Untyped λ-calculus with relative typing [⋆]

M. Randall Holmes

Math. Dept., Boise State University, Boise, Idaho, USA 83725

Abstract. A system of untyped λ-calculus with a restriction on function abstraction using relative typing analogous to the restriction on set comprehension found in Quine's set theory "New Foundations" is discussed. The author has shown elsewhere that this system is equiconsistent with Jensen's *NFU* ("New Foundations" with urelements) + Infinity, which is in turn equiconsistent with the simple theory of types with infinity. The definition of the system is given and the construction of a model is described. A semantic motivation for the stratification criterion for function abstraction is given, based on an abstract model of computation. The same line of semantic argument is used to motivate an analogy between the notion of "strongly Cantorian set" found in "New Foundations" and the notion of "data type"; an implementation of absolute types as domains of retractions with strongly Cantorian ranges is described. The implementation of these concepts in a theorem prover developed by the author is sketched.

1 Introduction

We discuss a system of λ-calculus which is, strictly speaking, untyped, but in which abstraction is limited by a relative typing scheme. The criterion used to limit abstraction is essentially the criterion of "stratification" used to limit set comprehension in Quine's set theory "New Foundations", in a version appropriate to a theory of functions. We have shown elsewhere (in [7]) that the system we present is equivalent in consistency strength and expressive power to the theory *NFU* + Infinity ("New Foundations" with urelements plus the Axiom of Infinity) introduced and shown to be consistent relative to *ZFC* by Jensen in [11].

We define the system and describe a model. We then present a semantic motivation for the system in terms of (very abstract) theoretical computer science ideas. We show how to reintroduce a notion of absolute type into this system, via the concept of "strongly Cantorian set" peculiar to Quine-style set theory, and argue for its appropriateness in terms of our semantic motivation. We conclude by sketching some features of an actual implementation of these ideas in a theorem prover.

We introduced systems of synthetic combinatory logic equivalent in strength to "New Foundations" and some of its fragments in [7] and [8]; the description of the kind of λ-calculus discussed here is implicit in those papers. An extensive

[⋆] Supported by the U. S. Army Research Office, grant no. DAAH04-94-G-0247

discussion of the set theory *NFU* with some discussion of the possible application of related systems of combinatory logic in computer science is found in our [9]; the definition of the λ-calculus we discuss here is briefly stated in that paper. Good references for the kind of set theory alluded to here are Rosser's [12] (old) and Forster's excellent [4] (current).

2 A Simply Typed λ-Calculus

It seems best to us to motivate the untyped system we will present by first introducing a typed version. The type labels in the typed system will be the natural numbers $0, 1, 2 \ldots$. Type $n + 1$ can be understood as the type $n \to n$ of functions from type n to type n, and each type n is identified with the type $n \times n$, the cartesian product of type n with itself. This very simple type structure is analogous to the type structure of the simple theory of types of Russell and Whitehead, as simplified by Ramsay, in which each type can be thought of as the power set of the preceding type. More complicated function and product types can be coded in this type system by the device of using (possibly iterated) constant functions of an object to represent it in higher types. For example, maps from type $n + 1$ to type n could be encoded as maps of type $n + 2$ all of whose values are constant functions (of type $n + 1$, used to represent their type n values). In this way, all the types of a more usual typed λ-calculus with all cartesian product and arrow types can be identified with subtypes of the more limited supply of types in this system.

Atomic terms are constants π_1, π_2 in each type above 0 , a constant eq in each type above 1 and a countable supply of variables of each type. If T and U are terms of type n, (T, U) is a term of type n (this expresses the identification of type n with $n \times n$). If T is a term of type $n+1$ and U is a term of type n, $T(U)$ is a term of type n. If T is a term of type n and x is a variable of type n, then $(\lambda x)(T)$ is a term of type $n+1$. (These two conditions express the identification of type $n + 1$ with type $n \to n$.)

Observe that the types of all subterms of a term can be inferred from the type of the larger term. We will take advantage of this and generally not express type superscripts.

The axioms of the theory are those of first-order logic plus the following non-logical axioms:

Proj: $\pi_i(x_1, x_2) = x_i$ for $i = 1, 2$
Prod: $(f, g)(x) = (f(x), g(x))$
Prodλ: $(\lambda x)(T, U) = ((\lambda x)(T), (\lambda x)(U))$
Surj: $(\pi_1(x), \pi_2(x)) = x$
Abst1: $(\lambda x)(T)(x) = T$
Eq: $eq(x, y) =$ if $x = y$ then π_1 else π_2
Nontriv: $\pi_1 \neq \pi_2$

Each of these axioms is "typically ambiguous"; any type which makes sense may be assigned to the two sides of an axiom (and then the types of subterms can be inferred as observed above). The axioms (Prodλ) and (Abst1) are axiom schemes with term parameters represented by T and U.

An axiom of extensionality

Ext: if for all x, $f(x) = g(x)$, then $f = g$

is certainly consistent with this system but will not be adjoined to it for our purposes. A model of this system is easily obtained by taking as type 0 any infinite set X and selecting a surjective map from $X \times X$ onto X to determine the pairing relation, then identifying type 1 with X^X and in general, if the set Y implements type n, using Y^Y to implement type $n + 1$. Axiom (Prod) determines the definition of the ordered pair in each type above 0. The axiom (Ext) is satisfied in this model, of course.

3 The Stratified λ-Calculus

Just as in Russell's type theory, the phenomenon of "typical ambiguity" presents itself. Each definable object and theorem has a precise analogue in each higher type (not necessarily in each lower type). Following Quine in his motivation of the set theory "New Foundations", we suggest that the types may be identifiable with one another, in which case the type indices can profitably be omitted from the theory entirely, obtaining a type free theory, but that only those instances of (Abst1) should be retained which involve λ-terms which would well-formed in the typed theory with a suitable assignment of types. Thus, for example, in the untyped λ-calculus each function f has the fixed point $(\lambda x)(f(x(x)))((\lambda x)(f(x(x))))$, but this cannot be typed with integer types as above, and so this λ-term is not associated with an instance of (Abst1) (in fact, it is not even a well-formed term under the rules stated below).

We present the term formation criteria of the type free theory in the form of two mutually recursive definitions.

Definition 1. An "atomic term" of the type free theory is any one of π_1, π_2, eq, or one of a countable supply of variables. A "well-formed term" of the type free theory is either an atomic term, a string of the form (T, U) or $T(U)$, where T and U are well-formed terms, or a string of the form $(\lambda x)(T)$, where T is a well-formed term and x is a variable which does not occur in T with "relative type" (see Definition 2) other than 0. The class of well-formed terms is the smallest class closed under these constructions.

Definition 2. Let T be a well-formed term of the type-free theory. The "relative type" of each occurrence of a subterm of T is an integer chosen as follows: the relative type of T with respect to itself is 0; if an occurrence of the subterm (U, V) has relative type n, the obvious occurrences of U and V are assigned relative type n; if an occurrence of the subterm $U(V)$ has relative type n, the obvious occurrences of U and V are assigned relative types $n + 1$ and n, respectively; if an occurrence of the subterm $(\lambda x)(U)$ has relative type n, then the relative types assigned to the obvious occurrences of x and U will be $n-1$ (x will not occur in U with a relative type other than 0 by Definition 1 above).

Note that negative relative types are possible; this exploits Wang's observation in [14] that the simple theory of types or the type theory described above can be extended to allow all negative types, which follows from a simple compactness argument.

We give an example. The term $x(x)$ is clearly well-formed. The relative type of the first occurrence of x in this term is 1; the relative type of the second occurrence is 0. Since the variable x occurs in the term $x(x)$ with a relative type other than 0, the term $(\lambda x)(x(x))$ is *not* well-formed (more generally, the term $(\lambda x)(f(x(x)))$ used in the construction of a fixed point of f is not well-formed).

We call λ-terms which are well-formed in the type-free theory "stratified" λ-terms, and we call this theory the stratified λ-calculus.

The axioms of the stratified λ-calculus are typographically the same as those given for the typed theory above, except, of course, that type indices are not to be supplied.

Theorem 3. *The stratified λ-calculus is consistent (if the usual set theory is consistent).*

Proof. A model for the stratified λ-calculus may be constructed as follows (this construction is given in the author's Ph. D. thesis [6]). Choose an infinite base set X as in the construction of a model of the typed theory, supplied with a bijection with $X \times X$ used for the representation of pairs of elements of X as elements of X. Let $\perp$ denote a fixed element of X (it is necessary for the pair $(\perp, \perp)$ to be represented by $\perp$). We indicate the construction of a cumulative system of types indexed by the ordinals based on X. If the type indexed with α is represented by a set Y, the type indexed by $\alpha+1$ will be represented by the set Y^Y of all functions from Y to Y. The difficulty with this is that Y^Y does not include Y; we resolve this by choosing a subset of type $\alpha+1$ at each stage to be identified with type α (details of this identification follow). Each element of type 0 is identified with its constant function in type 1. When type α has been embedded in type $\alpha+1$, we represent the object of type $\alpha+1$ associated with an object x of type α as x^+. Let f be an element of type $\alpha+1$; we identify it with the function f^+ in type $\alpha+2$ which takes each x^+ (of type $\alpha+1$) to $(f(x))^+$ and each element of type $\alpha+1$ not identified with any element of type α to $\perp$. This is how we embed type $\alpha+1$ in type $\alpha+2$. The cumulative nature of the type system thus defined allows the type indexed by a limit ordinal λ to be defined as the union of all types $\alpha < \lambda$. Note that an object of a limit type λ has already been assigned a value as a function of type $\alpha+1$ for each type $\alpha < \lambda$, and its values as a function in each type are identified by construction, so it has already been in effect assigned a value as a function of type $\lambda+1$ (with value $\perp$ everywhere off its natural domain). Thus, each type λ can be embedded in type $\lambda+1$, which is needed for the process of cumulative embedding of types described above to continue past limit ordinals. Essentially this technique of making function types cumulative was introduced by Dana Scott in connection with domain theory (see [13]). It is straightforward to verify that this cumulatively typed structure satisfies all the axioms of the typed theory, if the pair in higher types is defined in accordance with axiom (Prod).

Now consider this structure in a nonstandard model of set theory with an automorphism j moving a nonstandard ordinal α upward. The model of the stratified λ-calculus we construct will be type α in this structure. Pairing in the model will coincide with pairing in type α. Function application $f(x)$ for f and x objects of type α will be redefined as $j(f)(x)$. The effect of this redefinition is to cause the functions in type $j^{-1}(\alpha)+1$ to be assigned the extensions of the functions in type $\alpha+1$ (which are all the functions from type α into itself); objects of higher type will each be assigned the same extension as some element of this type. The denotation of $(\lambda x)(T)$ will be the unique object of type $j^{-1}(\alpha)+1$ with the correct extension, for each stratified term T (that each such object exists requires verification). It is straightforward to verify that all axioms of stratified λ-calculus hold in this model; proofs in [9] can be adapted to this purpose, for instance (originally derived from the model construction for *NFU* given by Maurice Boffa in [1]). $\square$

We recapitulate an example from above to make it clearer what is happening. The term $x(x)$ presents no difficulty (for x some fixed object); it will be represented by $j(x)(x)$, where j is the external automorphism. However, we cannot expect there to be a function $(\lambda x)(x(x))$ in the model of stratified λ-calculus, because it would be interpreted by the inverse image under j of a map taking each x in type α to $j(x)(x)$, and the latter map cannot be expected to exist (and indeed will not exist) because j is an external automorphism of the nonstandard model of set theory used.

We have demonstrated in [7] that the consistency strength of the stratified λ-calculus described here is exactly that of *NFU* + Infinity; the strength of the latter theory is known to be the same as that of the theory of types with infinity. Stronger extensions of these theories are possible, paralleling the strong extensions possible for the usual set theory.

Note that the model is NOT extensional. The type-free theory with (Ext) has the same strength as Quine's "New Foundations" with full extensionality, whose consistency remains an open question. But the non-extensional version is mathematically entirely adequate for applications. A canonical function with each extension is available in type $j^{-1}(\alpha)+1$ in the model; it happens that there are infinitely many additional objects with each extension in the higher types. A disconcerting feature of "New Foundations" is that the Axiom of Choice can be disproven; this is not the case in *NFU* or in the stratified λ-calculus without extensionality. The model of the stratified λ-calculus constructed above will satisfy AC (suitably expressed) if the underlying set theory satisfies AC.

Axiom (Abst1) can be used to define a notion of β-reduction (applicable only to stratified λ-terms); the fact that the theory with integer types is a subsystem of the usual typed λ-calculus with arrow types and cartesian products, mod an identification of types, allows us to see that this notion of reduction is Church-

Rosser and strongly normalizing. A direct proof of this is given in the author's thesis [6]. The fact that the theory with integer types is a subsystem of the usual typed λ-calculus with arrow types and cartesian products also makes it clear that the stratified λ-calculus is very weak, considered as a programming language. It can be strengthened by the addition of a stratified iteration or recursion combinator (acting on Church numerals) (I thank a referee for bringing the need for this comment to my attention).

It is possible to replace the axiom scheme (Abst1) with finitely many instances of the scheme and obtain a system of synthetic combinatory logic, in which the variable binding operator is eliminated. We introduce a new term construction: if T is a term, $\mathrm{K}[T]$ is a term (the constant function of T), and if $\mathrm{K}[T]$ is assigned relative type n, T is assigned relative type $n-1$. New atomic terms Abst and Λ are introduced. The axioms used to replace (Abst1) are

Const: $\mathrm{K}[x](y) = x$
Abst2: $\mathrm{Abst}(f)(g)(x) = f(\mathrm{K}[x])(g(x))$
Lambda: $\Lambda(x)(y) = x(y); \Lambda(\Lambda(x)) = \Lambda(x); \Lambda(x, y) = (\Lambda(x), \Lambda(y))$
Ext2: if $f(x) = g(x)$ for all x, then $\Lambda(f) = \Lambda(g)$

The interpretation of $\mathrm{K}[T]$ is obvious; Abst is a variation of the S combinator which respects stratification; Λ is the canonical retraction $(\lambda f)(\lambda x)(f(x))$ sending each object to a uniquely determined object with the same extension, for which special axioms are needed to preserve the weak extensionality implicit in the fact that substitutions for terms containing variables can be made into λ-terms. This synthetic combinatory logic also has a typed version with integer types as above. The notion of relative type for subterms of terms of the synthetic theory is defined as in Definition 2, except that the clause for λ-terms is replaced by a clause to the effect that if an occurrence of $\mathrm{K}[U]$ in a term T is assigned type n relative to T, then the obvious occurrence of U is assigned relative type $n-1$.

In our paper [7], we proved the following abstraction theorem:

Theorem 4. *For each term T of the language of the synthetic theory and variable x which occurs in T with no relative type other than 0, there is a term "$(\lambda x)(T)$" in which the variable x does not occur such that $(\lambda x)(T)(x) = T$ is a theorem of the synthetic theory.*

Note that the term formation rules and axioms of the synthetic theory make no reference to relative type at all; relative type is introduced only in order to state the abstraction theorem above. This is analogous to the situation with the set theories: *NF* or *NFU* can be finitely axiomatized using finitely many instances of stratified comprehension (so not mentioning stratification in the definition of the theory) from which stratified comprehension can be proven as a meta-theorem. Such an axiomatization was first presented by Hailperin in [5]; we give a more accessible one in [9].

4 A Semantic Motivation for the Stratified $\boldsymbol{\lambda}$-Calculus

We introduce an (extremely abstract) model of computation in terms of which we will give a semantic motivation for the stratification criterion for functional abstraction (it is worth noting that this is also an indirect semantic motivation for the set theories like NF, which are often regarded as motivated purely by a "syntactical trick").

Our abstract machine has an infinite set X of "addresses". There is a bijection pair:$X \times X \to X$: we use pair(x, y) to represent the pair of addresses (x, y). We refer to functions from X into X as "abstract programs"; the "state" of our machine is specified by a function state$X \to X^X$; state(x) may be thought of as the program stored in address x.

When x and y are addresses, we use (x, y) and $x(y)$ to abbreviate pair(x, y) and state$(x)(y)$, respectively. We introduce $(\lambda x)(T)$, for each term T and variable x, to represent an address (if there is such an address – an arbitrary one is chosen if there is more than one) such that $(\lambda x)(T)(x) = T$ for each address x. λ-terms can be thought of as program specifications.

We now consider the "program" $\Delta = (\lambda x)(x(x))$ as a sample of the kind of program which is ruled out by the stratification criterion for function abstraction. We suggest that there is an a priori reason to rule out any obligation to provide a program with specification Δ, based essentially on the notion of security of abstract data types. Moreover, this a priori reason, considered in full generality, motivates precisely the stratification criterion for function abstraction.

In our model of programming, programs are stored in and manipulated by reference to addresses. An abstract program is simply a function from X into X; but a program as implemented on our abstract machine has the additional feature of being stored in a particular address. Security of the data type "program" as represented on our machine requires that when we manipulate an address as representing a program, we use no information except information about its extension as a function. But the program Δ tells us to take the program stored in address x and apply it to address x; here we are expected to consider x both as a program and as the address in which that program is stored. A formal way to see that Δ violates the security of the program data type is to consider that a permutation of the assignments of programs to addresses could send the program stored in x to a new address y, and that the new value of $y(y)$ could differ from the old value of $x(x)$, although the new extension of y would be precisely the old extension of x. (A very precise result illuminating the relationship between stratification and permutations in a set-theoretical context is found in Forster's [4] (p. 87); similar reasoning is applied to typed λ-calculus in [3]) A preliminary version of the criterion which excludes program specifications like Δ would assert that if a parameter is used to represent an abstract program we should not have access to information about its identity as an address and vice versa.

The criterion can be sharpened by observing that there are more than two abstractions implemented by addresses on our machine: each address is an address and represents an abstract program via the state function; but one can also interpret the values of the abstract program represented by an address as abstract programs themselves, and thus interpret an address as a function from abstract programs to abstract programs. In this way we obtain a sequence of abstractions which are implemented by addresses on our machine precisely analogous to the sequence of types denoted by integers above; the objects at each level of abstraction are functions from the set of objects of the previous level of abstraction into itself.

The criterion restricting program specifications (λ-terms) is that each parameter in a specification should appear as representing an object at just one level of abstraction in this scheme; and this is precisely the criterion of stratification. To exploit the fact that the same address represents objects at two different levels of abstraction would be a violation of the security of the representations of the various levels of abstraction. The treatment of pairing has not been discussed, but it is precisely analogous to what is done above; it is technically tricky to see that axiom (Prod) and similar axioms are reasonable assumptions (that they are practicable is verified by our model construction above).

The failure of extensionality in the stratified λ-calculus as modelled above does not compromise the data type security argument. If we suppose that multiple addresses on our machine represent the same program, we must conclude that the proper identity criteria for addresses (considered as programs, as they are at all levels of abstraction except level 0) must be extensional; addresses are to be identified if they contain the same abstract program. We nonetheless end up with failure of extensionality when we attempt to interpret programs whose extensions do not respect this relation of identity of extension (which send diverse objects with the same extension to objects of different extensions); such programs must be assigned extensions which do respect the new identity criterion, presumably extensions already implemented by programs respecting the identity criteria, and will correspond to the many non-canonical programs with each extension found in the model described above. An example of a method for making such assignments would be to assign a given program a default value $\perp$ at each "value" (extension) at which it originally returned values with different extensions. The argument sketched in this paragraph is based on Marcel Crabbés proof that the theory *SC* with the axiom scheme of stratified comprehension alone interprets *NFU* (with its apparently stronger form of weak extensionality), given in [2].

As in the set theory "New Foundations" and its variants such as *NFU*, all objects are actually of the same kind, but abstraction (in set theory, set comprehension) is limited by a scheme of relative typing. The idea that all objects are ultimately of the same sort, but that they should be treated in abstractions (program specifications) as belonging to specific roles, is a very reasonable general outlook in computer science, where all objects, of whatever intended type, are in fact implemented as sequences of binary digits, but the type of the intended referent of an object controls the ways in which it can be manipulated in a program.

Of course, it is not necessary to use relative typing schemes in programming; it is not strictly necessary to acknowledge overtly that all objects are of one underlying sort. But it is often tempting, for it is often the case that an operation which is useful on one type is actually useful on objects of many related types or even of all types. A certain amount of polymorphism is implemented in our scheme; a very simple example is the presence of the identity function $(\lambda x)(x)$ which acts on all objects whatsoever. More useful further examples of polymorphic objects definable in stratified λ-calculus are the Church numerals "n" $= (\lambda f)(\lambda x)(f^n(x))$ for each natural number n. The polymorphism provided by this scheme is somewhat limited, however, by comparison with the kind of polymorphism provided in the second-order polymorphic λ-calculus, or, to give a concrete example, in the type system of the computer language *ML*. The motive behind polymorphic type systems is similar to the motive behind the attempt to adopt a relative typing scheme. We will discuss the implementation of data types usual to computer science below; we do not take them to be implemented by the very simple scheme of relative types used in the definition of stratification!

5 Interpreting Data Types in the Stratified $\boldsymbol{\lambda}$-Calculus

The scheme of relative typing which underlies the definition of stratification is a very simple type structure, not adequate for the needs of computer science. We present a second notion of absolute type, essentially orthogonal to the notion of relative type used in stratification, which corresponds to a fundamental concept of set theory in the style of "New Foundations" which has no analogue in the usual set theory *ZFC*.

We briefly describe the analogous situation in set theory. Cantor's theorem in the usual set theory establishes that the power set (the collection of all subsets) of a set A must have a larger cardinality than A. If this argument is applied to the universal set V, we obtain "Cantor's paradox"; the power set of the universe, the collection of all sets, is shown to be larger than the universe! Of course, the conclusion to be drawn from this is that there is no such set V in the usual set theory. In *NF* and related set theories, there *is* a universal set, so this argument must break down somehow. The outcome is best understood by referring to the theory of types, where A and its power set are objects of different types which cannot be compared, and the result analogous to Cantor's theorem proves instead that the set of all one-element subsets of A (the set of singletons of elements of A) is smaller in cardinality than the power set of A. The rather odd result in *NF* and the related set theories is that the universe V is larger than the set of all singletons. The map $(\lambda x)(\{x\})$, which would provide a bijection between these two sets, is seen not to exist, which is not especially surprising, since its definition is unstratified.

But Cantor's theorem will hold in its original form for any set A which has the same cardinality as its image under the singleton operation. Such sets are said to be "Cantorian". A set A is said to be "strongly Cantorian" if the restriction of the singleton operation to A defines a function; such a set is certainly Cantorian. The properties "Cantorian" and "strongly Cantorian" are unstratified and do not define sets.

The "class" of strongly Cantorian sets includes all sets of standard finite size. If the assertion that the set of natural numbers is strongly Cantorian (it is provably Cantorian) is adopted as an axiom, then all finite sets are strongly Cantorian as well. This axiom, Rosser's Axiom of Counting, is known to be consistent with *NFU* (and strengthens it essentially); its analogue will hold in our model of stratified λ-calculus iff the automorphism j used in the construction fixes each natural number. The "class" of strongly Cantorian sets is closed under cartesian product, power set, and the construction of function spaces. Strongly Cantorian sets are "small" in relation to the big sets like V; it has been suggested that they are the analogues of the small sets studied in the usual set theory, whose criterion for sethood is "limitation of size". This position is strengthened by the fact that stratification restrictions can be subverted for variables restricted to strongly Cantorian sets; the type of any reference to such a variable can be shifted by replacing references to the object with references to its singleton (we will show how this is done in the context of stratified λ-calculus below). We have discussed in our paper [9] and our unpublished [10] how this analogy can be shown to be precise under suitable assumptions.

We suggest, and will motivate in terms of our abstract model of programming, the proposition that the notion "strongly Cantorian set" is a reasonable analogue in our scheme to the notion of "data type". We see above that the usual types of booleans and natural numbers can be accommodated, and that the class is closed under the most obvious type constructors.

We represent sets in our theory as ranges of retractions (maps t such that $t(t(x)) = t(x)$ for all x). The best analogue of the singleton map in our theory is the operation which takes each x to $K[x] = (\lambda y)(x)$, its constant function. A retraction t represents a strongly Cantorian set iff there is a map K_t such that $K_t(t(x)) = K[t(x)]$ for each x. Notice that the relative type of x can be changed by applying this equation, if we know that x belongs to the range of t (and so is fixed by t); this allows the subversion of the stratification restrictions for variables restricted to strongly Cantorian sets. For example, if t is a retraction whose range is strongly Cantorian, we can define $\Delta_t = (\lambda x)(t(x)(t(x)))$ as $(\lambda x)(K_t^{-1}(K[t(x)])(t(x)))$; the first λ-term requires definition because it is unstratified and so ill-formed; but the trick of replacing the occurrence of $t(x)$ at relative type 1 with $K_t^{-1}(K[t(x)])$ lowers the type of the first occurrence of x to 0, making the term stratified. An occurrence of $t(x)$ can be raised in type by replacing it with $K_t(t(x))(0)$ (the use of 0 is just as a convenient constant). The net effect is that variables with a type retraction t applied to them (effectively, variables restricted to the range of t) do not need to satisfy the stratification restrictions, if t is a retraction with strongly Cantorian domain.

What does the existence of K_t tell us about the range of t in the context of our abstract model of computation? It allows us to correlate $K_t(t(x))$, with $K[t(x)]$; if the occurrence of $t(x)$ is to be understood as taking the role of an address on the right, it must be understood as taking the role of an abstract program on the left. In other words, K_t is a representation as a function of the correlation of objects in the range of t considered as programs and as addresses, which is exactly what the stratification criterion operates to prevent in general. Metaphorically, K_t tells us exactly how the programs in the range of t are stored in the "memory" of our abstract machine; a class of objects which we know how to store in memory seems like a reasonable abstract description of the concept "data type". Concretely, we have seen above that commonly used data types are subsumed under this notion. Type constructors are represented by operations (not always functions) on retractions in a straightforward way. Notice that types themselves, being functions, are first-class objects in this scheme.

We are aware that there is nothing novel about the use of retractions to represent types; this is a very commonly used technical device. The (hoped-for) novelty is all in the use of the notion of "strongly Cantorian set", in the guise "retraction with strongly Cantorian range".

We exhibit the construction of arrow types. Given a retraction s and a retraction t, we can construct a retraction $s \rightarrow t$ whose domain is the set of functions from the domain of s to the domain of t easily: $s \rightarrow t = (\lambda f)(t \circ f \circ s)$ (composition of functions is a stratified construction). An important thing to note is that the relative type of $s \rightarrow t$ is one higher than that of s and t; thus, the arrow type constructor is not a function (the cartesian product constructor does turn out to be a function). This construction is not presented as being original in any way; we are simply pointing out its features in the context of stratified λ-calculus.

We have investigated the representation of more complex type systems, including the second-order polymorphic λ-calculus mentioned above. To achieve polymorphism of this kind, in which objects can be constructed with type parameters, it is necessary to have an actual strongly Cantorian set of types to which we restrict ourselves; the collection of all absolute types (strongly Cantorian sets) is not a set at all, much less a strongly Cantorian set, so cannot be an absolute type. One reason that the set of types in such a scheme must be strongly Cantorian is that the arrow type constructor is type-raising; on a strongly Cantorian set of types, it can be represented by a function, but not on the whole domain of types. It seems more natural in this context to have a hierarchy of ever more inclusive notions of type, analogous to the universes of Martin-Löf type theory (though certainly different in detail).

We think that it is interesting that the notion of "strongly Cantorian set", peculiar to the Quine-style set theories, may turn out to have a useful role in our extension of this set theoretical paradigm to computer science. We also think that it is interesting to see the notion of type break into two orthogonal parts, the relative type scheme restricting the formation of global abstractions and the representation of data types as local domains on which we have additional information which allows us to subvert these global restrictions.

6 The Use of these Concepts in a Theorem Proving Project Sketched

We now discuss the ways in which we have used these concepts in the actual implementation of an equational theorem prover (a technical report on this project should be available from the author fairly shortly). This project was originally not intended to implement stratified λ-calculus; but it turned out that implementing stratification was the easiest way to modify an earlier version of the prover to support higher-order reasoning.

The prover we have implemented avoids the use of bound variables; there are no λ-terms or other variable-binding operations. The situation where stratification is applied is in definitions of functions and operators. The syntax of the input language allows unary prefix and binary infix operators. Each operator is assigned a relative type for each of its arguments (only if these are zero is the operator actually a function). For example, the operator @ (function application) has relative type 1 for its first argument and relative type 0 for its second argument. If the user attempts to define a function $\mathit{\Delta}$ by the equation $\mathit{\Delta}@x = x@x$, the prover will attempt to type all subterms of the expression and discover that x cannot be consistently assigned the same type everywhere, so the definition will be rejected: if the term $x@x$ is assigned type 0, we must assign type 1 to the first occurrence of x and type 0 to the second occurrence of x, and when the same variable must be assigned two different types the stratification procedure fails. It is also possible to define new prefixes and infixes which have different relative types from their arguments (and so are not genuine functions); the operators which form singletons and constant functions or the arrow type constructor operating on retractions would be examples. Their definitions also require attention to stratification (the technical term for such operators is "stratified but inhomogeneous").

An amusing fact about the implementation of stratification is that it is a matching operation. A usual application of matching checks whether each variable in one term is structurally correlated with the same subterm of another term, checking for parallelism of structure and consistency of assignments to the same variable everywhere in the term. The stratification checker in the prover assigns an integer to each subterm of a term (this always succeeds), but needs to check further whether the assignments of integers to occurrences of variables are consistent; it shares essential code with the matching functions of the prover. Thomas Forster has made the same observation (oral communication).

If only stratified equational axioms are used, the prover can be understood as working in the typed theory described above, with a system of type inference replacing explicit typing, which allows polymorphic function definitions introducing the defined function in all types simultaneously; this parallels the mathematical practice of those who work in Russell's type theory. However, the absence of explicit typing makes it tempting to introduce unstratified axioms (an example would be the defining equation of any K_t for a retraction t with strongly Cantorian domain). The subversion of the type system by introducing K_t's has been seen above to be useful in allowing more general function abstraction for terms with variables restricted by application of retractions to strongly Cantorian domains; once unstratified equations are introduced, we are working in the stratified λ-calculus (given the absence of bound variables, we may to be said to be working in the equivalent synthetic system).

Automatic features of the prover allow the absolute typing scheme using retractions to be largely inferred rather than explicit as well; occurrences of retractions can be removed where unnecessary (where their presence can be restored by a type inference) automatically and reintroduced equally automatically when needed; they can be treated as if they were type labels in appropriate contexts. Type inference for the usual constructed types can be implemented in a straightforward fashion. Since types in the sense of strongly Cantorian domains are represented by retractions, they are first-class objects in this system, which allows definition of new type constructors and implementation of extended type inference schemes by the user.

7 Acknowledgements

I appreciate the financial support of the U. S. Army Research Office for my automated theorem proving research. I appreciate the comments of the anonymous referees of this paper, whose specific concerns I have addressed in several places. A general concern raised by more than one referee is addressed here: the motive of this paper is to suggest that results originally proved in the completely theoretical area of *NF*-style set theory may have practical applications (or at least, applications closer to practical considerations); in this kind of paper it is not surprising that the main results are proved elsewhere and also that there is perhaps too much philosophical speculation of a vague nature. The aim of bringing these matters to the attention of a more practically minded community is to diminish this vagueness, and I thank the referees for helping me to make this possible.

References

1. Boffa, M. "*ZFJ* and the consistency problem for *NF*", in *Jahrbuch der Kurt Gödel Gesellschaft*, 1988, pp. 102-6.
2. Crabbé, M. On *NFU*. *Notre Dame Journal of Formal Logic*, vol. 33 (1992), pp. 112-119.
3. Forster, T. E. "A semantic characterization of the well-typed formulae of λ-calculus". *Theoretical Computer Science*, vol. 110 (1993), pp. 405-418.
4. Forster, T. E. *Set theory with a universal set*, Oxford logic guides no. 20. Clarendon Press, Oxford, 1992.
5. Hailperin, T. "A set of axioms for logic". *Journal of Symbolic Logic*, vol. 9 (1944), pp. 1-19.
6. Holmes, M. R. "Systems of combinatory logic related to Quine's 'New Foundations' ", Ph.D. thesis, State University of New York at Binghamton, 1990.
7. Holmes, M. R. "Systems of combinatory logic related to Quine's 'New Foundations' ". *Annals of Pure and Applied Logic*, vol. 53 (1991), pp. 103-133.
8. Holmes, M. R. "Systems of combinatory logic related to predicative and 'mildly impredicative' fragments of Quine's 'New Foundations' ". *Annals of Pure and Applied Logic*, vol. 59 (1993), pp. 45-53.

9. Holmes, M. R. “The set-theoretical program of Quine succeeded, but nobody noticed”. *Modern Logic*, vol. 4, no. 1 (1994), pp. 1-47.
10. Holmes, M. R. “Strong axioms of infinity in *NFU*”, preprint.
11. Jensen, R. B. “On the consistency of a slight (?) modification of Quine’s *NF*”. *Synthese*, vol. 19 (1969), pp. 250-63.
12. Rosser, J. B. *Logic for Mathematicians*. McGraw-Hill, reprinted (with appendices) by Chelsea, New York, 1978.
13. Scott, Dana. “Continuous lattices”, in *Springer Lecture Notes in Mathematics*, no. 274, pp. 97-136.
14. Wang, H. “Negative types”. *Mind*, vol. 61 (1952), pp. 366-8.

This article was processed using the LaTeX macro package with LLNCS style