their representation as sequences. So that the queue operations are defined in terms of sequence operations. Subsequently, the properties of the definiendum are fixed by the properties of the definiens. This interpretation is adopted in some approaches to specification [5]. Alternatively, definitions are taken to be "implicit" where an abstract type is fixed by the relationships between these functions expressed mathematically: an "implicit" definition involves an axiomatic account that lays out the formal relationships between the functions of the type. We shall adopt this approach.

However, similar conceptual points arise for both styles. In particular, under both regimes abstract types define abstract objects. This rules out any account where the functions are characterized by some form of physical implementation in which the objects are physical and the operations mechanical operations on these objects. This would not characterize an abstract type. So, to be clear, the characterization we employ follows a traditional philosophical distinction: abstract types are mathematical objects and non-abstract ones refer to physical structures. But be aware that we shall use the terms "abstract" and "concrete" as relative terms. In particular, "concrete types" maybe be mathematical. In contrast, the term "physical type" is taken only to refer to a physical structures. (Eventually within computer science, as opposed to mathematics, all abstract types must be given physical representation. This is in keeping with what we have argued elsewhere: computer science is centrally concerned with the construction of technical artifacts [2]. So abstract types must eventually morph into physical ones.)

To make our general notion more applicable, a few extensions and qualifications are necessary. Our structures may take the following form.

$$\boldsymbol{A}[T] =< A[T], \Omega >$$

This introduces a type that is "polymorphic" in T, where the latter is a parameter of the abstract type, i.e., it introduces a whole collection of types parameterized in T. Moreover, the functions of the abstract type are implicitly polymorphic, i.e., they maybe predicatively applied to any T. Additionally, these structures are subject to some axiomatic constraints.

We shall also require relations as well as functions in our structures. Here we represent them as Boolean valued functions $(T \Rightarrow Bool)$, where T is a type and the type of Boolean values is taken to be the following structure.

$$\boldsymbol{Bool} =< Bool, \Delta >$$

Here $Bool$ is the enumerated type $\{true, false\}$ and $cond$ is the only function in Δ with the following functionality.

$$cond : Bool \otimes T \otimes T \Rightarrow T$$

where $A \otimes B$ represents the Cartesian product of two types and $A \Rightarrow B$ the type of functions from A to B. Note that Cartesian products are taken to bind more strongly than function spaces, e.g., $Bool \otimes T \otimes T \Rightarrow T$ stands for $(Bool \otimes T \otimes T) \Rightarrow T$.

The function *cond* is taken to satisfy the following equality conditions.

$$cond(true, t, t') =_{Bool} t \qquad cond(false, t, t') =_{Bool} t'.$$

In order to state the axioms, such as these for *cond*, we assume that every type T comes equipped with an "external" notion of equality $=_T$. For example, $=_{Bool}$ is the "external" equality for the Boolean type. Here we shall often abbreviate $b =_{Bool} true$ as just, b i.e., when used as a logical assertion. Where it is clear which types are involved we shall drop the subscript on equality.

These equalities are not part of the "programming language" but part of its semantic theory [27,28] . You cannot employ them in the text of function definitions, i.e., programs. Note that *cond* is polymorphic in that it applies to arbitrary types T. In particular it applies to *Bool* itself. So, all the standard logical connectives, $\wedge, \vee, \rightarrow, \neg$ may be defined (or "programmed") in terms of the conditional, e.g.,

$$\rightarrow (b, b') \doteq cond(b, b', true).$$

More often than not, we shall employ infix notation and write $b \rightarrow b'$, etc.

In addition, we assume that each type T comes equipped with an "internal" Boolean function of equality that is part of the " functional programming language".

$$eq_T : (T \otimes T) \Rightarrow Bool.$$

That is, we may write function definitions using it. In particular, for complex type constructors, internal equality is usually definable in terms of the internal equality for its components. For example, for Cartesian products, we define it point-wise.

$$eq_{A \otimes B}((x, y), (u, v)) \doteq eq_A(x, u) \wedge eq_B(y, v).$$

For the types of this paper these two notions of equality will be provably co-extensional. However, both notions are necessary. Semantic judgments about the language are made with the external notion, whereas programs in the language employ the internal one.

This provides the basic notion of "abstract type". There are some central examples that will be used to illustrate and make the various issues concrete.

3. Some Data Types

Lists form a paradigm example of an abstract type. The polymorphic version takes the following form.

$$\boldsymbol{L}[T] =< L[T], \{nil, cons, rec, head, tail\} >$$

For instance, we might form lists of numbers $\boldsymbol{L}[N]$ or lists of Boolean values $\boldsymbol{L}[Bool]$ etc. Formally, the first and last pairs of operations have the following functionalities.

$$nil : L[T]$$

$$cons : T \otimes L[T] \Rightarrow L[T]$$

$$head : L[T] \Rightarrow T$$

$$tail : L[T] \Rightarrow L[T]$$

The functions *nil* and *cons* are the constructors of the type; they dictate the form of the elements. There is an unsaid assumption that the type of lists, over a given type, is the smallest type that contains the empty list *nil* and is closed under the *cons* operator. Implicitly, this justifies the use of induction as a means of reasoning about these types. The exact form of induction will depend upon the logical system employed. For example, a formulation in first-order logic would support first-order induction, and this is sufficient for the present application. However, a full development of the underlying theoretical

framework is a topic for another occasion, but see the section on foundations for some further elaboration of what this might look like.

The interaction of these constructors with the destructors, *head* and *tail*, is governed by the following axioms.

$$head(cons(t,l)) = t \qquad tail(cons(t,l)) = l$$

The axioms are silent on what happens outside these constraints. The final operator of the type, the function *rec*, is a (polymorphic) recursion operator over lists.

$$rec : L[T] \otimes A \Rightarrow C$$

where A and C are arbitrary types. Given $g : A \Rightarrow C$ and $f : T \otimes A \otimes C \Rightarrow C$, the recursion operator is taken to satisfy the following recursion equations, where the type A is an optional parameter, i.e., there may not be one.

$$rec(nil, a) = g(a)$$

$$rec(cons(t,l), a) = f(t, a, rec(l, a))$$

The function *rec* enables the definition of new functions, i.e., programming with lists in the functional style is largely driven by the definition of recursive functions.

Our second abstract type is a polymorphic version of *queues*. While queues and lists have the same structural signature,

$$\boldsymbol{Q}[T] =< Q[T], \{emp, enqueue, rec, front, dequeue\} >,$$

lists operate a "last-in" and "first-out" regime: queues reverse matters and employ a "last-in" and "last-out" one. More explicitly, the functions of the structure have the following types.

$$emp : Q[T]$$

$$enqueue : Q[T] \otimes T \Rightarrow Q[T]$$

$$rec : Q[T] \otimes A \Rightarrow C$$

$$dequeue : Q[T] \Rightarrow Q[T]$$

$$front : Q[T] \Rightarrow T$$

They are are taken to satisfy the following equality axioms. These are a little more messy than those for lists. Notice in particular that *dequeue* has to recursively unpack the structure of the queue to reach the front.

$$front(enqueue(emp, t)) = t \qquad dequeue(enqueue(emp, t)) = emp$$

$$front(enqueue(q, t)) = front(q) \quad where \quad q \neq emp$$

$$dequeue(enqueue(q, t)) = enqueue(dequeue(q), t) \quad where \quad q \neq emp$$

Despite these differences, we can employ recursion on queues in a parallel fashion to that for lists. Given $g : A \Rightarrow C$ and $f : T \otimes A \otimes C \Rightarrow C$, recursion over queues satisfies the following.

$$rec(emp, a) = g(a)$$

$$rec(enqueue(q, t), a) = f(t, a, rec(q, a))$$

We shall also employ restricted abstract types $\boldsymbol{L}[T]^-$ and $\boldsymbol{Q}[T]^-$ where $\Delta_L = \{nil, cons, rec\}$ and $\Delta_Q = \{emp, enqueue, rec\}$. All the "programming" in the next section only employs these restricted structures.

4. Some Infrastructure

We require some infrastructure to facilitate the formulation of data abstraction, and this is obtained by programming within these structures. (The reader unfamiliar with functional programming might consult [14,15] for a relevant introduction.) Most of the functions or "programs" contained here employ the above form of recursive definition. Indeed, for pedagogical reasons, we shall often not explicitly conform to the exact syntax for recursive definitions but use a more familiar and natural recursive style. We illustrate matters with lists, but everything works just as well for queues.

Figure 1 provides a recursive definition of the *append* operation that glues two lists together.

$$append(nil, l) \doteq l$$
$$append(cons(t, l'), l) \doteq cons(t, append(l', l))$$

Figure 1. $append : L[T] \otimes L[T] \Rightarrow L[T]$.

Here $f : T \otimes L[T] \Rightarrow L[T]$ is given as $f(t, l) = cons(t, l)$.

Given internal equality for the type T, Figure 2 provides a definition of internal equality for lists.

$$eq(nil, nil) \doteq true$$
$$eq(cons(a, l), nil) \doteq false$$
$$eq(nil, cons(a, l)) \doteq false$$
$$eq(cons(a, l), cons(b, k)) \doteq eq(a, b) \wedge eq(l, k)$$

Figure 2. List Equality.

A few more functions are required for the theoretical development. Membership in lists is defined in Figure 3.

$$mem(nil, x) \doteq false$$
$$mem(cons(a, l), x) \doteq cond(eq_T(a, x), true, mem(x, l))$$

Figure 3. $mem : L[T] \otimes T \Rightarrow Bool$.

Here *mem* is defined as a recursive function where the generating function is defined as $f(l, a, x, c) \doteq cond(eq_T(a, x), true, c)$. We shall employ more familiar notation and write $a \in l$ for $mem(l, a)$. By definition, the empty list behaves as expected: i.e., as a Boolean valued term $x \in nil$ equals false.

Given a Boolean valued function $g : L[T] \Rightarrow Bool$, we may define (Figure 4) "quantification" over lists.

$$\forall x \in nil.g[x] \doteq true$$
$$\forall x \in con(a, l).g[x] \doteq g[a] \wedge \forall x \in l.g[x].$$

Figure 4. Quantification Over Lists.

We may then define "extensional equivalence" for lists, ($\equiv$, Figure 5), again as a Boolean valued function.

$$u \equiv v \doteq (\forall x \in u.x \in v) \leftrightarrow (\forall x \in v.x \in u)$$

Figure 5. $\equiv$: $L[T] \otimes L[T] \Rightarrow Bool$.

Finally, we shall say that function $g : L[T] \otimes A \Rightarrow L[T]$ is *Extensional* if

$$\forall u : L[T].\forall v : L[T].u \equiv v \rightarrow \forall x : T.g(u,x) \equiv g(v,x).$$

We shall use this shortly. All of the above are also definable for queues.

Our objective is to employ these types as a basis for abstracting more abstract ones. In particular, we aim to abstract the following "more abstract" type.

5. Finite Sets

The abstract type of finite sets underpins the style of specification common to logical specification languages such as Z [29] , B [30] and VDM [5]. Indeed, there is also at least one programming language with finite sets as its central data type [31]. Our objective is to abstract the abstract type of finite sets from lists/queues. For pedagogical reasons, we first put in place this target of abstraction. It has the following signature.

$$\boldsymbol{S}[T] =< S[T], \{\phi, \oplus, rec\} >$$

Here $S[T]$ consists of "finite sets" whose elements are selected from T. The type is defined axiomatically as follows. We employ the standard constant for the empty set, and the function $\oplus$ adds a single element to a set. It satisfies two axiomatic constraints that demand that we ignore duplicates and the order of the elements.

$$(Dup) \quad \oplus(t,s) = \oplus(t,\oplus(t,s))$$

$$(Ord) \quad \oplus(t,\oplus(t',s)) = \oplus(t',\oplus(t,s))$$

As we shall see shortly, these guarantee extensionality for sets. This paves the way for the formulation of recursion over finite sets. This follows the pattern of recursion for lists. However, it has to be restricted: *rec* must be a function and, given the above constraints on set equality, for this to hold, we require *rec* to satisfy the following.

$$(Duprec) \quad rec(\oplus(t,\oplus(t,s)),a) = rec(\oplus(t,s),a)$$

$$(Ordrec) \quad rec(\oplus(t,\oplus(t',s)),a) = rec(\oplus(t',\oplus(t,s),a))$$

Consequently, the generating function $f : T \otimes A \otimes C \Rightarrow C$ must satisfy

$$(Dupf) \quad f(t,a,f(t,a,c)) = f(t,a,c)$$

$$(Ordf) \quad f(t,a,f(t',a,c)) = f(t',a,f(t,a,c))$$

Such functions we shall call *legitimate* generators for recursion. *Dupf* and *Ordf* guarantee that recursion is functional. Conversely, if *rec* is a function that satisfies the recursion equations, then the generating function f will be legitimate. Notice that the legitimacy constraint may also be applied to lists, i.e., we can restrict list recursion to legitimate generators.

In particular, the following functions are supported by such recursions.

$$mem : S[T] \otimes T \Rightarrow Bool$$

This is defined exactly as membership in lists. Again, we shall write $a \in s$ for $mem(s,a)$.

We may define quantification with respect to sets in a parallel way to lists. But now we may define internal equality (Figure 6).

$$eq_{S[T]}(s,s') \doteq (\forall x \in s.x \in s') \wedge (\forall x \in s'.x \in s)$$

Figure 6. Equality.

Given this, we can use induction on sets, together with *Dup* and *Ord*, to show that extensionality holds, i.e.,

$$\forall x : S[T].\forall y : S[T].eq_{S[T]}(s, s') \rightarrow s =_{S[T]} s'$$

So, "sets" behave as expected. We are now in a position to develop our approach to data abstraction.

6. Frege on Abstraction

The word "abstraction" is used throughout computer science with varying degrees of explicitness and precision. Actually, it is doubtful that there is just one computational usage. We concentrate on "data abstraction" [4,5], and here the Fregean perspective seems to provide the beginnings of some conceptual clarification, as well as the basis for some mathematical precision.

The traditional analysis of abstraction has its roots in Locke [21,32]. He has it that abstraction is a mental process in which new abstract ideas are formed by reflecting upon several objects or ideas, and omitting the features that distinguish them.

> The same Colour being observed to day in Chalk or Snow, which the Mind yesterday received from Milk, it considers that Appearance alone, makes it representative of all of that kind; and having given it the name Whiteness, it by that sound signifies the same quality wheresoever to be imagin'd or met with; and thus Universals, whether Ideas or Terms, are made. [32]

Seemingly, general terms stand for abstract ideas that are created by separating these ideas from the spatial and temporal qualities of particular things. For instance, one is given a range of white things of varying shape and sizes, and one ignores the respects in which they differ. In this way we come to idea of "whiteness".

Influential as it has been, it does not provide a clear basis for any precise mathematical account. Fortunately, contemporary work in the foundations of mathematics [17,19,23], based upon Frege's remarks on mathematical abstraction [22], has laid the groundwork for such an account.

> The judgment 'Line a is parallel to line b', in symbols: a || b, can be taken as an identity. If we do this, we obtain the concept of direction, and say: 'The direction of line a is equal to the direction of line b'. Thus we replace the symbol || by the more generic symbol =, through removing what is specific in the content of the former and dividing it between a and b. We carve up the content in a way different from the original way, and this yields us a new concept. [22]

Frege observes that many of the singular terms that appear to refer to abstract entities are formed by means of functional expressions. For example, the following would appear to pick out new abstract objects, namely *directions* and *collections*.

- *The direction of a line.*
- *The collection of elements in a list.*

While it is true that many singular terms formed by means of functional expressions denote ordinary concrete objects: e.g., 'the present pope', 'the source of corona 19', the functional terms that pick out abstract entities are distinctive in the sense that associated with such a functional expression there is an "equation" of the following form.

1. *The direction of a line A =The direction of line B* if and only if *A is parallel to B.*
2. *The collection of elements in list l = the collection of elements in list k*

Inspired by these examples, an abstraction principle may be formulated as a bi-conditional of the following form:

$$\forall x : K.\forall y : K.h(x) = h(y) \leftrightarrow R(x, y),$$

where h is a term forming operator, R is an equivalence relation, and K is the "kind" of objects over which we are quantifying and abstracting. Such principles of abstraction are intended to be mechanisms for the abstraction of new "kinds of things". Given a kind of thing K, abstraction introduces a new kind of thing H such that:

$$\forall z : H.\exists x : K.h(x) = z.$$

This insists that h is a surjective term forming operator from K to H. So, given the *kind* of thing that are lines, the abstraction principle introduces the *kind* of thing that are directions.

Our goal is to use this insight to provide a mathematical foundation for data abstraction where *kinds* are replaced by abstract data types.

7. Abstracting Abstract Types

When applied to data types we require a little more than the above general account of abstraction. More exactly, when dealing with data types we need to explicitly take into account not just the objects but also the functions of the type. Our objective with the new abstract type is to maintain the signature of the concrete type but with a "re-carving" of the content.

The following concept is a modification of the standard notion to cover functions that take and return values (e.g., *Bool*) outside the type under scrutiny.

Definition 1. *Let* $\boldsymbol{A} =< A, \Omega >$ *be any abstract type then* $R : A \otimes A \Rightarrow Bool$*, an equivalence relation on* A*, is a Congruence Relation if every* $f : \Omega$*,*

1. If $f : B \otimes A \otimes D \Rightarrow A$*, then* $\forall x : B.\forall u, u' : A.\forall y : D.R(u, u') \rightarrow R(f(x, u, y), f(x, u', y))$.

2. If $f : B \otimes A \otimes D \Rightarrow C$*, then* $\forall x : B.\forall u, u' : A.\forall y : D.R(u, u') \rightarrow f(x, u, y) =_C f(x, u', y)$.

With this in place we may formulate our notion of data abstraction. Assume that we are given a "concrete" data type

$$\boldsymbol{C} =< C, \Lambda >,$$

together with a congruence relation R on C. Then, the following *Principle of Abstraction* is taken to introduce a new type C/R whose elements have the form $h(c)$ for $c : C$, and whose internal equality conditions are given by the following principle of abstraction.

$$(abst) \qquad \forall x : C.\forall y : C.eq_{C/R}(h(x), h(y)) = R(x, y).$$

In other words, we are postulating a new type via its internal equality. Here the relation R is a defined Boolean function of the concrete structure. The new type is introduced whose equality relation is stipulated axiomatically to agree with R. Finally, the function h is stipulated to be surjective, i.e.,

$$\forall z : C/R.\exists x : C.h(x) = z.$$

This is to guarantee that there are no elements in the abstracted type except those that are abstracted.

Given that R is a congruence, we may lift the functions from the concrete type to the abstract one. For example, where for $g : \Lambda$ with functionality $g : C^n \Rightarrow C$, we "lift" the function to $\widehat{g} : (C/R)^n \Rightarrow C/R$ defined as follows.

$$\widehat{g}(h(x_1), \ldots, h(x_n)) \doteq h(\widehat{g}(x_1, \ldots, x_n))$$

This yields a new abstract type.

$$\boldsymbol{C/R} =< C/R, \Lambda/R >$$

where Λ/R is the type of all such $\widehat{g}$.

We illustrate the idea by abstracting finite sets from lists. We shall deal with the case of queues, and how matters are related, later.

Example 1. *We first provide the "axis of abstraction". This is provided by the notion of "extensional equivalence" for lists. The following principle of abstraction generates a new type $L[T]/\equiv$ (written as $\mathbf{S}[T]$).*

$$\forall u : L[T].\forall v : L[T].eq_{S[T]}(set(u), set(v)) = u \equiv v$$

This provides the actual carrier of the type. But now we have to define the various operations following the pattern of the general case.

$$\phi \doteq set(nil)$$

$$a \oplus set(l) \doteq set(cons(a, l))$$

$$rec(set(z), a) \doteq set(rec(z, a))$$

where both set and list recursion are restricted. Each of these new operators is extensional, i.e., congruent relative to extensional equality.

We have thus extracted a structure that satisfies the axioms for sets: the conditions *Dup* and *Ord* follow since equality is extensional equivalence. To complete the process of abstraction we can "kick away" the dependence on lists and axiomatize finite sets directly in terms of *Dup* and *Ord*.

We might be tempted to interpret these "Fregean abstract types" as quotient types: the new type A/R would then be identified as the set of equivalence classes of A induced by the congruence relation R. This would be the standard mathematical approach. But this interpretation is not the Fregean one. It does not have the same ontological force. Fregean abstractions are not new sets: principles of abstraction in the Fregean mold introduce new sui-generis notions that are not part of the existing background ontology. Fregean abstraction principles are taken to provide a mechanism for the creation of new abstract structures. Indeed, abstraction in this guise provides an explanation of how new abstract structures might come about.

8. Representation

On the face of it, representation is the opposite of abstraction. Here one begins with an abstract type and seeks to represent it in a more concrete one [4,5]. More precisely, given an abstract type

$$\boldsymbol{A} =< A, \Omega >,$$

the objective is to locate a more concrete one,

$$\boldsymbol{C} =< C, \Lambda > .$$

This is taken to "represent" the abstract one just in case there exits a function,

$$F : C \Rightarrow A,$$

the "Representation Function", from C into A that reflects the structural connections between them.

Definition 2. *Let $\boldsymbol{C} =< C, \Lambda >$ and $\boldsymbol{A} =< A, \Omega >$ be abstract types, then $F : C \Rightarrow A$, an is a Homomorphism if every $f_A : \Lambda$*

1. *If $f_C : B \otimes C \otimes D \Rightarrow C$, then $\forall x : B.\forall u : C.\forall y : D.(F(f_C(x, u, y)) =_A f_A(x, F(u), y))$.*
2. *If $f_C : B \otimes C \otimes D \Rightarrow E$, then $\forall x : B.\forall u : C.\forall y : D.f_A(x, F(u), y) =_E f_C(x, u, y)$.*

One further demand is that homomorphisms that are taken to be representational mappings are surjective, i.e., all abstract entities must have a concrete representation. So, a representation function must be a surjective homomorphism from the structure $\boldsymbol{C}$ onto the structure $\boldsymbol{A}$.

Example 2. *In the representation of sets as lists we are required to locate a representing homomorphism:*

$$set : L[T] \Rightarrow S[T].$$

And this is straightforward: we follow the structure of lists.

$$set(nil) \doteq \phi$$

$$set(cons(t, l)) \doteq t \oplus set(l)$$

This is surjective: to prove it we employ the set-induction. So everything in $S[T]$ has the form $set(x)$ for some $x : L[T]$. Let

$$f : L[T] \otimes A \Rightarrow C$$

be extensional. Then we may extend this to a function

$$\hat{f} : S[T] \otimes A \Rightarrow C$$

by

$$\hat{f}(set(x), a) \doteq f(x, a)$$

We can prove for restricted recursion for lists, this time by list-induction, that for all $z : L[T]$,

$$set(rec(z, a)) = rec(set(z), a),$$

where on the right hand side $\widehat{f}$ is employed in the definition of rec. So, as expected, set extends to the recursion operator.

There is another notion of representation that refers to the relation between a physical system and the external physical world. This is a case of the physical system "modelling" the physical world. Our notion of representation exists between a physical system and an abstract one. However, the word "implementation" is often used for both notions of representation.

9. Programming and Computation

What is a "program" and what is a "computation" differ at different levels of abstraction. Sets are more abstract than lists and this impacts upon the style of programming, and what computations are supported. Programming with sets must respect extensionality. An example of this is the *append* program for lists. This gives rise to the *union* program for sets (Figure 7).

$$union(\phi, s) \doteq s$$

$$union(\oplus(t, s'), s) \doteq cons(t, append(s', s))$$

Figure 7. $union : S[T] \otimes S[T] \Rightarrow S[T]$.

The function *append* is a representation of *union* where the two are related under the *set* function extended to set recursion. Conversely, given *append* we may abstract or lift the function to obtain the *union* operation on sets. This is possible because *append* is an instance of restricted recursion. So, computations generated by these two functions mirror each other. From the representational perspective, sets fix the specifiable computations on lists; with abstraction only certain list computations give rise to set theoretic ones.

However, although the following, Figure 8, is programmable with lists, it is not with sets: it is not a restricted recursion.

$$reverse(nil) \doteq nil$$

$$reverse(cons(t,l)) \doteq append(reverse(l), cons(t, nil))$$

Figure 8. Reversing a List.

So, there are computations with lists that cannot be carried out with sets, and these involve non-extensional functions. If we move down one level to the representation of lists in store in the form of linked lists, there will be computations on linked lists (e.g., those involving pointers to locations) that have no analogues with the abstract notion of list: not all physical computations will have abstract analogues. Different levels of abstraction support different notions of computation. Any computation programmable at a more abstract level is representable lower down but not visa-versa. This brings us nicely to the following section.

10. Levels of Abstraction

The computer science literature contains a great many informal discussions of the phrase, "level of abstraction" including its wider philosophical application [33]. There is also a pioneering formal analysis on the identity of computational artifacts [34]. However, there are few if any, conceptually motivated formal accounts of such levels within the literature on data abstraction. Our objective is to provide one based upon the above concept of "Fregean abstraction".

Any approach to abstraction in the Fregean mold faces the threat of paradox [19,20]. The neo-Fregeans Hale and Wright [23] respond by severely restricting the class of acceptable abstraction principles. Their approach is "static" in the sense that they hold the domain of the overall theory fixed. The alternative approach is "dynamic" [35] in the sense that abstraction with respect to a given domain may result in a new domain: i.e., we do not work in some fixed formal system but allow systems to grow by abstraction, and by abstractions built upon previous abstractions. This is the present approach. (Formally, we expect the addition of new abstract types to be conservative but we need a more precise formal framework to state and prove this.)

We illustrate matters with the move from list to sets that hides various possible levels of abstraction. Consider the principles *Dup* and *Ord*. These two principles are associated with two intermediate levels of abstraction: one jettisons order and the other eliminates duplicates. Consider the second. The following Boolean valued function, Figure 9, insists that two lists u and v are permutations of each other.

$$perm(nil, nil) \doteq true$$
$$perm(cons(x,u), nil) \doteq false$$
$$perm(nil, cons(x,u)) \doteq false$$
$$perm(cons(x,u), cons(y,v))$$
$$\doteq$$
$$(eq_T(x,y) \wedge perm(u,v)) \vee (member(x,v) \wedge member(y,u) \wedge perm(delete(y,u), delete(x,v)))$$

Figure 9. $perm : L[T] \otimes L[T] \Rightarrow Bool$.

Here $delete(y,u)$ deletes y from u. This facilitates the following principle of abstraction: two lists are equivalent if they are permutations of each other. Let $u, v : L[T]$ then

$$\forall u : L[T].\forall v : L[T].(eq_{\mathbf{B}[T]}(bag(u), bag(v)) = perm(u,v)).$$

As indicated by the name, this introduces *Bags* as a new abstract type which we symbolize as follows.

$$\boldsymbol{B}[T] =< B[T], empty, plus, rec >$$

Bags are permutation independent (*Ord*). Recursion is now restricted to generating functions that are order-independent, i.e., satisfy $Ordf$.

Conversely, given both types, $L[T]$ and $B[T]$, the representation function is then given as follows.

$$bag : L[T] \Rightarrow B[T].$$

This follows the same pattern as the representation function for sets.

$$bag(empty) \doteq \phi$$

$$bag(cons(t,l)) \doteq plus(t, bag(l))$$

$$rec(bag(z), a) \doteq bag(rec(z, a))$$

This type, which in terms "levels of abstraction", is intermediate between lists and sets.

Alternatively, beginning with lists, we can abstract by removing duplicates. To this end we define a function, Figure 10, that eliminates duplicates.

$$duplicate(nil) \doteq nil$$

$$duplicate(cons(u,l)) \doteq cond(u \in l, duplicate(l), cons(u, duplicate(l)))$$

Figure 10. $duplicate : L[T] \Rightarrow L[T]$.

We are then able to set up an equivalence between lists.

$$l \asymp k \doteq duplicate(l) = duplicate(k)$$

Finally, we abstract on this relation.

$$\forall u : L[T].\forall v : L[T].(eq_{\boldsymbol{N}[T]}(nd(u), nd(v)) = u \asymp v)$$

This introduces a new type—lists without duplicates.

$$\boldsymbol{N}[T] =< N[T], none, ndcons, rec >$$

Here recursion is restricted to generating functions that satisfy $(Dupf)$.

These two abstractions can be applied not just to lists but to the result of each other: they maybe composed in any order. Either way, the result of the composition is a type whose elements satisfy both *Ord* and *Dup*, i.e., this gets us to $\boldsymbol{S}[T]$ by two possible routes.

We can put matters as follows.

Definition 3. *Let* $\boldsymbol{C} =< C, \Lambda >$ *and* $\boldsymbol{A} =< A, \Omega >$ *be abstract types. Then* $\boldsymbol{A}$ *is at* Least as Abstract *than* $\boldsymbol{C}$ *if there is a surjective homomorphism from* $\boldsymbol{C}$ *to* $\boldsymbol{A}$*. It is* Strictly Less Abstract *if in addition there is no homomorphism in the opposite direction.*

For example, we have surjective homomorphism operating between the following abstract types—and none in the other directions.

$$\boldsymbol{L}[T]^{-} \Rightarrow \boldsymbol{B}[T] \Rightarrow \boldsymbol{S}[T]$$

Abstraction is a layered dynamic process, and the above examples, demonstrate, in a precise way, how different levels can be formulated and combined under composition to yield even more abstract notions.

11. Abstraction over Families

There are many possible representations of finite set theory: lists, queues, bags, arrays, and sequences all provide such. Moreover, each of these possible representations can be used as the source of abstraction. Indeed, the various mechanisms of abstraction are somehow an encoding of what all these representations have in common. Additionally, what they all have in common is that they are "containers" whose central property is to hold elements.

More generally, how do we abstract a type from a family of abstract types that share some "common" features? To address this we require the notion of a family of abstract types.

Definition 4. *An* ***I****-Indexed Family of Abstract Typesis a family*

$$F =< A_i, \Delta_i >_{i:I}$$

where each type has the same functional signature. A simple example employs our existing types.

Example 3. *Consider the abstract types* $\boldsymbol{L}[T]^-, \boldsymbol{B}[T]$ *and* $\boldsymbol{Q}[T]^-$ *where* $\Delta_L = \{nil, cons, rec\}$, $\Delta_Q = \{emp, enqueue, rec\}$, $\Delta_B = \{empty, plus, rec\}$. *These provide the constituents of a simple indexed family*

$$C =< C_i, \Delta_i >_{i:I}$$

with three components indexed by the enumerated type $I = \{L, B, Q\}$ *and where* $C_B = B[T]$, $C_Q = Q[T]^-$ *and* $C_L = L[T]^-$.

To abstract over such a family we need to be able to articulate what they have in common, and this is achieved by isolating the appropriate notion of congruence for such families. For this it is technically convenient to represent them as single structure over which congruence relations can be formulated in the standard way. Consequently, we form the disjoint union of the types in the family.

Definition 5. *The* Disjoint Union *of the family* $\boldsymbol{F} =< A_i, \Delta_i >_{i:I}$ *is the structure*

$$\boldsymbol{A} = (A, \Delta)$$

where

(i) $a : A \leftrightarrow \exists i : I.\exists z : A_i.a = (i, z)$

(ii) $f : \Delta \leftrightarrow \exists i : I.\exists g : \Delta_i.\forall x_1 : A_i \ldots\ldots \forall x_n : A_i.f((i, x_1), \ldots, (i, x_n)) = g(x_1, \ldots, x_n)$

Example 4. *The disjoint union of the family of abstract types*

$$C =< C_i, \Delta_i >_{i:I}$$

is the abstract type, "container".

$$\mathbf{C}[T] \doteq (C[T], \Delta)$$

Explicitly, $\Delta = \{nil, cons, rec\}$ *where, according to the definition, these are defined as follows. We illustrate with* cons.

$$cons(t, (L, l)) \doteq cons(t, l)$$

$$cons(t, (B, b)) \doteq plus(t, b)$$

$$cons(t, (Q, q)) \doteq enqueue(t, q)$$

While these form a family with just three members, there are other possible "container" types such as Sequences *and* Arrays, *and this analysis can be extended in a straightforward way.*

Let $F =< A_i, \Delta_i >_{i:I}$ *be any indexed family. The representations of a structure B given in terms of individual members of the family, i.e.,*

$$h_i : A_i \Rightarrow B$$

maybe turned into a representation of B in terms of the disjoint union in the obvious way.

$$h(i,a) \doteq h_i(a)$$

So, in particular, we get a representation of $\boldsymbol{C}[T]$ *into* $\boldsymbol{S}[T]$.

We are now in a position to define abstraction over such families.

Definition 6. *Abstraction over Families. Let*

$$\boldsymbol{A} = (A, \Delta)$$

be a disjoint union of the family $F =< A_i, \Delta_i >_{i:I}$ *and let* $R : A \otimes A \Rightarrow Bool$ *be a congruence relation over* $\boldsymbol{A}$. *Abstraction introduces a new type*

$$\boldsymbol{A}/R =< \boldsymbol{A}/R, \Delta/R >$$

with equality given by the following abstraction principle.

$$\forall u : A.\forall v : A.eq_{\boldsymbol{A}/R}([i,a],[j,b]) = R((i,a),(j,b))$$

where the new terms take the form $[i,a]$. *Given R is a congruence we may lift the functions of the individual types to* $\boldsymbol{A}$ *as standard, i.e.,* Δ/R *is the set*

$$\{\widehat{f}.f : \Delta\}$$

where

$$\widehat{f}([(i,a_1)],..,[(i,a_m)]) \doteq [f((i,a_1),.....,(i,a_m))].$$

Example 5. *As an example we demonstrate how to abstract sets from containers. Using recursion on the structures we are able to define a congruence relation over* $\boldsymbol{C}[T]$. *We illustrate with that part of the relation that links lists and bags (Figure 11).*

$$\mathcal{E}((L,nil),(B,empty)) \doteq true$$
$$\mathcal{E}((L,nil),(B,add(t,b))) \doteq false$$
$$\mathcal{E}((L,cons(t,l)),(B,empty)) \doteq false$$
$$\mathcal{E}((L,cons(t,l)),(B,add(t,b))) \doteq \mathcal{E}((L,l),(B,b))$$
$$\mathcal{E}((L,cons(t,l)),(B,add(t',b))) \doteq mem(t,b) \wedge mem(t',l) \wedge \mathcal{E}((L,delete(l,t')),(B,delete(b,t)))$$

Figure 11. Extensional Equivalence.

The definitions of the other pairs follow suite: together they define extensional equivalence between the types.

How is this related to abstraction relative to a single structure? The congruence relation $R : A \otimes A \Rightarrow Bool$ has restrictions to pairs of types in the family.

$$R_{ij} : A_i \otimes A_j \Rightarrow Bool$$

$$R_{ij}(a,b) \doteq R((i,a),(j,b))$$

In particular, consider the special case $R_i \doteq R_{ii}$, and the corresponding restricted abstraction:

$$\forall u : A_i.\forall v : A_i.eq_{A_i/R_i}([i,a],[i,b]) = R_i(a,b).$$

Here the congruence is restricted to the elements of a single abstract type. This yields a new abstract type:

$$\boldsymbol{A}_i/R_i =< A_i/R_i, \Delta_i/R_i > .$$

Example 6. *In the case of our running example, the abstracted type is isomorphic* ($\simeq$) *to what is obtained from abstraction over a single relation i.e.,*

$$(\boldsymbol{C}[T]/\mathcal{E}) \simeq (\boldsymbol{L}[T]^{-}/\equiv) \simeq (\boldsymbol{B}[T]/\equiv) \simeq (\boldsymbol{Q}[T]^{-}/\equiv) \simeq \mathbf{S}[T]$$

So, abstracting on a family of containers results in the same type (up to isomorphism) as abstracting from a single member.

In fact, generally, such single abstractions are isomorphic to the family abstraction.

Theorem 1. *For each* $i : I, \boldsymbol{A}/R \simeq \boldsymbol{A}_i/R_i$

This justifies the use of our original form of abstraction where a new type is abstracted from an individual type.

12. Abstract and Physical

We have used the term "abstract" in an ambiguous way. Firstly, we have used the term in a relative sense as a member of an abstract/concrete pair, i.e., one member of a pair is more "abstract" than another. But there is also the absolute sense of the term in which a type is mathematical. So far all types have in this sense been "absolutely" abstract. And this applies to all those we have referred to as "concrete". Of course, in computer science, at some point in the levels of representation/abstraction, the notion of being "concrete" will give way to the "physical" [36], where a physical type refers to physical things, and as such it is not determined by mathematical means but by the physical properties of its objects and operations. In other words, a physical type

$$\boldsymbol{P} =< P, \Pi >,$$

has physical objects and operations. A simple example concerns digital circuits. Such circuits are not mathematical objects but physical devices. Instances include binary adders, *and* and *or* gates, etc. Complex circuits result from composing simpler ones.

While there are significant differences that emanate from the different natures of abstract and physical devices, the representational/abstraction distinction is still applicable. What then becomes of the processes of representation and abstraction?

Given an abstract type

$$\boldsymbol{A} =< A, \Omega >$$

and a physical one

$$\boldsymbol{P} =< P, \Pi >,$$

$\boldsymbol{P}$ will be taken to be a representation of $\boldsymbol{A}$ just in case there is a surjective mapping, a correspondence, from the physical device to the abstract one that preserves the structure. This mirrors the demands of the so called *simple mapping* account of computation [37] where each pair of functions in the concrete and abstract types are subject to the constraints of that account. This is reflected in the homomorphic nature of the mapping.

Physical representation may be understood in terms of the concepts of technical artifacts [38–41]. Technical artefacts are physical objects. However, they cannot be definitively characterized by enumerating their physical properties since this has no place for their functional features. Likewise, they cannot they be completely characterized by an intentional conceptualization: their functionality must be realized in an adequate physical structure. Any adequate conceptualization must take into account both aspects and involve both functional and structural features. See [2,38–43] for further discussion of this perspective.

Applied to the present setting, the abstract type can be taken as the "functional description of the artifact" and informs us "what the thing is for". The physical type is characterized by a physical description of its objects and operations and acts as the "structural description" of the artifact.

Unfortunately, we inherit all the philosophical concerns of the simple mapping account of the nature of physical computation [44]. In particular, the homomorphism constraint is taken to be too easy to satisfy: almost anything that can be put into correspondence with the abstract type counts as a representation. Moreover, these triviality results apply also to the abstract/concrete relation (that is defined in terms of homomorphism), namely, that every concrete data type is a representation of every abstract one.

However, there is more going on here than mere extensional agreement that is witnessed by the homomorphism. In particular, this is not the only relationship between physical and abstract devices. Abstracting the abstract device from concrete instances requires the uncovering of a congruence relation. Given a collection of physical devices

$$\boldsymbol{P}$$

and a congruence relation R on $\boldsymbol{P}$, we can abstract to obtain an abstract type

$$\boldsymbol{P}/R.$$

This yields an abstract device whose mathematical properties are inherited from the structural properties of the various physical devices, i.e., those that survive the congruence. At this point what is a physical property is abstracted as a mathematical one. This is line with the Wittgenstein perspective that mathematical theorems are 'hardened' empirical regularities, upon which the former are supervenient. A mathematical 'proposition' functions as if it were an empirical proposition "hardened into a rule" [45]. Such abstractions provide one mechanism for transforming empirical propositions to mathematical ones. Once more, we illustrate with digital circuits. We say that two physical circuits P and Q are "equivalent", if they have the same input/output. They may differ in terms of their size and the other physical properties possessed by electronic circuits. When we abstract on this relation we obtain a data type of "abstract circuits".

The representation/abstraction duality is still present in the physical set-up. In the representational case, the use of the concepts of technical artifacts to conceptualize matters seems appropriate. Not so in the case of abstraction. Abstracting from a collection of physical devices yields an abstract characterization of them. Logically, this creates a new concept; it is not artifact specification [10] but the creation of an abstract notion.

We shall return to the abstract/physical case when we have made some more general observations about the differences between representation and abstraction. However, the general point is clear. There is more to the relationship between the physical and abstract devices than their extensional agreement: there is also an intentional aspect that manifest itself as one of either representation or abstraction. This takes some of the sting out of the "pancomputationalism [37]" claim .

13. Mathematical Duality

We earlier alluded to the relationship between "abstraction" and "representation". In particular, we stated that the two are related via two standard mathematical notions ("congruence" and "homomorphism"). This provides us with a way of conceptualizing the connections between representation and abstraction, and ensures us that we have a complementary pair of notions.

The abstraction process itself begins with a congruence relation and implictly defines a homomorphism via the principle of abstraction. The converse is also true: every homomorphism gives rise to a congruence. The following is routine to check.

Theorem 2. *Given types* $\mathcal{A} =< A, \Omega_{\mathcal{A}} >$ *and* $\mathcal{B} =< B, \Omega_{\mathcal{B}} >$ *with the same signature, and a surjective homomorhism,* $F : A \Rightarrow B$ *we can construct a congruence relation by*

$$R(x, y) \doteq (F(x) = F(y))$$

Then $\mathcal{A}/R =< A/R, \Omega/R >$ *is isomorphic to* $\mathcal{B}$.

So formally, $\boldsymbol{A}$ is an abstraction of $\boldsymbol{C}$ if and only if $\boldsymbol{C}$ is a representation of $\boldsymbol{A}$. This is the standard mathematical relationship between homomorphisms and equivalence/congruence relations. Moreover, there are parallel mathematical obligations. In the representation scenario we have to show that the specified mapping is a surjective homomorphism; in the abstraction case we have to show that the defined relation is a congruence. They are mathematical siblings: if one is true so is the other. However, as we remarked before, Fregean abstractions do not result in quotient types but introduce new concepts and structures that are sui-generis.

But even given the mathematical confluence between the two, there is much more to the relationship between concrete and abstract structures than this mathematical one. We shall get to this shortly but first we need to explore the process of abstraction a little more.

14. Fully Abstract Structures

Are there abstract types that are, in some sense, as abstract as possible. One motivation for this is pragmatic: a desideratum for specification and design is to isolate and hide as many implementation details as possible. Another is located in a more austere interpretation of the notion of abstract data type. Being fixed by the functions of the type could be taken as a demand that the elements of the type are "indiscernible" with respect to the functions of the type, i.e., they fix the notion of equality for the type. To proceed we require following notion.

Definition 7. *A type* $\mathcal{B} =< B, \Omega >$ *is a* Fully Abstract Structure *of a class C of types, of the same signature, if for any* $\mathcal{A}$ *in C there exists a surjective homomorphism from* $\mathcal{A}$ *onto* $\mathcal{B}$.

Example 7. *Consider the class* $\{\boldsymbol{L}[T]^{-}, \boldsymbol{B}[T], \boldsymbol{S}[T]\}$. $\boldsymbol{S}[T]$ *is the most abstract.*

Given this "Galois" style connection between our notions of congruence and homomorphism, we can also characterize fully abstract structures in terms of congruence relations.

Theorem 3. *A structure* $\mathcal{B}$ *is a fully abstract type of the class C iff for any* $\mathcal{A}$ *in the class there exists a congruence R on* $\mathcal{A}$ *such that* $\mathcal{A}/R$ *is isomorphic to* $\mathcal{B}$.

Proof. Assume that $\mathcal{B}$ is fully abstract for the class C. Assume that $\mathcal{A}$ is a member of C. Then there is a homomorphism H from $\mathcal{A}$ into $\mathcal{B}$. Define the derived congruence as before:

$$a \equiv b \doteq H(a) = H(b).$$

We know from a previous theorem that, $\mathcal{A}/R$ is isomorphic to $\mathcal{B}$. Conversely, assume $\mathcal{A}$ is in the class and there exists a congruence R on $\mathcal{A}$ such that $\mathcal{A}/R$ is isomorphic to $\mathcal{B}$. Let G be this isomorphism. Abstraction itself provides a surjective homomorphism F from $\mathcal{A}$ onto $\mathcal{A}/R$. The composition $G \circ F$ yields the required homomorphism from $\mathcal{A}$ onto $\mathcal{B}$. □

But how do we construct such fully abstract types in a uniform way? This brings the second motivation into the picture. In the next definition we assume that equality $(=_{\mathcal{A}})$ for the structure $\mathcal{A} =< A, \Omega >$ is not an element of Ω.

Definition 8. *Let* $\mathcal{A} =< A, \Omega >$ *be any type. Then for each* $O : \Omega$ *we define*

$$a \approx_O a' \doteq I(a, a') \wedge O(a, a')$$

where

$$I(a, a') \doteq \forall z : A.O(a, z) \leftrightarrow O(a', z)$$

$$O(a, a') \doteq \forall z : A.O(z, a) \leftrightarrow O(z, a')$$

The first insists that they are indistinguishable as inputs, and the second they are so as outputs. So, $a \approx_O a'$ insists that a and a' are indistinguishable relative to the operation O. The Natural Congruence *for $\mathcal{A}$, $\approx_{\mathcal{A}}$, is then defined as:*

$$a \approx_{\mathcal{A}} a' \doteq \forall O : \Omega.a \approx_O a',$$

which demands that a and a' are indistinguishable relative to the whole structure.

For following we have only to observe that any congruence will by definition imply the natural one.

Theorem 4. *Let $\mathcal{A} =< A, \Omega >$ be any type. Then*
(1) $\approx_{\mathcal{A}}$ is a congruence relation on $\mathcal{A}$
(2) If R is any congruence on $\mathcal{A}$, then $R(a, b) \rightarrow a \approx_{\mathcal{A}} b$

We may now construct the corresponding abstract type.

Definition 9. *Let $\mathcal{A}$ be any state-based structure. Then*

$$\mathcal{A}/\approx_{\mathcal{A}}$$

is the Natural Abstraction of $\mathcal{A}$.

In what sense is the natural abstraction fully abstract?

Theorem 5. *Let $\mathcal{A} =< A, \Omega >$ be any type. Let $C_{\mathcal{A}}$ be the class of types that are obtained by abstraction via congruence relations on $\mathcal{A}$. Then*
(1) The natural abstraction of $\mathcal{A}$ is a fully abstract structure of $C_{\mathcal{A}}$.
(2) Every fully abstract type of the class $C_{\mathcal{A}}$ is isomorphic to its natural abstraction.

Proof. The required homomorphism maps the elements of the given congruence to those of the natural abstraction. For the second part, let $\mathcal{A}$ be fully abstract. Then there exits some congruence R on $A/\approx$ such that $\mathcal{A}$ is isomorphic to $(\mathcal{A}/\approx_A)/R$. By Theorem 2, $(\mathcal{A}/\approx_A)/R$ must be isomorphic to $\mathcal{A}/\approx$. □

Every fully abstract structure is obtained from a natural abstraction, and natural abstractions are those that insist that equality if fixed by the operations. So, fully abstract types become a candidate for a more austere characterization of the notion of abstract data type.

15. Semantics

While representation and abstraction are mathematically equivalent, in the sense that the notions of "homomorphism" and "congruence relation" are mathematical duals, the activities of representation and abstraction are intentionally quite different. They have different goals and starting points; what governs what is different; what is correct is different.

Buried in these remarks are two substantial philosophical differences.

One of these concerns their semantic impact. Consider the representation case. The abstract type provides a semantic interpretation for the concrete one in that it provides the correctness conditions for the concrete one: the operations of the concrete type must be in harmony with those of the abstract one. This is fleshed out by the demand that the representation function is a homomorphism. This is a minimalist requirement for the role

of semantics: it must supply the conditions of correctness for the use of expressions in the language [46].

> In general, a semantic account of a language of any kind must tell us when we are using an expression correctly, and when we are not. The fact that the expression means something implies that there is a whole set of normative truths about my behavior with that expression; namely, that my use of it is correct in application to certain objects and not in application to others. .... The normativity of meaning turns out to be, in other words, simply a new name for the familiar fact that, regardless of whether one thinks of meaning in truth-theoretic or assertion-theoretic terms, meaningful expressions possess conditions of correct use. Kripke's insight was to realize that this observation may be converted into a condition of adequacy on theories of the determination of meaning: any proposed candidate for the property in virtue of which an expression has meaning, must be such as to ground the 'normativity' of meaning-it ought to be possible to read off from any alleged meaning constituting property of a word, what is the correct use of that word. [46]

A semantic account must provide us with an account of what constitutes correct use. Specifically, operations of the concrete type must be correct relative to the corresponding operations of the abstract one. In this sense the representation has a semantic function.

Abstraction principles also have semantic significance: the semantic interpretation of the functional expressions on the left hand side would appear to be given by the congruence relations on the right hand side, i.e., the latter are semantically prior to the functional expression on the left. For example, in order to understand the term '*direction*' is to know that '*the direction of a*' and '*the direction of b*' refer to the same entity if and only if the lines *a* and *b* are parallel. In other words, an understanding of the concept of a direction presupposes an understanding of the concept of parallelism, but not vice-versa. This is the perspective of semantic abstractionism [17,23]. Roughly, this insists that our capacity to have singular thoughts about objects of a certain type is fixed by the truth-conditions of identity judgments about objects of that type. In the case of data types, there is a semantic role played by the concrete type together with the defined congruence relation that supplies equality for the new type. The relation is a defined relation of the concrete type: it is expressed in the language of the concrete type, the semantics of which is given. The operations of the abstracted type are then given in terms of the more concrete ones. In this sense, the abstract structure is semantically dependent upon the concrete one. Another perspective on this is provided in [35] who documents three underlying principles for any semantic theory that tie together semantics with abstraction. With its roots in Frege, the semantics of a language relies on there being a notion of reference for the terms of the language. This notion is fixed by the objects in the semantic domains. And these are fixed by the equality conditions supplied by abstraction.

While both representation and abstraction have a semantic component, the correctness conditions are reversed. For representation, it is the abstract type that provides the correctness conditions for the concrete one. For abstraction, it is the concrete type that furnishes the semantic interpretation for the abstract one.

16. Ontology

The most obvious difference between representation and abstraction concerns their underlying ontological assumptions. In the representational case, the construction of a homomorphism from the concrete to the abstract presupposes that both data types already exist. In contrast, in the abstraction scenario, the abstract structure is ontologically dependent upon the more concrete one: it is created from it by abstraction. Abstraction principles are taken to introduce new terms referring to sui-generis objects, and thereby provide a mechanism for the creation of new abstract types. In our example, finite sets are introduced via abstraction from finite lists. However, in the case of representation, no new objects with their types are introduced. We are given both structures ahead of time.

This ontological difference between representation and abstraction can be further illustrated by considering the the way these notions are employed in set theory [47] . We illustrate with the most fundamental of all data types, the natural numbers. Consider what might be taken as the abstract type of natural numbers.

$$\boldsymbol{N} =< N, 0, succ, rec >$$

where *rec* represents a scheme of numerical recursion. There are various representations of this structure in set theory. For example, the Von-Neumann representation of the natural numbers takes the following form.

$$rep : \boldsymbol{V} \Rightarrow \boldsymbol{N}$$

where

$$rep(\phi) \doteq 0 \qquad rep(\{n, \{n\}\}) \doteq succ(rep(n))$$

and where $\boldsymbol{V}$ is the following structure.

$$\boldsymbol{V} =< V, \phi, succ, rec > .$$

ϕ is the empty set and $succ(n) \doteq \{n, \{n\}\}$ etc. The Zermelo representation is only slightly different. Here $succ(n) \doteq \{n\}$. Indeed, in principle, any of the other possible systems of set theoretic numerals may be used to represent the natural numbers, and they all generate true arithmetical statements. But there is a significant underlying assumption here, namely that there is an independent notion of "natural number" that is being represented: representations must be representations of something. This representational scenario is at the heart of the foundations of mathematics based upon set theory. Its function is to demonstrate that all of mathematics can be "represented" in set theory.

The abstractionist approach is entirely different. First notice that representations are not intended to be definitions, and there are good reasons for not so taking them. There is an immediate concern: which one do we chose? Do we select the Von-Neumann or Zermelo numerals—or one of the other myriad of possibilities? Each of them attributes a different collection of set-theoretic, non-arithmetic, properties to numbers. For example, Zermelo and Von-Neumann differ on whether n is a member of $n + 1$. While different choices attribute different sets of set-theoretic properties, from a numerical perspective, there appears to be no good reason why one account is superior to another. On the other hand, both accounts cannot both be "correct" since they contradict each other, e.g., over membership. This predicament is often called *Benacerraf's identification problem* [48]. According to it, numbers cannot be defined as sets. Moreover, such identifications do not respect mathematical practice: we do not treat numbers as sets. Even in standard expositions of set theory, where one begins with the Von-Neumann account, this is quickly discarded, and only the arithmetical properties of the numbers employed. The non-arithmetical properties are never used in the development of mathematics inside set theory.

All this is a criticism only of the definitional perspective. From the representational one, multiple representations are not surprising, indeed they are to be welcomed since the aim of specification is to jettison or hide many of the incidental properties that give rise to such multiplicity. Generally, for abstract types, we would expect the representations to have additional properties and operations that go beyond the abstract version.

Dedekind [49] earlier claimed that set theoretic formalizations are not faithful as definitions, and links it to the need for some kind of abstraction.

> The real numbers should not be identified with the corresponding cuts because those cuts have "wrong properties"; namely, as sets they contain elements, something that seems "foreign" to the real numbers themselves. Similarly, the natural numbers should not be ascribed set-theoretic or other "foreign" properties; they too should be conceived of "purely arithmetically". If one wishes to pursue your approach I should advise not to take the class itself (the system of mutually

> similar systems) as the number (Anzahl, cardinal number), but rather something new (corresponding to this class), something the mind creates.

Dedekind is alluding to the creation of new things, and here abstraction comes to the fore. One interpretation of this is that numbers are to be abstracted: we need to abstract away from all the idiosyncratic properties of the various set-theoretic definitions, and just leave the numerical structure. To see how, consider an indexed family of possible numerical structures.

$$\boldsymbol{Num} =< Num_i, \Delta_i >$$

where $\Delta_i = \{0^i, succ^i, rec^i\}$. These might include the Von-Neumann and Zermelo structures. All such set theoretic numeral systems are structurally isomorphic. Employing this as a congruence over the dependent sum of $\boldsymbol{Num}$ we obtain a structure:

$$\boldsymbol{N}/\simeq$$

which is isomorphic to each of the representational systems. In particular, the Von-Neumman system satisfies the following.

$$\boldsymbol{V} \simeq (\boldsymbol{V}/\simeq)$$

However, the elements of $(\boldsymbol{V}/\simeq)$ are not sets but sui-generis objects whose totality of properties are numerical, and precisely those common to all numeral representations. In this way we abstract away from the details of individual numeral systems to obtain the new abstract structure of the natural numbers. (See [50] who employs this technique to develop a Fregean approach to Structuralism [51]. From this perspective abstract data types are "structures" and the data items themselves are "positions" in the structure [50].) Abstraction brings new sui-generis notions into existence. Not so with the representation.

17. Foundations

While we have employed a set-theoretic framework for our exposition, the use of set theory as the appropriate medium for the formalization of computational notions is not unproblematic. The arguments of [48] would appear to apply equally to other discrete notions: presumably, lists and finite sets are subject to the same analysis as numbers. Lists have an informal interpretation in which they have no additional set-theoretic properties but only those that pertain to list processing and manipulation. In our intuitive everyday notion of a list, lists themselves are not sets. Even within the set-theoretic regime, applying abstraction to any set-theoretic representation of a data type results in new primitive notions: the present theory of finite sets reflects a vastly different notion of "set" to that given by the Zermelo–Fraenkel axioms.

However, it is not just the data items themselves that raise issues. Feferman [52] introduces two criteria for judging the success or otherwise of the formalization of an informal mathematical notion. Let T be a formal theory of an informal body of mathematics M.

1. T is "adequate" for M, if every concept, argument, and result of M is represented by a (basic or defined) concept, proof, and a theorem, respectively, of T.
2. T is "faithful" to M, if every basic concept of T corresponds to a basic concept of M and every axiom and rule of T corresponds to or is implicit in the assumptions and reasoning followed in M (i.e., T does not go beyond M conceptually or in principle).

The Zermelo–Fraekel set theory does not provide a faithful formalization of the computer science notion of type. The primary purpose of the latter is a way of classifying the objects of the language: it acts as barrier to semantic nonsense. Furthermore, for practical reasons, type membership must be a decidable judgment: we require a type-checker to be part of the implementation of the language. While set theory is an adequate formalization of the computational notion of type, it is not faithful: it goes beyond the reasoning associated with the role of types.

This argues for a more radical approach in which the computational notion of type is taken as the fundamental one. Indeed, this is implicit in our informal development which took place against the backdrop of functional programming [14,15]. To formalize matters we require a mathematical framework in which both the syntactic and semantics aspects of functional languages can be formalized. While different functional languages, with different basic types and different type constructors, would give rise to different theories of types, Constructive type theory [53] provides a possible framework. But there are others. Feferman's theories of operations and types [52], the theory of constructions [54,55], Typed predicate logic [9,36,56] and Turner's theories of operations and types [57], might all be employed. Finally, Type theory and category theory are close relations. This would also be a possible framework for abstraction—some might think the natural one.

18. Conclusions

Abstraction and representation are two of the fundamental notions of contemporary computer science. While they are mathematically dual notions they are philosophically quite different.

While they both have semantic import, they differ in terms of what provides the semantic interpretation of what. For abstraction, the source type provides the semantic foundations for the abstracted one by supplying its equality conditions. For representation, the abstract type supplies the correctness conditions for the source type acting as a representation.

Ontologically, they differ in that representation assumes that both structures are in place, whereas abstraction brings new structures into being. Abstraction generates new types, representation enables their use in computation by supplying a more concrete representation.

These differences apply equally well to the case where one of the data types is physical. This demonstrates that the relationship between the physical and abstract types goes beyond extensional agreement: it brings an intentional aspect that is materialized by the intentions to locate a representation for one type in another or abstract a new type from an existing one—or a family of such.

Funding: There is no attached funding for this research. However, the paper was written as part of PROGRAMME ANR (https://programme.hypotheses.org/) project: What is a program? Historical and philosophical perspectives. And we acknowledge financial assistance for workshops and conferences.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data sharing not applicable.

Acknowledgments: Thanks to Cliff Jones for many useful discussions on data abstraction.

Conflicts of Interest: There is no conflict of interest and the paper does not raise any ethical concerns.

References

1. Turner, R.; Angius, N. The Philosophy of Computer Science. In *The Stanford Encyclopedia of Philosophy (Winter 2020 Edition)*; Zalta, E.N., Ed.; Stanford University: Stanford, CA, USA, 2020. Available online: https://plato.stanford.edu/archives/win2020/entries/computer-science/ (19 January 2021).
2. Turner, R. *Computational Artifacts: Towards a Philosophy of Computer Science*; Springer: Berlin/Heidelberg, Germany, 2018.
3. Hoare, C.A.R., II. Notes on Data Structuring. In *Structured Programming*; Academic Press: Cambridge, MA, USA, 1972.
4. Dale, N.L.; Walker, H.M. *Abstract Data Types: Specifications, Implementations, and Applications*; Jones & Bartlett Learning: Burlington, MA, USA, 1996; ISBN 978-0-66940000-7.
5. Jones, C.B. *Software Development: A Rigorous Approach*; Prentice Hall International: Upper Saddle River, NJ, USA, 1980; ISBN 0-13-821884-6.
6. Broy, M.; Wirsing. M. A Systematic Study of Models of Abstract Data Types. *Theor. Comput. Sci.* **1984**, *33*, 139–174.
7. Liskov, B.; Zilles, S. Programming with abstract data types. *ACM SIGPLAN Not.* **1974**, *9*, 50–59, doi:10.1145/800233.807045.
8. Wirth, N. *The Programming Language Pascal (Revised Report)*; ETH Zürich: Zürich, Switzerland, 1973.

9. Turner, D.A. Miranda: A Non-Strict Functional Language with Polymorphic Types. In Proceedings of the IFIP Conference on Functional Programming Languages and Computer Architecture, Nancy, France, 16–19 September 1985; Springer Lecture Notes in Computer Science; Volume 201, pp. 1–16.
10. Turner, R. Specification. *Minds Mach.* **2011**, *21*, 135–152.
11. Backus, J. The History of Fortran I, II, and III. *ACM SIGPLAN Not.* **1978**, *13*, 165–180, doi:10.1145/960118.808380.
12. Bjorner, D.; Jones, C.B. *The Vienna Development Method: The Meta-Language, Lecture Notes in Computer Science 61*; Springer: Berlin/Heidelberg, Germany; New York, NY, USA, 1978; ISBN 978-0-387-08766-5.
13. Bowen, J. *Formal Specification and Documentation Using Z: A Case Study Approach*; International Thomson Computer Press: New York, NY, USA, 1996; ISBN 1-85032-230-9.
14. Thompson, S. *Haskell: The Craft of Functional Programming (International Computer Science Series) (English Edition)*; Addison-Wesley: Boston, MA, USA, 2011.
15. Milner, R.; Tofte, M.; Harper, R.; MacQueen, D. *The Definition of Standard ML (Revised)*; MIT Press: Cambridge, MA, USA, 1997; ISBN 978-0-262-63181-5.
16. Abrial, J.R.; Schumanand, S.A.; Meyer, B. A Specification Language. In *On the Construction of Programs*; Macnaghten, A.M., McKeag, R.M., Eds.; Cambridge University Press: Cambridge, UK, 1980; ISBN 0-521-23090-X.
17. Ebert, P.; Rossberg, M. *Abstractionism: Essays in Philosophy of Mathematics*; Oxford University Press: Oxford, UK, 2016.
18. Burgess, J.P. Book Review: Kit Fine. "The Limits of Abstraction". *Notre Dame J. Form. Log.* **2003**, *44*, 227–251.
19. Fine, K. *The Limits of Abstraction*; Clarendon Press: Oxford, UK, 2008.
20. Heck, R.G. The Development of Arithmetic in Frege's Grundgesetze der Arithmetik. *J. Symb. Log.* **1993**, *58*, 579–601.
21. Mancosu, P. *Abstraction and Infinity*; Oxford University Press: Oxford, UK, 2016.
22. Frege, G. *Posthumous Writings*; White., R., Translator; Basil Blackwell: Oxford, UK, 2003.
23. Hale, B.; Wright, C. *The Reason's Proper Study: Essays toward a Neo-Fregean Philosophy of Mathematics*; Oxford University Press: New York, NY, USA, 2001; 472p.
24. Wright, C. *Frege's Conception of Numbers as Objects*; Aberdeen University Press: Aberdeen, UK, 1983.
25. Thomas, P.; Robinson, H.; Emms, J. *Abstract Data Types : Their Specification, Representation, and Use*; Clarendon Press: Oxford, UK; Oxford University Press: New York, NY, USA, 1988.
26. Jones, C.B.; Turner, R. Abstraction in State Based Systems. Forthcoming.
27. Abramsky, S.; Jung, A. Domain theory. In *Handbook of Logic in Computer Science. III*; Abramsky, S., Gabbay, D.M., Maibaum, T.S.E., Eds.; Oxford University Press: Oxford, UK, 1994; pp. 1–168, ISBN 0-19-853762-X.
28. Milne, R.E.; Strachey, C. *A Theory of Programming Language Semantics*; Springer: Berlin/Heidelberg, Germany, 1976; ISBN 978-1-5041-2833-9.
29. Spivey, J.M. The Z Notation: A reference manual. In *International Series in Computer Science*, 2nd ed.; Prentice Hall: Upper Saddle River, NJ, USA, 1992.
30. Abrial. J.R. *The B-Book: Assigning Programs to Meanings*; Cambridge University Press: Cambridge, UK, 1996.
31. Schwartz, J.T.; Dewar R.B.K.; Dubinsky, E.; Schonberg, E. *Programming with Sets: An Introduction to SETL (Monographs in Computer Science)*; Springer: Berlin/Heidelberg, Germany, 1986.
32. Locke, J. *An Essay Concerning Human Understanding*; Nidditch, P.H., Ed.; Oxford University Press: Oxford, UK, 1690/1975.
33. Floridi, L. The Method of levels of abstraction. *Minds Mach.* **2008**, *18*, 303–329.
34. Angius, N.; Giuseppe, P. The logic of identity and copy for computational artefacts. *J. Log. Comput.* **2018**, *28*, 1293–1322.
35. Linnebo, O. *Thin Objects an Abstractionist Account*; Oxford University Press: Oxford, UK, 2018.
36. Turner, R. *Computable Models*; Springer: Berlin/Heidelberg, Germany, 2009.
37. Piccinini, G. *Physical Computation: A Mechanistic Account*; OUP Oxford: Oxford, UK, 2015.
38. Kroes, P.; Meijers, A. The Dual Nature of Technical Artifacts—presentation of a new research programme. *Techne Res. Philos. Technol.* **2002**, *6*, 4–8.
39. Kroes, P.A. The dual nature of technical artifacts. *Stud. Hist. Philos. Sci.* **2006**, *37*, 1–4.
40. Kroes, P. Technological explanations: the relation between structure and function of technological objects. *Soc. Philos. Technol. Q. Electron. J.* **1998**, *3*, 124–134.
41. Kroes, P. Engineering and the Dual Nature of Technical Artefacts. *Camb. J. Econ.* **2010**, *34*, 51–62.
42. Anderson, N.G. Information Processing Artifacts. *Minds Mach.* **2019**, *29*, 193–225.
43. Kroes, P. *Technical Artefacts: Creations of Mind and Matter: A Philosophy of Engineering Design*; Springer: Berlin/Heidelberg, Germany, 2012.
44. Piccinini, G. Computation in Physical Systems. In *The Stanford Encyclopedia of Philosophy (Summer 2017 Edition)*; Zalta, E.N., Ed.; Stanford University: Stanford, CA, USA, 2020. Available online: https://plato.stanford.edu/archives/sum2017/entries/computation-physicalsystems/ (accessed on 29 May 2017).
45. Wittgenstein, L. *Remarks on the Foundations of Mathematics*, 2nd ed.; von Wright, G.H., Rhees, R., Anscombe, G.E.M., Eds.; MIT Press: Cambridge, MA, USA, 1983.
46. Boghossian, P. The Rule-following Considerations. *Mind* **1989**, *98*, 507–549.

47. Bagaria, J. Set Theory. In *The Stanford Encyclopedia of Philosophy (Spring 2020 Edition)*; Zalta, E.N., Ed.; Stanford University: Stanford, CA, USA, 2020. Available online: https://plato.stanford.edu/archives/spr2020/entries/set-theory/ (12 February 2019).
48. Benacerraf, P. What Numbers Could not Be. *Philos. Rev.* **1965**, *74*, 47–73.
49. Dedekind, R. *Gesammelte Mathematische Werke*; Fricke, R., Noether, E., Ore, Ö., Eds.; Vieweg: Braunschweig, Germany, 1969; Volume 1–3.
50. Linnebo, O.; Pettigrew, R. Two Types of Abstraction for Structuralism. *Philos. Q.* **2014**, *64*, 267–283.
51. Reck, E.; Schiemer, G. Structuralism in the Philosophy of Mathematics. In *The Stanford Encyclopedia of Philosophy (Spring 2020 Edition)*; Zalta, E.N., Ed.; Stanford University: Stanford, CA, USA, 2020. Available online: https://plato.stanford.edu/archives/spr2020/entries/structuralism-mathematics/ (accessed on 18 Novmber 2019).
52. Feferman, S. Constructive theories of functions and classes. In *Studies in Logic and the Foundations of Mathematics*; Logic Colloquium 78 (Mons, 1978); Elsevier: Amsterdam, The Netherlands, 1979; pp. 159–224.
53. Dybjer, P.; Palmgren, E. Intuitionistic Type Theory. In *The Stanford Encyclopedia of Philosophy (Summer 2020 Edition)*; Zalta, E.N., Ed.; Stanford University: Stanford, CA, USA, 2020. Available online: https://plato.stanford.edu/archives/sum2020/entries/type-theory-intuitionistic/ (accessed on 8 June 2020).
54. Barendregt, H. Lambda Calculi with Types. In *Handbook of Logic in Computer Science*; Oxford University Press: Oxford, UK, 1993; Volume 2, ISBN 0-19-853761-1.
55. Coquand, T.; Huet, G. The Calculus of Constructions. *Inf. Comput.* **1988**, *76*, 95–120.
56. Turner, R. Available online: https://www.academia.edu/1739390/Typed_Predicate_Logic (accessed on 20 June 2020).
57. Turner, R. *Constructive Foundations for Functional Languages Raymond Turner*; Mcgraw Hill Book Co Ltd.: New York, NY, USA, 1991.