Preamble

In 1988, in [3], I introduced what was then a trick by which the strength of a recursion scheme over the natural numbers could be limited. The idea is to separate the object naturals – the inputs and the output of the computed function – from those natural that are used merely to control the inner workings of the computing algorithm. In this way diagonalization or ‘self reference’ is obstructed. In due course other people used a similar technique (sometimes without really appreciating what was being done). Eventually the trick metamorphed into a technique, here called tiering but sometimes called stratification.

In the autumn of 1998 I gave a series of talks to the Logic Group at the University of Leeds in which I outlined this general method of tiering. Those talks were written up as the Technical Report [4]. (That document is not worth reading. It contains several mistakes. This despite some members of the audience knowing more about certain aspects of the subject than I do.)

In due course I wrote up a shortish survey article [6]. By its very nature that document is not meant to be a definitive account. For one thing it is still far too early to have have such an account. Also such a document would be far to long to publish in the Bulletin. However, before, during, and after writing that paper I was compiling a set of notes on the topic. As always, these notes were primarily for myself, but I have no objection to letting them loose on the general mathematical public. These are those notes. It is citation [18] of [6].

This document is much longer than I originally intended. However, rather than try to cut it down by omitting some material, I have left it as it finished up. This means that you probably shouldn’t read it page by page.

I suggest the best way to read this document is to start by looking at [6]. As you find bits of that paper that you don’t understand or would like to know more about, then dip into this document at the appropriate place.

Parts of these notes are quite well organized, and other parts may seem more of a ramble than an account. There are parts where I have not been able to sort out the material as it should be done, or even where I don’t know the relevant results. I time I may revise the document to rectify its deficiencies. At the top of this page you will see banner saying

This version produced on ***

with a date. The original document was put on my web-page on DATE. This should indicate whether any changes have been made.
1 Introduction—To be done

Things to sort out

- In the real world, for \( l \)-placed functions we flit between curried and uncurried at will. Thus we might write either of

\[
fx_l \cdots x_1 \quad f(x_l, \ldots, x_1)
\]

for a typical value of such a function.

Within a system we always work with fully curried versions.

- A 0-placed function is a constant. We tacitly ignore this problem.

- \( x' \) convention for naturals

- I, we, you.

- Questions and suggestions for further work section.
1.1 THEOREM. (Bellantoni and Cook) A tiered version of Primitive Recursive Arithmetic produces only the class \( \mathcal{H} \) of humble functions.

(Leivant) A tiered version of Gödel’s \( T \) produces only the class \( \mathcal{L} \) of latent functions.

<table>
<thead>
<tr>
<th>Numeric = natural number (does not include reals)</th>
</tr>
</thead>
<tbody>
<tr>
<td>In the first instance we are concerned with first order natural number functions of a type</td>
</tr>
<tr>
<td>( \mathbb{N}^l \rightarrow \mathbb{N} )</td>
</tr>
<tr>
<td>for some ( l &lt; \omega ). For our purposes it is often more convenient to view these in Curried form, that is, receiving the arguments in sequence rather than as a tuple. We set</td>
</tr>
<tr>
<td>( \mathbb{N}(0) = \mathbb{N} )</td>
</tr>
<tr>
<td>( \mathbb{N}(1 + l) = \mathbb{N} \rightarrow \mathbb{N}(l) )</td>
</tr>
<tr>
<td>(for ( l &lt; \omega )) to generate a sequence ( (\mathbb{N}(l) \mid l &lt; \omega) ) of functions spaces each of which has the form</td>
</tr>
<tr>
<td>( \mathbb{N} \rightarrow \cdots \rightarrow \mathbb{N} \rightarrow \mathbb{N} )</td>
</tr>
<tr>
<td>where, for ( \mathbb{N}(l) ), there are ( l ) argument types in sequence and one target type. A function</td>
</tr>
<tr>
<td>( f : \mathbb{N} \rightarrow \cdots \rightarrow \mathbb{N} \rightarrow \mathbb{N} )</td>
</tr>
<tr>
<td>can receive arguments</td>
</tr>
<tr>
<td>( fx_1, \ fx_1x_{l-1}, \ \cdots \ fx_1 \cdots x_2, \ fx_1 \cdots x_2x_1 )</td>
</tr>
<tr>
<td>in sequence to produce several auxiliary functions and an eventual value ( fx_1 \cdots x_1 ).</td>
</tr>
<tr>
<td>Of course, as well as first order functions we also use some higher order functions. For instance, each recursor is such a higher order function. The syntactic mechanism we employ must be flexible enough to handle and such higher order gadget we may need.</td>
</tr>
</tbody>
</table>

[Held in 165../05-.../002.. Last changed April 13, 2006]

2 Numeric applied \( \lambda \)-calculi

One of the themes of these notes is that information about the complexity of a function can be gleaned from an inspection of the syntax used to specify that function. This is not a profound remark, but neither is it trite. However, we do wish to turn the observation into something of a method. To do that we need to set up the specifying syntactic machinery with some precision, more than normal in the day to day business of mathematics. Again this is not a profound observation. Or rather, it shouldn’t be, but remember Peano had a bit of trouble getting a similar, and deeper, message across.

In this section I will set up part of the syntactic machinery of an arbitrary tiered system \( \lambda T \).

For those of you who know about these things \( \lambda T \) is an applied \( \lambda \)-calculus generated from certain kinds of primitive types and constants. These are given in Definitions 2.3 and 2.7 below. In this section I deal only with the typing discipline, the derivation system. In
the next section I deal with the orthodox semantics, and then I deal with the computation mechanism, the reduction relation, in the section after that. You might want to skip this and the next two sections until something crops up that you don’t understand.

For those of you who don’t know about these things, let me warn you that it can be rather dull and seem a bit like nit-picking to no great purpose. When you first meet this stuff it can look a bit mysterious until suddenly it clicks, and then it seems mostly trivial. As you read this and the next two sections remember one thing: it’s about handling syntax in an efficient manner, nothing more.

Ok? Let’s get stuck in!

2.1 DEFINITION. The full function and product hierarchy $\mathcal{F}(N)$ over $N$ is the least family of sets which contains $N$ and also contains

$$S \rightarrow T \quad L \times R$$

whenever it contains $S, T, L, R$.

We refer to a member of $\mathcal{F}(N)$ as a housing space or a space for short.

Thus, in this document, a space or housing space is a function space or a product space built up from $N$ by a finite number of applications of the indicated constructions. Of course, this terminology is just a convenience, and shouldn’t be used in a wider context when there may be other kinds of spaces around.

We are interested in the members of these spaces.

The members of $N$ are just $0, 1, 2, 3, \ldots$, the natural numbers.

The members of the next two generated spaces

$$N \rightarrow N \quad N \times N$$

are, respectively, the 1-placed first order functions

$$f : N \rightarrow N$$

and the pairs $(l, r)$ of natural numbers.

The members of

$$N \times N \rightarrow N \quad N \rightarrow (N \rightarrow N) \quad (N \rightarrow N) \rightarrow N \quad N \rightarrow N \times N$$

are

- the 2-placed first order functions over $N$
- the 2-placed first order functions over $N$ in curried form
- certain kinds of functionals
- parallel pairs of 1-placed functions

respectively.

As we move up the hierarchy the housing spaces become more and more exotic. Rarely in mathematics do we meet the members of most of these spaces, but we still have to take account of them, if only to determine the boundary between the commonplace and the esoteric.

We want a precisely defined method of describing at least some of the members of the spaces in $\mathcal{F}(N)$. To produce that we first need a system of naming the spaces in $\mathcal{F}(N)$. That part is easy, we we simply mimic these constructions in syntactic form.
2.2 DEFINITION. The types (of an applied \(\lambda\)-calculus) are built up from a given stock of atoms using arrow (function space) and cross (product space) formation.

- Each atom is a type.
- If \(\pi, \sigma\) are types then so is \((\pi \to \sigma)\).
- If \(\sigma, \rho\) are types then so is \((\sigma \times \rho)\).

Here the brackets are part of the syntax; they ensure each type is uniquely parsed.

The type structure of an applied \(\lambda\)-calculus is determined entirely by its given stock of atoms. Definition 2.2 doesn’t say what these are, or can be (except, of course, there must be at least one of them). For a tiered system \(\lambda T\) these atoms are restricted as follows.

2.3 DEFINITION. The type structure of a tiered system \(\lambda T\) is determined by its atoms. There is always the monochrome atom \(N\) together with an indexed family \((N[i] \mid i \in I)\) of coloured atoms.

The monochrome atom \(N\) is the canonical name of the set \(\mathbb{N}\) of natural numbers. The index set \(I\) may be empty. In that case there are no coloured atoms, and we have a standard numeric applied \(\lambda\)-calculus. However, as well as \(N\) a tiered system can have other names for \(\mathbb{N}\). Thus as well as the standard monochrome natural numbers we also allow red, blue, green, vermilion, puce, mauve, \ldots natural numbers. The set \(I\) indexes the other colours used.

These coloured numbers are not just duplicates of the monochrome numbers. As we develop the systems you will see there may be some things that can be done with red numbers that can not be done with blue numbers, and green numbers are different again. The coloured natural numbers form part of a control mechanism for the monochrome natural numbers.

We write \(\xi\) for a typical atom, \(N\) or \(N[i]\).

The atoms and types are merely pieces of syntax. They are designed to name certain concrete spaces, the members of \(\mathcal{F}\mathcal{H}(\mathbb{N})\), the full hierarchy over \(\mathbb{N}\), but it is important not to confuse the naming syntax with the named space.

2.4 DEFINITION. Let \(\lambda T\) be a tiered system with colours \(I\). For each type \(\tau\) its intended or orthodox interpretation \([\tau]\) is generated by recursion over its structure. We set

\[
[N] = \mathbb{N} \quad [N[i]] = \mathbb{N}
\]

for each colour \(i \in I\), and

\[
[\pi \to \sigma] = [\pi] \to [\sigma] \quad [\sigma \times \rho] = [\sigma] \times [\rho]
\]

for all types \(\pi, \sigma, \rho\).

In other words the construction \([\cdot]\) passes through the types in a natural way. Notice the dual use of ‘\(\to\)’ and ‘\(\times\)’ here. One is part of the syntax, and the other is the set-theoretic constructor. This will not cause problems. More importantly, notice that a use of \([\cdot]\) loses the colour coding. There is information in the tiered (that is, coloured) syntax.
that is invisible in the intended meaning. Of course, we could produce a semantics for these types in any cartesian closed category with a natural number object. That is useful in more sophisticated developments, but is not needed here. We model $\lambda T$ entirely within the category of sets. Indeed, I am not sure if all the ramifications of a non-standard semantics for a tiered system have been worked out.

It is worth emphasizing the use product types. Often it is a matter of personal preference whether we use these or not. They do make the applied $\lambda$-calculus more expressive, but also involve more work. However, here some results are quite sensitive to the existence of products, and I will indicate when this is so. I suspect there are some aspects of the use of products which have not yet been uncovered.

(To be consistent, if we are going to use products then we ought to have another atom to name the empty product, the final object of the modelling category, which in our case is any singleton set. That will not be needed here, but it will be important if we ever look at a non-orthodox semantics.)

Types name housing spaces, the members of $\mathfrak{H}(\mathbb{N})$. However, we are more interested in the inhabitants of these housing types, the functions and pairs in the spaces. To name these inhabitants we use a syntactic category of terms. This is more complicated than the syntactic category of types. We first generate the raw terms and then we extract the well-formed terms.

Before I give the next definition let me sort out what may be a source of confusion.

A category is a mathematical structure consisting of objects and arrows with certain other features. They occur all over mathematics (sometimes without even being noticed). Here they can be used to model certain aspects of $\lambda$-calculi. However, as mentioned above, in these notes we need only the category of sets and functions (and to understand that you don’t even need to know what a category is).

A syntactic category is a collection of pieces of syntax with some common features. It is not a category made from syntax. The terminology ‘syntactic category’ is a bit unfortunate since it can be misunderstood. However, it seems to be the most common phrase used to mean ‘set of syntax’.

Here is the definition of the syntactic category of raw terms.

\textbf{2.5 DEFINITION.} The \textit{raw terms} (of an applied $\lambda$-calculus) are generated from a given stock of \textit{constants} and an unlimited stock of \textit{identifiers}.

- Each constant and each identifier is a term.
- If $q, p$ are terms then $(qp)$ is a term.
- If $x$ is an identifier, $\sigma$ is a type, and $r$ is a term, then $(\lambda x : \sigma . r)$ is a term.

The brackets are part of the syntax to ensure that each term is uniquely parsed.

As stated in this and Definitions 2.2, the brackets in types and terms are part of the official syntax. They ensure that each constructed string can be uniquely parsed. However, when displayed with all of its brackets on show a type or term can look unnecessarily cluttered. We thus use various informal convention for omitting brackets. There are also several other syntactic conventions which help to make a displayed type or term look neater. In order not to break up the flow of the development, I will put all these
conventions at the end of this section. However, some of these conventions may creep before they are explained, so keep your eyes open.

The raw terms are determined by the stocks of identifiers and constants. An identifier is sometimes called a variable, and we see the use of one in the compound \((\lambda x : \sigma . r)\). A tiered system \(\lambda T\) is partly determined by the choice of constants. I will say what these can be shortly at a more convenient point.

The generation of raw terms is a kind of free market economy in which all kinds of nonsense is allowed. (It is precisely this aspect which makes the untyped \(\lambda\)-calculus so powerful.) Many raw terms are grammatically incorrect.

For instance, in the construct
\[
q \ p \\
\hline
(qp)
\]
the idea is that \(q\) is a name of a function, \(p\) is a name of an input to the function, and then \((qp)\) is a name of the output from that function. Of course, for this to make sense there has to be a certain compatibility between \(q\) and \(p\).

To extract the meaningful terms we use a derivation system. This needs a bit of a preamble to get to it.

We introduce a few more syntactic categories.

A **statement** is a paired term \(t\) and type \(\tau\)
\[
t : \tau
\]
where \(t\) is the subject and \(\tau\) is the predicate \(\tau\). A **declaration** is a statement
\[
x : \sigma
\]
where \(x\) is an identifier. A **context** \(\Gamma\) is a list
\[
x_m : \sigma_m, \ldots, x_1 : \sigma_1
\]
of declarations. A context is **legal** if the declared identifiers \(x_1, \ldots, x_m\) are distinct. A **judgement**
\[
\Gamma \vdash t : \tau
\]
is a statement \(t : \tau\) in context \(\Gamma\). Here the gate ‘\(\vdash\)’ and the colon ‘\(\:\)’ are merely punctuation devices.

We wish to read this judgement as

Within the legal context \(\Gamma\), the well-formed term \(t\) inhabits the acceptable type \(\tau\)
but to do so we must provide a justification for this. In our case every type is acceptable, and a context is legal precisely when it declares distinct identifiers. It is the well-formedness of a term which requires some work. This is the job of the derivation system.

**2.6 DEFINITION.** A derivation
\[
(\nabla) \quad \Gamma \vdash t : \tau
\]
is a finite rooted tree of judgements grown according to the rules given in Table 1.

A term \(t\) is **well-formed** (in a context) if it is the subject of the root judgement (in that context) of some derivation.
Table 1: The construction rules for derivations

This definition deserves a longer explanation.

First of all, what is a finite rooted tree? Table 2 gives a particular example. It’s a finite poset with a unique least node, its root, and finitely many maximal nodes, its leaves. The crucial property is that for each leaf \( L \), the lower section below \( L \) is linearly ordered. The trees we meet here have at most binary splitting. Thus each non-leaf node is created, from above, in one of two ways.

In the example of Table 2 most non-leaf nodes are created by a binary splitting; there are just three nodes of the other kind.

A derivative is such a tree where each node is a judgement. We meet a derivation with the same shape as that of Table 2 a little later. The most important aspect of a derivation is the internal relationships between the various judgements. These are designed to organize the syntax of terms in a coherent fashion. Let’s look at these restrictions.

The raw terms are generated, in part, from the given stock of constants. Each such constant \( k \) comes with an attached type \( \kappa \), its housing type, and the statement

\[
k : \kappa
\]

is an axiom. For each legal context \( \Gamma \) the judgement

\[
\Gamma \vdash k : \kappa
\]

is an Axiom. (The distinction between axiom and Axiom may seem a bit pedantic, but it does have its uses. An Axiom is an axiom in context, so an axiom can be used in many different Axioms.)

Each such Axiom is allowed as a derivation of just one node. This explains the first rule of Table 1.

The only other kind of single node derivation is a Projection. Let \( \Gamma \) be any legal context, and let \( x : \sigma \) be a declaration occurring in \( \Gamma \). Thus \( \Gamma \) has the form

\[
\Lambda, x : \sigma, \Sigma
\]
Table 2: An example finite rooted tree

where \( \Lambda \) is the left part and \( \Sigma \) is the right part of \( \Gamma \), and either or both of these may be empty. This explains the second rule of Table 1.

Each derivation is a finite tree of judgements with a single root and a certain (finite) number of leaves. Each leaf is either an Axiom or a Projection, as described above. Each derivation is generated from its leaves to its root by a number of applications of

Weakening    Introduction    Elimination

the three rules of construction.

Let’s look first at the only binary splitting rule, Elimination.

Suppose we have two derivations

\[
\begin{array}{c}
\\vdash q : \pi \rightarrow \tau \\
\\vdash p : \pi
\end{array}
\]

with the same root context \( \Gamma \) and a root predicate compatibility as indicated by the double occurrence of the type \( \pi \). In such circumstances we are allowed to form

\[
\begin{array}{c}
\\vdash q : \pi \rightarrow \tau \\
\\vdash p : \pi
\end{array} \quad \Rightarrow \quad \Gamma \vdash (qp) : \tau
\]

\((E)\)

as a larger derivation. This use of Elimination mimics the term construction

\[
\frac{q \quad p}{(qp)}
\]

above but, of course, there is more going on in the derivation. In particular, the rule insists that the type of the input \( p \) matches the type of the source of the function \( q \).
Here we have labelled the use of Elimination

\[ (E) \]

by ‘E’. However, since Elimination is the only binary splitting rule, this labelling is hardly ever necessary. The two remaining rules

\[
\begin{align*}
\text{Weakening} & : \quad \cdot \quad \cdot \quad (W) \\
\text{Introduction} & : \quad (I)
\end{align*}
\]

both have a single numerator

\[ (W) \quad (I) \]

and so with these a label can be useful.

Introduction merely moves the right-most declaration across the gate to form an abstraction. The predicate is altered accordingly.

Weakening allows the creation of a new declaration without changing the statement. However, note that the new context \( \Gamma, x : \sigma \) must be legal. This means that \( x \) is fresh for \( \Gamma \), in other words it is not already declared in \( \Gamma \). The Weakening rule makes the system much more flexible, it avoids some silly problems with the renaming of bound identifiers. However, using Weakening means that some judgements have many derivations. Thus, at least initially, we should distinguish between a derivation and a derivable judgement.

Weakening is the only rule that enlarges the context, and the resulting context must be legal. Thus, after a few moment’s thought we see that in a derivation each occurring context is legal.

As in Table 1 we often write

\[
(\nabla) \quad \Gamma \vdash t : \tau
\]

to indicate that \( \nabla \) is a derivation with root judgement \( \Gamma \vdash t : \tau \).

If you have already seen a formal derivation in an applied \( \lambda \)-calculus then I suspect you don’t want to see another. If you haven’t seen one, then you should. It will help you to understand how the derivation system works. Shortly I will give an example in a tiered system \( \lambda T \), but to do that I must first say what axioms are allowed.

Earlier I told you to keep your eyes open, and you may already have missed a couple of uses of the syntactic conventions. Several more occur in the following. In particular, you should make sure you understand the dual use of \( (\cdot)' \) as an abbreviating device. Look at Definition 2.13 and the remarks following that.

2.7 DEFINITION. For a tiered system \( \lambda T \) the constants \( k \) and associated housing axioms \( k : \kappa \) are of a limited form.

(Pairing gadgets) For each pair \( \sigma, \rho \) of types there are three pairing gadgets.

\[
\begin{align*}
\text{Pair}_{\sigma, \rho} & : \sigma \rightarrow \rho \rightarrow \sigma \times \rho \\
\text{Left}_{\sigma, \rho} & : \sigma \times \rho \rightarrow \sigma \\
\text{Right}_{\sigma, \rho} & : \sigma \times \rho \rightarrow \rho
\end{align*}
\]

More often than not we omit the subscripts on these constants.

(Numeric gadgets) Each atom \( \xi = N \) or \( \xi = N[i] \) comes furnished with a pair of constants

\[
0_\xi : \xi \\
S_\xi : \xi'
\]

the zero and the successor over \( \xi \).
There is a battery of iterators

$$I_{\xi, \tau} : \xi \rightarrow \tau''$$

where \(\xi\) is an atom and \(\tau\) is a type. The selection of the pairs \(\xi\) and \(\tau\) which carry an iterator determines the strength (or weakness) of the system.

We often drop the various subscripts and write

\[
P L R 0 S I
\]

for these constants when this should not lead to confusion.

You can probably see what the pairing and numeric gadgets are for, but perhaps the role of the iterators is not so clear. It is the battery of iterators which determines the power of a tiered system. Along with the colour coding they limit the recursions allowed in the system. We will spend quite a bit of time on this aspect in this document.

Each tiered system \(\lambda T\) is determined by two features. The tiering aspects are controlled by the coloured atoms that are allowed, and these are indexed by \(I\). More importantly, the recursion aspects are controlled by the selection of iterators. Given \(I\) there are two extremes. We may have no iterators, or all possible iterators. Of course, these are not interesting examples; it is the judicious choice of iterators which makes tiering useful.

It is time we looked at some examples of derivations. To help with these we introduce a couple more notions.

2.8 DEFINITION. For terms \(t, s\) the formal iterates

\[
(t^m s \mid m \in \mathbb{N})
\]

of \(t\) on \(s\) are generated by

\[
t^0 s = s \quad t^m s = t(t^{m'} s)
\]

for each \(m \in \mathbb{N}\). (Remember that \(m' = m + 1\).)

For each atom \(\xi\) and each \(m \in \mathbb{N}\) we set

\[
[m_\xi] = S^m_0 \xi
\]

to obtain the authentic numerals over \(\xi\).

When generated in an appropriate context formal iterates are well-formed terms.

2.9 EXAMPLE. Suppose we have a pair of derivations

\[
\vdots \\
\Gamma \vdash t : \sigma' \\
\vdots \\
\vdots \\
\Gamma \vdash s : \sigma
\]

for some context \(\Gamma\), type \(\sigma\), and terms \(t\) and \(s\). We can then form

\[
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \
11
and more generally we have a derivation

\[ \Gamma \vdash t^m \cdot s : \sigma \]

for each \( m \in \mathbb{N} \). Notice that the size of this derivation depends on the size of \( m \).

As a particular case of this we have

\[
\begin{align*}
\vdash S_\xi : \xi' & \vdash 0_\xi : \xi \\
\vdash S_\xi : \xi' & \vdash \Gamma 1_\xi : \xi \\
\vdash S_\xi : \xi' & \vdash \Gamma 2_\xi : \xi \\
& \vdash \Gamma 3_\xi : \xi
\end{align*}
\]

and more generally

\[ \vdash \Gamma m_\xi : \xi \]

for each \( m \in \mathbb{N} \). This shows that the numerals are well-formed in the empty context, and hence in any legal context.

The next example illustrates how an iterator can be used.

2.10 EXAMPLE. Suppose we work in a tiered system \( \lambda T \) which has an iterator

\[ 1 : \xi \rightarrow \sigma'' \]

for some atom \( \xi \) and type \( \sigma \). Then we certainly can form

\[
\vdash 1 : \xi \rightarrow \sigma'' \vdash \Gamma m_\xi : \xi
\]

for any \( m \in \mathbb{N} \). Suppose we also have a pair of derivations

\[
\vdash t : \sigma' \vdash s : \sigma
\]

as in Example 2.9. With these we can form

\[
\begin{align*}
\vdash 1 : \xi \rightarrow \sigma'' & \vdash \Gamma m_\xi : \xi \\
& \vdash \Gamma m_\xi : \sigma'' (W) \\
& \vdash \Gamma m_\xi t : \sigma' \\
& \vdash \Gamma m_\xi ts : \sigma
\end{align*}
\]

where \((W)\) indicates the appropriate number of uses of Weakening needed to build up the context \( \Gamma \).
The final example is a bit more complicated. However, it does illustrates the use of the Pairing gadgets and it will occur again later.

2.11 EXAMPLE. Consider a tiered system which has an iterator

\[ I : Q \rightarrow \mathcal{P}'' \]

where

\[ \mathcal{P} = \mathcal{N} \times \mathcal{N} \]

and \( Q \) is some atom (which may be \( \mathcal{N} \)). This system has pairing constants

\[ P : \mathcal{N} \rightarrow \mathcal{N} \rightarrow \mathcal{P} \quad L : \mathcal{P} \rightarrow \mathcal{N} \quad R : \mathcal{P} \rightarrow \mathcal{N} \]

where these have been abbreviated to save space.

Using these consider the two raw terms

\[ \llbracket D \rrbracket = (\lambda w : \mathcal{P} . ((P(S(Lw)))(Lw))) \]
\[ \llbracket d \rrbracket = (\lambda x : Q . (R(\llbracket D \rrbracket)(((Ix)(\llbracket D \rrbracket)((P0)0)))))) \]

where \( S \) is \( S_N \) and \( 0 \) is \( 0_N \). Here I have stuck strictly to the letter of the law and inserted all the required brackets. Let’s show that

\[ \vdash \llbracket D \rrbracket : \mathcal{P} \rightarrow \mathcal{P} \quad \vdash \llbracket d \rrbracket : Q \rightarrow \mathcal{N} \]

and hence both are well-formed. Before we do that I suggest you write down the parsing tree of these two terms, with the leaves at the top and the roots \( \llbracket D \rrbracket \) and \( \llbracket d \rrbracket \) at the bottom.

Now consider the derivation given in Table 3. This is a little cramped, and I have had to omit some of the typing information, but it’s just about possible to read it. Notice also that I have left out some of the brackets where this doesn’t cause any confusion.

The two contexts are

\[ \Gamma = w : \mathcal{P} \quad \Sigma = x : Q \]

respectively. You should check that the derivation is correct, by reading it from leaves to root. Notice how it follows the parsing tree of \( \llbracket D \rrbracket \) and \( \llbracket d \rrbracket \), with a bit of extra information in the middle. In particular, notice how Weakening is used at the position indicated by \((W)\). Weakening could be used at other parts of the derivation. In particular, on the right hand side we could have a derivation

\[ \vdash P00 : \mathcal{P} \]
\[ \Sigma \vdash P00 : \mathcal{P} \]

in place of the indicated one.

Notice that the shape of the derivation is just that of the tree in Table 2.

That’s enough examples of derivations for now.
Table 3: An example derivation

**Syntactic conventions**

At this juncture it is convenient to mention some informal conventions we use when displaying syntax.

As stated in Definitions 2.2 and 2.5, the brackets in types and terms are part of the official syntax. They ensure that each constructed string can be uniquely parsed. For instance, the strings

\[(\zeta \to (\xi \to (\tau \to (\sigma \to \rho))))) \quad (((((ts)r)q)p)\]

are, respectively, a types built from earlier types \(\rho, \sigma, \tau, \xi, \zeta\), and a term built from earlier terms \(p, q, r, s, t\). However, as intelligent creatures we don’t need all those brackets to understand which string is meant. We can certainly omit the outside brackets, and with appropriate conventions we can omit some of the others as well.

**2.12 BRACKETTING CONVENTIONS.** For types \(\rho, \sigma, \tau\) we let

\[\tau \to \sigma \to \rho\] abbreviate \(((\tau \to (\sigma \to \rho))\)

and for terms \(r, s, t\) we let

\[tsr\] abbreviate \(((ts)r)\)

unless it is confusing to do so.

The are two type constructors \(\times\) and \(\to\), and we let \(\times\) take precedence over \(\to\). Thus

\[\sigma \times \rho \to \tau\] abbreviate \(((\sigma \times \rho) \to \tau)\)

for types \(\rho, \sigma, \tau\).
We make iterated use of the first two conventions. Thus the two strings above are
\[ \zeta \to \xi \to \tau \to \sigma \to \rho \quad t s r q p \]
in abbreviated form. Notice that the way brackets are re-instated is different for types and terms. To remember this difference simply think of evaluating a function given in curried form.

There is another notational tick we use a lot.

2.13 DEFINITION. For each type \( \tau \) we let
\[ \tau' \text{ abbreviate } (\tau \to \tau) \]
the ‘next most complicated’ type after \( \tau \).

For each type \( \tau \) we generate the **stacked powers** \( (\tau^i) \mid i < \omega \) of \( \tau \) by
\[ \tau^{(0)} = \tau \quad \tau^{(i')} = \tau^{(i)}' \]
(for \( i < \omega \)).

For instance
\[ \tau^{(0)} = \tau \]
\[ \tau^{(1)} = \tau' = (\tau \to \tau) \]
\[ \tau^{(2)} = \tau'' = (\tau' \to \tau') = (\tau \to \tau) \to (\tau \to \tau) \]
with
\[ \tau^{(3)} = \tau''' = ((\tau \to \tau) \to (\tau \to \tau)) \to ((\tau \to \tau) \to (\tau \to \tau)) \]
and then \( \tau^{(4)} \) is almost too long to fit across the page. You can see why this convention is useful.

We also use these conventions with members of \( \mathcal{F} \mathcal{H}(N) \). In particular, each \( S \in \mathcal{F} \mathcal{H}(N) \) generates a tower of spaces \( S, S', S'', S''' \ldots \) within \( \mathcal{F} \mathcal{H}(N) \).

As indicated, we use \((\cdot)'\) on types and spaces to produce the ‘next most complicated’ one. We also use the same convention in a different place. As explained in Section 1, for \( x \in N \) we write
\[ x' \text{ for } x + 1 \]
since this is the ‘next most complicated’ number.

There is another couple of abbreviating tricks we use. Consider the term
\[ (\lambda z : \tau . (\lambda y : \sigma . (\lambda x : \rho . t))) \]
displayed in its official syntax. We sometimes write this as
\[ \lambda z : \tau, y : \sigma, x : \rho . t \]
where the single \( \lambda \) now controls \( x, y, \) and \( z \). When the three types are the same, say \( \sigma \), we can write
\[ \lambda z, y, x : \sigma . t \]
for this term. There are other variations on this theme that we sometime use.
3 Orthodox semantics

As I stressed in Section 2, the derivation system of a tiered system $\lambda T$ is entirely about certain syntactic manipulations. According to the rules certain manipulations are allowed and others are not. The rules are designed to generated derivations

$$(\nabla) \quad \Gamma \vdash t : \tau$$

each of which we think of as a certification of correctness of the root judgement $\Gamma \vdash t : \tau$. We have seen that some judgements can have more than one derivation.

Of course, that is not the whole story. The system $\lambda T$ has an ulterior motive; correct judgements are supposed to describe certain mathematical gadgets, and the rules are set up with this in mind. Let’s try to expose this chicanery.

By Definition 2.4 we know that each type $\tau$ has an orthodox interpretation $[\tau]$, a set in the hierarchy $\mathfrak{H}(\mathbb{N})$. This idea can be extended to contexts. Thus for each context

$$\Gamma = x_m : \sigma_m, \ldots, x_1 : \sigma_1$$

we set

$$[\Gamma] = S_m \times \cdots \times S_1$$

where $S_i = [\sigma_i]$ for each $1 \leq i \leq m$, to produce the orthodox interpretation of $\Gamma$.

The idea is that each derivation $\nabla$, as above, describes a function

$$[\Gamma] \xrightarrow{[\nabla]} [\tau]$$

called the meaning of $\nabla$. Informally, this is nothing more than the function you first thought of when you looked at the subject $t$ of the root judgement of $\nabla$. In this section I will give a more detailed description of how we pass from the syntax $\nabla$ to the semantics $[\nabla]$. However, I must warn you that I won’t be entirely honest. There are some details which I will ‘conveniently’ ignore. This shouldn’t damage your understanding of the process, but may stimulate your curiosity. The root of the problem is the way we handle the empty context. I will give a more detailed description of the process in Section 23.

Before I begin the description proper, let me explain why this is the orthodox semantics, and not some other semantics.

As I mentioned briefly in Section 2 we interpret a system $\lambda T$ within the category of sets. There are also other categories in which $\lambda T$ can be interpreted, and each of these gives a non-orthodox semantics. However, that technique is not needed here.

To generate the function $[\nabla]$ we look at the derivation $\nabla$, pass from the leaves to the root, and attach a function at each node. We attach a rather simple function to each leaf; we will look at these in more detail shortly. We pass across the three construction rules, Weakening, Introduction, Elimination, using the ‘intended meaning’ of each rule.

For a use of Weakening

$$\Gamma \vdash t : \tau$$

$$\Gamma, x : \sigma \vdash t : \tau$$

we already have an interpretation

$$G \xrightarrow{[t]} T$$
of the numerator. Here $G = \Gamma$ and $T = \tau$, and I have written $[t]$ for the function. Strictly speaking this is a bit sloppy since the function depends on the whole of the derivation, not just the root subject $t$. However, this is a common practice, and with a bit of care you won’t come to much harm. This convention can be justified, but we need quite a bit more development to do that.

We have the meaning $[t]$ of the numerator of the rule. To obtain the meaning of the denominator we convert this into the function

$$G \times S \xrightarrow{[t]^+} T$$

by adding a dummy input to $[t]$ using $S = [\sigma]$ as the dummy input position. In other words, $[t]^+$ is the composite function

$$G \times S \xrightarrow{\text{proj}} G \times S \xrightarrow{[t]^+} T$$

where the function $\text{proj}$ forgets the dummy input to $S$. A use of Introduction

$$\Gamma, x : \sigma \vdash r : \rho \quad \Gamma \vdash t : \tau$$

is handled by un-currying. The numerator gives us a function

$$G \times S \xrightarrow{[r]} R$$

and we convert that into the function

$$G \xrightarrow{[t]} (S \rightarrow R)$$

by simply taking a different view of the function $[r]$.

A use of Elimination

$$\Gamma \vdash q : \pi \rightarrow \tau \quad \Gamma \vdash p : \pi$$

is handled by evaluation. Each component of the numerator

$$G \xrightarrow{[q]} (P \rightarrow T) \quad G \xrightarrow{[p]} P$$

selects a function and input

$$P \xrightarrow{[q](\star)} T \quad [p](\star) \in P$$

for each $\star \in G$, to give some value $([q](\star))(([p](\star)) \in T$. The denominator is the function

$$G \xrightarrow{\star} \Gamma \xrightarrow{([q](\star))(([p](\star))}$$

which returns that value for each selector $\star \in G$. You might want to think about this rule for a while. You might realize that you have seen this rule before not too far away from the $\lambda$-calculus.

17
You might also think about what happens if the context is empty, that is \( \mathcal{G} = \emptyset \) in these illustrations. It might seem that things become a mess. Well, everything can be done quite smoothly, but we need to take a bit of time to set up the details properly.

It remains to say how we handle leaves.

The meaning of a Projection

\[
\Gamma \vdash x : \sigma
\]

is obvious, it is the projection function

\[
\mathcal{G} \rightarrow \mathcal{S}
\]

which simply selects the input labelled by \( x \). For instance the meaning of

\[
y : \tau, x : \sigma, w : \rho \vdash x : \sigma
\]

is the projection function

\[
\mathcal{T} \times \mathcal{S} \times \mathbb{R} \rightarrow \mathcal{S} \\
(y, x, w) \mapsto x
\]

which selects the middle of the three inputs.

What about an Axiom

\[
(A) \quad \Gamma \vdash k : \kappa
\]

based on an axiom \( k : \kappa \)?

Each constant \( k \) has an intended interpretation \([k]\) which is a member of the intended interpretation \([\kappa]\) of the housing type \( \kappa \) of \( k \). The intended interpretation of the pairing gadgets should be obvious, the pair forming operation or the left or right projection. For each atom \( \xi \) we have \([\xi] = \mathbb{N}\) and then

\[
[0_\xi] = 0 \quad [S_\xi] = S
\]

where \( S \) is the successor function on \( \mathbb{N} \). For the intended interpretation of an iterator

\[
1 : \xi \rightarrow \tau''
\]

let \( \mathcal{T} = [\tau] \) and then

\[
[1] : \mathbb{N} \rightarrow \mathcal{T}''
\]

is given by

\[
[1]mts = t^m s
\]

for all \( m \in \mathbb{N}, t \in \mathcal{T}', s \in \mathcal{T} \). Thus \([1]\) is the function which when supplied with a natural number \( m \) will return the \( m \)-fold iterator on \( \mathcal{T} \).

These intended interpretations induce the semantics for each Axiom. For the Axiom \( A \), as above, the intended interpretation \([k]\) gives the constant function

\[
[\Gamma] \overset{[A]}{\longrightarrow} [\kappa]
\]

which sends each element of \([\Gamma]\) to \([k]\) in \([\kappa]\). When the context \( \Gamma \) is empty, this meaning is just the member \([k]\) of \([\tau]\).

Some examples of this process will help.
3.1 EXAMPLE. Consider the authentic numerals
\[ [m_\xi] = S_\xi^n 0_\xi \]
as given by Definition 2.8. Here \( \xi \) is one of the atoms, it may be monochrome or it may be one of the coloured ones.

The derivation on the left indicates that the first few of these are well-formed in the empty context, and it is obvious how we can continue this process.

\[
\begin{array}{c}
\vdash S_\xi : \xi' & \vdash 0_\xi : \xi \\
\vdash S_\xi : \xi' & \vdash 1_\xi : \xi \\
\vdash S_\xi : \xi' & \vdash 2_\xi : \xi \\
\vdash \ldots & \\
\end{array}
\]
\[
\begin{array}{c}
S : N \rightarrow N & 0 \in N \\
S : N \rightarrow N & 1 = S0 \in N \\
S : N \rightarrow N & 2 = S1 \in N \\
S : N \rightarrow N & 3 = S2 \in N \\
\end{array}
\]

The tree on the right shows how we generate the meaning of each numeral. Thus we have
\[ [m_\xi] = m \]
for each \( m \in \mathbb{N} \).

More generally, a derivation
\[ (\nabla) \quad \Gamma \vdash [m_\xi] : \xi \]
in a non-empty context produces a constant function
\[ G \quad \mathbb{N} \]
with constant value \( m \in \mathbb{N} \) as indicated by the numeral.

Notice how this orthodox semantics destroys any colour coding that might exists. This is typical of the passage from syntax to semantics.

We have discovered that the syntactic numerals are there to name the actual natural numbers. Perhaps now we should do a bit of arithmetic. Nothing too hard mind you.

3.2 EXAMPLE. Consider the tiered system \( \lambda B \) which has just two atoms. As well as the monochrome atom \( N \) there is a magenta atom \( M \). Suppose also that \( \lambda B \) has just one iterator
\[ I : M \rightarrow N'' \]
and the usual numeric and pairing gadgets. This is one of the systems we will look at quite a lot.

Consider the raw terms
\[
\begin{align*}
\text{Add} & : \lambda v : M, w : N \cdot I(vS(w) \\
\text{Mlt} & : \lambda v : M, u : M \cdot I(u(\text{Add}(v))0
\end{align*}
\]
of \( \lambda B \). Here \( S \) is \( S_N \) and \( 0 \) is \( 0_N \), the monochrome numeric gadgets, and \( I \) is the single iterator of \( \lambda B \), as give above. Note carefully how the identifiers \( u, v, w \) are used in these two examples. There is something going on here.
\[
\begin{array}{c}
\Gamma \vdash I : \mathcal{M} \to \mathcal{N}'' \\
\Gamma \vdash v : \mathcal{M} \\
\therefore \Gamma \vdash lv : \mathcal{N}'' \\
\Gamma \vdash S : \mathcal{N}'' \\
\Gamma \vdash lvS : \mathcal{N}'' \\
\Gamma \vdash w : \mathcal{N} \\
\therefore \Gamma \vdash lvw : \mathcal{N} \\
v : \mathcal{M} \vdash \ldots : \mathcal{N}'' \\
\vdash \text{Add} : \mathcal{M} \to \mathcal{N}''
\end{array}
\]

\[
\vdash I : \mathcal{M} \to \mathcal{N}'' \\
\vdash u : \mathcal{M} \\
\vdash \text{Add} : \mathcal{M} \to \mathcal{N}'' \\
\vdash \text{Add} v : \mathcal{N}'' \\
\vdash \text{Add} v 0 : \mathcal{N} \\
v : \mathcal{M} \vdash \ldots : \mathcal{M} \to \mathcal{N} \\
\vdash \text{Mlt} : \mathcal{M} \to \mathcal{M} \to \mathcal{N}
\]

Table 4: Two example derivations in $\lambda \mathcal{B}$

With the contexts
\[
\Gamma = v : \mathcal{M}, w : \mathcal{N} \\
\Sigma = v : \mathcal{M}, u : \mathcal{M}
\]
we can generate the two derivation of Table 4. Thus the terms $\text{Add}$ and $\text{Mlt}$ are well formed (in the empty context) with
\[
\vdash \text{Add} : \mathcal{M} \to \mathcal{N} \to \mathcal{N} \\
\vdash \text{Mlt} : \mathcal{M} \to \mathcal{M} \to \mathcal{N}
\]
respectively. These should give us functions
\[
A : \mathcal{N} \to (\mathcal{N} \to \mathcal{N}) \\
M : \mathcal{N} \to (\mathcal{N} \to \mathcal{N})
\]
which we can think of in more standard form
\[
\begin{array}{c}
\mathcal{N} \times \mathcal{N} \xrightarrow{A} \mathcal{N} \\
\mathcal{N} \times \mathcal{N} \xrightarrow{M} \mathcal{N}
\end{array}
\]
and the syntax used should be a clue as to what these are.

Consider the orthodox semantics of the top derivation. With a bit of patience you should be able to show that the ante-penultimate judgement
\[
\Gamma \vdash lvw : \mathcal{N}
\]
that is
\[
v : \mathcal{M}, w : \mathcal{N} \vdash lvw : \mathcal{N}
\]
gives a function

\[
\begin{array}{ccc}
N \times N & \longrightarrow & N \\
(m, n) & \longmapsto & S^m n
\end{array}
\]

where \(S\) is the successor function on \(N\). In other words this is the function

\[
(m, n) \longmapsto n + m
\]

and so \textbf{Add} names the addition function. If you have never done this kind of thing before, then I suggest you go through the construction using the rules set down on the previous pages. You should also worry a bit about how to get from this judgement to the ultimate one. This is the bit I have not really told you about.

Now look at the lower derivation. In particular, look at the part that finishes

\[
\Sigma \vdash \text{Add}v : \mathcal{N}'
\]

quite near the top of the whole derivation. The meaning of that is the function

\[
\begin{array}{ccc}
N \times N & \longrightarrow & \mathcal{N}' \\
(n, m) & \longmapsto & (\cdot + n)
\end{array}
\]

where as yet the input \(m\) is playing no visible role. Using this we see that the antepenultimate judgement

\[
\Sigma \vdash Iu(\text{Add}v)0 : \mathcal{N}
\]

gives a function

\[
\begin{array}{ccc}
N \times N & \longrightarrow & N \\
(n, m) & \longmapsto & (\cdot + n)^m 0
\end{array}
\]

that is

\[
(n, m) \longmapsto n \times m
\]

and so \textbf{Mlt} names the multiplication function. ■

Have another look at that example, and try to work out why we need two atoms \(\mathcal{N}\) and \(\mathcal{M}\). Surely we could replace \(\mathcal{M}\) be \(\mathcal{N}\) and do exactly the same thing. The example produces the two simplest arithmetical operations, addition and multiplication. See if you can produce the next operation, exponentiation, within the system. After you have worried for a while you can look at the next example.

3.3 EXAMPLE. Consider the tiered system \(\lambda F\) which has just one atom \(\mathcal{N}\) and just one iterator

\[
1 : \mathcal{N} \rightarrow \mathcal{N}''
\]

and the usual numeric and pairing gadgets. This system \(\lambda F\) is like the system \(\lambda B\) of Example 3.2 except now the coloured atom has been replaced by \(\mathcal{N}\). In particular anything that can be done in colour in \(\lambda B\) can also be done in monochrome in \(\lambda F\). But \(\lambda F\) can do much more.
Consider the raw terms

\[
\begin{align*}
\text{Add} & \quad \lambda u : \mathcal{N}, v : \mathcal{N}. \text{IuSv} \\
\text{Mlt} & \quad \lambda v : \mathcal{N}, u : \mathcal{N}. \text{Iu(Addv)0} \\
\text{Exp} & \quad \lambda v : \mathcal{N}, u : \mathcal{N}. \text{Iu(Mltv)\Gamma} \\
\text{Stk} & \quad \lambda w : \mathcal{N}, v : \mathcal{N}, u : \mathcal{N}. \text{Iu(Expv)w} \\
\text{Big} & \quad \lambda u : \mathcal{N}. \text{Iu(Stk\Gamma1\Gamma2\Gamma0)}
\end{align*}
\]

where $\{0\}, \{\Gamma\},$ and $\{2\}$ are the numerals over $\mathcal{N}$.

All of these terms are well formed (in the empty context). Derivations for \text{Add} and \text{Mlt} can be generated as in Example 3.2 (with $\mathcal{M}$ replaced by $\mathcal{N}$).

With the context

$$\Gamma = v : \mathcal{N}, u : \mathcal{N}$$

we may check that

$$\vdash \text{Exp} : \mathcal{N} \rightarrow \mathcal{N} \rightarrow \mathcal{N}$$

by two uses of Introduction. The top part of this derivation produces

$$\vdash (\text{Mltv}) : \mathcal{N}'$$

with meaning

$$\begin{array}{ccc}
\mathbb{N} \times \mathbb{N} & \longrightarrow & \mathbb{N}' \\
(n, m) & \longmapsto & (n \times \cdot)
\end{array}$$

where as yet the input $m$ is playing no visible role. Using this we see that the whole of the displayed derivation gives a function

$$\begin{array}{ccc}
\mathbb{N} \times \mathbb{N} & \longrightarrow & \mathbb{N} \\
(n, m) & \longmapsto & (n \times \cdot)^m 1
\end{array}$$

that is

$$(n, m) \longmapsto n^m$$

and so \text{Exp} names the 2-placed exponential function.

To deal with \text{Stk} let

$$\Sigma = v : \mathcal{N}, u : \mathcal{N}$$
we may check that

\[
\begin{array}{c}
\Sigma \vdash \text{Exp} : N \rightarrow N' \\
\Sigma \vdash v : N
\end{array}
\]

\[
\begin{array}{c}
\Sigma \vdash Iu : N'' \\
\Sigma \vdash (\text{Exp}) : N'
\end{array}
\]

\[
\begin{array}{c}
\Sigma \vdash Iu(\text{Exp}) : N'' \\
\Sigma \vdash w : N
\end{array}
\]

is the central part of the derivation leading to

\[
\vdash \text{Stk} : N \rightarrow N \rightarrow N \rightarrow N'
\]

by three uses of Introduction. The top part of this derivation produces

\[
\Sigma \vdash (\text{Exp}) : N'
\]

with meaning

\[
\begin{array}{c}
N \times N \times N \rightarrow N' \\
(n, m, l) \mapsto m^{ullet}
\end{array}
\]

where as yet the inputs \(n, l\) are playing no visible role. Using this we see that the whole of the displayed derivation gives

\[
\begin{array}{c}
N \times N \times N \rightarrow N \\
(n, m, l) \mapsto (m^{ullet})^l_n
\end{array}
\]

which is an important function we will meet in several places. It is the 3-placed function \(\mathbb{Z}\) generated by

\[
\mathbb{Z}(0, m, n) = n \\
\mathbb{Z}(l', m, n) = \begin{cases} 
  m^\mathbb{Z}(l, m, n) \\
  \mathbb{Z}(l, m, m^n)
\end{cases}
\]

for \(l, m, n \in \mathbb{N}\). There are alternative version of the recursion step, but both lead to the same stacking function.

Finally, we may generate a derivation

\[
\begin{array}{c}
\vdash \text{Stk} : -- \\
\vdash \Gamma : N
\end{array}
\]

\[
\begin{array}{c}
\vdash \text{Stk} \Gamma : -- \\
\vdash \Gamma_2 : N
\end{array}
\]

\[
\begin{array}{c}
\vdash \text{Stk} \Gamma \Gamma_2 : N'' \\
u : N \vdash 1u : N'' \\
\vdash \text{Stk} \Gamma \Gamma_2 : N''
\end{array}
\]

\[
\begin{array}{c}
\vdash \text{Stk} \Gamma \Gamma_2 : N'' \\
u : N \vdash 1u(\text{Stk} \Gamma \Gamma_2) : N'' \\
\vdash 0 \Gamma : N
\end{array}
\]

\[
\begin{array}{c}
\vdash \text{Big} : N \rightarrow N
\end{array}
\]

to show that Big is well-formed in the empty context.
The penultimate line of this derivation produces a function

\[
\begin{array}{c}
\mathbb{N} 
\end{array} \quad \begin{array}{c}
\mathbb{N}
\end{array}
\end{array}

\quad \begin{array}{c}
m \mapsto \mathfrak{N}(\cdot, 2, 1)^m 0
\end{array}

with initial values

\[
\begin{align*}
\text{Big}(0) &= 0 = 0 \\
\text{Big}(1) &= \mathfrak{N}(0, 2, 1) = 1 \\
\text{Big}(2) &= \mathfrak{N}(1, 2, 1) = 2 \\
\text{Big}(3) &= \mathfrak{N}(2, 2, 1) = 4 \\
\text{Big}(4) &= \mathfrak{N}(4, 2, 1) = 65,536
\end{align*}
\]

and then Big(5) is larger than any known measure of the universe.

The point of this last example is to show that some extremely fast growing functions are captured by the very simple system \(\lambda F\). In more common parlance, an iterated use of recursion can very quickly get out of hand.

It is worth looking at the terms

\[
\text{Add}, \text{Mlt}, \text{Exp}, \text{Stk}, \text{Big}
\]

which capture the functions. Count the number of nested uses if the iterator \(I\) in these terms. There are

\[
1, 2, 3, 4, 5
\]

respectively. These numbers give us a measure of complexity of the functions, and indicate that we can go quite a bit further. For instance

\[
\text{Six} = \lambda u : \mathcal{N}. \text{Big}^3\text{I}
\]

will give a function that increases faster than Big, and then

\[
\text{Seven} = \lambda u : \mathcal{N}. \text{Six}^0\text{I}
\]

will produce an even faster function, and so on. We consider a more organized version of this material in Section 16.

The system \(\lambda F\) is monochrome, but what about the similar and slightly coloured system \(\lambda B\) of Example 3.2? It turns out that exponentiation, even \(x \mapsto 2^x\), can not be captured in \(\lambda B\). The use of the coloured atom \(\mathcal{M}\) to control the iterator severely weakens the power of the system. This, in a nut shell, is what tiering is about.

Of course, the weakness of \(\lambda B\) in comparison with \(\lambda F\) is certainly not obvious, and requires some proof. That and similar results is what this document is about.

Earlier I said that the interpretation of the pairing gadgets

\[
\begin{array}{c}
\text{Pair}_{\sigma, \rho} \quad \text{Left}_{\sigma, \rho} \quad \text{Right}_{\sigma, \rho}
\end{array}
\]

should be obvious. This was perhaps a little glib, so let’s have a look at these.

3.4 EXAMPLE. Consider the pairing gadgets, as above, which for the sake of brevity we may write as \(P, L, R\). Consider an Axiom

\[
\Gamma \vdash P : \sigma \rightarrow \rho \rightarrow \sigma \times \rho
\]
using the pair forming constant. The orthodox semantics produces a function

\[ G \quad \rightarrow \quad (S \rightarrow R \rightarrow S \times R) \]

where \( S = [\sigma] \), and so on. This function looks a little strange but the uncurried form is familiar. It is just the pair forming function

\[ G \times S \times R \quad \rightarrow \quad S \times R \]

\[ \ast, s, r \quad \rightarrow \quad (s, r) \]

which forgets the input from \( G \) and pairs the two other inputs.

The two other Axioms

\[ \Gamma \vdash L : \sigma \times \rho \rightarrow \sigma \quad \Gamma \vdash R : \sigma \times \rho \rightarrow \rho \]

give functions

\[ G \quad \rightarrow \quad (S \times R \rightarrow S) \quad G \quad \rightarrow \quad (S \times R \rightarrow R) \]

which, in uncurried form

\[ G \times S \times R \quad \rightarrow \quad S \quad G \times S \times R \quad \rightarrow \quad R \]

\[ \ast, s, r \quad \rightarrow \quad s \quad \ast, s, r \quad \rightarrow \quad r \]

merely extract the relevant component.

For the final example of this section let’s briefly look at a derivation that uses these pairing constants.

3.5 EXAMPLE. Consider the derivation with root judgement

\[ \vdash \vec{d} : Q \rightarrow N \]

given in Table 3. With a bit of effort we can show that this names the predecessor function, the function \( d : N \rightarrow N \) given by

\[ dx = \begin{cases} 
  x - 1 & \text{if } x \neq 0 \\
  0 & \text{if } x = 0 
\end{cases} \]

for \( x \in \mathbb{N} \). If you try to prove this you may run into a snag. There is the informal trick used in the construction of the function. This will be explained in Section 13, when we look at this particular function in more detail.

As all these examples illustrate, the passage from derivation \( \nabla \) to meaning \([\nabla]\) can lose a lot of information. It looses all the colour coding, and there is an evaluation algorithm embedded in \( \nabla \) which is not visible in \([\nabla]\). Indeed, many different derivations have the same meaning. We will find out more about this in the next section.

As I said earlier, the informal description of the orthodox semantics given above is good enough most of the time. But, as you can see by going through the examples in detail, there is something not quite right. It gets into a tangle when it tries to handle the empty context. I will describe how to untangle this in Section 23.

Finally, let us set down why we are interested in this orthodox semantics.
3.6 DEFINITION. Consider a derivation

$$(F) \vdash [f] : \xi(l) \to \cdots \to \xi(1) \to \xi(0)$$

in some tiered system $\lambda T$ where $\xi(0), \ldots, \xi(l)$ are atoms of the system. The orthodox semantics produces a function $f = [f]$ in $N(l)$. The set $\mathfrak{M}(\lambda T)$ of all such first order functions named in this way is one measure of the strength of $\lambda T$. $\blacksquare$

Amongst other things, we are interested in the relationship between the syntactic facilities of $\lambda T$ and the complexity of $\mathfrak{M}(\lambda T)$.

[Last changed April 13, 2006]

4 Computation mechanism

We have seen that the orthodox semantics, the passage from syntax to meaning, forgets all the colour coding. It also forgets quite a lot more.

4.1 EXAMPLE. Suppose we have a pair of derivable judgements

$$\Gamma \vdash l : \sigma \quad \Gamma \vdash r : \rho$$

in some system. Using the pairing constants $P, L, R$ associated with the types $\sigma$ and $\rho$, we obtain a derivation

$$\begin{align*}
\Gamma \vdash P : - \quad & \Gamma \vdash l : \sigma \\
\Gamma \vdash P l : - \quad & \Gamma \vdash r : \rho \\
\Gamma \vdash L : \sigma \times \rho \to \sigma \quad & \Gamma \vdash P l r : \sigma \times \rho \\
\Gamma \vdash L(Pl) r : \sigma
\end{align*}$$

where, to save a bit of space, I have omitted some of the typing information.

The two given derivations have meanings

$$G \xrightarrow{l(\cdot)} S \quad G \xrightarrow{r(\cdot)} R$$

for some functions $l(\cdot)$ and $r(\cdot)$. Using these we can generate the meaning

$$G \xrightarrow{[L(P l r)]} S$$

of the root judgement. To do this it is easier to work with uncurried functions. We obtain

$$\begin{align*}
G \times S \times R \to S \times R \\
* \times \times \times \times (s, r) \\
G \times R \to S \times R \\
* \times \times (l(\cdot), r) \\
G \times S \times R \to S \\
* \times \times \times \times s \\
G \times S \times R \to S \times R \\
* \times \times (l(\cdot), r(\cdot)) \\
G \to S \\
* \xrightarrow{l(\cdot)} l(*)
\end{align*}$$
by passing through the derivation. This shows that the two judgements
\[ \Gamma \vdash l : \sigma \quad \Gamma \vdash L(Plr) : \sigma \]
have exactly the same meaning. This shouldn’t be a surprise, for that is part of the intention of the pairing gadgets.

Within a term there may be one or more subterms each of which, as far as meaning goes, is ‘equivalent’ to a simpler term. A primitive kind of such a term is called a redex. This is a contracted form of reducible expression. Here are the three kinds we need in these notes.

4.2 DEFINITION. For each pair of types \( \sigma, \rho \) each of the two terms
\[ \text{Left}_{\sigma,\rho}(\text{Pair}_{\sigma,\rho}lr) \quad \text{Right}_{\sigma,\rho}(\text{Pair}_{\sigma,\rho}lr) \]
is a pairing-redex. Here \( l \) and \( r \) are arbitrary terms.

For each iterator
\[ I : \xi \rightarrow \tau'' \]
over an atom \( \xi \), each of the two terms
\[ I0_\xi ts \quad I(S_\xi r)ts \]
is an iterator-redex. Here \( r, s, \) and \( t \) are arbitrary terms.

For each type \( \sigma \), identifier \( x \), and terms \( r, s \), the term
\[ (\lambda x : \sigma . r)s \]
is a \( \lambda \)-redex.

Often in the literature the word ‘redex’ means a \( \lambda \)-redex (because the other kinds of redexes don’t occur). Here we use ‘redex’ in the more general way.

Look at the redexes. (I suppose there are some people who think the plural of redex should be redices.) In each case there is an obvious simpler term with the same meaning.

4.3 DEFINITION. For each redex \( t^- \) we write
\[ t^- \triangleright t^+ \]
to indicate that \( t^+ \) is the immediate reduct of \( t^- \). Thus there are just five kinds of instances of this relation. Referring to Definition 4.2 we have
\[ \text{Left}(\text{Pair} lr) \triangleright l \quad \text{Right}(\text{Pair} lr) \triangleright r \]
\[ I0_\xi ts \triangleright s \quad I(S_\xi r)tx \triangleright t(lrts) \]
\[ (\lambda x : \sigma . r)s \triangleright r[x := s] \]
where \( r[x := s] \) is the result of replacing in \( r \) all free occurrences of \( x \) by \( s \).
Strictly speaking, before we give this definition we should describe the substitution algorithm which produces \( r[x := s] \) from two terms \( r, s \) and an identifier \( x \). For what we do here we can take this as ‘obvious’. However, if you have never seen this done formally before, you should be warned that there are some subtleties, but these won’t spoil your enjoyment here.

We refer to each instance 
\[ t^- \triangleright t^+ \]
of this relation as a 1-step reduction. From these we generate the reduction relation.

4.4 DEFINITION. The reduction relation
\[ t^- \triangleright \triangleright t^+ \]
is the transitive, compositional closure of the 1-step relation \( \triangleright \). Thus the above instance holds if we can move from \( t^- \) to \( t^+ \) in a finite number of steps where at each step we use an instance \( s^- \triangleright s^+ \) of the 1-step relation to replace a redex subterm \( s^- \) of \( t^- \) by its reduct \( s^+ \).

By intention the relation \( \triangleright \triangleright \) is unidirectional and irreflexive. We let \( \triangleright \triangleright \triangleright \) be the reflexive version of \( \triangleright \). Thus \( t^- \triangleright \triangleright \triangleright t^+ \) means \( t^- = t^+ \) or \( t^- \triangleright \triangleright t^+ \).

As always, a few good examples is worth a thousand words.

4.5 EXAMPLE. Consider a system with an iterator
\[ 1 : \xi \rightarrow \tau'' \]
over some atom \( \xi \). Then
\[ t^m s \triangleright \triangleright t^m s \]
for each \( m \in \mathbb{N} \) and terms \( s, t \). For instance
\[ t^4 \xi s = t(S(3 \xi) s) \triangleright t(l(3 \xi) t s) \]
\[ t^3 \xi s = t(S(4 \xi) s) \triangleright t(l(4 \xi) t s) \]
\[ t^2 \xi s = t(S(3 \xi) s) \triangleright t(l(3 \xi) t s) \]
\[ t^1 \xi s = t(S(0 \xi) s) \triangleright t(l(0 \xi) t s) \]
and
\[ l(0 \xi) t s \triangleright s \]
so that
\[ t^1 \xi s \triangleright t(l(0 \xi) t s) \triangleright t s = t^1 s \]
\[ t^2 \xi s \triangleright t(l(1 \xi) t s) \triangleright t(t^1 s) = t^2 s \]
\[ t^3 \xi s \triangleright t(l(2 \xi) t s) \triangleright t(t^2 s) = t^3 s \]
\[ t^4 \xi s \triangleright t(l(3 \xi) t s) \triangleright t(t^3 s) = t^4 s \]
to verify the reduction for \( m = 4 \). The general case follows by induction over \( m \).

Slightly more interesting reduction can be obtained using the arithmetical terms of Example 3.2.
4.6 **EXAMPLE.** Consider the term

\[
\text{Add} \quad \lambda u : \mathcal{M}, v : \mathcal{N} . luSv
\]

of \(\mathbf{AB}\) as in Example 3.2. We know that

\[\vdash \text{Add} : \mathcal{M} \rightarrow \mathcal{N} \rightarrow \mathcal{N}\]

is derivable and, by the orthodox semantics, the term names the addition operation. Using this term we can use the reduction relation to calculate the sum of two numbers.

We have two kinds of numerals, so let us write

\[\hat{m} \text{ for } \langle m \rangle_{\mathcal{M}} \quad \hat{n} \text{ for } \langle n \rangle_{\mathcal{N}}\]

too avoid too many subscripts.

For \(m, n \in \mathbb{N}\) we have

\[\vdash \text{Add} \hat{m} \hat{n} : \mathcal{N}\]

and the orthodox semantics says that this term names the number \(n + m\), but the term is clearly not the numeral \(\langle n + m \rangle\). To get to the numeral we have to go through a computation.

The subterm \(\text{Add} \hat{m}\) is a \(\lambda\)-redex with

\[\text{Add} \hat{m} \triangleright \lambda v : \mathcal{N} . \hat{m}Sv\]

as a 1-step redex removal. But now

\[\text{Add} \hat{m} \hat{n} \triangleright (\lambda v : \mathcal{N} . \hat{m}Sv)\hat{n} \triangleright \hat{m}S\hat{n}\]

using a second redex removal. We now have an iterator redex, to give

\[\text{Add} \hat{m} \hat{n} \triangleright \mathcal{N}^m \hat{n} = \langle n + m \rangle\]

as required. Notice how the colour of \(\hat{m}\) disappears at the last reduction.

There is also a second sequence of reduction that gets to the same result. Using Example 4.5 we have

\[\text{Add} \hat{m} \triangleright \lambda v : \mathcal{N} . \hat{m}Sv \triangleright \lambda v : \mathcal{N} . \mathcal{N}^m v\]

so that

\[\text{Add} \hat{m} \hat{n} \triangleright (\lambda v : \mathcal{N} . \mathcal{N}^m v)\hat{n} \triangleright \mathcal{N}^m \hat{n} = \langle n + m \rangle\]

using a \(\lambda\)-reduction at the last stage. \(\blacksquare\)

As this example shows, sometimes a term can be reduced in more than one way. We will discuss this in more detail later in this section.

4.7 **EXAMPLE.** Consider the raw term

\[
\text{Mlt} \quad \lambda v : \mathcal{M}, u : \mathcal{M} . lu(\text{Add}v)0
\]

of \(\mathbf{AB}\) as in Example 3.2. We know that

\[\vdash \text{Mlt} : \mathcal{M} \rightarrow \mathcal{M} \rightarrow \mathcal{N}\]

29
and, by the orthodox semantics, this term names the multiplication operation. Continuing with the notation of Example 4.6 we have

\[ \vdash \text{Mlt } \tilde{n} \tilde{m} : \mathcal{N} \]

for each \( m, n \in \mathbb{N} \), and this term names \( n \times m \).

By two \( \lambda \)-redex removals we have

\[ \text{Mlt } \tilde{n} \tilde{m} \triangleright I \tilde{m} (\text{Add} \tilde{n})^{0} \]

and then we have two options. We have either

\[ I \tilde{m} (\text{Add} \tilde{n})^{0} \triangleright (\text{Add} \tilde{n})^{m} \triangleright \tilde{n} \times \tilde{m} \]

or

\[ I \tilde{m} (\text{Add} \tilde{n})^{0} \triangleright \tilde{m} (\lambda v : \mathcal{N} \cdot \Gamma \tilde{n}^{v} \mathcal{S} \mathcal{v})^{m} \triangleright \tilde{n} \times \tilde{m} \]

where in each case the final reduction follows by induction on \( m \). The first case uses Example 4.6 and the second is a direct argument.

There can be several reduction paths from a term to another. As Example 4.7 hints at, some reductions may be more efficient than others. This is not something we look at in these notes.

For the final example of this section let’s look at the rather weird derivation of Example 2.11.

4.8 EXAMPLE. Consider the term

\[ \tilde{d} = \lambda x : Q \cdot R((lx \tilde{D})(P00)) \]

of Example 2.11 where I have now omitted some brackets. Let

\[ \tilde{m} = \tilde{m}_{Q} \]

for each \( m \in \mathbb{N} \). A \( \lambda \)-reduction followed by several iterator-reductions gives

\[ \tilde{d} \tilde{m} \triangleright R((\tilde{m} \tilde{D})(P00)) \triangleright R(\tilde{D}^{m}(P00)) \]

and we must now simplify the inner term. Try to reduce

\[ \tilde{D}^{l}(P^{l} T^{r} T^{r}) \]

for \( l, r \in \mathbb{N} \). You will then begin to understand the inner workings of the term \( \tilde{d} \).

We will return to the term \( \tilde{d} \) in due course.

There are several properties of the reduction relations that we must look at.

4.9 DEFINITION. A term \( t \) is normal if there are no reductions from \( t \). That is, if \( t \) has no subterm that is a redex.
(Normalization) For each derivation \( \Gamma \vdash t^- : \tau \)
there is a normal term \( t^+ \) with \( t^- \triangleright\triangleright t^+ \).

(Confluence) For each divergent wedge of reductions
\[
\begin{align*}
(L^-) & \quad t^- \triangleright\triangleright l & (R^-) & \quad t^- \triangleright\triangleright r \\
(\text{from a common source } t^-) & \quad \text{there is a convergent wedge} & & \\
(L^+) & \quad l \triangleright\triangleright t^+ & (R^+) & \quad r \triangleright\triangleright t^+
\end{align*}
\]
to a common sink \( t^+ \).

(Influence) For each derivable judgement
\[
(\nabla^-) \quad \Gamma \vdash t^- : \tau
\]
and reduction \( t^- \triangleright\triangleright t^+ \) from the root subject, there is a derivation
\[
(\nabla^+) \quad \Gamma \vdash t^+ : \tau
\]
of the reduced term \( t^+ \) in the same context and with the same housing type. This property is sometimes called subject reduction.

(Coherence) For each compatible derivation and reduction
\[
(\nabla^-) \quad \Gamma \vdash t^- : \tau \quad t^- \triangleright\triangleright t^+
\]
(as in the previous item) we have
\[
[\nabla^-] = [\nabla^+]
\]
that is, reduction preserves the meaning of a judgement.

Table 5: Four properties of the reduction relation

For example, each numeral \( \lbrack m_\xi \rbrack = S^m_\xi 0_\xi \)
is normal, and each of the terms
\[
\text{Add, Mlt, Exp,} \ldots
\]
(of Example 3.3) is normal.

Table 5 list four important properties of the reduction relation of a tiered system. We do not prove these properties here, but we do make quite a bit of use of them. Let’s look at the properties in turn.

(Normalization) Each derivation \( \Gamma \vdash t^- : \tau \)
certifies that the term $t^-$ makes sense, at least in the context $\Gamma$. This term $t^-$ need not be normal. However, this property ensures that

$$t^- \triangleright t^+$$

for at least one normal term $t^+$. Notice that here we use the reflexive version of the reduction relation. If we use $\triangleright$ then we have to say: either $t^- = t^+$ or $t^- \triangleright t^+$. The reflexive relation $\triangleright$ makes certain descriptions more concise.

As an example of this we have

$$\vdash \text{Big}[5] : \mathcal{N}$$

but this term is not normal. However, the property ensures that $\text{Big}[5]$ does reduce to a normal term. If you try to carry out this reduction you will do more damage to the universe than any politician could dream up.

(Confluence) We have seen that some terms can be reduced in different ways. However, in all the examples we have looked at we finish up with the same normal term no matter which reduction path we take. This is a general property.

The confluence property says that if we set off from a term $t^-$ and go in different directions, then no matter where we get to it is always possible to meet up later. Notice how the use of the reflexive relation makes this statement of this property quite neat.

There is a small point here which, so far, I have been rather quiet about. When are two terms $t_1$ and $t_2$ ‘the same’? The obvious answer is when they are exactly the same piece of syntax. However, we need a slightly more liberal notion.

Two terms $t_1$ and $t_2$ are ‘the same’ if they are alphabetic variants, that is they differ only in the choice of bound identifiers.

As an example the two terms

$$\lambda u_1 : \mathcal{N}, v_1 : \mathcal{N}. I u_1 S v_1 \quad \lambda u_2 : \mathcal{N}, v_2 : \mathcal{N}. I u_2 S v_2$$

are alphabetic variants and are regarded as the same term. Each names the addition operation.

(Influence) Suppose we have a derivation

$$(\nabla^-) \quad \Gamma \vdash t^- : \tau$$

and we take the root term $t^-$ and start to reduce it

$$t^- \triangleright t^+$$

to obtain some other term $t^+$. This reduction process seems to ignore the typing discipline, and at first sight it is not clear that the reduced term $t^+$ is well formed. The influence property ensures that it is.

(Coherence) This property ensures we can mix up the typing discipline and the reduction mechanism. Each derivable judgement

$$(\nabla^-) \quad \Gamma \vdash t^- : \tau$$

32
names some function, or element, $[\nabla^-]$. However, the name $t$ may not be in a the form that we want. Coherence ensures that we can start to reduce $t$ and, no matter which path we take, we will not change the meaning of the term. We have seen instances of this in Examples 4.6 and 4.7. Terms which name a natural number in an unnatural way have been reduced to the appropriate numeral.

We will make quite a lot of use of these four properties. As a taster let’s prove a small result which you may have already assumed is true without realizing it.

Consider Definition 3.6. This says each derivation

$$(F) \vdash \Gamma f : \xi(l) \rightarrow \cdots \rightarrow \xi(1) \rightarrow \xi(0)$$

is some tiered system gives a function

$$N \xrightarrow{f} N$$

in uncurried form. The definition gives the impression that $f$ depends only on the term $\Gamma f$. But how do we know this?

4.10 THEOREM. For each derivation $F$, as above, the meaning $[F]$ depends only on the term $\Gamma f$, and not the particular derivation.

Proof. Suppose we have two derivations $F_1, F_2$ of the same judgement. Each will give us a function

$$f_1 = [F_1], f_2 = [F_2]$$

of the indicated type. How do we know that $f_1 = f_2$? Let’s do this slowly.

We have a pair of derivations

$$(F_1) \vdash \Gamma f : \xi(l) \rightarrow \cdots \rightarrow \xi(1) \rightarrow \xi(0) \quad (F_2) \vdash \Gamma f : \xi(l) \rightarrow \cdots \rightarrow \xi(1) \rightarrow \xi(0)$$

using a common term $\Gamma f$. Consider $m_1, \ldots, m_l \in N$ let

$$\Gamma m_\xi(i) = \Gamma m_i, m_\xi(i)$$

for each $1 \leq i \leq l$, and form

$$t = \Gamma f^m m_\xi(l) \cdots m_\xi(1)$$

to obtain a second pair of derivations

$$(T_1) \vdash t : \xi_0 \quad (T_2) \vdash t : \xi_0$$

in a common term $t$. The orthodox semantics gives us two members

$$[T_1] = [F_1][\Gamma m_\xi(l)] \cdots [\Gamma m_\xi(1)] \quad [T_1] = [F_1][\Gamma m_\xi(l)] \cdots [\Gamma m_\xi(1)]$$

of $N$. In fact, we have

$$[T_1] = f_1(m_l, \ldots, m_1) \quad [T_2] = f_2(m_l, \ldots, m_1)$$

by unravelling the semantics, so it suffices to show $[T_1] = [T_2]$. 33
By (Normalization) we have
\[ t \leadsto s_1 \quad t \leadsto s_2 \]
for some normal terms \( s_1, s_2 \). Notice that, as yet, we cannot assert that \( s_1 = s_2 \). The reduction \( t \leadsto s_i \) may depend on the derivation \((T_i)\). Nevertheless, we can show \( s_1 = s_2 \), as follows.

By (Influence) we have derivations
\[
(S_1) \vdash s_1 : \xi_0 
\quad (S_2) \vdash s_2 : \xi_0
\]
and
\[
[T_1] = [S_1] 
\quad [T_2] = [S_2]
\]
holds by (Coherence). A use of (Confluence) gives
\[ s_1 \leadsto r \quad s_2 \leadsto r \]
for some term \( r \). But both \( s_1 \) and \( s_2 \) are normal, and hence
\[ s_1 = r = s_2 \]
as suggested.

Thus we have a pair of derivations
\[
(S_1) \vdash r : \xi 
\quad (S_2) \vdash r : \xi
\]
where \( \xi = \xi_0 \) and the common term \( r \) is normal. We now assert that \( r \) must be a numeral, say \( \lceil m \xi \rceil \) for some \( m \in \mathbb{N} \). This gives
\[ f_i(m_1, \ldots, m_1) = [T_i] = [S_i] = [\lceil m \xi \rceil] = m \]
for \( 1 = 1, 2 \), and hence
\[ f_1(m_1, \ldots, m_1) = f_2(m_1, \ldots, m_1) \]
as required. ■

There is just one problem with this argument. How do we know that the normal inhabitant \( r \) of \( \xi \) is a numeral? At this stage we don’t, but we will prove that result in Theorem 7.1. This may suggest there is some circularity in the arguments. In fact, the proof of Theorem 7.1 can be carried out much earlier, but it is more convenient to put it next to similar and more complicated arguments.

A closer look at the proof of this result gives us a bit more information.

4.11 SCHOLIUM. For each derivation \( F \), as above, the meaning \( [F] \) depends only on the unique normal form of the term \( \lceil f \rceil \).

Put another way, a derivation of a term can contain quite a lot more information than the orthodox semantics of that derivation.

[\textit{ Held in 165../05-version.../004.. Last changed April 13, 2006}]

34
5 Tiered systems and the classification programme

By definition a tiered system is an applied $\lambda$-calculus (with products) of the kind described is Section 2. Each system $\lambda T$ is determined by two factors, one minor and one major.

- The set $I$ indexes the coloured atoms $N[i]$ that are available. This determines the type structure of $\lambda T$, and some of the term structure. Remember that all atoms (monochrome and coloured) are furnished with a zero and a successor and there are three pairing gadgets for all product types.

- The battery of iterators
  
  $I_{r,\xi} : \xi \rightarrow \tau''$

  determines which recursions are available in the system. Remember that only selected pairs $\xi, \tau$ support an iterator, and $\xi$ is always an atom.

Of these by far the most important factor is the battery of iterators. It is these that determine the strength of the system.

Let’s look at some particular systems.

$\lambda A$ The simplest tiered system has just the monochrome atom $N$, no coloured atoms, and no iterators. This is a part of every other tiered system. An extensive analysis of this system has been known for many years. We look at this in some depth in these notes.

$\lambda B$ We enriched $\lambda A$ by adding one new furnished atom $N[\star]$ and an iterator

$I : N[\star] \rightarrow N''$

and no other gadgetry. This can be viewed as the system which underlies the result of Bellantoni and Cook given in Theorem 1.1 (although the original presentation wasn’t in these terms, and Theorem 1.1 isn’t exactly what Bellantoni and Cook proved).

$\lambda C$ We further enriched $\lambda A$ by adding a furnished atom and an iterator

$I_{r} : N[\tau] \rightarrow \tau''$

for each type $\tau$ generated from $N$. This includes $\lambda B$ since $N[\star] = N[N]$ and $I = I_{N}$. It is a more rounded tiered system, and it is considerably stronger than $\lambda B$. We will analyse this system in some detail.

$\lambda D$ This is a more economical version of $\lambda C$. We add to $\lambda A$ just one new furnished atom $N[\star]$ (as in $\lambda B$), but a whole battery of iterators

$I_{r} : N[\star] \rightarrow \tau''$

for each type $\tau$ generated from $N$. Anything that can be done in $\lambda C$ can be done in $\lambda D$, for we simply use $N[\star]$ in place of each $N[\tau]$. 

35
To obtain $\lambda E$ we iterate the trick which took us from $\lambda A$ to $\lambda D$. We add a sequence

$\mathcal{N}[0], \mathcal{N}[1], \ldots, \mathcal{N}[i], \ldots \ (i < \omega)$

of furnished atoms together with iterators

$I_{\tau,i} : \mathcal{N}[i] \rightarrow \tau''$

for each $i < \omega$ and each type $\tau$ built up from $\mathcal{N}, \mathcal{N}[0], \mathcal{N}[1], \ldots, \mathcal{N}[i-1]$. Thus $\lambda E$ is layered with $\lambda D$ appearing as layer $i = 0$.

$\lambda F$ This is a coalesced version of $\lambda B$. We add to $\lambda A$ one iterator

$I : \mathcal{N} \rightarrow \mathcal{N}''$

(without the coloured atom $\mathcal{N}[\star]$). Anything that can be done in $\lambda B$ can be done in $\lambda F$ (with $\mathcal{N}[\star]$ replaced by $\mathcal{N}$). This is an enfeebled version of the term calculus of Primitive Recursive Arithmetic. You may think of $\lambda B$ as a tiered version of $\lambda F$.

$\lambda G$ This is a coalesced version of $\lambda C$ and of $\lambda D$. There are no coloured atoms. We add to $\lambda A$ an iterator

$I_{\tau} : \mathcal{N} \rightarrow \tau''$

for each type $\tau$ (generated from $\mathcal{N}$). This is the term calculus of a version of Gödel’s T. You may think of $\lambda C$ as a 1-tiered version of $\lambda G$.

$\lambda L$ In a sense this is the free-est tiered system. At first sight the type structure looks rather intricate. Begin to generate the types from the monochrome atom $\mathcal{N}$. As each type $\tau$ is generated throw a new coloured atom $\mathcal{N}[\tau]$ into the pot from which further types can be generated. Also, for each generated type (in which there may be highly nested uses of atoms as indexes) there is an iterator

$I_{\tau} : \mathcal{N}[\tau] \rightarrow \tau''$

on the atom associated with $\tau$. Thus $\lambda C$ is the first layer of $\lambda L$. You may think of $\lambda L$ as a fully tiered version of $\lambda G$. This is essentially the system used by Leivant to prove the result of Theorem 1.1.

We will meet several other examples of tiered systems later in the notes.

How strong are these systems? We ought to be more precise with this question, since ‘strength’ can mean different things to different people. Roughly speaking in these notes we measure the strength of a system $\lambda T$ by the class first order functions that can be captured within the system. There are various ways to capture functions and these lead to different measures of strength. We deal with these different methods in Section 6, but for now the intuitive idea will suffice.

Consider any such tiered system $\lambda T$. Anything that can be done in $\lambda A$ can be done in $\lambda T$ (since $\lambda A$ is a part of $\lambda T$). Anything that can be done in $\lambda T$ can be mimicked in $\lambda G$ (at least for the kind of tiered systems we consider here). Thus the strength of $\lambda T$ will be somewhere between that of $\lambda A$ and $\lambda G$.

Depending on the methods we use the class of functions captured in $\lambda A$ lies somewhere between the class $\mathcal{T}$ of translations (at the weaker end) and the class $E_3$ of Kalmár
elementary functions (at the stronger end). The system $\lambda G$ is much stronger. Every function in the Grzegorczyk class $\mathcal{E}_{\varepsilon_0}$ can be captured in the system. If you are not sure what either of these classes are I give a brief survey of this system in Sections 19 and 21.

Thus, it seems we are looking for subclasses of $\mathcal{E}_{\varepsilon_0}$. However, the whole point of tiering is that it can severely weaken a system. For each tiered system we analyse here the strength is below $\mathcal{E}_3$. This is true even for the fully tiered version $\lambda L$ of $\lambda G$. The aim of these notes is to investigate just how tiering achieves this reduction in strength.

6 The various strengths of a system

Several times we have mentioned the strength of a tiered system $\lambda T$ but, as yet, we have not made this notion precise. In this section we correct that omission by attaching to $\lambda T$ four classes of functions

$$\mathcal{N}(\lambda T) \quad \mathcal{R}(\lambda T) \quad \mathcal{U}(\lambda T) \quad \mathcal{S}(\lambda T)$$

each of which can claim to be the strength of $\lambda T$. We find that

$$\mathcal{N}(\lambda T) = \mathcal{R}(\lambda T)$$

but the three classes

$$\mathcal{R}(\lambda T) \quad \mathcal{U}(\lambda T) \quad \mathcal{S}(\lambda T)$$

can be different. In later sections we determine these classes for certain particular systems $\lambda T$. This contributes to the short term classification programme.

In due course we will modify these notations and so refine these measure of complexity. However, these measures will do for now.

It will become clear that these are not the only possible measures of the strength of $\lambda T$. I will make some comments on other possibilities, but we will not pursue these directions here.

In this account we measure the strength of $\lambda T$ by a class of first order functions that are captured by $\lambda T$. Thus we are interested in functions

$$f : \mathbb{N}(l)$$

that is

$$f : \mathbb{N} \longrightarrow \cdots \longrightarrow \mathbb{N} \longrightarrow \mathbb{N}$$

where there are $l$ inputs and one output, each of which is in $\mathbb{N}$. But how can $\lambda T$ ‘capture’ such functions? There are several methods.

The most obvious method is the one that is hardly ever mentioned in the literature.

Consider atoms $\xi(0), \ldots, \xi(l)$ of $\lambda T$. Each $\xi(i)$ is either $\mathcal{N}$ or one of the coloured atoms of $\lambda T$. However, we do have

$$[\xi(i)] = \mathbb{N}$$

for each $0 \leq i \leq l$. 37
Consider a derivation 

\[(F) \vdash \Gamma : \xi(l) \to \cdots \to \xi(1) \to \xi(0)\]

in \(\lambda T\). This has an empty context and, as explained in Section 3, especially Definition 3.6, produces a function \(f : \mathbb{N}(l)\) obtained via the meaning \([F]\) of \(F\). Furthermore, by Theorem 4.10 we know that \(f\) depends only on the term \(\Gamma\), not the particular derivation \(F\) (or rather, we will know that once we have proved Theorem 7.1).

6.1 DEFINITION. Let \(\lambda T\) be a tiered system.

A first order function \(f : \mathbb{N}(l)\) is named in \(\lambda T\) if there is a derivation \((F)\), as above, which produces \(f\), that is the meaning \([F]\) selects \(f \in \mathbb{N}(l)\).

Let \(\mathcal{N}(\lambda T)\) be the class of functions named in \(\lambda T\). ■

Suppose the function \(f\) is named in \(\lambda T\). There is at least one derivation \(F\) involved with some term \(\Gamma\). We often say that \(\Gamma\) is a name of \(f\), or that \(f\) is named by \(\Gamma\). Thus a function may have many different names, and there will be significant differences between some of these names. We touch on this later in this section.

This is one method by which \(\lambda T\) can capture functions. You might ask why we capture only first order functions. This is a good point, and certainly the idea can be lifted to higher order functions. However, very little is known about the corresponding classes, even for familiar systems such as \(\lambda A\), \(\lambda F\), and \(\lambda G\). There is little point in making a definition if not much is done with it. This is a topic which, on the whole, is in the far distance of the long term classification programme.

There is another variant of this idea of naming which is worth pursuing. We will touch on this briefly later on, but won’t develop it in any depth. However, it should be given a good going over at some time.

As an example suppose we have a system with just one coloured atom \(S\), say sepia coloured, alongside the standard monochrome atom \(N\). (We actually look at such a system in Section 12.) How might we name a 2-placed function? There are eight possibilities, namely

\[\vdash \Gamma : \mathcal{N} \to \mathcal{N} \to \mathcal{N}\]
\[\vdash \Gamma : S \to \mathcal{N} \to \mathcal{N}\]
\[\vdash \Gamma : \mathcal{N} \to S \to \mathcal{N}\]
\[\vdash \Gamma : S \to S \to \mathcal{N}\]

and together with the four types with \(S\) in the target position. This gives eight classes of 2-placed functions that are named in this system, and these classes need not be the same.

For an arbitrary system \(\lambda T\) the class \(\mathcal{N}(\lambda T)\) can be refined and stratified by taking note of the colours used in a naming derivation. This will lead to a finer measure of the strength of \(\lambda T\). We don’t do this much in this document, but we do touch on it briefly in Section 16.

The remaining three classes are obtained using the computation mechanism of \(\lambda T\). Again we capture classes of first order function (and this time there are severe difficulties in lifting these ideas to higher order functions).

For the first computation method we again start from a derivation \((F)\), as above. Each atom \(\xi\) of \(\lambda T\) has its family of authentic numerals

\[\Gamma m\xi = S_{m\xi}^{\xi(0)}\xi\]
(for $m \in \mathbb{N}$) generated from the furnishings of $\xi$. Using a selected list $m(1), \ldots, m(l) \in \mathbb{N}$ of natural numbers we can form a compound term
\[ t = \Gamma f^\forall m(l)_{\xi(l)} \cdots m(1)_{\xi(1)} \]
with
\[ \vdash t : \xi(0) \]
that is $t$ inhabits the target atom $\xi(0)$. The indexing here
\[ \Gamma m(i)_{\xi(i)} \]
is a little finicky, but once we get used to the ideas involved we will use an abbreviated form.

We start to reduce this compound term $t$. By (Normalization) this will produce a normal term $t^*$ with $t \impliedby t^*$. By (Influence) we have $\vdash t^* : \xi(0)$. By Theorem 7.1 (to be proved in Section 7) we find that $t^*$ is a numeral $\Gamma m_{\xi(0)}$ over $\xi(0)$. Suppose $t \impliedby \Gamma m_{\xi(0)} \impliedby t \impliedby \Gamma n_{\xi(0)}$ are two such numerals. By (Confluence) there is a pair of reductions
\[ \Gamma m_{\xi(0)} \impliedby t^+ \quad \Gamma n_{\xi(0)} \impliedby t^+ \]
to a common term $t^+$. But both the numerals are normal, to give
\[ \Gamma m_{\xi(0)} = t^+ = \Gamma n_{\xi(0)} \]
and hence $m = n$.

In other words, for a term $\Gamma f$ of the indicated type, the reduction mechanism assigns to each $l$-tuple
\[ (m(l), \ldots, m(1)) \in \mathbb{N}^l \]
a unique $m \in \mathbb{N}$. Surely, this process has calculated a value of a function.

6.2 **DEFINITION.** Let $\Lambda T$ be a tiered system.

A first order function $f : \mathbb{N}(l)$ is **represented** in $\Lambda T$ if there is a derivation $(F)$, as above, such that
\[ \Gamma f^\forall m(l)_{\xi(l)} \cdots m(1)_{\xi(1)} \impliedby \Gamma m_{\xi(0)} \]
for each $m(0), m(1), \ldots, m(l) \in \mathbb{N}$ with $f(m(l), \ldots, m(1)) = m(0)$.

Let $\mathcal{R}(\Lambda T)$ be the class of functions represented in $\Lambda T$. 

We often say the term $\Gamma f$ is a representation of the function $f$, or that $f$ is represented by $\Gamma f$.

Notice that the reflexive version of the reduction relation is used here. This is to prevent a bit of silliness. Consider the constant
\[ S : \mathbb{N}' \]
which names the successor function on $\mathbb{N}$. For each $m \in \mathbb{N}$ we do **not** have
\[ S \Gamma m \impliedby \Gamma m + 1 \]
but we do have

\[ S^\gamma m^\gamma = \gamma m + 1^\gamma \]

and the reduction relation \( \triangleright \) is irreflexive, by definition. Thus, we do have

\[ S^\gamma m^\gamma \triangleright \gamma m + 1^\gamma \]

and so \( S \) does represent the successor function. This is one reason why the reflexive version of reduction is used. It is true that for most instances of a representation of a function we can use the irreflexive version, but occasionally we do need the reflexive version. Keep your eyes open for other examples of this.

As indicated in the build up to this definition, we need a background result before it can be made. That result is proved in Section 7. Later the methods used in the proof are developed to obtain further important results. However, when first seen these proofs can look like an exercise in nit-picking, 'What is the point of this?' you may ask. This section provides a partial answer to that question.

As with \( \mathfrak{R}(\lambda T) \) the class \( \mathfrak{R}(\lambda T) \) can be refined by taking note and influence of the colour of the atoms used in the representing derivation. That is something that should be done in the near future, for it will lead to interesting results. However, it is not done here.

Let's take another look at the derivation \((F)\). This provides two functions in \( \mathbb{N}(l) \), the one named and the one represented. Can these be different?

6.3 THEOREM. In a tiered system \( \lambda T \), each term

\[ \vdash \gamma f : \xi(l) \rightarrow \cdots \xi(1) \rightarrow \xi(0) \]

names precisely one function, represents precisely one function, and these two functions are the same.

Proof. Much of this proof has been indicated already, but let's set it down in one place.

Consider a derivation

\[ (F) \vdash \gamma f : \tau \]

of the judgement in question. By passing through the semantics of a \( F \) we obtain the unique function named by \( F \), the one selected by the meaning \([F]\) of \( F \). By Theorem 4.10 this function depends only on the term \( \gamma f \), not the particular derivation. Thus \( \gamma f \) names precisely one function. We also give a slightly different proof of this uniqueness here.

As indicated in the build up to Definition 6.2 the term \( \gamma f \) represents a unique function.

Let \( g \) be a function named by \( \gamma f \) and let \( h \) be the function represented by \( \gamma f \). Consider \( m(1), \ldots, m(l) \in \mathbb{N} \) and let

\[ m = g(m(1), \ldots, m(l)) \quad n = h(m(1), \ldots, m(l)) \]

be the values of the two functions. We show \( m = n \), so that \( g = h \) (and in consequence there is only one function named by the term \( \gamma f \)).

Let

\[ t = \gamma f^\gamma m(l)^\gamma \cdots m(1)^\gamma \]
(where, to save on clutter, we have omitted the typing subscripts). We have

\[ m = [t] \quad t \vdash \tau n \]

by the definition of naming and representing. However, (Coherence) applied to the reduction condition gives

\[ n = [\tau n] = [t] = m \]

as required.

This gives us the equality claimed earlier.

### 6.4 COROLLARY

For each tiered system \( \lambda T \) we have \( \mathfrak{N}(\lambda T) = \mathfrak{R}(\lambda T) \).

Naming and representing are two sides of the same coin. A name \( \tau f \) of a function \( f \) isolates that function and gives an algorithm for evaluation \( f \). To run the algorithm we supply \( \tau f \) with the appropriate list of numerals and start to reduce that compound term. Eventually this recursion process must stop, and can only stop at the numeral for the corresponding value of \( f \). Of course, there may be different reduction paths to that normal form, so the algorithm is non-deterministic, and there is scope for some strategic improvement in efficiency. A more serious problem is that any algorithm given by this particular name \( \tau f \) could be inherently inefficient in that there may be some other term \( \tau f' \) which names \( f \) and gives a better algorithm. The process by which we might convert \( \tau f \) into \( \tau f' \) is not understood.

This gives us one measure of the strength of a system \( \lambda T \), and this is perhaps the most important measure. We can get at it either by analysing the semantics of the system or by analysing its computation facilities. Furthermore, an analysis of one aspect can provide information about the other aspect.

We set up two further measures of the strength of \( \lambda T \). Both of these use the computation mechanism but with different numerals. The following definition explains why numerals \( \tau m \xi \) used so far are called authentic.

### 6.5 DEFINITION

For each type \( \tau \) and \( m \in \mathbb{N} \) we set

\[ \overline{m}_\tau = \lambda y : \tau', x : \tau. y^m x \]

to obtain the **simulated numerals** over \( \tau \).

Almost trivially we have

\[ \vdash \overline{m}_\tau : \tau'' \]

and the term \( \overline{m}_\tau \) is normal. However \( \overline{m}_\tau \) does not name a natural number. To see what it does name let \( T = [\tau] \), so that \( \overline{m}_\tau \) names a function \( M \in T'' \). It is not too hard to see that

\[ M f x = f^m x \]

for each \( f \in T', x \in T \). In other words \( \overline{m}_\tau \) names the \( m \)-fold iteration on \( T \).

The terms \( \overline{m}_\tau \) are often called **Church numerals**, and are used to capture functions. Before we give the general definition let’s look at some examples.
6.6 EXAMPLE. Let \( \zeta \) be an arbitrary type and consider the terms on the left below.

\[
\begin{align*}
\overline{S}_\zeta &= \lambda u : \zeta'' \cdot \lambda y : \zeta' \cdot x : \zeta \cdot y(uyx) \quad \zeta'' \rightarrow \zeta'' \\
\overline{A}_\zeta &= \lambda v,u : \zeta'' \cdot \lambda y : \zeta' \cdot x : \zeta \cdot (vy)(uyx) \quad \zeta'' \rightarrow \zeta'' \rightarrow \zeta'' \\
\overline{M}_\zeta &= \lambda v,u : \zeta'' \cdot \lambda y : \zeta' \cdot x : \zeta \cdot v(uy)x \quad \zeta'' \rightarrow \zeta'' \rightarrow \zeta'' \\
\overline{E}_\zeta &= \lambda v : \zeta''' \cdot u : \zeta'' \cdot \lambda y : \zeta' \cdot x : \zeta \cdot vuyx \quad \zeta''' \rightarrow \zeta'' \rightarrow \zeta''
\end{align*}
\]

For each such term \( \overline{F}_\zeta \) we find that

\[\vdash \overline{F}_\zeta : \tau\]

where \( \tau \) is the type given to its right. Observe that the type for \( \overline{E}_\zeta \) is not quite the same as that for \( \overline{A}_\zeta \) and \( \overline{M}_\zeta \).

To avoid a bit of subscript clutter let us write \( F \) for \( \overline{F}_\zeta \). Also let

\[
\overline{m} = \overline{m}_\zeta \quad \overline{n} = \overline{n}_\zeta
\]

for \( m, n \in \mathbb{N} \).

We deal with \( \overline{S}, \overline{A}, \overline{M} \) first, and then deal with \( \overline{E} \).

A simple exercise gives

\[
\begin{align*}
\overline{S}mst &\triangleright t(\overline{m}st) \triangleright t(\overline{m}s) = t^{m+1}s \\
\overline{A}ntmtst &\triangleright (\overline{n})(\overline{m}st) \triangleright t^n(\overline{m}s) = t^{m+n}s \\
\overline{M}ntmtst &\triangleright \overline{n}(\overline{m}t)s \triangleright (t^n)m s \triangleright t^{m\times n}s
\end{align*}
\]

for each \( m, n \in \mathbb{N} \) and terms \( s, t \). More generally we have

\[
\overline{S}m \triangleright 1 + m \quad \overline{A}nm \triangleright m + n \quad \overline{M}nm \triangleright m \times n
\]

for all \( m, n \in \mathbb{N} \). Here the subscripted outputs are

\[
\overline{(1 + m)}_\zeta \quad \overline{(m + n)}_\zeta \quad \overline{(m \times n)}_\zeta
\]

respectively. This shows that the terms \( \overline{S}, \overline{A}, \overline{M} \) in some way capture the functions

\[
suc = \text{successor} \quad \text{add} = \text{addition} \quad \text{mlt} = \text{multiplication}
\]

respectively.

We carry out a similar manipulation with \( \overline{E} \). However, this time we have to be a bit more careful because of the types involved.

Let

\[
\overline{m} = \overline{m}_\zeta \quad \overline{n} = \overline{n}_\zeta
\]

for \( m, n \in \mathbb{N} \). Then we have

\[
\overline{E}nmtst \triangleright \overline{n}mtst \triangleright \overline{m}sts \triangleright t^m s
\]

for all \( m, n \in \mathbb{N} \) and terms \( s, t \). More generally we have

\[
\overline{E}nm \triangleright \overline{m}^n
\]

or

\[
\overline{E}_\zeta \overline{m}_\zeta \overline{m}_\zeta \triangleright (\overline{m}^n)_\zeta
\]

42
with the types showing. Thus, in some way, \( E \) captures \( \exp = \text{exponentiation} \)

but where we now have to be careful with the order in which the inputs are consumed.

The proof of the crucial reduction

\[
m^n s \gg t^m s
\]

is not immediate. Try to prove this using the obvious induction over \( n \). You will soon run into an obstruction, then realize how to get over it, and learn something. ■

In some way the terms

\[
S_\zeta \quad A_\zeta \quad M_\zeta \quad E_\zeta
\]

capture the operations

\[
suc \quad \text{add} \quad \text{mlt} \quad \exp
\]

respectively. We make this precise as follows. This should be compared with Definition 6.2. The reflexive version of the reduction relation is used for the same reason.

6.7 DEFINITION. Let \( \lambda T \) be a tiered system.

A first order function \( f : \mathbb{N}(l) \) is \textbf{simulated} in \( \lambda T \) if there is a derivation

\[
\vdash \overline{f} : \tau(l)^{''} \rightarrow \cdots \tau(1)^{''} \rightarrow \tau(0)^{''}
\]

such that

\[
\overline{f} m(l)_{\tau(l)} \cdots m(1)_{\tau(1)} \gg \overline{m(0)_{\tau(0)}}
\]

for each \( m(0), m(1), \ldots, m(l) \in \mathbb{N} \) with \( f(m(l), \ldots, m(1)) = m(0) \). ■

Again the indexing is a little finicky, but in time we will condense it.

Notice that a simulation does not need any numeric gadgetry within the system, but some may need pairing gadgets. Example 6.6 shows that successor, addition, multiplication, and exponentiation are simulated by the terms \( S_\zeta, A_\zeta, M_\zeta, E_\zeta \), respectively. These are terms in \( \lambda \) the pure typed \( \lambda \)-calculus without products.

These examples also illustrate other aspects of the notion. The simulation of \( \text{suc}, \text{add}, \) and \( \text{mlt} \) is more uniform than that of \( \exp \). We make this precise.

6.8 DEFINITION. Let \( \lambda T \) be a tiered system.

(u) A first order function \( f \in \mathbb{N}(l) \) is \textbf{uniformly simulated throughout} \( \lambda T \) if for each type \( \zeta \) there is a term

\[
(U) \quad \vdash \overline{f} : \zeta^{''} \rightarrow \cdots \rightarrow \zeta^{''} \rightarrow \zeta^{''}
\]

which simulates \( f \).

Let \( U(\lambda T) \) be the class of first order functions uniformly simulated throughout \( \lambda T \).

(s) A first order function \( f \in \mathbb{N}(l) \) is \textbf{simulated throughout} \( \lambda T \) if for each type \( \zeta \) there are types \( \zeta_1, \ldots, \zeta_l \) and a term

\[
(S) \quad \vdash \overline{f} : \zeta^{''} \rightarrow \cdots \rightarrow \zeta_1^{''} \rightarrow \zeta^{''}
\]

which simulates \( f \).

Let \( G(\lambda T) \) be the class of first order functions simulated throughout \( \lambda T \). ■

43
There are, of course, other variants of this idea, but they are not needed here.

We now have three measures
\[ \mathfrak{R}(\lambda T) = \mathfrak{R}(\lambda T) \quad \mathfrak{U}(\lambda T) \quad \mathfrak{S}(\lambda T) \]
of the strength of a system \( \lambda T \). How do these compare?

Since each uniform simulation is a simulation we have \( \mathfrak{U}(\lambda T) \subseteq \mathfrak{S}(\lambda T) \). Example 6.6 shows that
\[ \text{suc, add, mlt} \in \mathfrak{U}(\lambda T) \quad \exp \in \mathfrak{S}(\lambda T) \]
for each system \( \lambda T \). Later we will see that \( \exp \notin \mathfrak{U}(\lambda A) \), so the two classes are different.

How does simulation compare with representing and naming. To illustrate this let’s look at the meaning of the term \( E \zeta \).

We have
\[ \vdash E \zeta : \zeta''' \to \zeta'' \to \zeta'' \]
where
\[ \zeta''' \to \zeta'' \to \zeta'' = \zeta''' \to (\zeta'' \to \zeta'') = \zeta''' \to \zeta''' = \zeta^{(iv)} \]
is the type involved.

Let \( \mathbb{Z} = \llbracket \zeta \rrbracket \), so we are looking for a member of
\[ \mathbb{Z}^{(iv)} = \mathbb{Z}''' \to \mathbb{Z}'' \]
(and, of course, \( \mathbb{Z} \) need not be the set of integers). Note that
\[ \mathbb{Z}^{(iv)} = \mathbb{Z}''' \to \mathbb{Z}' = \mathbb{Z}'' \to \mathbb{Z}' \to \mathbb{Z} \to \mathbb{Z} \]
by unravelling the type structure. The term \( E \zeta \) names a function
\[ E : \mathbb{Z}'' \to \mathbb{Z}'' \quad E : \mathbb{Z}''' \to \mathbb{Z}' \to \mathbb{Z} \to \mathbb{Z} \]
depending on how we want to view it. To see what this is consider arbitrary
\[ \phi \in \mathbb{Z}'' \quad F \in \mathbb{Z}'' \quad f \in \mathbb{Z}' \quad x \in \mathbb{Z} \]
so that
\[ E \phi F f x = \phi F f x \quad E \phi F f = \phi F f \quad E \phi F = \phi F \quad E \phi = \phi \]
and hence \( E \) is the identity function of \( \mathbb{Z}'' \). Thus we have a term which names a rather trivial function but simulates a function which most of the world doesn’t understand.

In fact, things can get worse.

6.9 Example. For each \( l < \omega \) consider the \((l + 1)\)-placed function \( \mathfrak{V}_l \in \mathbb{N}(l + 1) \) given by
\[ \mathfrak{V}_l(m_l, \ldots, m_1, m_0) = m_1^{m_l} \]
for \( m_0, \ldots, m_l \in \mathbb{N} \). Thus \( \mathfrak{V}_0 \) is the identity function and \( \mathfrak{V}_1 \) is exponentiation. More generally we have
\[ \mathfrak{V}_{l+1}(m_{l+1}, m_l, \ldots, m_1, m_0) = m_0^{m_{l+1}^{m_l^{m_{l-1}^{\cdots^{m_1^{m_0}}}}}} = \mathfrak{V}_l(m_{l+1}^{m_l^{m_{l-1}^{\cdots^{m_1^{m_0}}}}}) \]
Let $\tau$ be any type and consider the stacked powers $\tau^{(i)}$ as generated in Definition 2.13. We may check that the term

$$B(\tau, l) = \lambda u_1 : \tau^{(l+2)}, \ldots, u_1 : \tau^{(3)}, u_0 : \tau^{(2)}. \lambda y : \tau', x : \tau. u_1 \cdots u_1 u_0 y x$$

names an identity function but simulates $\nabla_l$. Notice that the simulation uses a different type of numeral for each input. $lacksquare$

These stacking functions play a pivotal role in determining which functions can or cannot be simulated (in a certain form). Modifying the construction we obtain a 3-placed function $\nabla$ generated by

$$\nabla(0, m, n) = n \quad \nabla(1 + l, m, n) = m^{\nabla(l,m,n)} = \nabla(l, m, m^n)$$

for each $l, m, n \in \mathbb{N}$. It is easy to show that for each $l$ the 2-placed function $\nabla(l, \cdot, \cdot)$ can be simulated in $\lambda A$. In contrast, it is known that the 1-placed function $\nabla(\cdot, 2, 0)$ can not be simulated. We use these functions again later in these notes.

It is this tension between representation and simulation that lies at the heart of the tiering method.

[Last changed April 13, 2006]

7 The exhaustive search method

One measure of the strength of a tiered system is the class $\mathcal{R}(\lambda T)$ of first order functions named in $\lambda T$. How might we determine this class? Whatever method we might try, we will surely have to investigate, and perhaps classify, certain derivations

$$(\nabla) \quad \Gamma \vdash t : \tau$$

in $\lambda T$. Of course, we are interested in only those derivations $\nabla$ where the root predicate $\tau$ is a first order type. Perhaps we can survey all derivations of the restricted kind. However, the derivation of such a first order judgement may pass through several much higher order judgements, so we may be forced into considering larger and larger classes of derivations. Fortunately there are some circumstances when such a search through all possible derivations can be carried out. In this section we give a couple of simple examples of this method. Later in these notes we will develop the method further. The technique we use is perhaps one of the most fundamental proof theoretic methods.

Consider a derivation $\nabla$, as above. This has a meaning

$$[\Gamma] \xrightarrow{[\nabla]} [\tau]$$

which is a function of the indicated concrete type. How might we determine what this is? This is where the properties

- Normalization
- Confluence
- Influence
- Coherence

of Section 4 are useful.

45
By the (Normalization) we know there is a normal term $t^\ast$ with $t \rhd t^\ast$. By (Confluence) this is essentially unique in that any other normal reduction of $t$ will differ from $t^\ast$ only in the choice of the bound variables. By (Influence) there is a derivation

$$(\nabla^\ast) \quad \Gamma \vdash t^\ast : \tau$$

and, in fact, this can be generated from $\nabla$ and an organized version of the reduction $t \rhd t^\ast$. By (Coherence) we have $[\nabla] = [\nabla^\ast]$ to show that $\nabla$ and $\nabla^\ast$ name the same gadget.

This shows that if we wish to survey a certain class of derivations $\nabla$ to extract the meaning of each, then we may suppose the root term $t$ is normal. This helps to cut down the number of derivations we have to consider.

As a first example suppose we have a derivation

$$(\nabla) \quad \vdash t : \xi$$

in the empty context with a normal subject term $t$ and an atomic predicate type $\xi$. What can $t$ be? We know some examples of such terms, namely the authentic numerals $\lceil m, \xi \rceil$ over $\xi$. Are there any more? It’s hard to think of any. In fact, we can show these numerals are the only possible inhabitants.

How can we do this? We start to analyse how $\nabla$ can arise. Any derivation must be either a leaf (an Axiom or a Projection) or be built using one of the three construction rules (Weakening, Introduction, Elimination). In our case, because the context is empty, $\nabla$ can not be a Projection nor can it arise by a Weakening. We thus find that either $\nabla$ is an Axiom or arises from certain shorter derivations in a certain way. It doesn’t take too long to see that the only possible Axiom is $\vdash 0_\xi : \xi$. To continue we begin to analyse how the shorter derivations can arise. In this way we unravel all possible derivations of the restricted form, and we (almost) find that each does produce a numeral.

This is the essence of the exhaustive search method. Given a derivation of a root judgement of a certain form, we begin to unravel all the possible ways that it can arise, and with luck we can extract some further information. With a bit of organization, a proof by induction is on the cards. However, as you can probably see, even when the given root is quite simple, the full derivation may use more complicated judgements, in which case the appropriate induction hypothesis is not immediately obvious.

We will use this method to solve the numeral problem just discussed. We work in an arbitrary tiered system but we must deal with a restricted class of derivations satisfying the global conditions set out in Table 6. At this stage it may not be clear why we need to consider product types. The mist should rise as we go through the proof of the following.

7.1 THEOREM. For each derivation in an arbitrary tiered system $\lambda T$

$$(\nabla) \quad \vdash t : \tau$$

where the root judgement meets the condition of Table 6, one of the following applies

(atom) If $\tau$ is an atom $\xi$ then $t = \lceil m, \xi \rceil$ for some $m \in \mathbb{N}$.

(product) If $\tau$ is a product type the $t$ is $\text{Pair} lr$ for some terms $l, r$.

depending on the shape of $\tau$. 

46
We consider derivations in an arbitrary tiered system $\lambda T$

$$\Gamma \vdash t : \tau$$

where the root judgement meets the following restrictions.

(\Gamma) The context is empty.

(t) The subject term $t$ is normal.

(\tau) The predicate type $\tau$ is one of

- (atom) An atom $\xi$
- (product) A product type $\sigma \times \rho$

for some $\xi, \sigma, \rho$.

### Table 6: Global conditions for Theorem 7.1

Proof. We proceed by induction over the height of $\nabla$. In this induction we may meet other derivations not in the restricted form and, of course, we can not apply an induction hypothesis to these. We survey all the possible ways that $\nabla$ can arise.

Firstly, $\nabla$ may be a leaf.

(\nabla Axiom) Since we are working in a tiered system $\lambda T$ there are six possible shapes for $t : \tau$.

\[(i) \quad 0_\xi : \xi \quad (iv) \quad \text{Pair} : \sigma \to \rho \to \sigma \times \rho \]
\[(ii) \quad S_\xi : \xi' \quad (v) \quad \text{Left} : \sigma \times \rho \to \sigma \]
\[(iii) \quad I : \xi \to \sigma'' \quad (vi) \quad \text{Right} : \sigma \times \rho \to \rho \]

Here $\xi$ is an atom but $\sigma, \rho$ are arbitrary. Of these only (i) satisfies the global restrictions on the type $\tau$, and this case does give a numeral.

(\nabla Projection) This case can not occur since the context is empty.

Secondly, $\nabla$ may arise by a use of one of the construction rules, These will involve shorter derivation to which, if the global restrictions are met, we may apply the induction hypothesis.

(\nabla Weakening) This case can not occur since the context is empty.

(\nabla Introduction) This case can not occur since it produces a subject type which is neither an atom nor a product.

(\nabla Elimination) Here we have two shorter derivations

$$\vdash s : \rho \to \tau \quad \vdash r : \rho$$

with $t = sr$. We look at the way the left hand one can arise. One such possibility is another use of Elimination. We anticipate this and unravel this left hand side as far as possible through all uses of Elimination. Thus we obtain a family of shorter derivations

$$(\text{Q}) \quad \vdash q : \pi_k \to \cdots \to \pi_1 \to \tau \quad (P_i) \quad \vdash p_i : \pi_i \quad i \leq i \leq k$$

where

$$t = q p_k \cdots p_1$$

47
with $q$ and each $p_i$ normal. (If any component of $t$ is reducible, then $t$ itself is reducible.) Observe that the subject type

$$\chi = \pi_k \to \cdots \to \pi_1 \to \tau$$

must have a special shape and there are restrictions on $\tau$. We consider how $Q$ can arise.

(Q Axiom) There are six possible cases as listed above.

(i) This can not arise since $\chi$ must be compound.

(ii) Here we must have

$$\vdash S_\xi : \xi \to \xi \quad (P) \vdash p : \xi$$

that is $k = 1$ with $\tau = \xi$ and $q = S_\xi$ to give $t = S_\xi p$ for some $p$. But now the shorter derivation $P$ meets the global restrictions, and so a use of the induction hypothesis gives $p = [m_\xi]$ for some $m \in \mathbb{N}$. Hence $t = [m(m + 1)]_\xi$, which is a numeral, as required.

(iii) Here we have at least

$$\vdash 1 : \xi \to \sigma'' \quad (P) \vdash p : \xi$$

for some atom $\xi$ and type $\sigma$. In other words $p$ is $p_k$ and $\xi$ is $\pi_k$ with the remaining $\pi_i$ and $\tau$ making up $\sigma''$.

Suppose $k = 1$. Then we have

$$\vdash lp : \sigma''$$

so that $t = lp$ with $\tau = \sigma''$. But the global restrictions do not allow $\tau$ to have this shape, and hence $k \neq 1$.

This shows that there is at least one more shorter derivation

$$\vdash r : \sigma'$$

to give

$$\vdash lpr : \sigma'$$

as an initial part of $t$. Again $\tau$ can not be $\sigma'$ so we have a further shorter derivation

$$\vdash s : \sigma$$

to give

$$\vdash lprs : \sigma$$

as an initial part of $t$. In other words

$$t = (lprs) \cdots$$

where there may be more terms to come. Now look at the shorter derivation $P$. This meets the global restriction, and hence the induction hypothesis gives $p = [m_\xi]$ for some $m \in \mathbb{N}$. With this we have

$$t = (lm_\xi r s) \cdots \Rightarrow r^m s \cdots$$

which contradicts the assumed normality of $t$. Thus this case can not arise.

(iv) Here we have at least

$$\vdash \text{Pair} : \sigma \to \rho \to \sigma \times \rho \quad (L) \vdash l : \sigma$$

with

$$\chi = \sigma \to \rho \to \sigma \times \rho$$

which tells us what $\tau$ can be. One of
(1) $k = 1$ with $\pi_1 = \sigma$ and $\tau = \rho \rightarrow \sigma \times \rho$

(2) $k = 2$ with $\pi_2 = \sigma, \pi_1 = \rho$ and $\tau = \sigma \times \rho$

must occur. In more detail we have one of the following.

(1) We have

\[
(Q) \vdash \text{Pair} : \sigma \rightarrow \rho \rightarrow \sigma \times \rho \quad (L) \vdash l : \sigma
\]

with $t = \text{Pair} l$ and $\tau = \rho \rightarrow \sigma \times \rho$.

(2) We have

\[
(Q) \vdash \text{Pair} : \sigma \rightarrow \rho \rightarrow \sigma \times \rho \quad (L) \vdash l : \sigma \quad (R) \vdash r : \rho
\]

with $t = \text{Pair} lr$ and $\tau = \sigma \times \rho$.

However, case (1) can be discounted because of the global conditions on $\tau$. Case (2) meets the required conclusion (product).

(v) Here we have

\[
(Q) \vdash \text{Left} : \sigma \times \rho \rightarrow \sigma \quad (P) \vdash p : \sigma \times \rho
\]

with

\[
t = \text{Left} p \cdots
\]

where there may be more terms to come. The shorter derivation $P$ meets the global restrictions, and hence the induction hypothesis gives $p = \text{Pair} lr$ for some terms $l, r$. But now

\[
t = \text{Left}(\text{Pair} lr) \cdots \triangleright l \cdots
\]

which contradicts the assumed normality of $t$. Thus this case can not arise.

(vi) As with (v), this case can not arise.

This completes the case where $Q$ is an Axiom.

(Q Projection) This case can not occur since the context is empty.

(Q Weakening) This case can not occur since the context is empty.

(Q Introduction) In this case we have $q = (\lambda x : \pi. r)$ for some term $r$, where $\pi = \pi_k$. But then with $p = p_k$ we see that the redex $(\lambda x : \pi. r)p$ is an initial part of $t$. Since $t$ is normal, this case can not arise.

(Q Elimination) This case is built into the unravelling of the top use of Elimination. This concludes the full proof. ■

This example is typical of many arguments used in Proof Theory. We proceed by induction over the syntax involved. However, the appropriate induction hypothesis is not immediately obvious, and there can be many case and subcases to consider. It is easy to overlook one such case. Furthermore, many of the cases are often straightforward, and only a few are crucial.

At a first reading of the proof it may not be clear why products types are involved. I suggest you try to re-do the proof without referring to product types, and see what happens when you get to case (V Elimination), subcase (Q Axiom), subsubcases (v, vi).

In Section 6 we used Theorem 7.1 to justify the idea of representing a function, as introduced in Definition 6.2. The proof of Theorem 6.3 also makes a tacit use of Theorem 7.1. Here is another version of that result.
7.2 THEOREM. Suppose, in some tiered system $\lambda T$, the term

\[
\vdash \Gamma : \xi_l \to \cdots \to \xi_1 \to \xi_0
\]

names the function $f : \mathbb{N}(l)$. Then

\[
\Gamma \vdash \lambda m : \xi_l \cdots \xi_1 \to \xi_0 \vdash fm_l \cdots m_1
\]

holds for all $m_1, \ldots, m_l \in \mathbb{N}$.

Proof. Given $m_1, \ldots, m_l \in \mathbb{N}$ let $m_0 = fm_l \cdots m_1$ be the value of the function at the given arguments. For each $0 \leq i \leq l$ we write $\Gamma m_i$ for $\Gamma m_{\xi_i}$ to save on clutter.

Let

\[
t = \Gamma m_l \cdots m_1
\]

so that

\[
(\nabla) \vdash t : \xi_0 \quad [t] = m_0
\]

hold. The derivation $\nabla$ can be obtained from the derivation giving $\Gamma$.

By (Normalization) there is a reduction $t \Downarrow t^*$ to some normal term $t^*$. By (Influence) we have $\vdash t^* : \xi_0$ and hence Theorem 7.1 gives $t^* = \Gamma n_0$ for some $n \in \mathbb{N}$. By (Coherence) we have

\[
m_0 = [t] = [t^*] = n_0
\]

to give the required result. \[\blacksquare\]

We now look at another example off the exhaustive search method. But this time we work within $\lambda A$, not an arbitrary $\lambda T$. As before we deal with a restricted class of derivations satisfying the following global conditions set out in Table 7.

What are we getting at here? As in the previous example the product types are needed to make a certain induction go through. We are primarily interested in the function case. Consider such a derivation

\[
(\nabla) \Gamma \vdash t : N(l)
\]

with predicate type

\[
N(l) = (N \to \cdots N \to N)
\]

where there are $l + 1$ occurrences of $N$. Let

\[
\Gamma = x_m : N, \ldots, x_1 : N
\]

be the context (where, of course, the empty case $m = 0$ is allowed). The meaning of $\nabla$ is a certain function

\[
\mathbb{N}^m \xrightarrow{F} (\mathbb{N}^l \to \mathbb{N})
\]

where we have used a bit of licence in the target type. By applying a sequence of Introductions to $\nabla$ we obtain a derivation

\[
\vdash \Gamma : N(m + l)
\]

where

\[
\Gamma = \lambda x_m : N, \ldots, x_1 : N. t
\]

50
We consider derivations in $\lambda A$

$$(\nabla) \quad \Gamma \vdash t : \tau$$

where the root judgement meets the following restrictions.

(Γ) The context $\Gamma$ is a list of declarations $x : \mathcal{N}$.

(t) The subject term $t$ is normal.

(τ) The predicate type $\tau$ is one of

- (function) A function type $\mathcal{N}(l)$
- (product) A product type $\sigma \times \rho$

for some $\sigma, \rho$.

Table 7: Global conditions for Theorem 7.4

is the new subject term. The meaning of this new derivation is a function

$$\mathbb{N}^{m+l} \xrightarrow{f} \mathbb{N}$$

where again we have used a bit of licence to reorganize the type. In fact, in a common or garden situation we would hardly distinguish between $F$ and $f$, and it is just this kind of licence we have used here.

This shows that each derivation $\nabla$ can be converted into a derivation in the empty context with essentially the same meaning $f$. To describe this manipulation we say $\nabla$ provides the function $f$. In fact, we are mainly concerned with derivations

$$\vdash \overset{f}{\Gamma} : \mathcal{N}(k)$$

in the empty context, but to handle these we need to deal with a larger family as set out in the global conditions for Theorem 7.4.

7.3 DEFINITION. A $k$-placed function $f \in \mathbb{N}(k)$ is a translation if it has one of the two forms

$$f(x_k, \ldots, x_1) = \begin{cases} n + x_i \\ n \end{cases}$$

for some $n \in \mathbb{N}$ and index $1 \leq i \leq k$.

Let $\mathfrak{T}$ be the class of translations.

We are now ready to determine $\mathfrak{R}(\lambda A)$. This is our second example of the exhaustive search technique. In the proof we stick as close as possible to the phraseology of the proof of Theorem 7.1. This will help to show what is routine and what is particular to this proof.

7.4 THEOREM. For each derivation in $\lambda A$

$$(\nabla) \quad \Gamma \vdash t : \tau$$

where the root judgement meets the condition of Table 7, one of the following applies
(function) If $\tau$ is a function type then $\nabla$ provides a translation in $T$.

(product) If $\tau$ is a product type then $t$ is $\text{Pair}lr$ for some terms $l, r$.

depending on the shape of $\tau$.

Proof. We proceed by induction over the height of $\nabla$. To do this we survey all the possible ways that $\nabla$ can arise.

Firstly, $\nabla$ may be a leaf.

($\nabla$ Axiom) Since we are working in $\lambda \text{A}$ there are five possible shapes for $t : \tau$.

(i) $0 : N$  (iv) $\text{Pair} : \sigma \rightarrow \rho \rightarrow \sigma \times \rho$

(ii) $S : N'$  (v) $\text{Left} : \sigma \times \rho \rightarrow \sigma$

(vi) $\text{Right} : \sigma \times \rho \rightarrow \rho$

Here $\sigma, \rho$ are arbitrary. Some of these meet the global conditions and some do not.

(i) This provides the $m$-placed translation which is identically 0.

(ii) This provides the $m + 1$-placed translation $f$ where

$$f(x, y) = Sy = y + 1$$

for the appropriate $x$ and $y$.

(iii) As indicated, this case doesn’t occur since there are no iterators in $\lambda \text{A}$.

(iv, v, vi) These cases don’t arise since each of

$$\sigma \rightarrow \rho \rightarrow \sigma \times \rho \qquad \sigma \times \rho \rightarrow \sigma \qquad \sigma \times \rho \rightarrow \rho$$

has the wrong shape to be a predicate type for these global conditions.

($\nabla$ Projection) This provides the $m$-placed translation $f$ given by

$$f(x) = x_j$$

where $j$ indicates the position of the projected declaration.

Secondly, $\nabla$ may arise by a use of one of the construction rules. These will involve shorter derivation to which, if the global conditions are met, we may apply the induction hypothesis.

($\nabla$ Weakening) Using the induction hypothesis we see that the numerator of this particular use provides some translation $g$, and then the denominator provides a translation $f$ obtained from $g$ by inserting a dummy argument.

($\nabla$ Introduction) As explained above, the numerator and denominator of such a use provide the same function.

($\nabla$ Elimination) Here we have two shorter derivations

$$\Gamma \vdash s : \rho \rightarrow \tau \quad \Gamma \vdash r : \rho$$

with $t = sr$. We look at the way the left hand one can arise. We track through its construction to unravel as many uses of Elimination and Weakening as possible. Thus we obtain a family of shorter derivations

$$(Q) \quad \Xi \vdash q : \pi_k \rightarrow \cdots \rightarrow \pi_1 \rightarrow \tau \quad (P_i) \quad \Pi_i \vdash p_i : \pi_i \quad i \leq i \leq k$$
where

\[ t = qp_k \cdots p_1 \]

with \( q \) and each \( p_i \) normal, and where each context \( \Xi \) and \( \Pi_i \) is a left hand part of \( \Gamma \). Note that \( k \geq 1 \). In this process we may have removed some uses of Weakening. These can be reinstated to obtain derivations

\[
(Q^+) \quad \Gamma \vdash q : \pi_k \to \cdots \to \pi_1 \to \tau \quad (P_i^+) \quad \Gamma \vdash p_i : \pi_i \quad i \leq i \leq k
\]

all in the original context. Since \( k \geq 1 \), each of these is strictly shorter than \( \nabla \). (In fact, we do not need to reinstate the missing parts of \( \Gamma \). However, doing so makes the proof a bit easier to follow.)

We consider how \( Q \) can arise. To do this let

\[
\chi = \pi_k \to \cdots \to \pi_1 \to \tau
\]

be its predicate type.

(Q Axiom) There are five possible cases as listed above.

(i) This can not arise since \( \chi \) must be compound.

(ii) Here we must have

\[
(Q^+) \quad \Gamma \vdash S : \mathcal{N} \to \mathcal{N} \quad (P^+) \quad \Gamma \vdash p : \mathcal{N}
\]

with \( k = 1 \) and with \( \tau = \mathcal{N} \) and \( t = Sp \) for some term \( p \). The shorter derivation \( P^+ \) meets the global restriction and hence, by the induction hypothesis, this derivation provides a translation. The function provided by \( \nabla \) is the composite of this function followed by the successor function, and so is a translation.

(iii) This case doesn’t occur since there are no iterators in \( \lambda \mathcal{A} \).

(iv) Here we have at least

\[
(Q^+) \quad \Gamma \vdash \text{Pair} : \sigma \to \rho \to \sigma \times \rho \quad (L^+) \quad \Gamma \vdash l : \sigma
\]

with \( \chi = \sigma \to \rho \to \sigma \times \rho \)

which tells us what \( \tau \) can be. One of

1. \( k = 1 \) with \( \pi_1 = \sigma \) and \( \tau = \rho \to \sigma \times \rho \)
2. \( k = 2 \) with \( \pi_2 = \sigma, \pi_1 = \rho \) and \( \tau = \sigma \times \rho \)

must occur. In more detail we have one of the following.

1. We have

\[
(Q^+) \quad \Gamma \vdash \text{Pair} : \sigma \to \rho \to \sigma \times \rho \quad (L^+) \quad \Gamma \vdash l : \sigma
\]

with \( t = \text{Pair}l \) and \( \tau = \rho \to \sigma \times \rho \).

1. We have

\[
(Q^+) \quad \Gamma \vdash \text{Pair} : \sigma \to \rho \to \sigma \times \rho \quad (L^+) \quad \Gamma \vdash l : \sigma \quad (R^+) \quad \Gamma \vdash r : \rho
\]

with \( t = \text{Pair}lr \) and \( \tau = \sigma \times \rho \).
However, case (1) can be discounted because of the global conditions on \( \tau \). Case (2) meets the required conclusion (product).

(v) Here we have at least

\[
(Q^+) \quad \Gamma \vdash \text{Left} : \sigma \times \rho \rightarrow \sigma \quad (P^+) \quad \vdash p : \sigma \times \rho
\]

where there may be other auxiliary derivations \( P_2^+, \ldots, P_k^+ \). The shorter derivation \( P^+ \) meets the global conditions and hence the induction hypothesis gives \( p = \text{Pair}lr \) for some terms \( l, r \). But now

\[
t = \text{Left}(\text{Pair}lr) \cdots
\]

which since

\[
\text{Left}(\text{Pair}lr) \triangleright l
\]

contradicts the assumed normality of \( t \). Thus this case can not arise.

(vi) As with (v), this case can not arise.

This completes the case where \( Q \) is an Axiom.

(Q Projection) This case can not occur since the only possible projected statement from this context \( \Gamma \) has the form \( x : \mathcal{N} \), and this type can not match \( \chi \).

(Q Weakening) This case is built into the unravelling of \( \nabla \).

(Q Introduction) In this case we have \( q = (\lambda x : \pi. r) \) for some term \( r \), where \( \pi = \pi_k \). But then with \( p = p_k \) we see that the redex \( (\lambda x : \pi. r)p \) is an initial part of \( t \). Since \( t \) is normal, this case can not arise.

(Q Elimination) This case is built into the unravelling of \( \nabla \).

This concludes the full proof.

As before we are not too concerned with predicate types that are products, nor with non-empty contexts. What we want from this result is the following.

7.5 THEOREM. We have \( \mathfrak{N}(\lambda A) = \mathfrak{T} \). That is, the class of first order functions named in \( \lambda A \) is precisely the class \( \mathcal{T} \) of translations.

Proof. It is a trivial exercise to show that each translation is named in \( \lambda A \). Conversely, by Theorem 7.4 (using an empty context) if a function is named in \( \lambda A \) then it is a translation.

The proofs of Theorems 7.1 and 7.4 give us two examples of the exhaustive search method. Once we have got used to the general method, with the several cases and subcases to look at, the idea is not too bad. At this stage it is worth considering how the method might be refined to perhaps produce other, more complicated, characterizations along the lines of Theorem 7.5. That result is concerned with representability in \( \lambda A \). Why don’t we try the same method on a slightly stronger system, say \( \lambda B \) or even \( \lambda F \)? Both of these present new problems to be handled in the proof.

Consider what happens if we try to modify Theorem 7.4 to \( \lambda F \). We can still use the same global conditions as before. However, \( \lambda F \) has an extra axiom

\[
1 : \mathcal{N} \rightarrow \mathcal{N}''
\]
and this will ensure that a larger class \( \mathfrak{F} \) of functions is produced. What is this class \( \mathfrak{F} \)? To find out observe that the new axiom will give more cases to be considered in the proof.

Firstly, within case (\( \nabla \) Axiom) there is a subcase (iii)

\[
(\nabla) \quad \Gamma \vdash I : \mathcal{N} \to \mathcal{N}' \to \mathcal{N} \to \mathcal{N}
\]

arising from the new axiom. However, this predicate type does not match the required shape of \( \tau \) (because of the \( \mathcal{N}' \)), so this case doesn’t arise.

Secondly, with case (\( \nabla \) Elimination) subcase (Q, Axiom) there is a subsubcase (iii) where we have

\[
(Q^+) \quad \Gamma \vdash I : \mathcal{N} \to \mathcal{N}' \to \mathcal{N} \to \mathcal{N}
\]

and at least the first two and perhaps all three of

\[
(E^+) \quad \Gamma \vdash e : \mathcal{N} \quad (H^+) \quad \Gamma \vdash h : \mathcal{N}' \quad (G^+) \quad \Gamma \vdash g : \mathcal{N}
\]

where \( e, h, g \) are terms. We have either of

\[
t = lehg \quad t = leh
\]

depending on whether or not \( G^+ \) occurs. However, each of \( E^+, H^+, G^+ \) is a shorter derivation that meets the global conditions so, by the induction hypothesis, they provide functions

\[
e(x) \quad h(x, \cdot) \quad g(x)
\]

within \( \mathfrak{F} \). (As before, the inputs \( x \) arise from the identifiers declared in the context \( \Gamma \).) We then find that the function produced is

\[
f(x, y, z) = h(x, \cdot)\cdot y
\]

with \( e(x) \) substituted for \( z \) and, when required, \( g(x) \) substituted for \( y \). Since

\[
f(x, y, 0) = y \quad f(x, y, z') = h(x, f(x, y, z))
\]

we see that \( f \) is primitive recursive in \( h \).

With a bit of cleaning up this argument leads to the following characterization.

7.6 THEOREM. Each first order functions named in \( \lambda \mathcal{F} \) is primitive recursive.

You may be wondering if this result could be improved to give a precise description of the class of first order functions named in \( \lambda \mathcal{F} \). I don’t know if there is such a characterization, nor do I know if it is worth the effort to look for one. We don’t get all primitive recursive functions. This problem is concerned with the way iterators and product types can interact to produce other forms of recursion. We will say more about this in Section 13.

It seems reasonable that we could take this kind of argument a bit further, perhaps eventually dealing with the whole of \( \lambda \mathcal{G} \). We will not do this here, but we will discuss some of the appropriate results Section 19.

For another variant of Theorem 7.5 consider what happens if we modify Theorem 7.4 to \( \lambda \mathcal{B} \). Again we can still use the same global conditions as before, and again the system \( \lambda \mathcal{B} \) has an extra axiom, but this time it is

\[
1 : \mathcal{N}[]) \to \mathcal{N}''
\]
which uses two atoms. Since $\lambda A$ is a part of $\lambda B$ we can still get at all translations and perhaps more using the new iterator. You should try working through the appropriate version of Theorem 7.4, which will involve deciding what the analogous class of functions should be. We return to this in Section 12.

8 Clones of functions

As part of the short term classification programme we must determine the classes

$$\mathcal{R}(\lambda T) \quad \mathcal{U}(\lambda T) \quad \mathcal{S}(\lambda T)$$

for some tiered systems $\lambda T$, and in particular for some of the systems listed in Section 3. We can expect that certain simple functions are easily captured and belong to some or all of these classes. Indeed, Example 6.6 shows that

$$\text{suc, add, mlt} \in \mathcal{U}(\lambda T) \quad \exp \in \mathcal{S}(\lambda T)$$

for each system $\lambda T$. We can also expect that these classes have certain closure properties. In this section we consider generalities such as these.

As mentioned in Section 1, the Introduction, we view each $l$-place function

$$f : \mathbb{N}^l \to \mathbb{N}$$

in uncurried form. Thus a typical value of this function $f$ has the form

$$f(x_l, \ldots, x_1)$$

where

$$(x_l, \ldots, x_1) \in \mathbb{N}^l$$

is an $l$-tuple of members of $\mathbb{N}$. Quite often the components of this tuple are not important to name separately, so we let

$$x = (x_l, \ldots, x_1)$$

to get

$$fx$$

as a typical value of $f$.

Here is a collection of simple functions which are not always mentioned but are always around to do a useful bookkeeping job.

8.1 DEFINITION. For each $1 \leq i \leq l$ the $i^{th}$ projection function

$$\binom{i}{i} : \mathbb{N}^l \to \mathbb{N}$$

is given by

$$\binom{i}{i}x = x_i$$

for each $x \in \mathbb{N}^l$, where $x = (x_l, \ldots, x_1)$, as above.
Projections can be used to add dummy inputs to a function and to shuffle around the existing inputs.

How can we capture these very tame beasts?

Let $\lambda T$ be an arbitrary tiered system. Almost trivially we can name each projection function in $\lambda T$, but because of the colour coding aspect it is still worth looking at how this is done.

Let $\xi(1), \ldots, \xi(l)$ be a list of atoms of $\lambda T$. For $1 \leq i \leq l$ let

$$\gamma(i)_{\xi} = \lambda u_i : \xi(l), \ldots, u_1 : \xi(1) . u_i$$

where the subscript $\xi$ is intended to list the atoms involved. A trivial exercise gives

$$\vdash \gamma(i)_{\xi} : \xi(l) \rightarrow \cdots \rightarrow \xi(1) \rightarrow \xi(i)$$

and just as easily we see that $\gamma(i)_{\xi}$ represents $(i)_{\xi}$.

Notice that we have actually done more than merely represent $(i)_{\xi}$: we have a multi-coloured representation. In other words, the colour coding can never be used to obstruct a projection.

The simulation of $(i)_{\xi}$ is just as easy.

Let $\zeta(1), \ldots, \zeta(l)$ be a list of types of $\lambda T$. For $1 \leq i \leq l$ let

$$\lambda u_i : \zeta(l)^{\prime\prime}, \ldots, u_1 : \zeta(1)^{\prime\prime} . u_i$$

where the subscript $\zeta$ is intended to list the types involved. A trivial exercise gives

$$\vdash \lambda u_i : \zeta(l)^{\prime\prime}, \ldots, u_1 : \zeta(1)^{\prime\prime} . u_i$$

and just as easily we see that $(i)_{\zeta}$ simulates $(i)_{\xi}$.

As an aside, there are certain times when

$$\lambda u_i : \zeta(l)^{\prime\prime}, \ldots, u_1 : \zeta(1)^{\prime\prime} . u_i$$

might be a better simulation of $(i)_{\xi}$, but we won’t go into that here.

These observations give the following.

8.2 LEMMA. For each tiered system $\lambda T$ we have

$$(i)_{\xi} \in \mathcal{R} (\lambda T) \quad (i)_{\zeta} \in \mathcal{U} (\lambda T) \subseteq \mathcal{S} (\lambda T)$$

for all $1 \leq i \leq l$.

As remarked above, we have proved more than this. We have complete freedom over the choice of the types involved with the representation or simulation. We make tacit use of this facility in various manipulations.

The next thing we look at is closure under composition. With this we have to be a little bit more careful. It is not just 1-placed functions that we compose. Also, when we get inside a system $\lambda T$ the colour coding may obstruct some compositions.
8.3 DEFINITION. Consider an \( l \)-placed function \( h \) over \( N \) together with a list \( g_1, \ldots, g_l \) of \( k \)-placed functions over \( N \). Thus there is precisely one function \( g_i \) to match each input to \( h \), and each of the function \( g_1, \ldots, g_l \) have the same number of inputs, namely \( k \). The composite

\[
f = h \circ (g_l, \ldots, g_1)
\]

of these functions is the \( k \)-placed function given by

\[
f x = h(g_l x, \ldots, g_1 x)
\]

for each \( x \in N^k \).

This notion of composition is just a particular case of the standard notion of composition for \( 1 \)-placed function. We can bundle up the list \( g_1, \ldots, g_l \) to form a single function

\[
N^k (g_l, \ldots, g_1) \to N^l
\]

and then the composite

\[
N^k (g_l, \ldots, g_1) \to N^l \xrightarrow{h} N
\]

is precisely the function \( h \circ (g_l, \ldots, g_1) \) of the definition.

Let \( \lambda T \) be an arbitrary tiered system. How might we represent the function

\[
f = h \circ (g_l, \ldots, g_1)
\]

in \( \lambda T \)? Of course, we must first represent \( g_1, \ldots, g_l, h \). Thus suppose the terms

\[
\vdash \lambda h^n : \xi(l) \to \cdots \to \xi(1) \to \xi
\]

\[
\vdash \lambda g_i^n : \eta(k) \to \cdots \to \eta(1) \to \xi(i)
\]

\[
\vdash \lambda g_1^n : \eta(k) \to \cdots \to \eta(1) \to \xi(1)
\]

represent the functions in \( \lambda T \). Here \( \xi(1), \ldots, \xi(l) \) and \( \eta(1), \ldots, \eta(k) \) are two list of atoms with a compatibility between the output types of the \( g_i \) and the input types of \( h \). Consider the term

\[
\lambda f^n = \lambda u_l : \eta(k), \ldots, u_1 : \eta(1). \lambda h^n s_i \cdots s_1
\]

where

\[
s_i = \lambda g_i^n u_l \cdots u_1
\]

for each \( 1 \leq i \leq l \). A simple calculation shows that

\[
\vdash \lambda f^n : \eta(k) \to \cdots \to \eta(1) \to \xi
\]

and, furthermore, this term \( \lambda f^n \) represents \( f \) in \( \lambda T \).

This seems easy enough, but it does uncover a problem. To capture a composition within a system \( \lambda T \) we must respect the colour coding of that system. Let’s look at an example.
8.4 EXAMPLE. Consider the tiered system $\lambda M$ with just two atoms, $\mathcal{N}$ the standard monochrome one, and $\mathcal{M}$ a coloured one, say a mauve one. Suppose also the system has just one iterator

$$J : \mathcal{M} \to \mathcal{N}^{\text{(iv)}}$$

with a rather strange type. Observe that $\lambda M$ is not the same as $\lambda B$ because of the target type of the iterator, but it does sit inside $\lambda C$.

For each $m \in \mathbb{N}$ let

$$\overline{m} = \overline{m_N} \quad \overline{m}_\mathcal{N} = \overline{m}_\mathcal{M}$$

the simulated and authentic numerals over $\mathcal{N}$ and the authentic numeral over $\mathcal{M}$.

It is quite easy to produce a term

$$\vdash D : \mathcal{N}'' \to \mathcal{N}''$$

which simulates doubling, that is

$$D \overline{m} \triangleright 2 \cdot \overline{m}$$

for each $m \in \mathbb{N}$. Using $D$ we let

$$2 = \lambda u : \mathcal{M} . J u D T S 0$$

where $0, S$ are the furnishings of $\mathcal{N}$ and $T$ is the simulated numeral over $\mathcal{N}$.

It is not too hard to show that

$$\vdash 2 : \mathcal{M} \to \mathcal{N}$$

and

$$2^\top m^\top \triangleright 2^\top m$$

for each $m \in \mathbb{N}$, so we have a representation of the function $2^\top$, exponentiation to base 2.

Before we continue it is perhaps worth seeing where this term $2$ comes from, for this will help later.

Let

$$E = \lambda u : \mathcal{M} . J u D T$$

so that

$$\vdash E : \mathcal{M} \to \mathcal{N}''$$

and

$$E^\top m^\top \triangleright 2^\top m$$

for each $m \in \mathbb{N}$. We have captured exponentiation to base 2 using a mixture of authentic and simulated numerals. We will develop this idea further in later sections.

Next consider the term

$$C = \lambda w : \mathcal{N}'' . w S 0$$

again using the furnishings of $\mathcal{N}$. We find that

$$\vdash C : \mathcal{N}'' \to \mathcal{N}$$

and

$$C m^\top \triangleright \overline{m}$$
for each \( m \in \mathbb{N} \). In other words, the term \( C \) converts simulated numerals into authentic numerals. This is another idea we develop further later.

Now consider the formal composite

\[
F = C \circ E = \lambda u : \mathcal{N}. C(Eu)
\]

of the two terms. We have

\[\vdash F : \mathcal{M} \rightarrow \mathcal{N}\]

and \( F \) represents exponentiation to base 2. This term \( C \circ E \) includes a redex \( Eu \) and by removing that we obtain the term \( \bar{2} \).

The term \( \bar{2} \) represents the function

\[ m \mapsto 2^m \]

an exponentiation, but what about the function

\[ m \mapsto 2^{2^m} \]

a super-exponentiation? Since this is just exponentiation composed with itself, there shouldn’t be a problem, should there? But there is.

The term

\[ \vdash \bar{2} : \mathcal{M} \rightarrow \mathcal{N} \]

represents \( 2^* \). To represent \( 2^{2^*} \) we appear to need some atom \( \mathcal{L} \) and a term

\[ \vdash \bar{2}' : \mathcal{L} \rightarrow \mathcal{M} \]

which also represents \( 2^* \). To obtain this term we can not just mimic the construction of \( \bar{2} \). This is because we don’t have an iterator \( K : \mathcal{L} \rightarrow \mathcal{M}^{(iv)} \).

The colour coding is beginning to bite. In fact, there is no representation of \( 2^{2^*} \) in \( \lambda M \). In particular \( \mathfrak{R}(\lambda M) \) is not closed under composition. Of course, it is easy to simulate \( 2^{2^*} \) in \( \lambda M \), but the system does not have enough facilities to convert that into a representation.

This shows that the class \( \mathfrak{R}(\lambda T) \) for a tiered system may not be closed under composition. Just before Example 7.4 I said this was a ‘problem’. This is not quite the way to put it, since this is precisely what tiering is about. A way of organizing a finer control on the construction of functions.

What about other methods of capture?

How might we simulate the function

\[ f = h \circ (g_l, \ldots, g_1) \]

in a system \( \lambda T \)? Of course, we must first simulate \( g_1, \ldots, g_l, h \). Thus suppose the terms

\[ \vdash \bar{h} : \zeta(l)'' \rightarrow \cdots \rightarrow \zeta(1)'' \rightarrow \zeta'' \]

\[ \vdash \bar{g}_l : \eta(k)'' \rightarrow \cdots \rightarrow \eta(1)'' \rightarrow \zeta(l)'' \]

\[ \vdots \]

\[ \vdash \bar{g}_i : \eta(k)'' \rightarrow \cdots \rightarrow \eta(1)'' \rightarrow \zeta(i)'' \]

\[ \vdots \]

\[ \vdash \bar{g}_1 : \eta(k)'' \rightarrow \cdots \rightarrow \eta(1)'' \rightarrow \zeta(1)'' \]
simulate these functions in $\lambda T$. Here $\zeta(1), \ldots, \zeta(l)$ and $\eta(1), \ldots, \eta(k)$ are two list of types. with compatibility between the output types of the $g_i$ and the input types of $h$. Consider the term

$$\overline{f} = \lambda u_1 : \eta(l)'' \ldots, u_1 : \eta(1)'' . \overline{hs}_1 \cdots s_1$$

where

$$s_i = \overline{g}_i u_i \cdots u_1$$

for each $1 \leq i \leq l$. A simple calculation shows that

$$\vdash \overline{f} : \eta(k)'' \rightarrow \cdots \rightarrow \eta(1)'' \rightarrow \zeta''$$

and, furthermore, this term $\overline{f}$ simulates $f$ in $\lambda T$.

Does this uncover another ‘problem’? Let’s wait and see.

Let’s now gather together what we have learned in this section.

Here is a useful bit of terminology which ought to be better known. Believe it or not, I first came across it in a rather dull book on universal algebra.

8.5 DEFINITION. A clone is a class of first order functions over $\mathbb{N}$ with varying number of inputs which contains all projections and is closed under composition. ■

It is not too hard to see that the class of all projections is a clone, and hence is the smallest possible clone. More generally, given any family of functions there is a smallest clone which contains these. This is the standard way of generating clones.

8.6 THEOREM. For each tiered system $\lambda T$ the class $R(\lambda T)$ need not be a clone, but the two classes

$$\mathcal{U}(\lambda T), \mathcal{S}(\lambda T)$$

are clones.

Proof. Example 8.4 shows that $\mathcal{R}(\lambda T)$ need not be a clone.

By Lemma 8.2, both the classes $\mathcal{U}(\lambda T), \mathcal{S}(\lambda T)$ contain all projections. Thus it suffices to show that each is closed under composition. We have done most of the work for this, but there is one aspect we must look at.

Let us show first that $\mathcal{U}(\lambda T)$ is closed under composition. Thus consider a function

$$f = h \circ (g_1, \ldots, g_1)$$

where

$$g_1, \ldots, g_1, h \in \mathcal{U}(\lambda)$$

and, of course, these are compatible for composition. Consider any type $\zeta$. Then, by Definition 6.8(u) of $\mathcal{U}(\lambda T)$ we have terms

$$\vdash \overline{h} : \zeta'' \rightarrow \cdots \rightarrow \zeta'' \rightarrow \zeta''$$

$$\vdash \overline{g}_i : \zeta'' \rightarrow \cdots \rightarrow \zeta'' \rightarrow \zeta''$$

$$\vdots$$

$$\vdash \overline{g}_i : \zeta'' \rightarrow \cdots \rightarrow \zeta'' \rightarrow \zeta''$$

$$\vdots$$

$$\vdash \overline{g}_i : \zeta'' \rightarrow \cdots \rightarrow \zeta'' \rightarrow \zeta''$$
which simulate these functions. The construction above gives a term
\[ \vdash \overline{f} : \zeta'' \rightarrow \cdots \rightarrow \zeta'' \rightarrow \zeta'' \]
which simulates \( f \).

Here the uniformity of the simulation helps to make the proof quite easy. We don’t have that uniformity with \( \mathcal{S}(\lambda T) \) so we can expect to do a bit more work.

To show that \( \mathcal{S}(\lambda T) \) is closed under composition consider a function
\[ f = h \circ (g_1, \ldots, g_l) \]
where
\[ g_1, \ldots, g_l, h \in \mathcal{S}(\lambda) \]
and, of course, these are compatible for composition. Consider any type \( \zeta \). Then, by Definition 6.8(s) of \( \mathcal{S}(\lambda T) \) we have a term
\[ \vdash \overline{h} : \zeta(l)'' \rightarrow \cdots \rightarrow \zeta(1)'' \rightarrow \zeta'' \]
which simulates \( h \). Here we don’t have much control over the input types \( \zeta(1), \ldots, \zeta(l) \). However, we now have terms
\[ \vdash \overline{g_i} : \eta_i(k)'' \rightarrow \cdots \rightarrow \eta_i(1)'' \rightarrow \zeta(i)'' \]
\[ \vdash \overline{g_1} : \eta_1(k)'' \rightarrow \cdots \rightarrow \eta_1(1)'' \rightarrow \zeta(1)'' \]
where we now have even less control over the array of inputs types. In particular, it seems that we can not arrange for the equalities required to carry out the construction above. What do we do now?

At this point we will postpone the remainder of the proof. The result is still true but we need a new idea. That is best described in a more general context, and that is the topic of the next section.

[Hold in 165../05-version../009.. Last changed April 13, 2006]

9 Extended polynomials and humble functions

From Corollary 6.4 and Theorem 7.5 we have
\[ \mathcal{N}(\lambda A) = \mathcal{R}(\lambda A) = \mathcal{T} \]
the class of translations. In this and the next section we obtain a precise description of the class \( \mathcal{U}(\lambda A) \), the class of functions uniformly simulated throughout \( \lambda A \). Later we will determine \( \mathcal{S}(\lambda A) \) together with the strengths of certain other tiered systems.

Let’s begin by looking at some examples of clones of functions, each of which is a plausible candidate as the strength of some system.
9.1 EXAMPLE. Let

\[
Z \in \mathbb{N}(1) \quad S \in \mathbb{N}(1) \quad A \in \mathbb{N}(2) \quad M \in \mathbb{N}(2) \quad N \in \mathbb{N}(1)
\]

be given by

\[
Zx = 0 \quad Sx = 1 + x \quad Ayx = x + y \quad Myx = x \times y
\]

and

\[
Nx = \begin{cases} 
0 & \text{if } x \neq 0 \\
1 & \text{if } x = 0 
\end{cases}
\]

for each \(x, y \in \mathbb{N}\). These are the zero function, successor, addition, multiplication, and the numeric negation. Using various combinations of these functions we can generate several clones, as follows.

- \(Z, S\) gives \(\mathcal{T}\) the translations
- \(Z, S, A\) gives \(\mathcal{L}\) the linear functions
- \(Z, S, A, M\) gives \(\mathcal{P}\) the polynomials
- \(Z, S, A, M, N\) gives \(\mathcal{E}\) the extended polynomials

In other words, each class is the smallest clone containing the indicated functions. This class is obtained from the functions and the projections by repeatedly forming compositions using earlier constructed functions. Remember that the projections can be used as bookkeeping gadgets, to shuffle around the inputs, introduce dummy inputs, and the like.

The translations \(\mathcal{T}\) are just as before. Each \(l\)-placed linear function \(f \in \mathcal{L}\) has the form

\[
f(x_1, \ldots, x_l) = a_0 + a_1 x_1 + \cdots + a_l x_l
\]

where the coefficients \(a_0, a_1, \ldots, a_l\) are fixed natural numbers. The polynomials are not quite the functions you might expect. They are formed using only positive coefficients and with no uses of subtraction, and so return only positive values.

The class \(\mathcal{T}\) is used in Theorem 7.5. The two classes \(\mathcal{L}, \mathcal{P}\) will play only walk-on roles here (but they are clearly important elsewhere). The fourth class is more important here and deserves some spotlight.

9.2 DEFINITION. The class \(\mathcal{E}\) of extended polynomial is the least clone containing the functions \(Z, S, A, M, N\), as described above.

Notice that

\[
\mathcal{T} \subseteq \mathcal{L} \subseteq \mathcal{P} \subseteq \mathcal{E}
\]

and each is a plausible candidate for the class of functions obtained in some way from a simple tiered system.

Because of the presence of numeric negation in its generating family the extended polynomials are, perhaps, not so familiar. Let’s have a look at some simple examples.

9.3 EXAMPLES. From numeric negation \(N\) and its composite \(N^2\) we have

\[
Nx = \begin{cases} 
0 & \text{if } x \neq 0 \\
1 & \text{if } x = 0 
\end{cases} \quad N^2x = \begin{cases} 
1 & \text{if } x \neq 0 \\
0 & \text{if } x = 0 
\end{cases}
\]
and
\[ Nx + N^2 x = 1 \]
for each \( x \in \mathbb{N} \).

The 3-placed function \( P \) given by
\[ P(z, y, x) = z \times (N^2 x) = y \times (Nx) \]
is an extended polynomial. We find that
\[ P(z, y, x) = \begin{cases} y & \text{if } x \neq 0 \\ z & \text{if } x = 0 \end{cases} \]
for \( x, y, z \in \mathbb{N} \). This is a kind of numerical implication and can be used in various coding tricks. ■

We will develop a better understanding of these functions in this section.

9.4 LEMMA. We have \( \mathfrak{C} \subseteq \mathfrak{U}(\lambda A) \).

Proof. By Theorem 8.6 we know that \( \mathfrak{U}(\lambda A) \) is a clone, so it suffices to show that \( \mathfrak{U}(\lambda A) \) contains the functions \( Z, S, A, M, N \).

Let \( \zeta \) be an arbitrary type. By Example 6.6 we know that \( S, A, M \) are simulated in \( \lambda A \) with types
\[ \zeta'' \rightarrow \zeta'' \quad \zeta'' \rightarrow \zeta'' \quad \zeta'' \rightarrow \zeta'' \quad \zeta'' \rightarrow \zeta'' \]
respectively. Also the terms
\[ \begin{align*}
\overline{\zeta z} &= \lambda u : \zeta'', y : \zeta', x : \zeta \cdot x \\
\overline{\zeta n} &= \lambda u : \zeta'', y : \zeta', x : \zeta \cdot u(\lambda v : \zeta \cdot x)(yx)
\end{align*} \]
simulate \( Z, N \) with indicated types, respectively. Only the claim for \( N \) is not immediate, so let’s deal with that.

To show that \( \overline{\zeta n} \) simulates \( N \) we may drop the subscripts.

Since \( \overline{s t} \Rightarrow s \) for all terms \( s, t \), we have
\[ \begin{align*}
\overline{\zeta 0} \Rightarrow \lambda y : \zeta', x : \zeta \cdot \overline{(-)}(yx) & \Rightarrow \lambda y : \zeta', x : \zeta \cdot yx = \overline{1} \\
\overline{\zeta m+1} \Rightarrow \lambda y : \zeta', x : \zeta \cdot \overline{-}(yx) & \Rightarrow \lambda y : \zeta', x : \zeta \cdot (-)^{m+1}(yx)
\end{align*} \]
as required. Here \( (-) \) is \( (\lambda v : \zeta \cdot x) \).

For each \( m \in \mathbb{N} \) we have
\[ \begin{align*}
\overline{\zeta m+1} & \Rightarrow \lambda y : \zeta', x : \zeta \cdot \overline{-}(yx) \\
& \Rightarrow \lambda y : \zeta', x : \zeta \cdot \overline{-}^{m}(yx) \\
& = \lambda y : \zeta', x : \zeta \cdot (\lambda v : \zeta \cdot x)^{r} \\
& \Rightarrow \lambda y : \zeta', x : \zeta \cdot x = \overline{0}
\end{align*} \]
as required. Here \( r \) is some compound term whose details are not important. ■

Eventually we will improve this inclusion to an equality. Along the way we obtain quite a bit more information about the behaviour of \( \lambda A \). Extracting that information is the main purpose of this and the next section.
Capturing a function by representation or by simulation have a certain commonality. For both we use the reduction facilities, but for the one we deal with authentic numerals whereas for the other we use simulated numerals. What can’t we use a common generalization where we handle a mixture of authentic and simulated numerals?

In this and the next section we look at one such modest extension. Later, in Section 11 we look at a more ambitious extension.

The atom $N$ supports both authentic $\langle m \rangle : N$ and simulated $\langle m \rangle : N'$ numerals. Both of these can be used to produce functions within $\lambda A$. Using all authentic numerals we have a representation, and using all simulated numerals we have a simulation. There is also the possibility of mixing these kinds of numerals.

Let $\hat{N}$ be either $N$ or $N'$, and let $\hat{m}$ be the corresponding numeral. Thus

\[ \hat{m} = \begin{cases} \langle m \rangle & \text{if } \hat{N} \text{ is } N' \\ \langle m \rangle & \text{if } \hat{N} \text{ is } N \end{cases} \]

where here we do not need to put in a subscript. With this we can look at types

$\hat{N} \rightarrow \cdots \rightarrow \hat{N} \rightarrow \hat{N}$

where each $\hat{N}$ is chosen independently of the others. Thus, if there are $(l + 1)$ occurrences of $N$, then there are $2^{l+1}$ such types.

9.5 DEFINITION. A first order function $f : N(l)$ is imitated in $\lambda A$ (in the narrow sense) if there is a derivation

\[ \vdash \hat{f} : \hat{N} \rightarrow \cdots \rightarrow \hat{N} \rightarrow \hat{N} \]

such that

\[ \hat{f} m(l) \cdots m(1) \triangleright m(0) \]

for each $m(0), m(1), \ldots, m(l) \in N$ with $f(m(l) \cdots m(1)) = m(0)$. ■

Later, in Section 11 we consider a more general notion of imitation. The phrase ‘in the narrow sense’ is an indication that there is a generalization to come. However, for the time being we can omit the phrase, since it is needed only when we get nearer to the extended notion.

Certain $l$-placed function $f$ can be imitated in $\lambda A$ in $2^{l+1}$ different ways. Each occurrence of $\hat{N}$ in the imitating type may be $N$ or $N'$. If all of these are $N$, then we have a representation. If all of these are $N'$, then we have a simulation.

Here we are particularly interested in those imitations where the target type is $N$, not $N'$. You may ask why we don’t want $N'$ as the target type. There is nothing mysterious here, and we will consider the more general kind of imitation later. It’s just that we don’t want to analyse it in any detail in this section.

However, before we look at these $N$-target imitations in more detail, let’s see what we can do with the others.

Consider an imitation

\[ \vdash \tilde{f} : \tilde{N} \rightarrow \cdots \rightarrow \tilde{N} \rightarrow N'' \]

of a function $f$ where the target type is $N''$. We can convert this into an imitation of the other kind without changing the imitated function. To do that consider the term

\[ C = \lambda w : N''. \ wS0 \]
used in Example 8.4. Here $S, 0$ are the furnishing of the atom $N$, and we have

$$\vdash C : N' \rightarrow N \quad \text{Cm} \not\vdash m$$

for each $m \in \mathbb{N}$. Using this we set

$$\hat{f} = \lambda u : \tilde{N}, \ldots, u_1 : \tilde{N} \cdot C(\tilde{f} u_1 \cdots u_1)$$

to obtain a term

$$\vdash \hat{f} : \tilde{N} \rightarrow \cdots \rightarrow \tilde{N} \rightarrow N$$

with the same input types but with $N$ as output type. Since

$$\hat{f}m(l) \cdots m(1) \not\vdash C(\tilde{f} m(l) \cdots m(1)) \not\vdash C(m(0)) \not\vdash m(0)$$

we see that $\hat{f}$ imitates the same function $f$.

We now turn to the main topic of this section.

Consider an $N$-target imitation $\hat{f}$ of an $l$-placed function $f$, as above. This imitation divides the inputs into two kinds, those where we use authentic numerals and those where we use simulated numerals. The output is always an authentic numeral. This imposes on $f$ some extra structure that is not visible in the set theoretic world.

Suppose $f$ is such an $(s + a)$-placed function, that is we have nominated $s$ inputs to be simulated and $a$ inputs to be authentic (and, of course, $s + a = l$). To prevent the notation getting out of hand we suppose all the simulated inputs have been collected at the left hand end and all the authentic inputs have been collected at the right hand end. Thus a typical value of $f$ has the form

$$f(y_s, \ldots, y_1; x_a, \ldots, x_1)$$

where we use the semicolon to separate the two kinds of inputs. The imitation has the form

$$(H(s; a)) \vdash \hat{f} : N'' \rightarrow \cdots \rightarrow N'' \rightarrow N \rightarrow \cdots \rightarrow N \rightarrow N$$

where there are $s$ occurrences of $N''$ and $a$ input types of $N$ as input types.

9.6 DEFINITION. Let $s, a, l \in \mathbb{N}$ with $s + a = l$. A first order function $f : \mathbb{N}(l)$ is humble of arity $(s; a)$ if it is imitated via a derivation $H(s; a)$, as above, where there are $s$ inputs types $N''$ and $a$ input types of $N$.

Let $\mathcal{H}(s; a)$ be the class of functions which are humble of arity $(s; a)$.

Let $\mathcal{H}$ be the class of functions which are humble of some arity. ■

In other words $\mathcal{H}$ is the disjoint union of the classes $\mathcal{H}(s; a)$ as $(s; a)$ ranges over all possible arities.

The manipulations just before the definition more or less prove the following.

9.7 THEOREM. Suppose $f \in U(\lambda A)$ is $l$-placed. Then $f$ is $(l, 0)$-humble.

Proof. Consider any $l$-placed function $f \in U(\lambda A)$. For each type $\zeta$ we have a simulation of $f$ of type

$$\zeta'' \rightarrow \cdots \zeta'' \rightarrow \zeta''$$
and, in particular, a term

\[ \vdash \tilde{f} : \mathcal{N}'' \to \cdots \to \mathcal{N}'' \to \mathcal{N}'' \]

which simulates \( f \). Using the manipulation above we convert this into a term

\[ \vdash \hat{f} : \mathcal{N}'' \to \cdots \to \mathcal{N}'' \to \mathcal{N} \]

which imitates \( f \).

This gives us an upper and a lower bound for the class \( \mathcal{U}(\lambda \mathcal{A}) \).

9.8 COROLLARY. We have \( \mathcal{E} \subseteq \mathcal{U}(\lambda \mathcal{A}) \subseteq \mathcal{H} \).

The imitating term of a humble function is allowed to used mixed kinds of inputs. Since we want a more detailed analysis of these functions, it might be a good idea to convert the imitating term into some kind of standard form.

Suppose we have an imitating term \( \hat{f} \) and suppose one of the input types is \( \mathcal{N} \) (not \( \mathcal{N}'' \)). Thus we have

\[ \vdash \hat{f} : \mathcal{N} \to \cdots \to \mathcal{N} \to \cdots \to \mathcal{N} \to \mathcal{N} \]

where we have isolated the input type of interest. Using the term \( C : \mathcal{N}'' \to \mathcal{N} \) we may set

\[ \tilde{f} = \lambda w_1 : \mathcal{N}, \ldots, u : \mathcal{N}'' \cdots w_1 : \mathcal{N} . \hat{f} w_1 \cdots (Cu) \cdots w_1 \]

to get

\[ \vdash \tilde{f} : \mathcal{N} \to \cdots \to \mathcal{N}'' \to \cdots \to \mathcal{N} \to \mathcal{N} \]

and produce a second term which imitates \( f \). This process changes the status of the nominated input numerals from authentic to simulated. By several applications see that each humble function can be imitated by a term

\[ \vdash \hat{f} : \mathcal{N}'' \to \cdots \to \mathcal{N}'' \to \mathcal{N} \]

where all the input numerals are simulated. Perhaps it is easier to analyse this kind of imitating term rather than one with mixed input types.

Perhaps it is, but it is not a good idea to do this. The passage \( \hat{f} \mapsto \tilde{f} \) looses some information (the status of the input types) and this is a crucial aspect of humble functions. We want to preserve this information. There is a convenient standard form for imitating terms, but it is not this one.

Consider an \( l \)-placed humble function with an imitating term

\[ \vdash \hat{f} : \mathcal{N} \to \cdots \to \mathcal{N} \to \mathcal{N} \]

using mixed input. A certain number of these, \( s \), will be simulated and a certain number, \( a \), will be authentic, with \( l = s + a \). It is more convenient to separate these two kinds than to change the status of the authentic ones. We want a term of the form

\[ \vdash \hat{f} : \mathcal{N}'' \to \cdots \to \mathcal{N}'' \to \mathcal{N} \to \cdots \to \mathcal{N} \to \mathcal{N} \]
where there are \( s \) input types \( \mathcal{N}'' \) and \( a \) input types \( \mathcal{N} \), and where this imitates essentially the same function with the inputs permuted to match the re-ordering of the input types. We can achieve this by setting

\[
\hat{f} = \lambda u_s, \ldots, u_1 : \mathcal{N}'', \lambda v_a, \ldots, v_1 : \mathcal{N}. \hat{f} w_l \cdots w_1
\]

where \( w_1, \ldots, w_l \) is the appropriate permutation of \( u_1, \ldots, u_s, v_1, \ldots, v_a \).

In short, and imitation within \( \lambda \mathcal{A} \) (in the sense of Definition 9.4) gives a function with arity, provided we preserve the distinction between simulated and authentic numerals. We can always permute the inputs and so obtain a standard ‘simulates-authentic’ form. If necessary, we can convert an authentic input into a simulated input, but this should not be done as a matter of routine.

[Last changed April 13, 2006]

10 Meek functions

By Corollary 9.8 we have

\[ \mathfrak{C} \subseteq \mathfrak{U}(\lambda \mathcal{A}) \subseteq \mathfrak{H} \]

where \( \mathfrak{C} \) is defined explicitly (as the least clone which contains certain generating functions), but where the other two are classes of functions that are captured in a certain way. In this section we show that these three classes are the same. In fact, we do much more.

By definition, the class \( \mathfrak{H} \) is the disjoint union of classes \( \mathfrak{H}(s; a) \) where the pair \( (s; a) \) indicates the arity of the members. In this section we obtain an explicit description of this class \( \mathfrak{H}(s; a) \). This isolates the roles played by the simulated and authentic numerals in the imitation of a humble function.

For the time being let’s not worry about the arity of a function. We will get to that in a short while.

10.1 DEFINITION. A decision function is an extended polynomial \( D \) which takes only the values 0 and 1.

A list \( D, D_1, \ldots, D_a \) of decision functions forms a partition if each has the same list of inputs and

\[ Dy + D_1y + \cdots D_ay = 1 \]

for each such list \( y \) of inputs. In other words, if for each \( y \) exactly one of \( D_y, D_1y, \ldots, D_ay \)

is 1 and the others are 0.

The numeric negation \( N \) is a decision function, as is \( N^2 \), and the pair form a partition. In general a decision function \( D \) can have several inputs, and then \( D \) tests for a certain relationship between its inputs. For instance, if \( E_y \) is (a typical value of) any extended polynomial then \( N(E_y) \) is (a typical value of) a decision function. This function \( N \circ E \) returns 1 precisely when the input gives a zero of the extended polynomial.

Suppose \( D, D_1, \ldots, D_a \) form a partition. Then, since each takes only values 0 and 1, we have

\[ Dy = N(\sum_i D_iy) \]
where the summation is taken over $1 \leq i \leq a$. Notice also that having the same list of inputs is not a serious restriction. This can always be achieved by using projections.

10.2 **EXAMPLE.** Let $D, D_1, \ldots, D_a$ form a partition and let $A, A_1, \ldots, A_a$ be a matching list of extended polynomials, all with the same list $y$ of inputs. The function $f$ given by

$$f_y = \begin{cases} A y & \text{if } D_i y = 1 \text{ for } 1 \leq i \leq a \\ A y & \text{if } D y = 1 \end{cases}$$

is an extended polynomial, since we have

$$f_y = ((A y) \times (D y)) + \sum_i (A_i y) \times (D_i y)$$

for each $y$. ■

You may think the notation here is a little eccentric. At this stage it is, but you will see why it is done this way shortly.

We now start to discuss functions with arity.

Given a first order function

$$f \in \mathbb{N}(l)$$

we divide its inputs into two two kinds, the ‘simulated’ and the ‘authentic’. Thus each value of such a function $f$ can be written

$$f(y_s, \ldots, y_1; x_a, \ldots, x_1)$$

where $x_1, \ldots, x_a$ are the authentic inputs and the $y_1, \ldots, y_s$ are the simulated inputs. We say $f$ has arity $(s; a)$ and sometimes write

$$f(y; x)$$

when the details of the lists $y$ and $x$ of inputs are not needed, in which case we also say that $f$ has arity $(y; x)$. (Strictly speaking we should allow the simulated and authentic inputs to occur in random order, but it makes the notation easier to handle if we tacitly assume the inputs have been permuted into the ‘simulated+authentic’ order.)

The following should be compared with Example 10.2.

10.3 **DEFINITION.** A function $f$ of arity $(s; a)$ is **meek** if it has the form

$$f(y; x) = \begin{cases} x_i + A_i y & \text{if } D_i y = 1 \\ A y & \text{if } D y = 1 \end{cases}$$

where $A, A_1, \ldots, A_a$ are extended polynomials and $D, D_1, \ldots, D_a$ form a partition.

Let $\mathcal{M}(s; a)$ be the class of meek functions of arity $(s; a)$.

Let $\mathcal{M}$ be the class of meek functions. ■

The main purpose of this section is to show

$$\mathcal{M}(s; a) = \mathcal{H}(s; a)$$
and so we obtain an explicit description of the imitating facilities of \(\text{\textsc{\textlambda}A}\). We prove the equality via two separate inclusions. One inclusion follows by a use of the exhaustive search technique, and the other follows by combining several tricks we have seen already.

Before we start those proofs let’s get a bit more familiar with the nature of meek functions.

Consider first a meek function \(f\) of arity \((s; 0)\) for some \(s\). This must use a partition which consists of a single extended polynomial \(D\). In other words, this function \(D\) is identically 1, and the meek function \(f\) is itself an extended polynomial. Thus each extended polynomial is meek provided we view all the inputs as simulated.

10.4 LEMMA. We have \(\mathcal{E} \subseteq \mathcal{M}\).

A meek function \(f\) of arity \((s; a)\) may use a partition for which one component is identically 1 (and all the others are identically 0). In this case we have one of

\[
  f(y, x) = x + Ay \quad f(y, x) = Ay
\]

for some extended polynomial \(A\) and where, in the left hand case, \(x\) is a selected authentic input. By taking \(A\) to be identically 0 or a projection function we see that we have the projections with arity. In other words for any standard projection function we can split the inputs into simulated and authentic as we please. Notice that if we try to do such a split with the inputs of a standard extended polynomial, then we may not get a meek function.

In practice some of the inputs of a meek function may not occur in some of the components \(A, A_1, \ldots, A_a, D, D_1, \ldots, D_a\). The missing ones can always be inserted as dummy inputs using projections.

10.5 LEMMA. We have \(\mathcal{M} = \mathcal{E}\), provided we ignore the arity of meek functions.

**Proof.** By Lemma 10.4 we have \(\mathcal{E} \subseteq \mathcal{M}\), so it suffices to show the converse inclusion.

Consider a meek function \(f\), as in Definition 10.3. Each component \(A, A_1, \ldots, A_a, D, D_1, \ldots, D_a\) is an extended polynomial. It is easy to check that

\[
  f(y; x) = ((Ay) \times (Dy)) + \sum_i (x_i + A_i y) \times (D_i y)
\]

so \(f\) itself is an extended polynomial.

This shows that each meek function can be viewed as an extended polynomial *provided we are prepared to lose the status of the inputs*. But we don’t want to loose that arity, so what do we do? That will take a bit of time to explain.

We need some closure properties of \(\mathcal{M}\). In particular, we need to know that \(\mathcal{M}\) is closed under composition. However, this is not just the usual composition described in Definition 8.3 but a restricted kind composition which respects arity. You might think you can guess what this is, but there is a slight twist.

Suppose we have a meek function

\[
  h(v_s, \ldots, v_1; u_a, \ldots, u_1)
\]
with indicated arity \((s; a)\). Suppose also we have matching lists of meek functions

\[ k_s y, \ldots, k_1 y \quad g_a(y; x), \ldots, g_1(y; x) \]

with the indicated arities. In particular, each function \(k\) has no authentic inputs. There is some arity \((t; b)\) such that each \(k\) has arity \((t; 0)\) and each \(g\) has arity \((t; b)\). We can now compose the function in the obvious way.

10.6 LEMMA. Given the data, as above, the composite function \(f\) where

\[ f(y; x) = h(k_s y, \ldots, k_1 y ; g_a(y; x), \ldots, g_1(y; x)) \]

is meek with the indicated arity.

Proof. This is simple exercise in boolean jiggery-pokery, and it is not too hard to work out. However, it is worth looking at the details for the tricks used can look a bit mysterious at first sight. In fact, they are quite simple but obscured by the notation. Furthermore, the details do tell us something about the nature of functions with arity in general and meek functions in particular.

We are given that \(h\) is meek so that

\[ h(v; u) = \begin{cases} u_i + B_i v & \text{if } E_i v = 1 \\ B v & \text{if } Ev = 1 \end{cases} \]

where \(B, B_i, E, E_i\) are extended polynomials and \(E, E_i\) form a partition. Here \(i\) runs through the length of \(u\). We are also give a family a extended polynomials \(k y\), one for each input \(v\). We combine these into a list \(k y\) (for we need never look inside this). Finally, we are given a family of meek functions

\[ g_i(y; x) = \begin{cases} x_j + A_{ij} y & \text{if } D_{ij} y = 1 \\ A_i y & \text{if } D_i y = 1 \end{cases} \]

where \(i\) runs through the length of \(u\) and \(j\) runs through the length of \(x\). For each such \(i\) the \(A_i, A_{ij}, D_i, D_{ij}\) are extended polynomials and \(D_i, D_{ij}\) form a partition.

A direct substitution gives

\[ f(y; x) = \begin{cases} g_i(y; x) + B_i (k y) & \text{if } E_i (k y) = 1 \\ B(k y) & \text{if } E(k y) = 1 \end{cases} \]

which is not in the form we require. By decomposing each \(g_i\) we get

\[ f(y; x) = \begin{cases} x_j + A_{ij} y + B_i (k y) & \text{if } E_i (k y) = 1 \text{ and } D_{ij} y = 1 \\ A_i y + B_i (k y) & \text{if } E_i (k y) = 1 \text{ and } D_i y = 1 \\ B(k y) & \text{if } E(k y) = 1 \end{cases} \]

which is nearer but still not in the required form. (What we are seeing here is that we could use a more liberal decomposition for the form of a meek function.)

Let

\[ G_i y = (E_i (k y)) \times (D_i y) \quad G_{ij} y = (E_i (k y)) \times (D_{ij} y) \]
for each \( i, j \). Note that each of these is a decision function. Using these set

\[
Fy = E(ky) + \sum \limits_i G_iy \quad F_jy = \sum \limits_i G_{ij}y
\]

for each \( j \) to produce a family of extended polynomials. We check that these are decision functions and form a partition.

For each \( y \) exactly one of

\[
E(ky) \quad E_i(ky)
\]

is 1 and all the others are 0. Let’s look at the possibilities. Suppose \( E(ky) = 1 \). Then \( E_iy = 0 \) for each \( i \), so that

\[
G_iy = G_{ij}y = 0
\]

for all \( i, j \) to give

\[
Fy = 1 \quad F_jy = 0
\]

for all \( j \), as it should be in this case. Suppose \( E(ky) = 0 \). There is some index \( \star \) with \( E_{\star}(ky) = 1 \) and \( E_i(ky) = 0 \) for all other \( i \). This gives

\[
G_{\star}y = D_{\star}y \quad G_{\star}jy = D_{\star}jy \quad G_iy = 0 \quad G_{ij}y = 0
\]

for all other \( i \) and all \( j \). Thus

\[
Fy = G_{\star}y = D_{\star}y \quad F_jy = G_{\star}jy = D_{\star}jy
\]

for all \( i \). But the \( D_{\star}, D_{\star}j \) form a partition so precisely one of the values \( Fy, F_jy \) is 1 and all other are 0.

Using these decisions functions we have

\[
f(y; x) = \begin{cases} 
  x_j + A_{ij}y + B_i(ky) & \text{if } G_{ij}y = 1 \\
  A_iy + B_i(ky) & \text{if } G_iy = 1 \\
  B(ky) & \text{if } E(ky) = 1
\end{cases}
\]

which is nearer but still not in the required form.

Let

\[
Cy = (B(ky) \times E(ky)) + \sum \limits_i ((A_iy + B_i(ky)) \times G_iy)
\]

to obtain an extended polynomial \( C \). For each \( y \) at most one of

\[
E(ky) \quad G_iy
\]

is 1 and all the others are 0. Thus one of

\[
Cy = B(ky) \quad Cy = A_iy + B_i(ky) \quad Cy = 0
\]

holds depending on how the previous decision values work out. Remembering the construction of the function \( F \) this shows that

\[
f(y; x) = \begin{cases} 
  x_j + A_{ij}y + B_i(ky) & \text{if } G_{ij}y = 1 \\
  Cy & \text{if } Fy = 1
\end{cases}
\]
which is closer but still not there.

For each index \( j \) we need to combine all the upper alternatives in which \( x_j \) occurs. For each \( j \) let

\[
C_j y = \sum_i ((A_{ij} y + B_{ij}(ky)) \times G_i y)
\]

to obtain a family of extended polynomials. Then remembering the construction of the functions involved we see that

\[
f(y; x) = \begin{cases} 
  x_j + C_j y & \text{if } F_j x = j \\
  C y & \text{if } F y = 1
\end{cases}
\]

is what we want.

There isn’t much going on in this proof. The notation is a bit finicky because we are trying to simulate boolean decisions using numeric functions (which is always a bit messy). Notice also that it is the simulated inputs that are controlling these decisions. In fact, it is not a bad idea to think of simulated inputs as control inputs. This is not a trite remark, it is the idea that lies at the heart of much of the tiering technique. The crucial aspect of a meek function \( f \) is that the authentic inputs play only a minor role in organizing its output, and it is that we want to analyse.

You should take another look at the statement of Lemma 10.6. The composition allowed is not quite as liberal as you might think. For example, suppose we have a meek function

\[
h(v; u)
\]

of arity \((1, 1)\) and suppose

\[
k(y; x) \quad g(y; x)
\]

are a pair of meek functions of some arity \((s, a)\). Then we can certainly form a function

\[
f(y; x) = h(k(y; x); g(y; x))
\]

by the ‘obvious’ kind of composition. However, Lemma 10.6 does not ensure that this \( f \) is meek. To preserve meekness only functions of arity \((-0)\) can be substituted into simulated inputs.

Example 9.1 gives some of the standard clones. There are many other clones that crop up, and most of these are closed under some kind of recursion. The best known example is the clone of primitive recursive functions (which is closed under more than just primitive recursion). The class \( \mathfrak{M} \) is also closed under some forms of recursion but, of course, forms in which the arities are taken into account. In fact, this is where the two kinds of inputs come into their own.

Suppose we have a meek function

\[
h(y; x, u)
\]

with the indicated arity. We select one of the authentic inputs \( u \) and conveniently place it at the right hand end (although, of course, it could be in any authentic position). By fixing the other inputs we obtain a 1-placed meek function

\[
h(y; x, \cdot)
\]
of arity \((0, 1)\). Any 1-placed function can be iterated and, by using the number of iterations as a second input we obtain a 2-placed function. In this way we obtain a new function \(f\) with

\[ f(v, y; x, u) = h(y; x, \cdot)^v u \]

where, by choice, the exponent \(v\) is treated as a new simulated input, and the base value \(u\) is treated as an authentic input (as is required by \(h\)). For instance,

\[ f(0; y; x, u) = u \]
\[ f(1; y; x, u) = h(y; x, u) \]
\[ f(2; y; x, u) = h(y; x, h(y; x, u)) \]

and so on.

10.7 LEMMA. Given the data, as above, the constructed function \(f\) is meek with the indicated arity.

Proof. The given meek function \(h\) has arity \((y; x, u)\), so that

\[ h(y; x, u) = \begin{cases} 
  u + Cy & \text{if } Fy = 1 \\
  x_i + B_i y & \text{if } E_i y = 1 \\
  Ay & \text{if } Dy = 1
\end{cases} \]

where \(A, B_i, C\) are extended polynomials and \(D, E_i, F\) form a partition. Let us assume that \(h\) has arity \((s, a + 1)\), so that \(i\) ranges through \(\{1, \ldots, a\}\).

Consider the extended polynomials

\[ K(v, y) = v \times (Cy) \times (N^2 v) \times (Fy) \]
\[ J(v, y) = Nv + (N^2 v) \times (Fy) \]
\[ I_i(v, y) = (N^2 v) \times (E_i y) \]
\[ H(v, y) = (N^2 v) \times (Dy) \]

where we have introduced a new simulated input \(v\). Thus each of these has arity \((1 + s, 0)\). Notice the use of numeric negation in these constructions.

It is easy to check and the \(H, I_i, J\) form a partition, but let’s do that just to be sure. Since \(D, E_i, F\) form a partition we have cases

<table>
<thead>
<tr>
<th>(Dy = 1)</th>
<th>(E_i y = 1)</th>
<th>(Fy = 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(J(v, y) = Nv)</td>
<td>(J(v, y) = Nv)</td>
<td>(J(v, y) = Nv + N^2 v = 1)</td>
</tr>
<tr>
<td>(I_i(v, y) = 0)</td>
<td>(I_i(v, y) = 0)</td>
<td>(I_i(v, y) = 0)</td>
</tr>
<tr>
<td>(I_\bullet(v, y) = 0)</td>
<td>(I_\bullet(v, y) = N^2 v)</td>
<td>(I_\bullet(v, y) = 0)</td>
</tr>
<tr>
<td>(H(v, y) = N^2 v)</td>
<td>(H(v, y) = 0)</td>
<td>(H(v, y) = 0)</td>
</tr>
</tbody>
</table>

where the top line indicates which part of the partition is selected by \(y\). Here \(\bullet\) is the selected index in the central case with \(i\) as a typical ‘other index’. In the left and central cases the parity of \(v\) decides between \(J\) and either \(H\) or \(I_\bullet\).
Using these we find that
\[
\begin{align*}
    h(y; x, \cdot)^v u = \begin{cases} 
        u & \text{if } Nv = 1 \\
        u + v \times (Cy) & \text{if } Nv = 0 \text{ and } F_y = 1 \\
        x_i + B_i y & \text{if } Nv = 0 \text{ and } E_i y = 1 \\
        Ay & \text{if } Nv = 0 \text{ and } D y = 1
    \end{cases}
\end{align*}
\]
which once checked, will complete the proof.

We look first at the left hand equality first.

Suppose that \( v = 0 \). Then
\[
    h(y; x, \cdot)^0 u = u \quad Nv = 1
\]
and the top case is selected. Otherwise we have \( Nv = 1 \) with \( v \neq 0 \) and one of the bottom three cases is selected, as follows.

Suppose \( D y = 1 \). Then
\[
    h(y; x, u) = Ay
\]
so that \( h(y; x, \cdot) \) is a constant function, and hence
\[
    h(y; x, \cdot)^v u = Ay
\]
for non-zero \( v \), as required.

Suppose \( E_i y = 1 \). Then \( h(y; x, \cdot) \) is another constant function, and hence
\[
    h(y; x, \cdot)^v u = x_i + B_i y
\]
for non-zero \( v \), as required.

Suppose \( E y = 1 \). Then
\[
    h(y; x, u) = u + C y
\]
so that
\[
    h(y; x, \cdot)^v = u + v \times (Cy)
\]
to give the required result.

Next we look at the right hand equality.

We have
\[
    I_i(v, y) = 1 \iff N^2 v = 1 = E_i y \iff Nv = 0 \text{ and } E_i y = 1 \\
    H(v, y) = 1 \iff N^2 v = 1 = D y \iff Nv = 0 \text{ and } D y = 1
\]
which deals with the bottom two cases.

Finally we have \( J(v, y) = 1 \) precisely when either
\[
    Nv = 1 \quad \text{or} \quad N^2 v = 1 = F y
\]
and
\[
    K(v, x) = \begin{cases} 
        0 & \text{if } Nv = 1 \\
        v \times (Cy) & \text{if } Nv = 0 \text{ and } F y = 1
    \end{cases}
\]
which leads to the required result.

You should take careful note of the kind of iteration allowed by Lemma 10.7. There are several other forms of iteration you might want to use, but the result may not cover those cases.

We will need a mixture of these two results.

10.8 LEMMA. Suppose the functions \( h(y; x, \cdot) \) and \( g(y; x) \) are meek with indicated arities. Suppose \( y \) is one of the simulated inputs. The function \( f \) given by

\[
f(y; x) = h(y; x, \cdot)^y g(y; x)
\]

is meek with indicated arity.

Proof. By Lemma 10.7 we have a meek function

\[
l(v, y; x, u) = h(y; x, \cdot)^v u
\]

of indicated arity. We use Lemma 10.6 to substitute into \( l \). Of course, we simply want to substitute for \( u \) and \( v \), but Lemma 10.6 doesn’t allow that, we have to substitute into every input position. (This is a little bit pedantic, for what we want to do turns out to be allowable, but let’s check that.)

We may view each projection function as a meek function by splitting the inputs into simulated or authentic in any way we like. We use these to substitute into most positions.

For each input in \( y \) we use the projection function of arity \( (y; 0) \) which selects that position. For input \( v \) we use the projection function of arity \( (y; 0) \) which selects the nominated input \( y \) from \( y \). Thus one of these projection functions is used twice to force \( v = y \), and the others essentially do nothing.

For each authentic input \( x \) we substitute the projection function of arity \( (y; x) \) which selects \( x \). So again these are essentially doing nothing.

Finally, for the authentic input \( u \) we substitute the given function \( g \).

We can now make a start on the proof of the equality \( \mathcal{M}(s; a) = \mathcal{F}(s; a) \). First the ‘easy’ inclusion.

10.9 THEOREM. For each arity \( (s; a) \) we have \( \mathcal{M}(s; a) \subseteq \mathcal{F}(s; a) \). That is, for each meek function \( f \) with typical value

\[
f(y_s, \ldots, y_1; x_a, \ldots, x_1)
\]

there is a term

\[
\vdash \widehat{f} : N'' \rightarrow \cdots \rightarrow N'' \rightarrow N \rightarrow \cdots N \rightarrow N
\]

which imitates \( f \). In this type there are \( s \) occurrences of \( N'' \) and \( a + 1 \) occurrences of \( N \).

Proof. We produce the term \( \widehat{f} \) by recursion on the number \( a \) of authentic inputs. Within the construction we use several tricks we have seen already.

For the base case \( a = 0 \) the function \( f \) is an \( s \)-placed extended polynomial \( A \) with a partition given by the function that is identically 1. By Lemma 9.4 the function \( A \) can be simulated by a term

\[
\vdash \alpha : N'' \rightarrow \cdots \rightarrow N'' \rightarrow N'
\]
with $s$ input types. Thus we have
\[ \alpha \overline{n} \triangleright \overline{A_n} \]
for each list $n$ from $\mathbb{N}$ of length $s$. Using the trick that produced the term $C$ in Example 8.4 and again between Definitions 9.5 and 9.6, we set
\[ \hat{\alpha} = \lambda v : \mathcal{N}^{''}. (\alpha v)S0 \]
to obtain a term which imitates the $s$-placed extended polynomial $A$ as a humble function of arity $(s, 0)$.

Next we look at the case $a = 1$. Although this is not strictly necessary, it does help to explain what happens in the general recursion step.

Thus we have a function $f$ of arity $(s, 1)$ where
\[ f(y; x) = \begin{cases} x + By & \text{if } Dy = 0 \\ Ay & \text{if } Dy = 1 \end{cases} \]
for each $x \in \mathbb{N}$ and $y \in \mathbb{N}^s$. Here $A, B, D$ are extended polynomials and so, by Lemma 9.4, there are terms $\alpha, \beta, \delta$ such that
\[ \alpha \overline{n} \triangleright \overline{A_n} \quad \beta \overline{n} \triangleright \overline{B_n} \quad \delta \overline{n} \triangleright \overline{D_n} \]
for all $n \in \mathbb{N}^s$. Let
\[ \hat{h} = \lambda v : \mathcal{N}^{''}, u : \mathcal{N}. (\beta v)Su \quad \hat{g} = \lambda v : \mathcal{N}^{''}, u : \mathcal{N}. (\alpha v)S0 \]
and
\[ \hat{f} = \lambda v : \mathcal{N}^{''}, u : \mathcal{N}. (\delta v)(\lambda w : \mathcal{N}. \hat{g}vu)(\hat{h}vu) \]
where in $\hat{f}$ we have used a trick similar to that in the simulation of the numeric negation in the proof of Lemma 9.4. We find that for each $n \in \mathbb{N}^s$ and $m \in \mathbb{N}$
\[ \text{if } Dn = 0 \text{ then } \hat{f} \overline{n}m \triangleright (\overline{\hat{h} \overline{n}m}) \triangleright (\overline{\beta \overline{n}m}) \triangleright (\overline{\alpha \overline{n}S0}) \triangleright (\overline{\hat{g} \overline{n}m} + \overline{B \overline{n}m}) \]
\[ \text{if } Dn = 1 \text{ then } \hat{f} \overline{n}m \triangleright (\overline{\hat{h} \overline{n}m}) \triangleright (\overline{\alpha \overline{n}S0}) \triangleright (\overline{\hat{g} \overline{n}m} + \overline{B \overline{n}m}) \]
where
\[ (-) = (\lambda w : \mathcal{N}. \hat{g} \overline{n}m) \triangleright (\lambda w : \mathcal{N}. (\alpha n)S0) \]
and the details of the term $r$ are not needed. This shows that $\hat{f}$ imitates $f$ as a humble function of arity $(s, 1)$.

For the recursion step $a \rightarrow a + 1$ we modify the trick used in the case $a = 1$. We are given a meek function $f$ of arity $(s, a + 1)$, so that
\[ f(y; x, w) = \begin{cases} w + Bx & \text{if } Ey = 1 \\ x_i + A_iy & \text{if } D_iy = 1 \\ Ay & \text{if } Dy = 1 \end{cases} \]
for each $y \in \mathbb{N}^s, x \in \mathbb{N}^s$ and $w \in \mathbb{N}$. Here the $A, A_i, B$ are extended polynomials, and the $D, D_i, E$ form a partition. Using numeric negation let $F$ be given by
\[ Fy = N(Ey) \]
to obtain a decision function. For each $i$ (with $1 \leq i \leq s$) let $G_i$ be given

$$G_i y = (D_i y) \times F y$$

to obtain a family of decision functions. Observe that for each $y$ there is at most one $i$ with $G_i y = 1$. Let $G$ be given by

$$G y = N(\sum_i G_i y)$$

so that the $G_i$ with $G$ form a partition. Using these we see that

$$g(y; x) = \begin{cases} x_i + A_i y & \text{if } G_i y = 1 \\ A y & \text{if } G y = 1 \end{cases}$$

gives a meek function of arity $(s, a)$. In particular, by recursion this $g$ can be imitated as a humble function by some term $\hat{g}$. We use this in an imitation of $f$.

Consider the function $k$ of arity $(s, a + 1)$ given by

$$k(y; x, w) = \begin{cases} w + B y & \text{if } F y = 0 \\ g(y, x) & \text{if } F y = 1 \end{cases}$$

for the appropriate $y, x$, and $w$. We check that this is nothing more than $f$.

If $F y = 0$ then $E y = 1$ and hence

$$k(y; x, w) = w + B y = f(y, x, w)$$
as it should be.

If $F y = 1$ then

$$k(y; x, w) = g(y, x)$$

and $E y = 0$ so that exactly one of $D_i y, D y$ is 1 and the others are 0. If $D_i y = 1$ then $G_i y = 1$ and hence

$$k(y; x, w) = g(y, x) = x_i + A_i y = f(y, x, w)$$
as it should be. Similarly, if $D y = 1$ then each $D_i y$ and each $G_i y$ is 0 so that $G y = 1$ and hence

$$k(y; x, w) = g(y, x) = A y = f(y, x, w)$$
as it should be.

Since $f = k$ we can use the form of $k$ to produce an imitation of $f$ as a humble function.

Both $B$ and $F$ are extended polynomials, so can be simulated by terms $\beta, \phi$, respectively. By recursion we have a term $\hat{g}$ which imitates $g$. Using these we find that the term

$$\hat{f} = \lambda v : N'', u, w : N. (\phi v)(\lambda w : N. \hat{g}vu)((\beta v)Sw)$$

imitates $f$. (Do not be mislead by the two different uses of ‘$w$’ in this term.)

We use the exhaustive search method to prove the converse inclusion $\mathcal{H} \subseteq \mathcal{M}$. However, as with all uses of this method, we have to prove something stronger.

78
We consider derivations in $\lambda A$

$$(\nabla) \quad \Gamma \vdash t : \tau$$

where the root judgement meets the following restrictions.

$(\Gamma)$ The context $\Gamma$ is a list of declarations $y : N''$ and $x : N$.

$(t)$ The subject term $t$ is normal.

$(\tau)$ The predicate type $\tau$ is one of

(function) A function type

$$\widehat{N} \rightarrow \cdots \rightarrow \widehat{N} \rightarrow N$$

where each input type $\widehat{N}$ is either $N''$ or $N$, and these choices are independent

(product) A product type $\sigma \times \rho$

for some $\sigma, \rho$.

Table 8: Global conditions for Theorem 10.10

By Definition 9.5 each humble function $f$ is given by a derivation

$$\vdash \widehat{f} : \widehat{N} \rightarrow \cdots \rightarrow \widehat{N} \rightarrow N$$

where each source type $\widehat{N}$ is either $N''$ of $N$. We wish to show that $f$ is meek where its arity can be read of from the source types of $\widehat{f}$. In other words, each source type $N''$ gives a simulated input and each source type $N$ gives an authentic input. The result is an extension of Theorem 7.4 and is proved in almost the same way. As there we need to look at a larger class of derivations, as set out in Table 8.

The clause (product) is needed because product predicate types occur quite naturally in the induction. We can ignore this point until we get into the proof.

The derivation is allowed to have a non-empty context $\Gamma$, but of a restricted kind. By several uses of (Introduction) we can convert the derivation $\nabla$ into a derivation

$$(\nabla^+) \quad \vdash \lambda \Gamma . t : \tau^+$$

in the empty context. We know that $\nabla$ and $\nabla^+$ have essentially the same meaning (modulo currying). By the various restrictions the new subject term $\lambda \Gamma . t$ imitates a humble function $f$, and this is the essential meaning of $\nabla$. As in Theorem 7.4 we say $\nabla$ provides $f$.

In the proof of the following we stick as close as possible to the phraseology of the proof of Theorem 7.4. This will emphasize the similarities and differences between the two proofs.

10.10 THEOREM. For each derivation of $\lambda A$

$$(\nabla) \quad \Gamma \vdash t : \tau$$

where the root judgement meets the global conditions of Table 8, one of the following applies
If $\tau$ is a function type then $\nabla$ provides a meek function of arity determined by the context $\Gamma$ and the predicate type $\tau$.

If $\tau$ is a product type the $t$ is $\text{Pair}lr$ for some terms $l, r$.

Depending on the shape of $\tau$.

**Proof.** It is convenient to write

$$f(y; x)$$

for a typical value of the function $f$ provided by $\nabla$ when the (function) case arises. Here we have split the inputs in accordance with their types, simulated of authentic. Notice that each input may arise from the context $\Gamma$ or from the predicate type $\tau$. Of course, we have taken the liberty of permuting the inputs to fit this pattern.

We proceed by induction over the height of $\nabla$. To do this we survey all the possible ways that $\nabla$ can arise.

Firstly, $\nabla$ may be a leaf.

**$(\nabla \text{ Axiom})$** Since we are working in $\lambda A$ there are five possible shapes for $t : \tau$.

\begin{align*}
(i) \quad & 0 : \mathcal{N} \\
(ii) \quad & S : \mathcal{N}' \\
(iv) \quad & \text{Pair} : \sigma \to \rho \to \sigma \times \rho \\
(v) \quad & \text{Left} : \sigma \times \rho \to \sigma \\
v(i) \quad & \text{Right} : \sigma \times \rho \to \rho
\end{align*}

Here $\sigma, \rho$ are arbitrary. Some of these meet the global conditions and some do not.

(i) This provides the meek function of arity $(y, x)$ which is is identically 0.

(ii) This provides the meek function $f$ of arity $(y, x)$ where

$$f(y, x) = x + 1$$

where this $x$ arises from the source of the predicate type $(\mathcal{N} \to \mathcal{N})$.

(iii) As indicated, this case doesn’t occur since there are no iterators in $\lambda A$.

(iv, v, vi) These cases don’t arise since each of

$$\sigma \to \rho \to \sigma \times \rho \quad \sigma \times \rho \to \sigma \quad \sigma \times \rho \to \rho$$

has the wrong shape to be a predicate type for these global conditions.

**$(\nabla \text{ Projection})$** Because of the nature of $\Gamma$ we see that the statement $t : \tau$ is either $y : \mathcal{N}''$ or $x : \mathcal{N}$ depending on which component of $\Gamma$ is projected from. The first of these matches neither (function) nor (product), so only the second can arise. This produces a projection viewed as a meek function.

Secondly, $\nabla$ may arise by a use of one of the construction rules. These will involve shorter derivation to which, if the global conditions are met, we may apply the induction hypothesis.

**$(\nabla \text{ Weakening})$** From the induction hypothesis we see that the numerator of this particular use provides some meek function $h$, and then the denominator provides a function $f$ obtained from $h$ by inserting a dummy argument. This can be achieved by a use of Lemma 10.6, to show that $f$ is meek.

**$(\nabla \text{ Introduction})$** Consider the numerator and denominator of any use of Introduction. The denominator predicate must be a function type, so here case (product) can not arise.
and (function) must arise. But now the numerator falls into case (function) and so, by the induction hypothesis, provides a meek function. Finally, remember that the numerator and denominator provide the same function.

(∇ Elimination) We track through the construction of ∇ to unravel as many uses of Elimination and Weakening as possible. Thus we obtain a family of shorter derivations

\[(Q) \quad \Xi \vdash q : \pi_k \rightarrow \cdots \rightarrow \pi_1 \rightarrow \tau \quad (P_i) \quad \Pi_i \vdash p_i : \pi_i \quad i \leq i \leq k\]

where

\[t = q p_k \cdots p_1\]

with \(q\) and each \(p_i\) normal, and where each context \(\Xi\) and \(\Pi_i\) is a left hand part of \(\Gamma\). Note that \(k \geq 1\). In this process we may have removed some uses of Weakening. These can be reinstated to obtain derivations

\[(Q^+) \quad \Gamma \vdash q : \pi_k \rightarrow \cdots \rightarrow \pi_1 \rightarrow \tau \quad (P_i^+) \quad \Gamma \vdash p_i : \pi_i \quad i \leq i \leq k\]

all in the original context. Since \(k \geq 1\), each of these is strictly shorter than ∇.

We consider how \(Q\) can arise. To do this let

\[\chi = \pi_k \rightarrow \cdots \rightarrow \pi_1 \rightarrow \tau\]

be its predicate type.

(Q Axiom) There are five possible as listed above.

(i) This can not arise since \(\chi\) must be compound.

(ii) Here we must have

\[(Q^+) \quad \Gamma \vdash S : N \rightarrow N \quad (P^+) \quad \Gamma \vdash p : N\]

with \(k = 1\) and with \(\tau = N\) and \(t = S p\) for some term \(p\). The shorter derivation \(P^+\) meets the global restriction and hence, by the induction hypothesis, this derivation provides a meek function \(g\). The function \(f\) provided by ∇ is given by

\[f(y; x) = 1 + g(y; x)\]

which is meek (by a rather simple case of Lemma 10.6).

(iii) This case doesn’t occur since there are no iterators in \(\lambda A\).

(iv) As in the proof of Theorems 7.1 and 7.4, here we have at least

\[(Q^+) \quad \Gamma \vdash \text{Pair} : \sigma \rightarrow \rho \rightarrow \sigma \times \rho \quad (L^+) \quad \Gamma \vdash l : \sigma\]

and possibly

\[(R^+) \quad \Gamma \vdash r : \rho\]

as well. In other words one of

\[t = \text{Pair}\ l \text{ with } \tau = \rho \rightarrow \sigma \times \rho \quad t = \text{Pair}\ l r \text{ with } \tau = \sigma \times \rho\]

holds. The first of these is discounted by the global conditions on \(\tau\), and the second meets the required conclusion (product).
(v) Here we have at least

\[(Q^+) \quad \Gamma \vdash \text{Left} : \sigma \times \rho \rightarrow \sigma \quad (P^+) \quad \vdash p : \sigma \times \rho\]

where there may be other auxiliary derivations \(P_2^+, \ldots, P_k^+\). The shorter derivation \(P^+\) meets the global conditions and hence the induction hypothesis gives \(p = \text{Pair}lr\) for some terms \(l, r\). But now

\[t = \text{Left}(\text{Pair}lr) \ldots\]

which since

\[\text{Left}(\text{Pair}lr) \triangleright l\]

contradicts the assumed normality of \(t\). Thus this case can not arise.

(vi) As with (v), this case can not arise.

This completes the case where \(Q\) is an Axiom.

\((Q\text{ Projection})\) This crucial case and the one that differs from the proof of Theorem 7.4. Let’s leave this until the end and get rid of the others first.

\((Q\text{ Weakening})\) This case is built into the unravelling of \(\nabla\).

\((Q\text{ Introduction})\) In this case we have \(q = (\lambda x : \pi \cdot r)\) for some term \(r\), where \(\pi = \pi_k\). But then with \(p = p_k\) we see that the redex \((\lambda x : \pi \cdot r)p\) is an initial part of \(t\). Since \(t\) is normal, this case can not arise.

\((Q\text{ Elimination})\) This case is built into the unravelling of \(\nabla\).

\((Q\text{ Projection – again})\) Because of the nature of \(\Xi\) we have either

\[\Xi \vdash y : \mathcal{N}''\] with \(\chi = \mathcal{N}''\) or \[\Xi \vdash x : \mathcal{N}\] with \(\chi = \mathcal{N}\)

depending on the declaration projected. But \(\chi\) must be compound, so only the first of these can arise. Thus we must have one of

\[\Xi \vdash y : \mathcal{N}'' \quad \Pi \vdash p : \mathcal{N} \quad \text{with} \quad t = yp\]

\[\Xi \vdash y : \mathcal{N}'' \quad \Pi \vdash p : \mathcal{N} \quad \Lambda \vdash r : \mathcal{N} \quad \text{with} \quad t = ypr\]

where \(\Xi, \Pi,\) and \(\Lambda\) (when it occurs) are left hand parts of \(\Gamma\).

As before, by reinstating the uses of Weakening we obtain

\[(Q^+) \quad \Gamma \vdash y : \mathcal{N}''\quad (P^+) \quad \Gamma \vdash p : \mathcal{N} \rightarrow \mathcal{N} \quad (R^+) \quad \Gamma \vdash r : \mathcal{N}\]

where in some cases \(R^+\) might not arise. These derivations are still shorter than \(\nabla\) (because of the missing use of Elimination).

We may apply the induction hypothesis to \((P^+)\) to obtain a meek function \(h\). In a typical value \(h(y; x, u)\) of \(h\) the right hand authentic input corresponds to the input type \(\mathcal{N}\) in the type of \(p\). All the other inputs of \(h\) arise from the context \(\Gamma\).

Consider \(\Gamma \vdash yp : \mathcal{N}'\) where \(yp\) is a subterm of \(t\). Think of the function \(f\) produced by this derivation. (Note that we are not invoking the induction hypothesis to claim that this function is meek.) The identifier \(y\) corresponds to one of the simulated inputs of \(h\). Thus, remembering how simulated numerals work, we have

\[f(y; x, u) = h(y; x, \cdot)^yu\]

to show, by Lemma 10.8, that, in fact, \(f\) is meek with indicated arity.
When $t = yp$ this is the function produced by $\nabla$.

When $t = ypr$ we apply the induction hypothesis to $(R)$ to obtain a function $g(y; x)$ which is meek with indicated arity. There is no input $u$ since all the inputs arise from the context $\Gamma$. With this we see that the composite

$$f(y; x) = h(y; x, \cdot)^pg(y; x)$$

is the function produced by $\nabla$. By Lemma 10.8 this is meek.

If you read through this proof again you will see that the induction is tracked by a recursive algorithm by which the imitation of a humble function can be converted into a description of a meek function in the canonical form of Definition 10.3. It is worth stating this consequence separately.

10.11 SCHOLIUM. We have $\mathcal{H}(s; a) \subseteq \mathcal{M}(s; a)$. Furthermore, each humble function can be produced as a meek by tracking through a witnessing derivation. In particular, the arity can be read off from the predicate type.

We now have enough information to add to Theorem 7.5. We have

$$\mathcal{M} = \mathcal{E} \subseteq \mathcal{U}(\lambda A) \subseteq \mathcal{H}$$

where Lemma 10.5 give the equality and Corollary 9.8 gives the inclusions. We also have

$$\mathcal{M}(s; a) = \mathcal{H}(s; a)$$

by Theorem 10.9 and Scholium 10.11. These give the following extension of Theorem 7.5.

10.12 THEOREM. We have

$$\mathcal{R}(\lambda A) = \mathcal{T} \quad \mathcal{U}(\lambda A) = \mathcal{E}$$

for the system $\lambda A$.

Proof. The left hand equality is just Theorem 7.5. The right hand equality is proved above.

The next class to look at is $\mathcal{G}(\lambda A)$. Recall that we still haven’t completed the proof that this is closed under composition. To do that we need a new idea.

[Held in 165.../05-Version../011.. Last changed April 13, 2006]

11 The network of numerals

A first order function can be captured in a tiered system in several different ways; by representation, by simulation, or by imitation (in the narrow sense). Later in this section we introduce a notion of imitation (in the broad sense) which covers all of these different modes of capture.

Once or twice we have seen how one kind of capture can be converted into another kind of capture. In this section we develop a more systematic method of manipulating
captures. This attaches to each tiered system a geometric object whose properties seem to be closely related to the strengths of the parent system.

To begin let's have another look at the trick we have used a couple of times. It is used explicitly in Example 8.4 between Definitions 9.5 and 9.6 and is hidden in the proof of Theorem 10.9.

11.1 EXAMPLE. Let \( \xi \) be an atom in some tiered system \( \lambda T \), and set

\[
C = \lambda w : \xi''. w \Sigma \lambda x. i
\]

using the furnishing of \( \xi \). We have

\[
\vdash C : \xi'' \to \xi
\]

with

\[
\boxed{C \triangleright m}
\]

for each \( m \in \mathbb{N} \). Consider terms

\[
(r) \quad \vdash [f^\triangleright] : \xi \to \xi
\]
\[
(c) \quad \vdash \overline{f} : \xi'' \to \xi''
\]
\[
(i) \quad \vdash \hat{f} : \xi \to \xi''
\]

which represent, simulate, and imitate a 1-placed function \( f \). (Here the imitation is in a broader sense to be developed in more detail later.) Thus

\[
(r) \quad \boxed{[f^\triangleright m] \triangleright \boxed{[f m]}}
\]
\[
(c) \quad \boxed{f m} \triangleright \boxed{f m}
\]
\[
(i) \quad \boxed{\hat{f} m} \triangleright \boxed{\hat{f} m}
\]

for each \( m \in \mathbb{N} \). Now let

\[
(r) \quad \tilde{f} = [f^\triangleright \circ C = \lambda u : \xi'' \cdot [f^\triangleright (u \Sigma \lambda x. i)]
\]
\[
(c) \quad \tilde{f} = C \circ \overline{f} = \lambda u : \xi''. (\overline{f} u) \Sigma \lambda x
\]
\[
(i_1) \quad \tilde{f} = \hat{f} \circ C = \lambda u : \xi''. \hat{f} (u \Sigma \lambda x)
\]
\[
(i_2) \quad \tilde{f} = C \circ \hat{f} = \lambda u : \xi''. (\hat{f} u) \Sigma \lambda x
\]
\[
(i_3) \quad \tilde{f} = C \circ \hat{f} \circ C = \lambda u : \xi''. ([\hat{f} (u \Sigma \lambda x)]) \Sigma \lambda x
\]

to convert terms

\[
(r) \quad \text{from } \vdash [f^\triangleright] : \xi \to \xi \quad \text{to } \vdash \tilde{f} : \xi'' \to \xi
\]
\[
(c) \quad \text{from } \vdash \overline{f} : \xi'' \to \xi'' \quad \text{to } \vdash \tilde{f} : \xi'' \to \xi
\]
\[
(i_1) \quad \text{from } \vdash \hat{f} : \xi \to \xi'' \quad \text{to } \vdash \tilde{f} : \xi'' \to \xi''
\]
\[
(i_2) \quad \text{from } \vdash \hat{f} : \xi \to \xi'' \quad \text{to } \vdash \tilde{f} : \xi \to \xi
\]
\[
(i_3) \quad \text{from } \vdash \hat{f} : \xi \to \xi'' \quad \text{to } \vdash \tilde{f} : \xi'' \to \xi
\]
respectively. We may check that each constructed term $\tilde{f}$ captures $f$ in the sense that

$$\tilde{f} m \Rightarrow \tilde{f} m$$

for each $m \in \mathbb{N}$. Here the input and output numerals are of the appropriate kind, and these are not the same except for $(i_1, i_2)$.

In this example we have converted one kind of capture of a function into another kind of capture of the same function. But there appears to be some cases missing. Why can’t we convert an imitation

$$\text{from } \tilde{f} : \xi'' \to \xi \text{ to } \tilde{f} : \xi \to \xi''$$

say? Think about it. The answer is important, and that is what this section is about.

We wish to move numerals between types. To do that we first uncover the paths along which they move.

Recall from Definition 2.2 how types are built up.

11.2 DEFINITION. For types $\tau, \gamma$ we write

$$\tau \downarrow \gamma$$

and say $\tau$ immediately descends to $\gamma$ if one of

- $\tau = \pi \to \sigma$ where $\gamma = \sigma$
- $\tau = \sigma \times \rho$ where $\gamma = \sigma$ or $\gamma = \rho$

holds for types $\pi, \sigma, \rho$.

Note that this is not just an immediate unravelling of the structure of the type $\tau$. For $\tau = \pi \to \sigma$ we do not have $\tau \downarrow \pi$ unless, of course, $\pi = \sigma$.

11.3 DEFINITION. For types $\tau, \gamma$ we write

$$\tau \Downarrow \gamma$$

and say $\tau$ descends to $\gamma$ if one of

- $\tau = \gamma$
- $\tau = \pi \to \sigma$ where $\sigma \Downarrow \gamma$
- $\tau = \sigma \times \rho$ where $\sigma \Downarrow \gamma$ or $\rho \Downarrow \gamma$

holds for types $\pi, \sigma, \rho$.

This, of course, is a definition by recursion over the structure of $\tau$.

In other words, the descent relation $\Downarrow$ is the reflexive, transitive closure of the 1-step relation $\downarrow$ of immediate descent.

To determine whether or not $\tau \Downarrow \gamma$ we begin to unravel the construction of $\tau$ and consider all relevant sub-types until we meet $\gamma$. The only restriction is that for an arrow type $\tau = \pi \to \sigma$ we cannot follow $\pi$, we must follow $\sigma$. This produces a descent path
from the type $\tau$ to the type $\gamma$. For each type $\tau$ there is at least one atom $\xi$ with $\tau \downarrow \xi$, and every maximal descent path from a type must end in an atom.

Consider a descent

$$\tau \downarrow \xi$$

from an arbitrary type $\tau$ to an atom $\xi$. These types carry numerals

$$\overline{m_\tau} : \tau'' \quad \overline{m_\xi} : \xi$$

simulated over $\tau$ and authentic over $\xi$. Is there any way we can convert one kind into the other? The descent is determined by a descent path

$$\tau = \tau_0, \tau_1, \ldots, \tau_i, \ldots, \tau_l = \xi$$

where

$$\tau_i \downarrow \tau_{i+1}$$

for each $0 \leq i < l$. We can convert numerals $\overline{m_\tau}$ into numerals $\overline{m_\xi}$ by following this path and converting numerals at each step. There are two extreme ways we might do this. The first is to produce simulated numerals of types

$$\tau'' = \tau_0'', \tau_1'', \ldots, \tau_i'', \ldots, \tau_l'' = \xi''$$

with a use of the term $C : \xi'' \rightarrow \xi$ at the end. We look at the details of this shortly. The other way uses a generalization of the authentic numerals.

11.4 DEFINITION. For each type $\tau$ the lifted furnishings of $\tau$

$$0_\tau : \tau \quad S_\tau : \tau'$$

are generated by recursion over $\tau$.

- If $\tau$ is an atom then these furnishings are as given.

- If $\tau = \pi \rightarrow \sigma$ then

$$0_\tau = \lambda p : \pi.0_\sigma \quad S_\tau = \lambda v : \tau, p : \pi. S_\sigma(v p)$$

using the furnishings of $\sigma$

- If $\tau = \sigma \times \rho$ then

$$0_\tau = \text{Pair} 0_\sigma 0_\rho \quad S_\tau = \lambda v : \tau. \text{Pair}(S_\sigma(\text{Left} v))(S_\rho(\text{Left} v))$$

using the furnishings of $\sigma$ and $\rho$.

Of course, in the product clause the pairing gadgets should be correctly indexed.

For each type $\tau$ and $m \in \mathbb{N}$ we set

$$\overline{m_\tau} = S''_\tau 0_\tau$$

to obtain the lifted $\tau$-numerals. ■
It is routine to check that the types of $S_\tau$ and $0_\tau$ are as given. Also we have

$$\vdash \lceil m_\tau \rceil : \tau$$

for each $m \in \mathbb{N}$.

When $\tau$ is an atom these furnishings and numerals are just the authentic gadgets. However, when $\tau$ is not an atom the numeral $\lceil m_\tau \rceil$ need not be normal and we do not have $\lceil m_\tau \rceil = m$. These lifted numerals are designed merely to provide stepping stones between the authentic and the simulated numerals.

With lifted numerals we can generalize the term $C$ of Example 11.1 and earlier.

Using the furnishings of the type $\tau$ we set

$$C_\tau = \lambda w : \tau''. w S_\tau 0_\tau$$

to produce a term of type $\tau'' \to \tau$.

11.5 LEMMA. (a) For each type $\tau$ we have

$$C_\tau m_\tau \triangleright \lceil m_\tau \rceil$$

for each $m \in \mathbb{N}$.

(b) For each compound type

$$\tau = \pi \to \sigma \quad \tau = \sigma \times \rho$$

we have

$$\lceil m_\tau \rceil \triangleright \lambda p : \pi. \lceil m_\sigma \rceil \quad \lceil m_\tau \rceil \triangleright \text{Pair} \lceil m_\sigma \rceil m_\rho$$

for each $m \in \mathbb{N}$.

Proof. (a) This is straightforward (and we have done the calculation a couple of times already).

(b) We verify the left hand reduction by induction over $m$.

The base case, $m = 0$, is an equality (which is why the result is stated with $\triangleright$ as the reduction).

For the induction step, $m \mapsto m + 1$, we have

$$\lceil (m + 1)_\tau \rceil = S_\tau \lceil m_\tau \rceil \triangleright \lambda p : \pi. S_\sigma \lceil m_\sigma \rceil p \triangleright \lambda p : \pi. S_\sigma \lceil m_\sigma \rceil = \lambda p : \pi. \lceil (m + 1)_\sigma \rceil$$

as required. Here the second step unravels $S_\tau$ and the third uses the induction hypothesis.

For the right hand reduction we observe that the construction of $S_\tau$ gives

$$\text{Left}(S_\tau t) \triangleright \text{Left}(\text{Pair}(S_\sigma (\text{Left} t))(S_\sigma (\text{Right} t))) \triangleright S_\sigma (\text{Left} t)$$
$$\text{Right}(S_\tau t) \triangleright \text{Right}(\text{Pair}(S_\sigma (\text{Left} t))(S_\sigma (\text{Right} t))) \triangleright S_\sigma (\text{Right} t)$$

for each term $t$.

Using this we check that

$$\text{Left} \lceil m_\tau \rceil \triangleright \lceil m_\sigma \rceil \quad \text{Right} \lceil m_\tau \rceil \triangleright \lceil m_\rho \rceil$$

for each $m \in \mathbb{N}$. As expected, we proceed by induction over $m$, and we may deal only with the left hand reduction.
For the base case, \( m = 0 \), we have
\[
\text{Left}^\tau_0 = \text{Left}^\tau_0 = \text{Left}(\text{Pair}^\sigma_0^\rho) \Downarrow 0_\sigma = \tau_0^\sigma
\]
by the construction of \( 0_\tau \).

For the induction step, \( m \mapsto m + 1 \), we have
\[
\text{Left}^\tau(m + 1) = \text{Left}(S^\tau(m) \Downarrow S^\sigma(\text{Left}^\tau m)) \Downarrow S^\sigma(m) = \tau_0
\]
using the observation above and the induction hypothesis.

Finally we verify the required reduction by a third induction over \( m \).

As before, the base case, \( m = 0 \), is an equality.

For the induction step, \( m \mapsto m + 1 \), we have
\[
\tau(m + 1) = S^\tau(m) \Downarrow S^\sigma(\text{Left}^\tau m) \Downarrow S^\sigma(m) = \tau_0 = \tau_0
\]
as required. Here the second step unravels the construction of \( S_\tau \), and the third step uses the preliminary reduction above. Notice that the current induction hypothesis is not actually used.

We extend the idea of part (a). Consider types \( \tau, \sigma \) and select
\[
\hat{\tau} = \tau'' \text{ or } \tau \quad \hat{\sigma} = \sigma'' \text{ or } \sigma
\]
where these choices are independent (so there are four cases). For each \( m \in \mathbb{N} \) set
\[
\hat{m}_\tau = \tau m_\tau \text{ or } \hat{m}_\tau = \tau m_\tau \quad \hat{m}_\sigma = \tau m_\sigma \text{ or } \hat{m}_\sigma = \tau m_\sigma
\]
to match the choices.

11.6 DEFINITION. Continuing with the notation above, a numeral converter is a term
\[
\vdash L : \hat{\tau} \rightarrow \hat{\sigma}
\]
(derivable in the system of interest) such that
\[
L\hat{m}_\tau \Downarrow \hat{m}_\sigma
\]
holds for all \( m \in \mathbb{N} \). The important point here is that the same number \( m \in \mathbb{N} \) is used to generate the two numerals but the housing type changes.

The identity terms
\[
\text{Id}_{\tau''} : \tau'' \rightarrow \tau'' \quad \text{Id}_\tau : \tau \rightarrow \tau
\]
are trivial numeral converters (converting simulated numerals into simulated numerals and lifted numerals into lifted numerals, respectively). The term
\[
C_\tau : \tau'' \rightarrow \tau
\]
given above is a numeral converter, from simulated to lifted.

If

\[ L : \hat{\tau} \rightarrow \hat{\sigma} \quad M : \hat{\sigma} \rightarrow \hat{\rho} \]

are numeral converters (where the two \( \hat{\sigma} \) are the same) then the composite

\[ M \circ L : \hat{\tau} \rightarrow \hat{\rho} \]

is a numeral converter.

There are more interesting examples of numeral converters.

11.7 THEOREM. For each descent path \( \tau \downarrow \gamma \) there are numeral converters

\[ C_{\gamma,\tau} : \tau'' \rightarrow \gamma \quad D_{\gamma,\tau} : \tau \rightarrow \gamma \quad E_{\gamma,\tau} : \tau'' \rightarrow \gamma'' \]

taking

simulated to lifted \quad lifted to lifted \quad simulated to simulated

respectively.

Proof. We deal first with the terms \( D \) and \( E \) for a 1-step descent \( \tau \downarrow \gamma \). There are several cases to look at.

(\( \tau = \pi \rightarrow \sigma \), \( D_{\pi,\pi} \)) For a type \( \tau = \pi \rightarrow \sigma \) we set

\[ D_{\pi,\pi} = \lambda v : \tau . v \circ_0 \pi \]

to generate the 1-step term. Almost trivially this has the required type. For each \( m \in \mathbb{N} \) we have

\[ D_{\pi,\pi} \{ m \tau \} \overset{\Rightarrow}{\Rightarrow} \{ m \tau \circ_0 \pi \} \overset{\Rightarrow}{\Rightarrow} \{ m \sigma \} \]

where the second reduction follows by Lemma 11.5(b).

(\( \tau = \pi \rightarrow \sigma \), \( E_{\pi,\pi} \)) For the given type \( \tau \) let

\[ K = \lambda x : \sigma , p : \pi . x \quad B = \lambda y : \sigma' , v : \tau , p : \pi . y(vp) \]

to obtain terms

\[ K : \sigma \rightarrow \tau \quad B : \sigma' \rightarrow \tau' \]

with

\[ Ksp \overset{\Rightarrow}{\Rightarrow} x \quad Btsp \overset{\Rightarrow}{\Rightarrow} t(sp) \]

for all terms \( s, t, p \).

We show that

\[ (Bt)^m(Ks)p \overset{\Rightarrow}{\Rightarrow} t^m s \]

for all terms \( t, s, m \) and \( m \in \mathbb{N} \). Naturally, we proceed by induction over \( m \).

For the base case, \( m = 0 \), we have

\[ (Bt)^0(Ks)p = (Ks)p \overset{\Rightarrow}{\Rightarrow} s = t^0 s \]

by the construction of \( K \).

For the induction step, \( m \mapsto m + 1 \), with

\[ s' = ((Bt)^m s) \]

89
we have
\[ s'p \triangleright t^m s \]
by the induction hypothesis, so that
\[ (Bt)^{m+1}(Ks)p = Bts'p \triangleright t(s'p) \triangleright t(t^m s) = t^{m+1} s \]
as required.

Finally (for this case) with
\[ E_{\sigma,\tau} = \lambda w : \tau'', y : \sigma', x : \sigma . w(By)(Kx)0 \pi \]
we have
\[ E : \tau'' \rightarrow \sigma'' \]
by simply unravelling the construction of \( E \). For each \( m \in \mathbb{N} \) the previous calculations give
\[ \overline{E_{\sigma,\tau}} \triangleright \lambda y : \sigma, x : \sigma . m_\tau(By)(Kx)0 \pi \]
\[ \triangleright \lambda y : \sigma, x : \sigma . (By)^m(Kx)0 \pi \]
\[ \triangleright \lambda y : \sigma, x : \sigma . y^m x = \overline{m_\sigma} \]
as required.

\((\tau = \sigma \times \rho)\) For such a type there are two 1-step descents
\[ \tau \downarrow \sigma \quad \tau \downarrow \rho \]
the left case and the right case. These cases are dealt with in a similar way. We will look at the details for the left case only. Of course, we use the appropriately indexed constants
\[ \text{Pair} : \sigma \rightarrow \rho \rightarrow \tau \quad \text{Left} : \tau \rightarrow \sigma \quad \text{Right} : \tau \rightarrow \rho \]
and the reductions
\[ \text{Left(Pairlr)} \triangleright l \quad \text{Right(Pairlr)} \triangleright r \]
in constructions.

\((\tau = \sigma \times \rho, D_{\bullet,\bullet})\) Using Lemma 11.5 we see that
\[ D_{\sigma,\tau} = \text{Left}_{\sigma,\rho} \]
has the required reduction property.

\((\tau = \sigma \times \rho, E_{\bullet,\bullet})\) For the given type \( \tau \) let
\[ J = \lambda x : \sigma . \text{Pair} x 0 \rho \quad A = \lambda y : \sigma', v : \tau . \text{Pair}(y(\text{Left} v)(\text{Right} v)) \]
to obtain terms
\[ J : \sigma \rightarrow \tau \quad A : \sigma' \rightarrow \tau' \]
with
\[ \text{Left}(Js) \triangleright s \quad \text{Left}(Atr) \triangleright t(\text{Left} r) \]
for all terms \( r, s, t \).
We check that
\[ \text{Left}((At)^m r) \not\not\not t^m(\text{Left} r) \]
for all terms \( r, t \) and \( m \in \mathbb{N} \). Of course, we proceed by induction on \( m \).

The base case, \( m = 0 \), is immediate.

For the induction step, \( m \mapsto m + 1 \), with
\[ r' = (At)^m r \]
we have
\[ (At)^{m+1} r = Atr' \not\not\not \text{Pair}(t(\text{Left} r')(\_)) \]
so that
\[ \text{Left}((At)^{m+1} r) \not\not\not \text{Left}(\text{Pair}(t(\text{Left} r')(\_))) \not\not\not t(\text{Left} r') \not\not\not t(t^m(\text{Left} r)) = t^{m+1}(\text{Left} r) \]
as required. Here we use the construction of \( A \), the reduction above, and the induction hypothesis.

Finally, for this case, we show that
\[ E_{\sigma,\tau} = \lambda w : \tau'', y : \sigma', x : \sigma. \text{Left}(w(Ay)(Jx)) \]
will do. We have
\[ \vdash E : \tau'' \rightarrow \sigma'' \]
by an unravelling \( E \). For each \( m \in \mathbb{N} \) the previous calculations give
\[
\begin{align*}
\text{E}_{\tau''} & \not\not\not \lambda y : \sigma, x : \sigma. \text{Left}(\text{E}_{\tau''}(Ay)(Jx)) \\
& \not\not\not \lambda y : \sigma, x : \sigma. \text{Left}((Ay)^m(Jx)) \\
& \not\not\not \lambda y : \sigma, x : \sigma. y^n(\text{Left}(Jx)) \\
& \not\not\not \lambda y : \sigma, x : \sigma. y^m x = m_\sigma
\end{align*}
\]
as required.

This deals with the 1-step descents of the terms \( D \) and \( E \). Since numeral converters are closed under composition, longer descents for these terms are easily constructed.

For each type \( \tau \) the term \( C_{\tau} \) of Lemma 11.5 is the numeral converter \( C_{\tau,\tau} \). Longer descents for this case can be obtained by using composites with the \( D \) of \( E \) terms. 

The last couple of lines of this proof are perhaps a little terse, so let’s look at an example of that construction.

How can we produce a numeral converter
\[ C_{\sigma,\tau} : \tau'' \rightarrow \sigma \]
for a 1-step descent \( \tau \downarrow \sigma \)? There are two possible composites
\[
\begin{array}{c}
\tau'' \xrightarrow{E_{\sigma,\tau}} \sigma'' \\
C_{\tau} \xrightarrow{D_{\sigma,\tau}} \sigma
\end{array}
\]
using the terms $D, E$ constructed in the earlier part of the proof and two version of the term $C$ of Lemma 11.5. Thus either of the composites

$$D_{\sigma, \tau} \circ C_\tau \quad C_{\sigma} \circ E_{\sigma, \tau}$$

will produce a version of $C_{\sigma, \tau}$. Let’s look at some of the details of these terms.

There are, of course, two cases depending on the shape of $\tau$.

$(\tau = \pi \rightarrow \sigma)$ For this case we have

$$D_{\sigma, \tau} \circ C_\tau = \lambda w : \tau'' . D_{\sigma, \tau}(C_\tau w) \ll \lambda w : \tau'' . C_\tau w 0_\pi \ll \lambda w : \tau'' . w S_\sigma 0_\sigma 0_\pi$$

$$C_{\sigma} \circ E_{\sigma, \tau} = \lambda w : \tau . C_{\sigma}(E_{\sigma, \tau} w) \ll \lambda w : \tau . E w S_\sigma 0_\sigma \ll \lambda w : \tau . w (BS_{\sigma})(K0_\sigma)0_\pi$$

where

$$BS_{\sigma} \ll \lambda v : \tau, p : \pi . S_{\sigma}(vp) \quad K0_{\sigma} \ll \lambda p : \pi . 0_\sigma$$

brings about a further slight simplification for the second term.

It seems that the term obtained from $D_{\sigma, \tau} \circ C_\tau$ is more efficient.

$(\tau = \sigma \times \rho)$ For this case we have

$$D_{\sigma, \tau} \circ C_\tau = \lambda w : \tau'' . D(C_\tau w) = \lambda w : \tau''. \left\langle \text{Left}(Cw) \ll \lambda w : \tau''. \text{Left}(w S_\sigma 0_\tau) \right\rangle$$

$$C_{\sigma} \circ E_{\sigma, \tau} = \lambda w : \tau . C_{\sigma}(Ew) \ll \lambda w : \tau . E w S_\sigma 0_\sigma \ll \lambda w : \tau . \text{Left}(w (AS_{\sigma})(J0_\sigma))$$

where

$$AS_{\sigma} \ll \lambda v : \tau . \text{Pair}(S_{\sigma}(\text{Left}v)(\text{Right}v)) \quad J0_{\sigma} \ll \text{Pair}0_\sigma 0_\pi$$

brings about a further slight simplification for the second term.

Again it seems that the term obtained from $D_{\sigma, \tau} \circ C_\tau$ is more efficient.

These calculations suggest that for each 1-step descent $\tau \downarrow \sigma$ we take

$$C_{\sigma, \tau} = \lambda w : \tau . D_{\sigma, \tau}(w S_\tau 0_\tau)$$

for the numeral converter of type $\tau'' \rightarrow \sigma$. In both cases $D_{\sigma, \tau}$ is a selection gadget; when $\tau = (\pi \rightarrow \sigma)$ it evaluates at $0_\pi$, and when $\tau = \sigma \times \rho$ or $\tau = \rho \times \sigma$ it selects the ‘$\sigma$’ component.

11.8 EXAMPLE. Consider a 4-step descent

$$\tau \downarrow \chi \downarrow \sigma \downarrow \rho \downarrow \xi$$

where

$$\tau = \bullet \rightarrow \chi \quad \chi = \sigma \times \bullet \quad \sigma = \bullet \times \rho \quad \rho = \bullet \rightarrow \xi$$

with $\xi$ an atom. The details of the types $\bullet$ are not needed. Then the term

$$\lambda w : \tau''. D_{\xi, \rho}(D_{\rho, \sigma}(D_{\sigma, \chi}(D_{\chi, \tau}(w S_\tau 0_\tau)))$$

is a numeral converter (from simulated to authentic) of type $\tau'' \rightarrow \xi$. This uses the lifted numerals at $\tau$.

It may seem more natural to use the simulated numerals to get from $\tau''$ to $\xi''$. However, I suggest you use the calculations above to work out what this path gives. ■
There are certain aspects of these constructions that I have not investigated. Perhaps they should be.

These numeral converters exist in all tiered systems. Notice that each one either moves down a descent path or for some type converts the simulated numerals at that type into the lifted numerals at that type. Why can’t we go up a descent path or convert from authentic to simulated? For some systems we can, and to a large extent it is these other possible converters which determine the power of a tiered system.

For each iterator \( I : \xi \to \tau'' \) the term

\[
L = \lambda u : \xi . \lambda y : \tau', x : \tau . l u y x
\]

is a numeral converter, of the same type, from authentic to simulated numerals.

Conversely, if \( L : \xi \to \tau'' \) is a numeral converter then the term

\[
I = \lambda u : \xi . \lambda y : \tau', x : \tau . L u y x
\]

behaves as a pointwise iterator of the same type. Thus, for each \( m \in \mathbb{N} \) the compound \( L^m \) has the same effect as the iterator, and the term \( I \) can be used in place of the iterator.

11.9 LEMMA. Suppose the tiered system \( \lambda T \) has a numeral converter

\[
L : \xi \to \xi''
\]

(from authentic to simulated) for some atom \( \xi \). Then each generic \( G \)-function on a finite level is named in \( \lambda T \).

Proof. Using the given converter \( L \) let

\[
\text{ack} = \lambda y : \xi', x : \xi . L(\xi x)y x
\]

so that \( \vdash \text{ack} : \xi'' \) is derivable. Suppose \( \vdash \forall f : \xi' \) names a function \( f : \mathbb{N}' \). For each \( m \in \mathbb{N} \), with \( n = m' \) we have

\[
\text{ack}^f m \xi \quad \Rightarrow \quad L^f m \xi \quad \Rightarrow \quad \forall f^m \xi\]

which shows that \( \text{ack}^f \) names \( ack f \). Thus, by iteration, the terms

\[
\text{ack}^i S \xi \quad (i < \omega)
\]

name the the generic \( G \)-functions on the finite levels.

What does this network of numerals look like?

Take two copies of the decent paths, one above the other. On the lower sheet we have the lifted numerals (some of which are authentic), and on the upper sheet we have the simulated numerals. On either sheet we may move down the descent paths (using \( D \) or \( E \)), and at any point we may move from the upper sheet to the lower sheet (using \( C \)). There are certain upwards connections from the atoms. These correspond to the iteration terms (either constants of compound). In some circumstances there may be other connections.

The structure of this two sheeted network seems to be an important characteristic of the strength of the parent tiered system.
We conclude this section with a use of this network of numerals.

In Theorem 8.6 we asserted that for each tiered system $\lambda T$ the class $\mathcal{S}(\lambda T)$ of functions simulated throughout $\lambda T$ is a clone. However, we had to abandon the proof for we ran into a problem with the closure under composition. The difficulty was arranging the compatibility of various input and outputs types. The network of numerals enables us to overcome that difficulty.

We will prove something more than is required for that result, and for this we need to extend the notion of imitation introduced in Definition 9.5.

Each type $\zeta$ supports two kinds of numerals

\[ \bar{m} : \zeta'' \quad \Gamma m : \zeta \]

the simulated on the left and the lifted on the right. Of course, when $\zeta$ is an atom the lifted numerals are just the authentic numerals for that atom.

To develop a common account of these two kinds of numerals we use a bit a notation which, in fact, we saw at the beginning of this section. Let

\[ \hat{\zeta} = \zeta'' \quad \text{or} \quad \hat{\zeta} = \zeta \]

with

\[ \hat{m} = \bar{m} \quad \text{or} \quad \hat{m} = \Gamma m \]

depending on the circumstances. Thus we have

\[ \hat{m} : \hat{\zeta} \]

and we never try to mix up the kind of the numeral with the wrong kind of type.

More generally, suppose $\zeta(0), \ldots, \zeta(l)$ is a list of types. For each $0 \leq i \leq l$ let

\[ \hat{\zeta}(i) = \zeta(i)'' \quad \text{or} \quad \hat{\zeta}(i) = \zeta(i) \]

where these choices are independent. For each $0 \leq i \leq l$ let

\[ \hat{m}_i = \bar{m}_i \quad \text{or} \quad \hat{m}_i = \Gamma m_i \]

to match the choice of $\hat{\zeta}(i)$.

This convention enables us to look at types

\[ \hat{\zeta}(l) \rightarrow \cdots \rightarrow \hat{\zeta}(1) \rightarrow \hat{\zeta}(0) \]

where different occurrences of $\hat{\zeta}$ may have different meanings. With a bit of care this should not cause undue confusion.

11.10 DEFINITION. Let $\lambda T$ be a tiered system. A first order function $f : \mathbb{N}(l)$ is imitated in $\lambda A$ (in the broad sense) if there is a derivation

\[ \vdash \hat{f} : \hat{\zeta}(l) \rightarrow \cdots \rightarrow \hat{\zeta}(1) \rightarrow \hat{\zeta}(0) \]

such that

\[ \hat{f}\bar{m}(l) \cdots \bar{m}(1) \boxast \bar{m}(0) \]

for each $m(0), m(1), \ldots, m(l) \in \mathbb{N}$ with $f(m(l), \ldots, m(1)) = m(0)$. ■
Clearly, representation, simulation, and imitation (in the narrow sense) are all particular cases of imitation (in the broad sense). However, there is a crucial difference with this new notion. Consider an imitation (in the broad sense) of some function \( f \) (as in Definition 11.10), and suppose the target numerals are either authentic over some atom or simulated. Thus each such numeral is normal. This ensures that the imitation provides an algorithm for evaluating \( f \). To determine

\[
f(m(l), \ldots, m(1))
\]

for \( m(1), \ldots, m(l) \in \mathbb{N} \) we form the term

\[
\hat{f} \mathcal{m}(l) \cdots \hat{m}(1)
\]

and then start to reduce. It does matter which reduction path we take. By (Normalization) every reduction path will terminate eventually, and by (Confluence) there is essentially only one possible terminating term. That term is the numeral corresponding to the required value. However, in general a lifted numeral need not be normal. This means that an imitation (in the broad sense) of a function need not provide an algorithm for evaluating that function.

11.11 LEMMA. Let \( \lambda T \) be any tiered system and suppose the function \( f : \mathbb{N}(l) \) is imitated by a term \( \hat{f} \), as in Definition 11.10. Let \( \eta \) be any type with

\[
\eta \downarrow \zeta(i)
\]

for each \( 1 \leq i \leq l \), and let \( \zeta = \zeta(0) \). Then there is a term

\[
\vdash \hat{f} : \eta'' \to \cdots \to \eta'' \to \zeta(0)
\]

which imitates \( f \).

Proof. By Theorem 11.7 we have a numeral converter

\[
\mathcal{L}_i : \eta'' \to \hat{\zeta}(i)
\]

for each \( 1 \leq i \leq l \). A simple argument shows that

\[
\hat{f} = \lambda u_l, \ldots, u_1 : \eta'' \cdot \hat{f}(\mathcal{L}_l u_l) \cdots (\mathcal{L}_1 u_1)
\]

has the required type and imitates \( f \).

Of course, for types \( \zeta(1), \ldots, \zeta(l) \) there always is at least one type \( \eta \) with

\[
\eta \downarrow \zeta(i)
\]

for each \( 1 \leq i \leq l \). For instance, we can take any product of \( \zeta(1), \ldots, \zeta(l) \).

As indicated above, lifted numerals and imitations (in the broad sense) are merely useful technical devises. For instance, they help us out of an earlier difficulty.

In Theorem 8.6 we asserted that for each tiered system the class \( \mathcal{G}(\lambda T) \) is a clone, but got into a some difficulty with the proof. We can now get round that problem.
11.12 Lemma. For each tiered system $\lambda T$ the class $\mathcal{S}(\lambda T)$ is closed under composition.

Proof. Let us re-use the abandoned proof of Theorem 8.6. Thus, consider a composite

\[ f = h \circ (g_l, \ldots, g_1) \]

where

\[ g_1, \ldots, g_l, h \in \mathcal{S}(\lambda) \]

and suppose each $g_i$ is $k$-placed.

Consider any type $\zeta$. Then, by definition of $\mathcal{S}(\lambda T)$ there are terms

\[ \vdash \overline{h} : \zeta(l)'' \rightarrow \cdots \rightarrow \zeta(1)'' \rightarrow \zeta'' \]
\[ \vdash \overline{g_l} : \eta_l(k)'' \rightarrow \cdots \rightarrow \eta_l(1)'' \rightarrow \zeta(l)'' \]
\[ \vdash \overline{g_i} : \eta_i(k)'' \rightarrow \cdots \rightarrow \eta_i(1)'' \rightarrow \zeta(i)'' \]
\[ \vdash \overline{g_1} : \eta_1(k)'' \rightarrow \cdots \rightarrow \eta_1(1)'' \rightarrow \zeta(1)'' \]

which simulate the corresponding function. Consider any type $\eta$ with

\[ \eta \downarrow \eta_i(j) \]

for each $1 \leq i \leq l$ and $1 \leq j \leq k$. By Lemma 11.9 we may modify the terms $\overline{g_i}$ to achieve

\[ \eta_l(j) = \eta \]

for each $1 \leq i \leq l$ and $1 \leq j \leq k$. Using these modified terms let

\[ \overline{f} = \lambda u_k, \ldots, u_1 : \eta. \overline{h} (\overline{g_l} u) \cdots (\overline{g_1} u) \]

where $u$ abbreviates $u_l, \ldots, u_1$. An easy exercise shows that

\[ \vdash \overline{f} : \eta'' \rightarrow \cdots \eta'' \rightarrow \zeta'' \]

and that $\overline{f}$ simulates $f$. \qed

In Section 14 we will see other uses of the network of numerals.

[Hand in 165.../05-version.../012.. Last changed April 13, 2006]

12 A simple tiered system

These notes are suppose to be about tiered systems $\lambda T$, systems with a multicoloured array of natural number gadgets. Certainly, we have obtained a few results about arbitrary tiered systems $\lambda T$, such as

\[ \mathfrak{R}(\lambda T) = \mathfrak{R}(\lambda T) \quad \mathfrak{L}(\lambda T) \subseteq \mathcal{S}(\lambda T) \]

96
where the left hand class need not be closed under composition, but the other two are, and hence are clones.

So far, the only particular system we have analysed in any depth is the monochrome system $\lambda A$ which has only the minimal natural number facilities. From Theorems 7.5 and 10.12 we have

$$R(\lambda A) = \mathcal{T} \quad U(\lambda A) = \mathcal{E}$$

the classes of translations and extended polynomials, respectively. In particular, the class $R(\lambda A)$ is a clone. The proofs of these equalities give us quite a bit more information about the arities of the function involved. We also know that $U(\lambda A) \subseteq \mathcal{S}(\lambda A)$ since the predecessor function belongs to the larger class but not the smaller. However, as yet we don’t really know much about $\mathcal{S}(\lambda A)$.

We have analysed $\lambda A$ using the notions of representation, simulation, and imitation. In fact, this analysis uses a disguised form of tiering within $\lambda A$. In this section we make those aspects more explicit by working in a genuinely tiered system (even if this system is still rather short of natural number gadgets).

When working in $\lambda A$ we use two kinds of numerals, the authentic and the simulated. The lifted numerals are merely stepping stones to pass between these more important kinds. Much of what happens in $\lambda A$ exploits the interaction between the authentic and the simulated. This is at the heart of what is going on with tiering.

Let’s try to put the two kinds of numerals on a more equal footing, and design a tiered system $\lambda M$ which captures the same classes of functions solely by representation.

As always, we keep the monochrome atom $N$ with its furnishing. These give us the the authentic numerals. We add another (sepia coloured) atom $S$ to house the analogues of the simulated numerals. Of course $S$ is furnished with a zero and a successor constant, so now we will have to use subscripts to distinguish $N$-gadgets from $S$-gadgets. Thus we have two kinds of numerals

$$\gamma_{mS} = S^m_0S \quad \gamma_{nN} = S^n_N0N$$

which inhabit $S$ and $N$, respectively. We may also need some other gadgets in the form of iterators.

As an illustration consider how a 2-placed function could be named in this system. We need a term

$$\vdash \gamma f : I \to J \to K$$

where each of $I, J, K$ is one of the two atoms. This gives eight possibilities which we split into two kinds

$$\vdash \gamma f : I \to J \to S \quad \vdash \gamma f : I \to J \to N$$

according to the output type. In this way we see that $\lambda M$ can be used to name two kinds of functions with arity, those from names with target $S$ and those from names with target $N$. These two classes need not be the same. We wish to design $\lambda M$ so that the names with target $N$ give us precisely the meek functions and at this stage we don’t care too much what the target $S$ names give. (If this example seems familiar, it is because it was mentioned briefly towards the beginning of Section 6.)

12.1 DEFINITION. Consider a tiered system $\lambda M$ with just two atoms; the monochrome atom $N$, and a coloured atom $S$. For $s, a \in \mathbb{N}$ let

$$R(\lambda M \mid s; a \mid S) \quad R(\lambda M \mid s; a \mid N)$$
be the classes of functions of arity \((s; a)\) which are represented in the system by terms
\[ \vdash \overline{f} : S \rightarrow \cdots \rightarrow S \rightarrow N \rightarrow \cdots \rightarrow N \rightarrow T \]
where there are \(s\) input types \(S\) and \(a\) input types \(N\), and where
\[ T = S \quad T = N \]
for the two cases. □

This notation \(R(\lambda M \mid s; a \mid T)\) gives us quite detailed information, the name of the system, the arity of the function, and the target type. Our problem is to design a system \(\lambda M\) with
\[ R(\lambda M \mid s; a \mid N) = R(s; a) \]
and for the time being we don’t care what \(R(\lambda M \mid s; a \mid S)\) is.

What extra gadgetry does \(\lambda M\) need? Consider a meek function \(f\) of arity \((s; a)\). How might we find a name for \(f\)? We still have all the facilities of \(\lambda A\) available, so by Theorem 10.9 we can produce a term
\[ \vdash \hat{f} : N'' \rightarrow \cdots \rightarrow N'' \rightarrow N \rightarrow \cdots N \rightarrow N \]
which imitate \(f\). (Here there are \(s\) input types \(N''\) and \(a\) input types \(N\).) Suppose we can find a term
\[ J : S \rightarrow N'' \]
where
\[ J\overline{m_S} \not\Rightarrow m_N \]
for each \(m \in N\). In other words, \(J\) is a numeral converter. With \(J\) we may set
\[ \overline{f} = \lambda y_s, \ldots, y_1 : S. \hat{f}(Jy_s) \cdots (Jy_1) \]
to produce a name for \(f\) of the required form. (You might like to spend a few minutes worrying about where the authentic inputs have gone.)

Where can we find a term \(J\)? It looks very like an iterator doesn’t it? Let’s try adding the iterator
\[ I : S \rightarrow N'' \]
with its standard reduction properties. Given this we may set
\[ J = \lambda u : S, y : N', x : N. I y x \]
to obtain a term of the right type. Also
\[ J\overline{m_S} \not\Rightarrow \lambda y : N', x : N. J\overline{m_S} y x \not\Rightarrow \lambda y : N', x : N. y^m x = m_N \]
which is the required property.

So what system \(\lambda M\) have we produce so far? There are two types \(N, S\) (with furnish-ings) and just one iterator \(I : S \rightarrow N''\). In words what we have is nothing more than the system \(\lambda B\) of Section 3 (where there we had \(N[\cdot]\) rather than \(S\)). Let’s begin to develop a more detailed analysis of this system, but let’s continue to write ‘\(S\)’ in place of ‘\(N[\cdot]\)’.

From the observation above we have the following.

98
We consider derivations in $\lambda B$

$$(\nabla) \quad \Gamma \vdash t : \tau$$

where the root judgement meets the following restrictions.

$(\Gamma)$ The context $\Gamma$ is a list of declarations $y : S$ and $x : N$.

$(t)$ The subject term $t$ is normal.

$(\tau)$ The predicate type $\tau$ is one of

(function) A function type

$I \to \cdots \to I \to N$

where each input type $I$ is either $S$ or $N$, and these choices are independent

(product) A product type $\sigma \times \rho$

for some $\sigma, \rho$.

Table 9: Global conditions for Theorem 12.3

12.2 THEOREM. For each arity $(s; a)$ we have $M(s; a) \subseteq R(\lambda B \mid s; a \mid N)$. That is, each meek function $f$ of arity $(s; a)$ can be named in $\lambda B$ by a term

$$\Gamma \vdash \{f\} : S \to \cdots \to S \to N \to \cdots N \to N$$

where there are $s$ input types $S$ and $a$ input types $N$.

We have indicated a proof of this going via Theorem 10.9. However, it is instructive to go through a direct proof using the furnishings of $\lambda B$. You will find that it is not quite a straightforward as you first think, and learn something as well.

If our system $\lambda B$ is the right one, then there will be a converse to this result. In other words it should be the case that only meek functions can be named in $\lambda B$ in the manner indicated in Theorem 12.2. How can we prove this? There are two possibilities. The first one is to re-work the exhaustive search method to deal with this particular case. We will look at the details of this in a moment. A second possibility is to try, somehow, to use Scholium 10.11 to produce the required result. Perhaps we can somehow reverse the idea above, and show how certain properties of $\lambda B$ can be ‘interpreted’ in $\lambda A$. We show how to do this in Section 14 where we also deal with several other cases.

To obtain the converse of Theorem 12.2 we modify the proof of Theorem 10.10. We will not go through this proof in detail, but merely concentrate on those parts that are not routine.

As always we need to search through a larger class of derivations, this time as set out in Table 9.

12.3 THEOREM. For each derivation of $\lambda B$

$$(\nabla) \quad \Gamma \vdash t : \tau$$

where the root judgement meets the conditions of Table 9, one of the following applies
If $\tau$ is a function type then $\nabla$ provides a meek function of arity determined by the context $\Gamma$ and the predicate type $\tau$.

If $\tau$ is a product type the $t$ is $\text{Pair} lr$ for some terms $l, r$.

depending on the shape of $\tau$.

**Proof.** This proof follows the same lines as the proof of Theorem 10.10. We give only the details where these are significant differences from that proof.

As usual we proceed by induction over the height of $\nabla$ (and extract an algorithm by recursion over this height). We survey all the possible ways that $\nabla$ can arise.

Firstly, $\nabla$ may be a leaf.

(*$\nabla$ Axiom*) Since we are working in $\lambda B$ there are six possible shapes for $t : \tau$ (some of which come in two colours).

1. $0_S : S$ $0_N : N$ (iv) $\text{Pair} : \sigma \rightarrow \rho \rightarrow \sigma \times \rho$
2. $S_S : S'$ $S_N : N'$ (v) $\text{Left} : \sigma \times \rho \rightarrow \sigma$
3. $1 : S \rightarrow N''$ (vi) $\text{Right} : \sigma \times \rho \rightarrow \rho$

Here $\sigma, \rho$ are arbitrary. Of these only the $N$ subcases of (i, ii) meet the global conditions, and these provide rather simple meek functions. All the others, including (iii), do not meet the global conditions.

(*Projection*) This provides a projection function viewed as a meek function.

Secondly, $\nabla$ may arise by a use of one of the construction rules. Passing across a use of Weakening or Introduction is routine, so there is just one case to consider.

(*$\nabla$ Elimination*) As usual we track through the construction of $\nabla$ to unravel as many uses of Elimination and Weakening as possible. Thus we obtain a family of shorter derivations

$$(Q) \ \Xi \vdash q : \pi_k \rightarrow \cdots \rightarrow \pi_1 \rightarrow \tau \quad (P_i) \ \Pi_i \vdash p_i : \pi_i \quad i \leq i \leq k$$

where

$$t = qp_k \cdots p_1$$

with $q$ and each $p_i$ normal, and where each context $\Xi$ and $\Pi_i$ is a left hand part of $\Gamma$. By reinstating the uses of Weakening we obtain derivations

$$(Q^+) \ \Gamma \vdash q : \pi_k \rightarrow \cdots \rightarrow \pi_1 \rightarrow \tau \quad (P_i^+) \ \Gamma \vdash p_i : \pi_i \quad i \leq i \leq k$$

all in the original context, and all of which are strictly shorter than $\nabla$. We let

$$\chi = \pi_k \rightarrow \cdots \rightarrow \pi_1 \rightarrow \tau$$

and consider how $Q$ can arise.

(*$Q$ Axiom*) There are six possible as listed above. Of these most either can not arise or lead to simple meek functions. Only the case (iii) is not routine. For this case we have

$$(Q^+) \ \Gamma \vdash 1 : S \rightarrow N'' \rightarrow N \rightarrow N$$

together with at least the first two and possibly all three of

$$(E^+) \ \Gamma \vdash \gamma^+ : S \quad (H^+) \ \Gamma \vdash \eta^+ : N'' \quad (G^+) \ \Gamma \vdash \zeta^+ : N$$

100
with
\[
\begin{align*}
t = \Gamma e \tau h \tau & \quad t = \Gamma e \tau h \tau g \tau
\end{align*}
\]
depending on whether or not \( G^+ \) occurs. Here the component terms \( e \tau, h \tau, g \tau \) are normal (since \( q \) is normal).

Whatever these terms are they provide functions
\[
e(y; x) \quad h(y; u, x) \quad g(y; x)
\]
where the inputs \( y \) arise from declarations \( y : S \) in \( \Gamma \), the inputs \( x \) arise from declarations \( x : N \) in \( \Gamma \), and the extra input \( u \) of \( h \) arises from the source type in the predicate \( N' \).

Now, both \( H^+ \) and \( G^+ \) are shorter derivations and fall into the global conditions. Hence both \( h \) and \( g \) are meek functions of indicated arity. In particular, by Lemmas 10.7 and 10.8, the function
\[
l(v, y; x, u) = h(y; x, \cdot)^\nu u \quad l(v, y; x) = h(y; x, \cdot)^\nu g(y; x)
\]
(whichever occurs) is meek. The function \( f \) provided by \( \nabla \) is given by substituting \( e(y, x) \) for \( v \) in \( l \). Why should this function be meek?

At this stage we realize that we should have first obtained a characterization of what the functions \( e \) can be. In fact it is one of
\[
e(y, x) = y + c \quad e(y, x) = c
\]
where \( y \) is a selected input from \( y \) and \( c \in N \). Thus \( e \) is a meek translation of a restricted kind. Using this we see that \( f \) is given by one of
\[
\begin{align*}
f(y, x, u) &= h(y; x, \cdot)^{n+y} u \quad f(y, x) = h(y; x, \cdot)^{n+y} g(y, x) \\
f(y, x, u) &= h(y; x, \cdot)^{n} u \quad f(y, x) = h(y; x, \cdot)^{n} g(y, x)
\end{align*}
\]
depending on which case arise. Each one of these is a meek function.

All the other cases are routine.

This proof is a nice example of what can happen with the exhaustive search technique. At the beginning it looks as though we want to prove something quite simple, and we attempt to do that by induction over the relevant derivations. To do that we need a more intricate induction hypothesis and so have to prove something more complicated. For instance, in all the examples we have seen there has been a (product) clause, even though we are not interested in products. This is because the unravelling of the derivation forces us to consider how the product gadgets become involved.

The trick is to choose an induction hypothesis that is just strong enough, and doing that is not always simple. Sometimes one aspect has to be taken into account even though it hardly occurs in the derivation. Such things are easily overlooked. This has happened at the last step in the previous proof. Before that result we should have proved the following.

12.4 THEOREM. For each derivation of \( \lambda B \)
\[
(\nabla) \quad \Gamma \vdash t : \tau
\]
where the root judgement meets the conditions of Table 10, one of the following applies.
We consider derivations in $\lambda B$

$$(\nabla) \quad \Gamma \vdash t : \tau$$

where the root judgement meets the following restrictions.

(Γ) The context $\Gamma$ is a list of declarations $y : S$ and $x : N$.

(t) The subject term $t$ is normal.

(τ) The predicate type $\tau$ is one of

(function) A function type

$I \to \cdots \to I \to S$

where each input type $I$ is either $S$ or $N$, and these choices are independent

(product) A product type $\sigma \times \rho$

for some $\sigma, \rho$.

Table 10: Global conditions for Theorem 12.4

(function) If $\tau$ is a function type then $\nabla$ provides a translation $f$ of the one of the forms

$f(y;x) = y + n \quad f(y;x) = n$

where $y$ is the list of simulated inputs of type $S$, $x$ is the list of authentic inputs of type $N$, $y$ is a selected component of $y$, and $n$ is some constant.

(product) If $\tau$ is a product type the $t$ is $\text{Pair}lr$ for some terms $l, r$.

depending on the shape of $\tau$.

I won’t prove this. It is very like the proof of Theorem 7.4, and you should have very little difficulty with it now. However, notice that the authentic inputs $x$ have no influence at all on the output. This is because within the system $\lambda B$ there is no way of moving from $N$ to $S$.

Theorems 12.3 and 12.4 give us the following.

12.5 THEOREM. We have

$${\mathcal R}(\lambda B | S) = \mathfrak T \quad {\mathcal R}(\lambda B | N) = \mathfrak C$$

respectively.

Here

$${\mathcal R}(\lambda B | S) \quad {\mathcal R}(\lambda B | N)$$

are the respective unions of the classes

$${\mathcal R}(\lambda B | s : a | S) \quad {\mathcal R}(\lambda B | s : a | N)$$

as $(s;a)$ ranges over all arities.

[Held in 165.../05-Version../012.. Last changed April 13, 2006]
The translation technique

So far all our non-trivial results have been obtained using the exhaustive search method. In this section we introduce another technique, or rather we describe a simple example of another technique. It is sometimes possible to transform derivations in one system into derivations in another system. Using such transformations sometimes we can show that functions captured in one system can also be captured in another system, perhaps using a different method of capture.

In this section we describe a simple version of this idea, the translation technique. Later, in Section 14 we describe a more sophisticated version of this transformation idea.

Here we work entirely in $\lambda A$. We show how each imitation in $\lambda A$

$$(\hat{\nabla}) \vdash \hat{f} : \zeta(l)^{\prime\prime} \to \cdots \to \zeta(1)^{\prime\prime} \to \mathcal{N}$$

can be translated into a simulation in $\lambda A$

$$(\nabla) \vdash f : \eta(l)^{\prime\prime} \to \cdots \to \eta(1)^{\prime\prime} \to \eta^{\prime\prime}$$

of the same function $f$. Here the target type $\eta$ is arbitrary but the other source types $\eta(1), \ldots, \eta(l)$ are determined by the translation process and the given derivation. It will be clear that this kind of translation could be extended, but we don’t need to do that just yet.

The translation process is a bit tedious but it is worth going through it in detail. We set up translations

$$\tau \mapsto |\tau| \quad t \mapsto |t| \quad \nabla \mapsto |\nabla|$$

between types, terms, and derivations. These are mostly trivial but, of course, there are certain properties that have to be checked.

As mentioned, we work entirely in $\lambda A$. We select an arbitrary type $\eta$, and this determines everything that we do. We refer to $\eta$ as the control type of the translation.

13.1 DEFINITION. Let $\eta$ be an arbitrary type of $\lambda A$. The type translation

$$\tau \mapsto |\tau|$$

controlled by $\eta$, is generated by recursion over the structure of the type $\tau$ using

$$|\mathcal{N}| = \eta^{\prime\prime} \quad |\sigma \to \rho| = |\sigma| \to |\rho| \quad |\sigma \times \rho| = |\sigma| \times |\rho|$$

as the recursion clauses.

This idea is quite simple. Given a type $\tau$ locate all occurrences of the atom $\mathcal{N}$ in $\tau$. Replace each such occurrence by $\eta^{\prime\prime}$ to obtain the translated type $|\tau|$. In other words, we think of $\mathcal{N}$ as a variable and substitute $\eta^{\prime\prime}$ for this. For those who know about these things there is something polymorphic going on here, and perhaps the whole process can be describe within the second order $\lambda$-calculus.

Next we set up a translation of terms. To do this we remember that the type $\eta$ carries simulated furnishings

$$\overline{\eta} : \eta^{\prime\prime} \quad \underline{\eta} : \eta^{\prime\prime\prime}$$

which capture zero and successor, respectively.
13.2 DEFINITION. Let $\eta$ be an arbitrary type of $\lambda\mathcal{A}$. The term translation

$$t \mapsto |t|$$

controlled by $\eta$ is generated by recursion over the structure of the term $t$ using the following recursion steps.

For the constants we set

$$|0| = 0_\eta \quad |S| = S_\eta$$

with

$$|\text{Pair}_{\sigma,\rho}| = \text{Pair}_{|\sigma|,|\rho|} \quad |\text{Left}_{\sigma,\rho}| = \text{Left}_{|\sigma|,|\rho|} \quad |\text{Right}_{\sigma,\rho}| = \text{Right}_{|\sigma|,|\rho|}$$

using the translation of the indexing types.

We then pass this through all terms by

$$|x| = x \quad |k| = \text{as above} \quad |qp| = |q||p| \quad |\lambda x : \sigma . r| = \lambda x : |\sigma| . |r|$$

as the recursion clauses. Here $x$ is an identifier and $k$ is a constant. ■

This translation is a bit harder to describe succinctly. However, note that the resulting term $|t|$ contains no occurrences of the natural number gadgets $0$ and $S$. In fact, each authentic numeral has been translated into a simulated numeral over the control type $\eta$.

To see why we use this translation we need to press on a little bit further.

13.3 LEMMA. For each axiom $k : \kappa$ of $\lambda\mathcal{A}$ we have $\lambda\mathcal{A}$-derivation

$$(\nabla(k)) \vdash |k| : |\kappa|$$

in the empty context.

Proof. In more detail for the two axioms

$$0 : \mathcal{N} \quad S : \mathcal{N}$$

there are derivations

$$(\nabla(0)) \vdash 0_\eta : \eta'' \quad (\nabla(S)) \vdash S_\eta : \eta'' \to \eta''$$

constructed in the obvious fashion.

For the axioms

$$\text{Pair}_{\sigma,\rho} : \sigma \to \rho \to \sigma \times \rho \quad \text{Left}_{\sigma,\rho} : \sigma \times \rho \to \sigma \quad \text{Right}_{\sigma,\rho} : \sigma \times \rho \to \rho$$

the required derivations are even easier, since

$$|\text{Pair}_{|\sigma|,|\rho|}| : |\sigma| \to |\rho| \to |\sigma| \times |\rho| \quad |\text{Left}_{|\sigma|,|\rho|}| : |\sigma| \times |\rho| \to |\sigma| \quad |\text{Right}_{|\sigma|,|\rho|}| : |\sigma| \times |\rho| \to |\rho|$$

are also axioms. ■

Next we show how to translate derivations. We translate statements, declarations, contexts, and judgements in the obvious way.

$$|t : \tau| = |t| : |\tau| \quad |x : \sigma| = x : |\sigma| \quad \ldots, x : |\sigma|, \ldots = \ldots, x : |\sigma|, \ldots$$

$$|\Gamma \vdash t : \tau| = |\Gamma| \vdash |t| : |\tau|$$

Because we have changed very little and preserved almost all of the syntactic structure, the translation respects the typing discipline.
13.4 LEMMA. Each $\text{LA}$-derivation

\[ (\nabla) \quad \Gamma \vdash t : \tau \]

translates into a $\text{LA}$-derivation

\[ (|\nabla|) \quad |\Gamma| \vdash |t| : |\tau| \]

using the translated types and terms.

Proof. We allow $|\cdot|$ to pass across the construction rules for derivations in the obvious way. Lemma 13.3 deals with the axioms. The projections and steps across the rules are immediate. ■

Consider an imitation

\[ \vdash \widetilde{f} : \zeta(l)^{\prime\prime} \to \cdots \to \zeta(1)^{\prime\prime} \to N \]

of some function $f$. When translated this derivation produces a derivation

\[ \vdash \widetilde{f} : \eta(l)^{\prime\prime} \to \cdots \to \eta(1)^{\prime\prime} \to \eta'' \]

where $\widetilde{f} = |\widetilde{f}|$ and

\[ \eta(1) = |\zeta(1)|, \ldots, \eta(l) = |\zeta(l)| \]

are the types involved. Of course, $\eta$ is the type chosen to control the whole translation. This term $\widetilde{f}$ simulates a function. But is this the same as the function imitated by $\widetilde{f}$? We can not answer that by looking at the semantics, because this translation does not preserve meaning. We need to see how the translation interacts with the computation mechanism.

13.5 LEMMA. For each instance

\[ t^- \triangleright t^+ \]

of a 1-step reduction in $\text{LA}$, there is a 1-step reduction

\[ |t^-| \triangleright |t^+| \]

in $\text{LA}$.

Proof. We go through the 1-step reductions in turn, as listed in Definition 4.3.

The pairing reductions are immediate (since each pairing gadget translates into a pairing gadget of the same kind).

There are no iterator reductions to worry about (since there are no iterators in $\text{LA}$).

Finally consider a $\lambda$-redex removal

\[ (\lambda x : \sigma. r)s \triangleright r[x := s] \]

in $\text{LA}$. We have

\[ |(\lambda x : \sigma. r)s| = (\lambda x : |\sigma|. |r|)s \]

so it suffices to show that

\[ |r[x := s]| = |r|[x := |s|] \]

holds. This follows by a tedious induction over the structure of $r$. ■

The obvious induction now gives the following.
13.6 COROLLARY. For each instance
\[ t^- \rightarrow t^+ \]
of a reduction in \( \lambda \alpha \), there is a reduction
\[ |t^-| \rightarrow |t^+| \]
in \( \lambda \alpha \).

This is not quite enough for what we need. We must see what happens to numerals as they pass through the translation.

13.7 LEMMA. For each \( m \in \mathbb{N} \) and type \( \zeta \) we have
\[ |m\zeta| = \frac{m}{|\zeta|} \]
where \( \eta \) is the controlling type.

Proof. For each simulated numeral we have
\[ m\zeta = \lambda y : \zeta, x : \zeta \cdot y^m x \]
so that the claimed translation is immediate.

For an authentic numeral we have
\[ |\frac{m}{\zeta}| = S_{|\eta|/\eta}^{m\eta} \frac{m}{\eta} \]
where the reduction follows by a simple induction over \( m \).

At last we can prove the result we want.

13.8 THEOREM. Consider the translation controlled by an arbitrary type \( \eta \). Suppose
\[ \vdash \hat{f} : \zeta(l)^n \rightarrow \cdots \rightarrow \zeta(1)^n \rightarrow \mathcal{N} \]
is an imitation of a certain function \( f \), and let
\[ \vdash \overline{f} : \eta(l)^n \rightarrow \cdots \rightarrow \eta(1)^n \rightarrow \eta'' \]
be the translated derivation with \( \overline{f} = |\hat{f}| \). Then \( \overline{f} \) simulates \( f \).

Proof. Consider \( m(1), \ldots, m(l) \in \mathbb{N} \). We have
\[ \hat{f} m(l)_{\zeta(l)} \cdots m(1)_{\zeta(1)} \rightarrow |m(0)| \]
where \( m(0) = f(m(1), \ldots, m(l)) \). This is one occasion when it is sensible to display all the indexing types.

By Corollary 13.6 we have
\[ |\hat{f} m(l)_{\zeta(l)} \cdots m(1)_{\zeta(1)}| \rightarrow |m(0)| \rightarrow m(0)_{\eta} \]
where Lemma 13.7 gives the second reduction. That lemma also gives
\[ \overline{f} m(l)_{\eta(l)} \cdots m(1)_{\eta(1)} = |\hat{f} m(l)_{\zeta(l)} \cdots m(1)_{\zeta(1)}| \]
for the required result.

We give just one application of this result. In fact, Theorem 13.8 has been designed with this application in mind (which is why the statement isn’t as strong as it could be).
13.9 THEOREM. For each function $f \in \mathbb{N}(l)$ the following conditions

(i) $f$ is simulated throughout $\lambda A$, that is $f \in \mathcal{S}(\lambda A)$.

(ii) $f$ can be simulated by a term

$$\vdash \overline{f} : \zeta(l)'' \rightarrow \cdots \rightarrow \zeta(1)'' \rightarrow \mathcal{N}''$$

for some types $\zeta(1), \ldots, \zeta(l)$.

(iii) $f$ can be imitated by a term

$$\vdash \widehat{f} : \zeta(l)'' \rightarrow \cdots \rightarrow \zeta(1)'' \rightarrow \mathcal{N}$$

for some types $\zeta(1), \ldots, \zeta(l)$.

are equivalent.

Proof. (i)$\Rightarrow$(ii). If $f \in \mathcal{S}(\lambda A)$ then $f$ can be simulated throughout $\lambda A$, and so can be simulated in the required form by choosing $\mathcal{N}''$ as the target type.

(ii)$\Rightarrow$(iii). Suppose $f$ can be simulated by a term $\overline{f}$, as indicated. Using the numeral converter $C : \mathcal{N}'' \rightarrow \mathcal{N}$ we obtain a term $\widehat{f} = C \circ \overline{f}$ which imitates $f$.

(iii)$\Rightarrow$(i). This is an immediate consequence of Theorem 13.8. ■

We will develop this idea further in the next section.

14 The implementation technique

In Section 13 we described a method of transforming one kind of imitation of a function in $\lambda A$ into another kind of imitation of the same function in $\lambda A$. The trick that drives that transformation is to replace the use of authentic numerals by simulated numerals of an appropriate kind.

In this section we describe a slightly more sophisticated kind of transformation. We start from a tiered system $\lambda T$ which must satisfy an appropriate restriction (as given later). We show how each representation of a function in $\lambda T$ can be transformed into an imitation of the same function in $\lambda A$. In this transformation each authentic monochrome numeral is left intact, but each authentic coloured numeral is implemented by a simulated numeral. This shows that each such restricted system $\lambda T$ is comparatively weak, since each function that can be named in $\lambda T$ can be captured in $\lambda A$. However, the colour coding of $\lambda T$ may be used to stratify $\mathcal{R}(\lambda T)$ into layers of complexity. We illustrate that aspect in the next section.

One difference between the translation technique of Section 13 and the implementation technique described here is that for a translation there is some choice as indicated by the controlling type, whereas here the implementation is completely determined by the system $\lambda T$.

Each type $\tau$ and each term $t$ of $\lambda T$ is implemented in $\lambda A$

$$\tau \mapsto |\tau| \quad t \mapsto |t|$$
in a fairly routine manner.

\[ |\mathcal{N}| = \mathcal{N} \quad |\xi| = \text{as selected} \quad |\sigma \rightarrow \rho| = |\sigma| \rightarrow |\rho| \quad |\sigma \times \rho| = |\sigma| \times |\rho| \]

\[ |x| = x \quad |k| = \text{as selected} \quad |qp| = |q| |p| \quad |\lambda x : \sigma \cdot r| = \lambda x : |\sigma| \cdot |r| \]

Here \( \xi \) is a coloured atom of \( \lambda T \) and the selected type \( |\xi| \) will be described shortly. For the time being, let’s assume this has been done. Similarly, \( k \) is a constant of \( \lambda T \) and the selected term \( |k| \) has to be chosen with a modicum of care.

Consider an axiom

\[ k : \kappa \]

of \( \lambda T \). This gives a statement

\[ |k| : |\kappa| \]

of \( \lambda A \), but unless we are careful this could be nonsense. Whatever we do we must be able to prove the following analogue of Lemma 13.3.

**14.1 LEMMA.** For each axiom \( k : \kappa \) of \( \lambda T \) there is a \( \lambda A \)-derivation

\[(\nabla (k)) \vdash |k| : |\kappa| \]

in the empty context.

Of course, we can’t prove this just yet. We will prove it bit by bit as we describe how each constant is implemented. We use the result as a guide to what should be done.

Let’s look at the constants in the order listed in Definition 2.7.

Consider three pairing gadgets

\[ \text{Pair}_{\sigma,\rho} : \sigma \rightarrow \rho \rightarrow \sigma \times \rho \quad \text{Left}_{\sigma,\rho} : \sigma \times \rho \rightarrow \sigma \quad \text{Right}_{\sigma,\rho} : \sigma \times \rho \rightarrow \rho \]

de \( \lambda T \). The implementation of the housing types are

\[ |\sigma| \rightarrow |\rho| \rightarrow |\sigma| \times |\rho| \quad |\sigma| \times |\rho| \rightarrow |\sigma| \quad |\sigma| \times |\rho| \rightarrow |\rho| \]

which allows the following definition.

**14.2 DEFINITION.** Each pairing gadget is implemented by the corresponding pairing gadget.

\[ |\text{Pair}_{\sigma,\rho}| = \text{Pair}_{|\sigma|,|\rho|} \quad |\text{Left}_{\sigma,\rho}| = \text{Left}_{|\sigma|,|\rho|} \quad |\text{Right}_{\sigma,\rho}| = \text{Right}_{|\sigma|,|\rho|} \]

Each required derivations \( \nabla (-) \) is an Axioms.

The monochrome atom and numeric gadgets are even easier. Since these must be left intact we have the following.

**14.3 DEFINITION.** The monochrome furnishings

\[ 0 : \mathcal{N} \quad S : \mathcal{N}’ \]

are implemented as themselves. In particular, the required derivations \( \nabla (0) \) and \( \nabla (S) \) are Axioms.
The coloured numeric gadgets are different. The coloured numerals are implemented by simulated numerals. To do that we must first arrange the following

(ξ) For each coloured atom ξ we have |ξ| = ζ" for some type ζ of λA.

Just how we arrange this will be describe shortly when we deal with the iterators. This condition allows the following.

14.4 DEFINITION. Consider the furnishings

0ξ : ξ  Sξ : ξ'

of a coloured atom ξ. We have |ξ| = ζ" for some λA-type ζ. We set

|0ξ| = 0ζ  |Sξ| = Sζ

using the simulated furnishings of ζ". The required derivations

(∇(0ξ)) ⊢ 0ζ : ζ"  (∇(Sξ)) ⊢ Sζ : ζ"'

to show that [m] is implemented as itself. Coloured numerals are different. The implemented version need not be a numeral.

14.5 EXAMPLE. Let ξ be a coloured atom with |ξ| = ζ". Then for each m ∈ N we have

[mξ] = S"m0ξ

we have

|mξ| = |S"m0| = |S|m|0| = S"m0 = mξ

to show that [mξ] is implemented as itself. Coloured numerals are different. The implemented version need not be a numeral.

14.6 RESTRICTION. (First try) For each coloured atom ξ of λT there is a unique type τ and an iterator

I : ξ → τ"

on ξ. Each iterator arises in this way (so there are no iterators on N). Furthermore, it must be possible to construct τ without reference to ξ, either explicitly or implicitly. ■

109
The last part of this restriction is not exactly crystal clear is it? What is it trying to say? I must point out that the statement of this restriction in [6] as item 8.2 is incorrect. Thus I will now spend a little time explaining exactly what Restriction 14.6 is trying to say.

Perhaps the best way of explaining this restriction is to describe in full the algorithm for implementing types.

14.7 DEFINITION. For each type $\tau$ of $\lambda T$, the implementation $|\tau|$ is generated by following rules.

- For compound types we set
  
  $$ |\sigma \rightarrow \rho| = |\sigma| \rightarrow |\rho| \quad |\sigma \times \rho| = |\sigma| \times |\rho| $$

  for all types $\rho, \sigma$.

- For the monochrome atom $\mathcal{N}$ we set $|N| = \mathcal{N}$.

- For each coloured atom $\xi$ first locate the unique type $\tau$ with an iterator $I: \xi \rightarrow \tau''$ and then set $|\xi| = |\tau|''$.

There are no other rules.

14.8 EXAMPLE. Consider a system with just three atom $\mathcal{N}, \xi, \zeta$ where $\xi$ and $\zeta$ are coloured. Suppose the system has just two iterators

$$ I: \xi \rightarrow \xi'' \quad J: \zeta \rightarrow \zeta'' $$

on $\xi$ and $\zeta$, respectively. This system certainly satisfies the first part of Restriction 14.6.

How do we calculate $|\xi|$? We first calculate $|\zeta|$ and then set $|\xi| = |\zeta|''$. But $\zeta$ is a coloured atom, so to calculate $|\zeta|$ we first calculate $|\xi|$ and then set $|\zeta| = |\xi|''$. This, of course, leads to a circularity.

The intention is that this kind of circularity is prevented.

14.9 RESTRICTION. (Second try) For each coloured atom $\xi$ of $\lambda T$ there is a unique type $\tau$ and an iterator

$$ I: \xi \rightarrow \tau'' $$

on $\xi$. Each iterator arises in this way (so there are no iterators on $\mathcal{N}$). Furthermore, it must be possible to calculate

$$ \tau \longrightarrow |\tau| $$

using the algorithm of Definition 14.7.
Notice that under this restriction the condition \((\xi)\) on a coloured atom is met, and so we may use the implementation of Definition 14.4.

Almost trivially, the two systems \(\lambda B, \lambda C\) meet this restriction. With a little more work we see that \(\lambda L\) also meets this restriction. The systems \(\lambda D, \lambda E\) do not since, in these systems, for each coloured atom \(\xi\) there are iterators

\[ l_{\xi,\tau} : \xi \to \tau'' \]

for several types \(\tau\). The system \(\lambda F\) fails this test for the restriction insists that only coloured atoms support iterators. The system \(\lambda G\) gets nowhere near meeting the restriction.

With this restriction we can implement each iterator.

14.10 DEFINITION. Consider an iterator \(I\). By Restriction 14.9 this is attached to a unique coloured atom

\[ I : \xi \to \tau'' \]

and a unique target type \(\tau\). Also by Restriction 14.9 we can produce a type \(\zeta = |\tau|\) of \(\lambda A\). We set \(|\xi| = \zeta''\) and take

\[ |I| = \lambda w : \zeta'', y : \zeta', x : \zeta. wyx \]

as the implementation of \(I\). The required derivation

\[ \nabla(I) \vdash |I| : \zeta''' \]

is obvious. 

Each iterator \(I\) is implemented as the name of an identity function. This is why the required derivation \(\nabla(I)\) is obvious.

This completes the description of the implementation, and the proof of Lemma 14.1. From now on we assume we are dealing with a system \(\lambda T\) which meets the Restriction 14.9.

We now transform statement, contexts, and judgements in the obvious way, and a use of Lemma 14.1 give the following analogue of Lemma 13.4.

14.11 LEMMA. Each \(\lambda T\)-derivation

\[ \nabla \Gamma \vdash t : \tau \]

transforms into a \(\lambda A\)-derivation

\[ \nabla(|\Gamma|) \vdash |t| : |\tau| \]

using the implemented types and terms.

Clearly, because of the way the implementation handles iterators, it does not preserve meaning. So we need to see what it does to computations.

Here is the analogue of Lemma 13.5. Notice how the presence of iterators makes the situation more complicated.
14.12 LEMMA. For each instance

\[ t^- \triangleright t^+ \]

of a 1-step reduction in \( \lambda T \), we have

\[ |t^+| \triangleright t^0 \quad |t^-| \triangleright t^0 \]

for some term \( t^0 \) of \( \lambda A \).

Proof. In fact, for most instances the transformation produces a reduction

\[ |t^-| \triangleright |t^+| \]

in \( \lambda A \), and often this is a 1-step reduction.

This is the case with the 1-step reductions concerned with pairing gadgets. For instance, consider

\[ t^- = \text{Left}_{\rho,\sigma}(\text{Pair}_{\rho,\sigma}lr) \quad t^- = l \]

in \( \lambda T \). We have

\[ |t^-| = \text{Left}_{\rho,\sigma}(\text{Pair}_{\rho,\sigma}lr) |l| |r| \quad |t^-| = |l| \]

so that

\[ |t^-| \triangleright |t^+| \]

and we may take \( t^0 = |t^-| \). The other pairing reduction works in the same way.

Next consider an abstraction removal. Thus let

\[ t^0 = (\lambda x: \sigma . r)s \quad t^- = r[x := s] \]

be \( \lambda T \) terms. By a rather tedious induction we find that

\[ |t^-| = |r[x := s]| = |r[x := |s|] \]

so that

\[ |t^-| \triangleright |t^+| \]

and again we may take \( t^0 = |t^-| \).

Next we look at the two 1-step reduction associated with an iterator \( I \). Remember this has the form \( I : \xi \rightarrow \tau'' \) where \( \xi \) is a coloured atom, with \( |\tau| = \zeta \) for some type \( \zeta \) of \( \lambda A \). The term \( |\tau| \) names the identity function on \( \zeta'' \).

For the simple case we have

\[ t^- = I0_\xi ts \quad t^+ = s \]

so that

\[ |t^-| = |\tau|0_\xi |t||s| \triangleright 0_\xi |t||s| \triangleright |s| = |t^+| \]

where the first reduction follows by the implementation of \( I \) and the second follows by the nature of \( 0_\zeta \). Again we have \( |t^-| \triangleright |t^+| \).

Finally, we deal with the case

\[ t^- = I(S_\xi r)ts \quad t^+ = t(lrst) \]

which doesn’t produce a reduction in \( \lambda A \).
Let
\[ T = \text{lrt} \quad A = \text{|r|t|s|} \]
(so that \( T \) is a \( \lambda T \)-term and \( A \) is a \( \lambda A \)-term). Since \( |I| \) is an identity term we have
\[ |T| = |I|r|t|s| \Rightarrow A \]
and hence
\[ |t^+| = |t||T| \Rightarrow |t|A \]
in \( \lambda A \). Next, remembering the shape of \( |S_\xi| \), we have
\[ |t^-| = |I|(|S_\xi||r|)|t||s| \Rightarrow |S_\xi||r||t||s| \Rightarrow |t|(|r||t||s|) = |t|A \]
again in \( \lambda A \). Thus we have
\[ |t^-| \Rightarrow |t|A \quad |t^+| \Rightarrow |t|A \]
and we may take \( t^0 = |t|A \). ■

The obvious induction now gives the following analogue of Corollary 14.6.

14.13 COROLLARY. For each instance
\[ t^- \Rightarrow t^+ \]
of a reduction in \( \lambda T \), we have
\[ |t^+| \Rightarrow t^0 \quad |t^-| \Rightarrow t^0 \]
for some term \( t^0 \) of \( \lambda A \).

Before we get to the main result of this section we need one last bit of preparation. In fact, this is mostly notation.

consider any atom \( \xi \) of \( \lambda T \) this might the monochrome atom \( \mathcal{N} \) or it might be coloured. We let
\[ \hat{\xi} = \begin{cases} \mathcal{N} & \text{if } \xi \text{ is } \mathcal{N} \\ \zeta'' & \text{if } \xi \text{ is coloured} \end{cases} \]
where \( |\xi| = \zeta'' \) in the coloured case. Thus \( |\xi| = \hat{\xi} \) in all cases. Whatever it is \( \xi \) does have associated authentic numerals
\[ |\tilde{m}_\xi| = S_\xi^{\prime\prime}0_\xi \]
in \( \lambda T \). Let \( \tilde{m}_\xi \) be the corresponding numeral in \( \lambda A \). That is let
\[ \tilde{m}_\xi = \begin{cases} |\tilde{m}| & \text{if } \xi \text{ is } \mathcal{N} \\ \tilde{m}_\xi & \text{if } \xi \text{ is coloured} \end{cases} \]
where \( |\xi| = \zeta'' \) in the coloured case. We know that \( |\tilde{m}_\xi| = \tilde{m}_\xi \) in the monochrome case, but Example 14.5 shows that the coloured case is slightly more complicated. However, we do have the following.
LEMMA. For each atom $\xi$ of $\lambda T$ we have

$$\left|^{\sim} m_\xi^\gamma \right| \not\ll \hat{m}_\xi$$

for each $m \in \mathbb{N}$.

We can now pull all this together to bound the strength of a restricted tiered system.

THEOREM. Let $\lambda T$ be a tiered system which meets Restriction 14.9. Then we have $\mathcal{R}(\lambda T) \subseteq \mathcal{G}(\lambda A)$.

Proof. Consider any first order function $f : \mathbb{N}(l)$ with $f \in \mathcal{R}(\lambda T)$.

We are given a term $\vdash \left|^{\sim} f \right| : \xi_l \rightarrow \cdots \rightarrow \xi_1 \rightarrow \xi_0$

of $\lambda T$ which names $f$. Each of $\xi_0, \xi_1, \ldots, \xi_l$ is an atom, but some may be $\mathcal{N}$ and some may be coloured. As above let

$$\hat{\xi} = |\xi|$$

for each occurring atom $\xi$. By Lemma 14.11 the implementation transforms the naming derivation in $\lambda T$ into a derivation

$$\vdash \hat{f} : \hat{\xi}_l \rightarrow \cdots \rightarrow \hat{\xi}_1 \rightarrow \hat{\xi}_0$$

in $\lambda A$ (where $\hat{f} = |^{\sim} f|$). We show this imitates the same function $f$ in $\lambda A$, and then transform this imitation into a simulation of the same function.

Consider $m_1, \ldots, m_l \in \mathbb{N}$ and let $m_0 = f(m_l, \ldots, m_1)$. Since $^{\sim} f$ represents $f$ in we have

$$^{\sim} f \left|^{\sim} m_l^\gamma \cdots m_1^\gamma \not\ll \left|^{\sim} m_0^\gamma \right|$$

where $^{\sim} m_i^\gamma : \xi_i$ for each $0 \leq i \leq m$. By Corollary 14.13 we have reductions in $\lambda A$

$$\hat{f} \left|^{\sim} m_l^\gamma \cdots m_1^\gamma \not\ll t_0 \left|^{\sim} m_0^\gamma \right| \not\ll t_0$$

for some term $t_0$. By Lemma 14.14 we have

$$\hat{f} \left|^{\sim} m_l^\gamma \cdots m_1^\gamma \not\ll \hat{m_l} \cdots \hat{m_1} \left|^{\sim} m_0^\gamma \right| \not\ll \hat{m_0}$$

and hence Confluence gives

$$\hat{m_l} \cdots \hat{m_1} \not\ll n \quad \hat{m_0} \not\ll n$$

for some mediating term $n$. But now we observe that $\hat{m_0}$ is normal (whatever kind of atom $\xi_0$ is), and hence $n = \hat{m_0}$, to give

$$\hat{m_l} \cdots \hat{m_1} \not\ll \hat{m_0}$$

to show that $\hat{f}$ imitates $f$.

Next we take a closer look at the type of $\hat{f}$. After reordering the input types we find that

$$\vdash \hat{f} : \xi''_s \rightarrow \cdots \rightarrow \xi''_1 \rightarrow \mathcal{N} \rightarrow \cdots \rightarrow \mathcal{N} \rightarrow \hat{\xi}_0$$
for an appropriate number $\zeta_1, \ldots, \zeta_s$ of types (corresponding to the number of coloured input types of $f$). The system $\lambda A$ has a numeral converter $C : N'' \rightarrow N$, and a use of this produces an imitation

$$\vdash \tilde{f} : \zeta''_s \rightarrow \cdots \rightarrow \zeta''_1 \rightarrow N'' \rightarrow \cdots \rightarrow N'' \rightarrow \tilde{\xi}_0$$

of the same function.

If $\xi_0 = N$ then $f \in \mathcal{S}(\lambda A)$ by Theorem 13.9.

If $\xi_0$ is coloured then $\tilde{\xi}_0 = \zeta''$ for some type $\zeta$, and we have to do a little more. By Theorem 11.7 there is another numeral converter $C : \zeta'' \rightarrow N$, and a use of this gives an imitation

$$\vdash \tilde{f} : \zeta''_s \rightarrow \cdots \rightarrow \zeta''_1 \rightarrow N'' \rightarrow \cdots \rightarrow N'' \rightarrow N$$

of the same function, to show that $f \in \mathcal{S}(\lambda A)$, as required. ■

Notice that we don’t get $\mathcal{R}(\lambda T) \subseteq U(\lambda T)$. However, there is quite a bit more we can get out of this idea.

[Last changed April 13, 2006]

15 A fully tiered system

In this section we eventually get round to analysing the system $\lambda C$. Before that let’s collect together some of the information we have so far. We begin with a definition.

15.1 DEFINITION. A function $f \in N(l)$ is latent if it is imitated in $\lambda A$ by a term

$$\vdash \tilde{f} : \zeta(l)'' \rightarrow \cdots \rightarrow \zeta(1)'' \rightarrow N$$

for some types $\zeta(1), \ldots, \zeta(l)$.

Let $L$ be the class of latent functions. ■

In some ways this is not a very satisfactory definition. It doesn’t really tell us what these functions are. We will pick up more information in due course. However, we now have

$$\mathcal{R}(\lambda A) = \mathcal{T} \quad U(\lambda A) = \mathcal{E} \quad \mathcal{G}(\lambda A) = L$$

where the first two follow from Theorems 7.5 and 10.12, and third follows from Theorem 13.9. We have

$$\mathcal{R}(\lambda B \mid S) = \mathcal{T} \quad \mathcal{R}(\lambda B \mid N) = \mathcal{E}$$

from Theorem 12.5. This shows that the class $\mathcal{E} = U(\lambda A)$ of functions that are uniformly simulated throughout $\lambda A$ can be captured as the class of functions represented in a suitable tiered system. What about the class $L = \mathcal{G}(\lambda A)$? Can this be captured as the class of functions represented in a suitable tiered system? In this section we show that $\lambda C$ is such a system. In fact, we do more. We show how $L$ can be stratified using the type structure of $\lambda C$.

Before we get to that we need a bit of a preamble.

We first stratify the types of $\lambda A$. 115
15.2 DEFINITION. We set

\[ L(0) = \{ \mathcal{N} \} \quad L(r + 1) = \{ \tau, \sigma \to \rho, \sigma \times \rho \mid \rho, \sigma, \tau \in L(r) \} \]

for each \( r < \omega \), to stratify the types of \( \lambda \mathcal{A} \).

By construction we have a cumulative hierarchy of types

\[ (L(\star)) \quad L(0) \subseteq L(1) \subseteq \cdots \subseteq L(r) \subseteq \cdots \quad (r < \omega) \]

which eventually contains every type of \( \lambda \mathcal{A} \). There is a natural sense in which the types in these layers become more complicated as we move up the levels. In particular, we have

\[ \mathcal{N} \in L(0), \mathcal{N}' \in L(1), \ldots, \mathcal{N}^{(r)} \in L(r), \ldots \]

and

\[ \zeta \in L(r) \implies \zeta' \in L(r + 1) \]

for each type \( \zeta \) and \( r < \omega \).

15.3 DEFINITION. For each \( r < \omega \), a function \( f : \mathbb{N}(k) \) is \( r \)-latent, that is latent of level \( r \), if \( f \) is imitated by a term

\[ \vdash \hat{f} : \zeta(1)'' \to \cdots \to \zeta(1)' \to \mathcal{N} \]

for some types \( \zeta(1), \ldots, \zeta(l) \in L(r) \).

Let \( \mathcal{L}(r) \) be the class of \( r \)-latent functions.

By construction we have

\[ (\mathcal{L}(\star)) \quad \mathcal{L}(0) \subseteq \mathcal{L}(1) \subseteq \cdots \subseteq \mathcal{L}(r) \subseteq \cdots \quad (r < \omega) \]

to match the layering \( (L(r)) \). By Theorem 14.12 we have \( \bigcup \{ \mathcal{L}(r) \mid r < \omega \} = \mathcal{L} = \mathcal{G}(\lambda \mathcal{A}) \), so that hierarchy eventually contains every latent function.

This hierarchy prompts several questions. What is happening as we progress up the hierarchy? We already know the bottom layer.

15.4 THEOREM. We have \( \mathcal{L}(0) = \mathcal{E} = \mathcal{U}(\lambda \mathcal{A}) = \mathcal{R}(\lambda \mathcal{B} | \mathcal{N}) \).

Proof. The layer \( \mathcal{L}(0) \) is the class of functions \( f \) that are imitated by a term

\[ \vdash \hat{f} : \mathcal{N}'' \to \cdots \to \mathcal{N}'' \to \mathcal{N} \]

where each input type is \( \mathcal{N}'' \).

Consider any \( f \in \mathcal{U}(\lambda \mathcal{A}) \). There is a simulation

\[ \vdash \overline{f} : \mathcal{N}'' \to \cdots \to \mathcal{N}'' \to \mathcal{N}'' \]

of \( f \) in \( \lambda \mathcal{A} \). Using the numeral converter \( C : \mathcal{N}'' \to \mathcal{N} \) we may transform \( \overline{f} \) into an imitation \( \hat{f} \) of the same function. Thus \( \mathcal{U}(\lambda \mathcal{A}) \subseteq \mathcal{L}(0) \).
Consider any $f \in \mathcal{L}(0)$, with a imitation $\hat{f}$, as above. Since $\lambda A$ is a part of $\lambda B$, we have an imitation $\hat{f}$ in $\lambda B$ of the same function. But, in $\lambda B$, we have an extra atom $S$ and a numeral converter $J : S \to N''$ so we may transform $\hat{f}$ into a representation
\[ \vdash \hat{f} : S \to \cdots \to S \to N \]
of the same functions. Using Theorem 12.5 this shows that
\[ f \in \mathcal{R}(\lambda B \mid N) = \mathcal{E} \]
and hence $\mathcal{L}(0) \subseteq \mathcal{E}$.

From these two arguments we have
\[ \mathfrak{U}(\lambda A) \subseteq \mathcal{L}(0) \subseteq \mathcal{E} \]
which, since $\mathfrak{U}(\lambda A) = \mathcal{E}$, gives the required result. ■

This result is about $\lambda A$, the system that has only monochrome numerals and no iterators. However, note how the use of tiering, the use of numerals of different colours, plays a part in the proof. We move into the system $\lambda B$, and then invoke Theorem 12.5 which is proved using the exhaustive search technique. Here tiering seems to be a way of handling an exhaustive search. We develop this idea by producing systems stronger than $\lambda B$.

15.5 DEFINITION. For each $r < \omega$ let $\lambda B(r)$ be the tiered system which has a dedicated atom $N[\tau]$ and iterator
\[ I_\tau : N[\tau] \longrightarrow \tau'' \]
for each $\tau \in L(r)$. ■

Since $L(0) = \{ N \}$ we see that $\lambda B(0)$ has just one coloured atom $S = N[N]$ with its iterator
\[ I_N : S \longrightarrow N'' \]
and this is the only iterator of $\lambda B(0)$. In other words, $\lambda B(0)$ is the system $\lambda B$ analysed in Section 12. This is where the notation comes from.

We have an increasing chain
\[ \lambda B = \lambda B(0) \subseteq \lambda B(1) \subseteq \lambda B(2) \subseteq \cdots \subseteq \lambda B(r) \subseteq \cdots \quad (r < \omega) \]
of systems. In other words any derivation or computation of $\lambda B(r)$ is also a derivation or computation of $\lambda B(r + 1)$. The amalgam $\lambda B(\omega)$ of this chain is just the system $\lambda C$.

We are concerned with the representational strength of $\lambda B(r)$. More precisely, we are interested in the class
\[ \mathcal{R}(\lambda B(r) \mid N) \]
for each $r < \omega$. We know already that
\[ \mathcal{R}(\lambda B(0) \mid N) = \mathcal{E} = \mathcal{L}(0) \]
by Theorem 14.4. We obtained that results by an application of the exhaustive search method. Here we use the translation technique to obtain a more general result.
15.6 THEOREM. We have
\[ R(\lambda B(r) | N) = L(r) \]
for each \( r < \omega \).

Proof. As can be expected, we prove the two inclusions
\[ L(r) \subseteq R(\lambda B(r) | N) \quad R(\lambda B(r) | N) \subseteq L(r) \]
by quite different methods. Only the right hand inclusion needs the implementation technique.

Consider first \( f \in L(r) \). We have a simulation
\[ \vdash \overline{f} : \zeta(l)'' \rightarrow \cdots \rightarrow \zeta(1)'' \rightarrow \mathcal{N} \]
of \( f \) in \( \lambda A \) where each \( \zeta(i) \in L(r) \). This is also a simulation of \( f \) in \( \lambda B(r) \). But in that system we have iterators
\[ I_{\zeta(i)} : \mathcal{N}[\zeta(i)] \rightarrow \zeta(i)'' \]
and these combine with \( \overline{f} \) to produce a representation
\[ \vdash \overline{f'} : \mathcal{N}[\zeta(l)] \rightarrow \cdots \rightarrow \mathcal{N}[\zeta(1)] \rightarrow \mathcal{N} \]
of the same function.

This shows that \( L(r) \subseteq R(\lambda B(r) | N) \).

For the other inclusion consider \( f \in R(\lambda B(r) | N) \). We have a representation of \( f \) in \( \lambda B(r) \)
\[ \vdash f' : \xi(l) \rightarrow \cdots \rightarrow \xi(1) \rightarrow \mathcal{N} \]
where \( \xi(1), \ldots, \xi(l) \) are atoms. Some of these may be \( \mathcal{N} \) and some may be coloured. By taking note of these we see that the representation has the form
\[ \vdash f' : \mathcal{N}[\zeta(s)] \rightarrow \cdots \rightarrow \mathcal{N}[\zeta(1)] \rightarrow \mathcal{N} \rightarrow \cdots \rightarrow \mathcal{N} \rightarrow \mathcal{N} \]
where each input type \( \zeta \) is in \( L(r) \). I have taken the liberty of re-ordering the input types into simulated+authentic. We use the implementation technique to transform this into a derivation in \( \lambda A \).

Consider the associated iterator
\[ I_{\zeta} : \mathcal{N}[\zeta] \rightarrow \zeta'' \]
of any of these input types. The target type \( \zeta \) is a \( \lambda A \) type, so is implemented as itself, \( |\zeta| = \zeta \). This gives
\[ |\mathcal{N}[\zeta]| = \zeta'' \]
and so the representation gives an imitation
\[ \vdash \hat{f} : \zeta(s)'' \rightarrow \cdots \rightarrow \zeta(1)'' \rightarrow \mathcal{N} \rightarrow \cdots \rightarrow \mathcal{N} \rightarrow \mathcal{N} \]
of some function. As in the proof of Theorem 14.15, the term \( \hat{f} \) imitates the original function \( f \).
This is not quite in the required form since there may be some authentic input types \( \mathcal{N} \). However, using the numeral converter
\[
\mathcal{C} : \mathcal{N}'' \to \mathcal{N}
\]
(which exists in \( \lambda \mathcal{A} \)) we may transform the imitation \( \widehat{f} \) into an imitation
\[
\vdash \widehat{f} : \zeta(s)'' \to \cdots \to \zeta(1)'' \to \mathcal{N}'' \to \cdots \to \mathcal{N}'' \to \mathcal{N}
\]
of the same function. Since each \( \zeta \in L(r) \) and \( \mathcal{N} \in L(0) \), this shows that \( f \in \mathcal{L}(r) \).

Since \( \lambda \mathcal{C} \) is just the amalgam of the systems \( \lambda \mathcal{B}(r) \) we have the following.

15.7 COROLLARY. We have \( \mathcal{R}(\lambda \mathcal{C} | \mathcal{N}) = \mathcal{L} \).

The proof of Theorem 15.6 indicates that it should be possible to set up a more refined measure by taking note of the arity of a represented function. I won’t pursue that here.

The next job is to characterize the layers \( \mathcal{L}(1), \mathcal{L}(2), \ldots \) leading, perhaps, to a characterization of \( \mathcal{L} \). Unfortunately, I don’t know such a characterization, but I will give a few results that suggest a couple of conjectures.

We use exponentiation and iterated exponentiation. However, it is convenient to fix the base of the exponentiation as 2. The following is a modification of a part of Example 6.6. It is also related to Example 8.4, but the notation ‘\( 2 \)’ is now used in a slightly different way.

15.8 EXAMPLE. For an arbitrary type \( \zeta \) let
\[
\overline{2}_\zeta = \lambda u : \zeta'', y : \zeta', x : \zeta. u\overline{2}_\zeta y x
\]
where \( \overline{2}_\zeta : \zeta'' \) is the sumulated numeral for 2. Also let
\[
\overline{2} = \lambda u : \mathcal{N}'' . u\overline{2}_0
\]
where \( \overline{2} = \overline{2}_{\mathcal{N}'} \) and where \( \mathcal{S} \) and \( 0 \) are the authentic furnishings of \( \mathcal{N} \). We have
\[
\overline{2}_\zeta : \zeta'' \to \zeta'' \quad \overline{2} : \mathcal{N}'' \to \mathcal{N}
\]
and a routine argument gives
\[
\overline{2}_{\zeta \overline{m}_\zeta} \upmodels \overline{(2^m)\zeta} \quad \overline{2}\overline{m} \upmodels \overline{(2^{m\overline{m}})}
\]
for each \( m \in \mathbb{N} \). Here \( \overline{m} = \overline{m}_{\mathcal{N}'} \). Thus we have captured the exponential function \( 2^\ast \) (the function \( x \mapsto 2^x \)) in two different ways.

Since the type \( \zeta \) is arbitrary the left hand reduction show that \( 2^\ast \in \mathcal{S}(\lambda \mathcal{A}) = \mathcal{L} \). The right hand reduction improves this to \( 2^\ast \in \mathcal{L}(1) \).

In fact, the left reduction also give this tighter information since \( \overline{2} \) is a reduced form of \( \mathcal{C} \circ \overline{2}_{\mathcal{N}} \) where \( \mathcal{C} : \mathcal{N}'' \to \mathcal{N} \) is the usual numeral converter.
We now show that $2^* \not\in \mathcal{L}(0)$. To do that we use the stacking function $\mathfrak{m}$ which made an appearance at the end of Section 6. Here we expand on the remarks made there. As with exponentiation, here we fix the base as 2. Thus we use a chain of functions

$$\mathfrak{m}(r, 2) : \mathbb{N} \rightarrow \mathbb{N}$$

generated by

$$\mathfrak{m}(0, 2)x = x \quad \mathfrak{m}(r + 1, 2)x = \begin{cases} 2^{\mathfrak{m}(r, 2)x} & \text{for } r < \omega \\ \mathfrak{m}(r, 2)2^x & \end{cases}$$

for $r < \omega$ and $x \in \mathbb{N}$. There are alternative step clauses and each is useful in different circumstances.

We also use a kind of diagonal limit

$$\mathfrak{r} : \mathbb{N} \rightarrow \mathbb{N}$$

of these functions given by

$$\mathfrak{r}(x) = \mathfrak{m}(x, 2)0$$

(for $x \in \mathbb{N}$). This is an important function that marks the boundary between two classes of functions. We will see something of this in the next proof.

15.9 LEMMA. We have $2^* \in \mathcal{L}(1) - \mathcal{L}(0)$.

Proof. The imitation $\mathfrak{m}$ of Example 15.8 shows that $2^* \in \mathcal{L}(1)$.

By way of contradiction suppose $2^* \in \mathcal{L}(0)$. Then, since $L(0) = \{\mathcal{N}\}$, there is some term

$$\mathfrak{m} : \mathcal{N}'' \rightarrow \mathcal{N}''$$

which imitates $2^*$. Using $\mathfrak{m}$ let

$$\mathfrak{r} = \lambda u : \mathcal{N}^{(iv)} . u\mathfrak{m}^{0_{\mathcal{N}}}$$

so that

$$\mathfrak{r} : \mathcal{N}^{(iv)} \rightarrow \mathcal{N}''$$

and, with $m = \overline{m_{\mathcal{N}''}}$, we have

$$\mathfrak{m}^{m_{\mathcal{N}}} \gg \mathfrak{m}^{0_{\mathcal{N}}} \gg \mathfrak{m}(m)_{\mathcal{N}}$$

for each $m \in \mathbb{N}$. This shows that $\mathfrak{r} \in \mathcal{L}(2)$.

In fact, it can be shown that $\mathfrak{r} \notin \mathcal{L}$, which is the contradiction. $\blacksquare$

The proof that $\mathfrak{r} \notin \mathcal{L}$ isn’t hard but it needs a bit of preparation. We look at this proof in Section 21 where it fits better with the other material.

The functions $\mathfrak{m}(r, 2)$ also provide canonical examples of the levels of $\mathcal{L}$.

15.10 LEMMA. For each $r < \omega$ and type $\zeta$, there is a term

$$\mathfrak{m}(r, 2)_{\zeta} : \zeta^{(r+2)} \rightarrow \zeta''$$

which simulates the function $\mathfrak{m}(r, 2)$. 120
Proof. We generate the terms $\mathcal{R}(r, 2)\zeta$ by recursion over $r$ with variation of the type $\zeta$. To do this we use the terms $\mathcal{L}_{\zeta} : \zeta'' \to \zeta''$ of Example 15.8.

We set

$$\mathcal{R}(0, 2)\zeta = \lambda u : \zeta'' y : \zeta, x : \zeta . uyx \quad \mathcal{R}(r + 2, 2)\zeta = \mathcal{L}_{\zeta} \circ \mathcal{R}(r, 2)\zeta'$$

for each $r < \omega$ and type $\zeta$. We verify that these terms do the required job by induction over $r$ with variation of $\zeta$.

For the base case, $r = 0$, almost trivially we have $\mathcal{R}(0, 2)\zeta : \zeta'' \to \zeta''$ and this term simulates the identity function, that is $\mathcal{R}(0, 2)$.

For the induction step, $r \mapsto r + 1$, assuming $\mathcal{R}(r, 2)\zeta : \zeta'' \to \zeta''$ for a fixed $r$ and all types $\zeta$, we have $\mathcal{R}(r, 2)\zeta' : \zeta'' \to \zeta''$ to give $\mathcal{R}(r + 1, 2)\zeta : \zeta'' \to \zeta''$ as required. Similarly, with $r' = r + 1$, for each $m \in \mathbb{N}$ we have

$$\mathcal{R}(r', 2)\zeta_m \mathcal{R}_{\zeta''}(r, 2)\zeta_m \mathcal{R}_{\zeta''}(r + 1, 2)\zeta_m \mathcal{L}_{\zeta''} \mathcal{R}(r, 2)\zeta_m \mathcal{R}_{\zeta''}(r + 2, 2)\zeta_m = \mathcal{R}(r', 2)\zeta_m$$

as required.

This show that $\mathcal{R}(r, 2) \in \mathfrak{G}(\lambda A) = \mathfrak{L}$ for each $r \in \mathbb{N}$. In fact, a closer inspection of the types involved gives the following.

15.11 COROLLARY. For each $r \in \mathbb{N}$ we have $\mathcal{R}(r, 2) \in \mathfrak{L}(r)$.

Proof. We certainly have a simulation $\mathcal{R}(r, 2) : \mathcal{N}'' \to \mathcal{N}''$ of $\mathcal{R}(r, 2)$, and then a use of the numeral converter $C : \mathcal{N}'' \to \mathcal{N}$ gives a term $\mathcal{R}(r, 2) : \mathcal{N}''(r) \to \mathcal{N}$ which imitates $\mathcal{R}(r, 2)$. Since $\mathcal{N}''(r) \in L(r)$, this gives the required result.

It can be shown that that this result is optimal, namely that $\mathcal{R}(r + 1, 2) \notin \mathfrak{L}(r)$ for each $r \in \mathbb{N}$. Roughly speaking we show that for $r \neq 0$, each $f \in \mathfrak{L}(r)$ is bounded above by some finite iterate of $\mathcal{R}(r, 2)$, and then observe that $\mathcal{R}(r + 1, 2)$ can not be bounded in this way. The proof of this is a little more delicate, and will not be included in these notes. I do not know if it can be done using the exhaustive search method.

[Held in 165../05../015.. Last changed April 13, 2006]
16 Several systems compared.

In this section we compare the systems

$$\lambda A \quad \lambda B \quad \lambda C \quad \lambda L$$

and the chain of systems

$$\lambda B = \lambda B(0) \subseteq \cdots \subseteq \lambda B(r) \subseteq \cdots \subseteq \lambda B(\omega) = \lambda C$$

between $\lambda B$ and $\lambda C$. We gather together all the results obtained so far, and add a few more. We use the hierarchy

$$\mathcal{T} \subseteq \mathcal{E} = \mathcal{L}(0) \subseteq \cdots \subseteq \mathcal{L}(r) \subseteq \cdots \subseteq \mathcal{L}(\omega) = \mathcal{L}$$

as a yardstick by which to measure the various systems. We find that each system can capture at most the first order functions in $\mathcal{L}$, but do so in several different ways.

Recall that $\mathcal{T}$ and $\mathcal{E}$ are, respectively, the class of translations and the class of extended polynomials. We have various explicit descriptions of these functions. The class $\mathcal{L}$ of latent functions is less clear.

The various results we discuss are given in Table 11. Down the left hand column we list the systems we consider. For each such system $\lambda T$ the next three columns list

$$\mathcal{R}(\lambda T) \quad \mathcal{U}(\lambda T) \quad \mathcal{G}(\lambda T)$$

respectively. At least that is so for the $\mathcal{R}$- and the $\mathcal{G}$-column. The $\mathcal{U}$-column is almost empty. That is because I do not know the relevant results. The numbers under each equality indicate where that result is proved in these notes. As you can see, several results are obtained in this section.

Let’s look at the table in more detail.

Across the top row we have the results for $\lambda A$. These have been dealt with in some depth earlier.

The next two rows deal with $\lambda B$. This system has just two atoms, $\mathcal{N}$ and $\mathcal{S}$. We have seen that representations with these different targets lead to different classes of functions. These are given in the $\mathcal{R}$-column. We look at the $\mathcal{G}$-column later in this section.

The next four rows deal with the systems $\lambda B(r)$ for arbitrary $r$, and the system $\lambda C$. These systems have the standard monochrome atom $\mathcal{N}$, and several other atoms $\mathcal{S}$. Representation with target $\mathcal{N}$ have been dealt with earlier, as indicated in the table. It turns out that representations with target $\mathcal{S}$ (for any given coloured atom $\mathcal{S}$) gives a rather feeble class of functions, namely $\mathcal{T}$. Let’s deal with those before looking at the remainder of the table.

In Theorem 12.4 we stated without proof a result which gives

$$\mathcal{R}(\lambda B \mid \mathcal{S}) = \mathcal{T}$$

for the only coloured atom $\mathcal{S}$ of $\lambda B$. In fact, almost the same proof gives the result for $\lambda C$.

This is yet another example of the exhaustive search method.
### Table 11: The classes captured by various systems

<table>
<thead>
<tr>
<th></th>
<th>$\lambda$A</th>
<th>$\lambda$B</th>
<th>$\lambda$C</th>
<th>$\lambda$L</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R(\lambda A)$</td>
<td>$\mathcal{I}$</td>
<td>$\mathcal{E}$</td>
<td>$\mathcal{L}$</td>
<td>$\mathcal{E}$</td>
</tr>
<tr>
<td>$\mu(\lambda A)$</td>
<td>$\mathcal{C}$</td>
<td>$\mathcal{E}$</td>
<td>$\mathcal{L}$</td>
<td>$\mathcal{L}$</td>
</tr>
<tr>
<td>$G(\lambda A)$</td>
<td>$\mathcal{L}$</td>
<td>$\mathcal{L}$</td>
<td>$\mathcal{L}$</td>
<td>$\mathcal{L}$</td>
</tr>
<tr>
<td>$\lambda$B</td>
<td>$R(\lambda B</td>
<td>N)$</td>
<td>$\mathcal{E}$</td>
<td>$\mathcal{L}$</td>
</tr>
<tr>
<td>$\mu(\lambda B</td>
<td>N)$</td>
<td>$\mathcal{E}$</td>
<td>$\mathcal{L}$</td>
<td>$\mathcal{L}$</td>
</tr>
<tr>
<td>$G(\lambda B</td>
<td>N)$</td>
<td>$\mathcal{L}$</td>
<td>$\mathcal{L}$</td>
<td>$\mathcal{L}$</td>
</tr>
<tr>
<td>$\lambda$B</td>
<td>$R(\lambda B</td>
<td>S)$</td>
<td>$\mathcal{I}$</td>
<td>$\mathcal{L}$</td>
</tr>
<tr>
<td>$\mu(\lambda B</td>
<td>S)$</td>
<td>$\mathcal{I}$</td>
<td>$\mathcal{L}$</td>
<td>$\mathcal{L}$</td>
</tr>
<tr>
<td>$G(\lambda B</td>
<td>S)$</td>
<td>$\mathcal{L}$</td>
<td>$\mathcal{L}$</td>
<td>$\mathcal{L}$</td>
</tr>
<tr>
<td>$\lambda$C</td>
<td>$R(\lambda C</td>
<td>N)$</td>
<td>$\mathcal{L}$</td>
<td>$\mathcal{L}$</td>
</tr>
<tr>
<td>$\mu(\lambda C</td>
<td>N)$</td>
<td>$\mathcal{L}$</td>
<td>$\mathcal{L}$</td>
<td>$\mathcal{L}$</td>
</tr>
<tr>
<td>$G(\lambda C</td>
<td>N)$</td>
<td>$\mathcal{L}$</td>
<td>$\mathcal{L}$</td>
<td>$\mathcal{L}$</td>
</tr>
<tr>
<td>$\lambda$L</td>
<td>$R(\lambda L</td>
<td>S)$</td>
<td>$\mathcal{L}$</td>
<td>$\mathcal{L}$</td>
</tr>
<tr>
<td>$\mu(\lambda L</td>
<td>S)$</td>
<td>$\mathcal{L}$</td>
<td>$\mathcal{L}$</td>
<td>$\mathcal{L}$</td>
</tr>
<tr>
<td>$G(\lambda L</td>
<td>S)$</td>
<td>$\mathcal{L}$</td>
<td>$\mathcal{L}$</td>
<td>$\mathcal{L}$</td>
</tr>
</tbody>
</table>

16.1 **Theorem.** For each derivation in $\lambda$C

\[
(\nabla) \quad \Gamma \vdash t : \tau
\]

where the root judgement meets the condition of Table 12, one of the following applies

- (function) If $\tau$ is a function type then $\nabla$ provides a translation of one of the non-$N$ inputs.
- (product) If $\tau$ is a product type then $t$ is $\text{Pair}lr$ for some terms $l, r$.

depending on the shape of $\tau$.

**Proof.** Before we begin the proof proper, let’s make clear exactly what the (function) conclusion means.

Let $f$ be the function provided by $\nabla$. Some of the inputs to $f$ will arise form the context $\Gamma$ and some will arise from the input components $\xi_n, \ldots, \xi_1$ of $\tau$. We may divide these inputs into two kinds, those from an occurrence of $N$ and those from an occurrence of a coloured atom. Thus, after a suitable shuffling, a typical value of $f$ has the form

\[
f(y; x)
\]

where $x$ is the list of monochrome inputs and $y$ is the list of coloured inputs. (We could further subdivide $y$ by colour, but that is not necessary here.) We show that $f$ has one of the two forms

\[
f(y; x) = y + c \quad f(y; x) = c
\]

where $y$ is a nominated coloured input and $c$ is some fixed constant.
We consider derivations in \( \lambda C \)

\[
(\nabla) \quad \Gamma \vdash t : \tau
\]

where the root judgement meets the following restrictions.

(\(\Gamma\)) The context \( \Gamma \) is a list of declarations \( z : \xi \) where \( \xi \) is an atom (and different atoms, including \( \mathcal{N} \), may occur in the context).

(\(t\)) The subject term \( t \) is normal.

(\(\tau\)) The predicate type \( \tau \) is one of

- (function) A function type
  \[
  \xi_n \rightarrow \cdots \rightarrow \xi_1 \rightarrow \xi_0
  \]
  where \( \xi_0, \ldots, \xi_n \) are atoms, and \( \xi_0 \) is not \( \mathcal{N} \).
- (product) A product type \( \sigma \times \rho \)

for some \( \sigma, \rho \).

Table 12: Global conditions for Theorem 16.1

As usual, we prove this result by induction over the height of \( \nabla \). To do this we survey all the possible ways that \( \nabla \) can arise.

Firstly, \( \nabla \) may be a leaf.

(\(\nabla\) Axiom) Since we are working in \( \lambda C \) there are six possible shapes for \( t : \tau \) (some of which come in several varieties).

\[
\begin{align*}
(i) & \quad 0_\xi : \xi \\
(ii) & \quad S_\xi : \xi' \\
(iii) & \quad \text{I} : \xi \rightarrow \sigma'' \\
(iv) & \quad \text{Pair} : \sigma \rightarrow \rho \rightarrow \sigma \times \rho \\
(v) & \quad \text{Left} : \sigma \times \rho \rightarrow \sigma \\
(vi) & \quad \text{Right} : \sigma \times \rho \rightarrow \rho
\end{align*}
\]

(i) For this case \( \tau \) is \( \xi \), so that \( \xi \) is coloured. The function provided is identically 0.

(ii) For this case \( \tau \) is \( \xi' \), so that \( \xi \) is coloured. This provides the function \( f \) where

\[
f(y, y; x) = Sy = y + 1
\]

where the input \( y \) arise from the input position in \( \tau \).

(iii) Here we have

\[
\Gamma \vdash \text{I} : \xi \rightarrow \sigma' \rightarrow \sigma \rightarrow \sigma
\]

for some coloured atom \( \xi \) and \( \lambda A \)-type \( \sigma \). However, the occurrence of \( \sigma' \) means that the shape of the type of \( \text{I} \) does not match the required shape of \( \tau \). Thus this case does not arise.

(iv,v,vi) The shape of these types do not match the required shape of \( \tau \). Thus these cases don’t arise.

(\(\nabla\) Projection) Here we have

\[
\Gamma \vdash z : \xi
\]

where \( z : \xi \) occurs in \( \Gamma \). Also \( \tau \) is \( \xi \), so that \( \xi \) is coloured. Thus the provided function is

\[
f(y; x) = y
\]

124
where \( y \) is the input corresponding to the projected declaration.

Secondly, \( \nabla \) may arise by a use of one of the construction rules. These will involve shorter derivation to which, if the global conditions are met, we may apply the induction hypothesis.

(\( \nabla \) Weakening) Using the induction hypothesis we see that the numerator of this particular use provides some translation \( g \) of the appropriate kind. The denominator provides the translation \( f \) obtained from \( g \) by inserting a dummy input.

(\( \nabla \) Introduction) In the usual fashion, the numerator and denominator of such a use provide the same function.

(\( \nabla \) Elimination) As usual we track through the construction of \( \nabla \) to unravel as many uses of Elimination and Weakening as possible. Thus we obtain a family of shorter derivations

\[
(Q) \quad \Xi \vdash q : \pi_k \to \cdots \to \pi_1 \to \tau \quad (P_i) \quad \Pi_i \vdash p_i : \pi_i \quad i \leq i \leq k
\]

where

\[
t = qp_k \cdots p_1
\]

with \( q \) and each \( p_i \) normal, and where each context \( \Xi \) and \( \Pi_i \) is a left hand part of \( \Gamma \). By reinstating the uses of Weakening we obtain derivations

\[
(Q^+) \quad \Gamma \vdash q : \pi_k \to \cdots \to \pi_1 \to \tau \quad (P_i^+) \quad \Gamma \vdash p_i : \pi_i \quad i \leq i \leq k
\]

all in the original context, and all of which are strictly shorter than \( \nabla \). We let

\[
\chi = \pi_k \to \cdots \to \pi_1 \to \tau
\]

and consider how \( Q \) can arise. Observe that \( k \geq 1 \), so that \( \chi \) can not be an atom.

(\( Q \) Axiom) There are six possible as listed above.

(i) This can not arise since \( \chi \) must be compound.

(ii) Here we must have

\[
(Q^+) \quad \Gamma \vdash S_\xi : \xi \to \xi \quad (P^+) \quad \Gamma \vdash p : \xi
\]

for some atom \( \xi \). In particular, we have \( k = 1 \) and with \( \tau = \xi \), so that \( \xi \) is coloured, and \( t = Sp \) for some term \( p \). The shorter derivation \( P^+ \) meets the global restriction and hence, by the induction hypothesis, this derivation provides a translation of the appropriate kind. The function provided by \( \nabla \) is the composite of this function followed by the successor function, and so is a translation of the appropriate kind.

(iii) We show that this case doesn’t occur. By way of contradiction, suppose it does arise. Then we have at least

\[
\Gamma \vdash 1 : \xi \to \sigma'' \quad \Gamma \vdash p : \xi
\]

for some coloured atom \( \xi \) and \( \lambda A \) type \( \sigma \). This means that

\[
t = 1p \cdots
\]

and \( \tau \) is embedded in the type \( \sigma'' \). But then \( \tau \) is a \( \lambda A \)-type, which can not be since its target component \( \xi_0 \) is coloured.
(iv) Here we have at least

\[ (Q^+) \quad \Gamma \vdash \text{Pair} : \sigma \rightarrow \rho \rightarrow \sigma \times \rho \quad (L^+) \quad \Gamma \vdash l : \sigma \]

with

\[ \chi = \sigma \rightarrow \rho \rightarrow \sigma \times \rho \]

which tells us what \( \tau \) can be. One of

1. \( k = 1 \) with \( \pi_1 = \sigma \) and \( \tau = \rho \rightarrow \sigma \times \rho \)
2. \( k = 2 \) with \( \pi_2 = \sigma, \pi_1 = \rho \) and \( \tau = \sigma \times \rho \)

must occur. In more detail we have one of the following.

1. We have

\[ (Q^+) \quad \Gamma \vdash \text{Pair} : \sigma \rightarrow \rho \rightarrow \sigma \times \rho \quad (L^+) \quad \Gamma \vdash l : \sigma \]

with \( t = \text{Pair}l \) and \( \tau = \rho \rightarrow \sigma \times \rho \).

2. We have

\[ (Q^+) \quad \Gamma \vdash \text{Pair} : \sigma \rightarrow \rho \rightarrow \sigma \times \rho \quad (L^+) \quad \Gamma \vdash l : \sigma \quad (R^+) \quad \Gamma \vdash r : \rho \]

with \( t = \text{Pair}lr \) and \( \tau = \sigma \times \rho \).

However, case (1) can be discounted because the shape of this \( \tau \) doesn’t meet the required global conditions. Case (2) meets the required conclusion (product).

(v) Here we have at least

\[ (Q^+) \quad \Gamma \vdash \text{Left} : \sigma \times \rho \rightarrow \sigma \quad (P^+) \quad \vdash p : \sigma \times \rho \]

where there may be other auxiliary derivations \( P_2^+, \ldots, P_k^+ \). The shorter derivation \( P^+ \) meets the global conditions and hence the induction hypothesis gives \( p = \text{Pair}lr \) for some terms \( l, r \). But now

\[ t = \text{Left}(\text{Pair}lr) \cdots \]

which since

\[ \text{Left}(\text{Pair}lr) \not\vdash l \]

contradicts the assumed normality of \( t \). Thus this case can not arise.

(vi) As with (v), this case can not arise.

This completes the case where \( Q \) is an Axiom.

(Q Projection) This case can not occur since the only possible projected statement from this context \( \Gamma \) has the form \( z : \xi \), and this type can not match \( \chi \).

(Q Weakening) This case is built into the unravelling of \( \nabla \).

(Q Introduction) In this case we have \( q = (\lambda x : \pi . r) \) for some term \( r \), where \( \pi = \pi_k \).

But then with \( p = p_k \) we see that the redex \( (\lambda x : \pi . r)p \) is an initial part of \( t \). Since \( t \) is normal, this case can not arise.

(Q Elimination) This case is built into the unravelling of \( \nabla \).

This concludes the full proof.

With this we have the following result which adds to the \( \mathcal{R} \)-column of Table 11
16.2 COROLLARY. We have
\[ R(\lambda B | S) = R(\lambda B(r) | S) = R(\lambda C | S) = \mathcal{I} \]
where \( S \) is any coloured atom of the system in question.

Proof. We have
\[ \mathcal{I} \subseteq R(\lambda B | S) \subseteq R(\lambda B(r) | S) \subseteq R(\lambda C | S) \subseteq \mathcal{I} \]
where Theorem 16.1 gives the extreme right hand inclusion, and the others are trivial. ■

The proof of Theorem 16.1 gives us more precise information, in the sense that we can determine exactly the translations arise. This information is not needed here, but may be useful in the future.

Let us now turn to the system \( \lambda L \), a system we haven’t looked at before. This has a rather intricate type structure and, at first sight, appears to be quite strong. However, as we now show, the system satisfies Restriction 14.9.

What do we have to do? Given a type \( \tau \) of \( \lambda L \) we must be able to calculate its implementation \( |\tau| \) as a \( \lambda A \)-type using the algorithm set of in Definition 14.7. This, of course, requires a calculation of \( |\xi| \) for each coloured atom \( \xi \) that occurs in \( \tau \), and this requires a calculation of \( |\sigma| \) for certain other types, which in turn brings in other coloured atoms, and so on. We have to show that this process terminates. To do that we first give a more detailed description of the type structure of \( \lambda L \).

We generate the types of \( \lambda L \) in phases.

(Phase (0)) We use the monochrome atom \( N \) to generate of the \( \lambda A \)-types. Think of these types as being generated in some systematic order. As each such types \( \tau \) is generated we create a new coloured atom \( \xi = N[\tau] \) together with an iterator
\[ I : \xi \rightarrow \tau'' \]
dedicated to \( \tau \). As each such coloured atom \( \xi \) is created we put it to one-side for use later. Notice that for such an atom we have
\[ |\xi| = |\tau''| = \tau'' \]
so that its implementation is well-defined.

(Phase (1)) At the end of phase (0) we have many new atoms. In fact, we have a bijective correspondence between the new atoms and the \( \lambda A \)-types. Each corresponding pair \( \xi \) and \( \tau \) is indexed by an iterator \( I \), as above.

We now use \( N \) and the new coloured atoms from phase (0) to generate all possible types. Some of these will be repetitions of types dealt with in phase (0), so we discard these. Think of these new types as being generated in some systematic order. As each such types \( \tau \) is generated we create a new coloured atom \( \xi = N[\tau] \) together with an iterator
\[ I : \xi \rightarrow \tau'' \]
dedicated to \( \tau \). As each such coloured atom \( \xi \) is created we put it to one-side for use later. Once these new types have been generated we have the \( \lambda C \)-types.
Consider any such type \( \tau \). This is built up from \( \mathcal{N} \) and certain coloured types \( \xi \) created in phase (0). Each such atom has a well defined implementation, and hence we can calculate \( |\tau| \). For instance, suppose

\[
\tau = \xi \rightarrow \mathcal{N}
\]

for some coloured atom \( \xi \). We have \( |\xi| = \sigma'' \) for some \( \lambda \mathcal{A} \)-type \( \sigma \), and then

\[
|\tau| = |\xi| \rightarrow |\mathcal{N}| = \zeta'' \rightarrow \mathcal{N}
\]

by a 2-step calculation. Notice that this \( \tau \) looks like a rather simple type, but its implementation exposes some hidden complexities.

(Phase (2)) At the end of phase (1) we have many more new atoms. In total, we have \( \mathcal{N} \), the coloured atoms created in phase (0), and the coloured atoms created in phase (1). We use all these atoms to generate all possible types, except we discard those types dealt with in phase (0) or phase (1). Think of these new types as being generated in some systematic order. As each such types \( \tau \) is generated we create a new coloured atom \( \xi = \mathcal{N}[\tau] \) together with an iterator

\[
1 : \xi \rightarrow \tau''
\]

dedicated to \( \tau \). As each such coloured atom \( \xi \) is created we put it to one-side for use later.

Consider any type \( \tau \) generated in this phase. This is built up from \( \mathcal{N} \) and certain coloured types created in phase (0), and certain coloured types created in phase (1). Each such atom has a well defined implementation, and hence we can calculate \( |\tau| \). For instance, suppose

\[
\tau = \chi \rightarrow \xi
\]

for some coloured atom \( \xi \) created in phase (0), and for some coloured atom \( \chi \) created in phase (1). We have \( |\xi| = \sigma'' \) for some \( \lambda \mathcal{A} \)-type \( \sigma \). The atom \( \chi \) will have been created as the source of some iterator of type

\[
\chi \rightarrow \zeta''
\]

where \( \zeta \) is some type generated in phase (0). This well have an implementation \( |\zeta| = \rho'' \) for some \( \lambda \mathcal{A} \)-type \( \rho \). But now

\[
|\chi| = |\zeta||'' = \rho^{(iv)}
\]

to give

\[
|\tau| = |\chi| \rightarrow |\xi| = \rho^{(iv)} \rightarrow \sigma''
\]

to show that \( \tau \) is more complicated than it looks.

(Phase \((r + 1)\)) At the end of phase \((r)\) we have many atoms. We have \( \mathcal{N} \) together with the coloured atoms created in earlier phases. We use all these atoms to generate all possible types, except we discard those types dealt with in the earlier phases. Think of these new types as being generated in some systematic order. As each such types \( \tau \) is generated we create a new coloured atom \( \xi = \mathcal{N}[\tau] \) together with an iterator

\[
1 : \xi \rightarrow \tau''
\]
dedicated to $\tau$. As each such coloured atom $\xi$ is created we put it to one-side for use in the next phase.

Consider any type $\tau$ generated in this phase. This is built up from $N$ and certain coloured types created in the earlier phases. Each such atom has a well defined implementation, and hence we can calculate $|\tau|$. However, note that this calculation will involve tracking through all the earlier phases.

By definition, $\lambda L$ is that system with the type structure and iterators generated by this infinite process. Thus we have the following.

16.3 LEMMA. *The system $\lambda L$ satisfies Restriction 14.9.*

Since $\lambda C$ is a part of $\lambda L$, this result with Corollary 15.7, Theorem 14.15, and Theorem 13.9 gives the following.

16.4 COROLLARY. *We have $\mathcal{R}(\lambda L | N) = \mathcal{R}(\lambda L) = \mathcal{L}$.*

We are going to improve this result by showing that we may replace the target type $N$ by any atom $S$ of $\lambda L$. This result gives the remaining entry in the $\mathcal{R}$-column of Table 11.

To prove the result we exploit the homogeneous, or polymorphic, nature of $\lambda L$. This need a bit of explaining.

Consider any finite set $\tau$ of types of $\lambda L$. These types are built up from certain atoms of $\lambda L$, but these construction don’t tell us the whole story. We require a full construction of these types. To explain what this is let’s start to unravel the construction of the types.

The types are built up from certain atoms

$$\xi_0, \xi_1, \ldots, \xi_i$$

where we let $\xi_0 = N$ (even this does not actually occur in the types). Each coloured atom $\xi$ in this list is introduced into $\lambda L$ as the source of some iterator

$$l : \xi \longrightarrow \sigma''$$

for some ‘simpler’ type $\sigma$. In other words, we have $\xi = N[\sigma]$. Consider the types

$$\sigma_1, \ldots, \sigma_l$$

attached to $\xi_1, \ldots, \xi_i$. These types are built up from certain atoms

$$\xi_{l+1}, \ldots, \xi_m$$

at some earlier phase (as explained above). These new atoms are attached to types

$$\sigma_{l+1}, \ldots, \sigma_m$$

which are built up from atoms

$$\xi_{m+1}, \ldots, \xi_n$$

at an even earlier phase, and so on.
In this way we produce a complete unravelling of the construction of the types in $\tau$. In particular, we generate a finite set of atoms needed for the constructions.

The full construction of $\tau$ starts from $\mathcal{N}$ and first generates finitely many $\lambda\text{A}$-types. Some of these generated types $\sigma$ are used to create new atoms

$$\xi = \mathcal{N}[\sigma]$$

along with the associated iterator. These new atoms, along with $\mathcal{N}$, are used to generate further types, some of which create further new atoms, which are then used to generate more types, and so on. Eventually, after a finite number of steps, we have all the information needed to generate each member of the finite set $\tau$ of types.

The full construction of $\tau$ can be organized as follows.

We start with $\mathcal{N}$ and first generate finitely many $\lambda\text{A}$-types. For certain of these types a new coloured atom is created. We then use all these atoms to generate further types, some of which are used to create new atoms, which are then used to generate further types, and so on. Eventually, after a finite number of steps, we have generated all the types in $\tau$.

Let $\lambda L(\tau)$ be that part of $\lambda L$ in which all this takes place.

16.5 EXAMPLE. Consider the type

$$\tau = \chi \rightarrow \xi \rightarrow \mathcal{N}$$

where

$$\xi = \mathcal{N}[\xi'], \chi = \mathcal{N}[\chi']$$

are the two coloured atoms. The list

$$\mathcal{N}, \sigma = (\mathcal{N} \rightarrow \mathcal{N}), \xi = \mathcal{N}[\sigma], \rho = (\xi \rightarrow \xi), \chi = \mathcal{N}[\rho], \zeta = (\xi \rightarrow \mathcal{N}), \tau = (\chi \rightarrow \zeta)$$

is the full construction of the type $\tau$. ■

We can now invoke the homogeneity of $\lambda L$. Consider the full construction of some finite set $\tau$ of types. This construction can be mimicked starting from any coloured atom $\mathcal{S}$. We replace the use of $\mathcal{N}$ be $\mathcal{S}$. Each types $\sigma$ that was generated is replaced by some type $\sigma(\mathcal{S})$ where $\mathcal{S}$ now plays the role of $\mathcal{N}$. Each atom

$$\xi = \mathcal{N}[\sigma]$$

that was created is replaced by the atom

$$\xi(\mathcal{S}) = \mathcal{N}[\sigma(\mathcal{S})]$$

attached to the replacement of $\sigma$. In this way we obtain the $\mathcal{S}$-variant $\tau(\mathcal{S})$ of each member $\tau$ of $\tau$.

16.6 EXAMPLE. Consider the type

$$\tau = \chi \rightarrow \xi \rightarrow \mathcal{N}$$

where

$$\xi = \mathcal{N}[\xi'], \chi = \mathcal{N}[\chi']$$

130
as in Example 16.5. The full construction of \( \tau \) is given in the left hand column.

\[
\begin{align*}
N & \quad N(S) = S \\
\sigma = (N \rightarrow N) & \quad \sigma(S) = (S \rightarrow S) \\
\xi = N[\sigma] & \quad \xi(S) = N[\sigma(S)] \\
\rho = (\xi \rightarrow \xi) & \quad \rho(S) = (\xi(S) \rightarrow \xi(S)) \\
\chi = N[\rho] & \quad \chi(S) = N[\rho(S)] \\
\zeta = (\xi \rightarrow N) & \quad \zeta(S) = (\xi(S) \rightarrow S) \\
\tau = (\chi \rightarrow \zeta) & \quad \tau(S) = (\chi(S) \rightarrow \zeta(S))
\end{align*}
\]

Let \( S \) be any coloured atom, and consider the construction in the right hand column. This generates a type

\[
\tau(S) = \chi(S) \rightarrow \xi(S) \rightarrow S
\]

of the same shape as \( \tau \) but where each use of \( N \), explicit or implicit, is replaced by a use of \( S \).

The homogeneity of \( \lambda L \) gives the following.

16.7 LEMMA. We have

\[
R(\lambda L | N) \subseteq R(\lambda L | S)
\]

for each coloured atom \( S \).

Consider any \( f \in R(\lambda L | N) \). This represented by some term

\[
(\nabla) \vdash f : \xi_1 \rightarrow \cdots \xi_l \rightarrow N
\]

for some atoms \( \xi_1, \ldots, \xi_l \). We survey the derivation \( \nabla \), and obtain a full construction of all the types used. This may involve atoms other than the ones in the type of \( f \). The term \( f \) may use various iterators, but the associated atom of each will occur in this full construction.

We mimic the full construction with \( N \) replaced by the atom \( S \). In this way we obtain a derivation

\[
(\nabla(S)) \vdash f(S) : \xi_1(S) \rightarrow \cdots \xi_l(S) \rightarrow S
\]

where the term \( f(S) \) has the same shape as \( f \) but could use different iterators. By a rather tedious induction we see that \( f(S) \) also represents \( f \), to give \( f \in R(\lambda L | S) \), as required.

With this we can complete the \( \mathcal{R} \)-column of Table 11.

16.8 THEOREM. For each atom \( S \) of \( \lambda L \) we have \( \mathcal{R}(\lambda T | S) = \mathcal{L} \).

Proof. We have

\[
\mathcal{L} = R(\lambda L | N) \subseteq R(\lambda L | S) \subseteq R(\lambda L) \subseteq \mathcal{L}
\]

by Corollary 16.4 and Lemma 16.7.

All that remains now is to complete the \( \mathcal{S} \)-column of Table 11.
16.9 THEOREM. We have
\[ \mathcal{S}(\lambda T) = \mathcal{L} \]
for each system \( \lambda T \) between \( \lambda A \) and \( \lambda L \).

Proof. Trivially, we have
\[ \mathcal{L} = \mathcal{S}(\lambda A) \subseteq \mathcal{S}(\lambda T) \subseteq \mathcal{S}(\lambda L) \]
so that an inclusion \( \mathcal{S}(\lambda L) \subseteq \mathcal{L} \) will suffice. In fact, we show \( \mathcal{S}(\lambda L) \subseteq \mathcal{R}(\lambda L) \) and then invoke Corollary 16.4.

Consider any \( f \in \mathcal{S}(\lambda L) \). We have a simulation
\[ \overline{f} : \zeta_l'' \rightarrow \cdots \rightarrow \zeta_1'' \rightarrow \zeta_0'' \]
of \( f \) in \( \lambda L \) we may choose \( \zeta_0 \) in any way we please (but have to accept \( \zeta_1, \ldots, \zeta_l \) as they come out). Let us choose \( \zeta_0 \) to be \( \mathcal{N} \). Within \( \lambda L \) we have numeral iterators
\[ l_i : \xi_i \rightarrow \zeta_i'' \]
for \( 1 \leq i \leq l \), and we have a numeral converter
\[ C : N'' \rightarrow \mathcal{N} \]
and with these we can modify \( \overline{f} \) to produce a representation
\[ \overline{f^*} : \xi_l \rightarrow \cdots \rightarrow \xi_1 \rightarrow \xi_0 \]
of the same function \( f \). Hence \( f \in \mathcal{R}(\lambda L) \), as required. \( \blacksquare \)

On this point it is worth comparing the two systems \( \lambda C \) and \( \lambda L \). The syntactic structure of \( \lambda C \) is straightforward, but that of \( \lambda L \) is rather intricate. However, the two systems have exactly the same strength, each can name only the latent functions. As a tiered system \( \lambda C \) is much better than \( \lambda L \). This is because the colour coding is used to stratify the strength of \( \lambda C \), there is a difference between terms with target \( \mathcal{N} \) and terms with a coloured target. In \( \lambda L \) all the colours do the same job, and the system merely replicates the same stuff over and over again. The use of colour coding in \( \lambda C \) is one aspect that ought to be developed further.

[Last changed April 13, 2006]
Notice that \( d_0 = 0 \). This is to ensure that the function keeps within the natural numbers.

You might think it a little odd to have a whole section devoted to one simple function. However, the section is really about the way the existence or not of pairing gadgets, and numeric gadgets can have an effect on the way we capture functions. The function \( d \) is used to illustrate this.

In Example 6.6 and the proof of Lemma 9.4 we saw how the commonplace numeric functions could be simulated in any tiered system. In fact, if you take another look at those simulations you will see that only the facilities of \( \lambda \), the pure simply typed \( \lambda \)-calculus without products, are used. In a similar way the predecessor function can also be simulated throughout \( \lambda \).

### 17.2 Definition

Let \( \zeta \) be an arbitrary type, and let

\[
\overline{d} = \lambda x : \zeta^{(5)} . x(-3-)(-2-)\overline{T}
\]

where

\[
(-3-) = \lambda w : \zeta''' , z : \zeta', y : \zeta , x : \zeta . z(-3b-)(-3a-)
\]

\[
(-3b-) = \lambda u : \zeta . w\overline{uyx}
\]

\[
(-3a-) = y(w\overline{uyx})
\]

\[
(-2-) = \lambda z : \zeta'' , y : \zeta' , x : \zeta . z(\lambda u : \zeta . x)x
\]

\[
\overline{T} = \lambda y : \zeta' , x : \zeta'' . yx
\]

\[
\overline{U} = \lambda y : \zeta' , x : \zeta'' . x
\]

so that \( \overline{d} \) has type \( \zeta^{(5)} \rightarrow \zeta^{(2)} \) (by a simple calculation).

Checking that \( \overline{d} \) does have the indicated type is straightforward, but the resulting derivation is a bit convoluted. You should write down the whole tree sometime.

More importantly, it can be shown that

\[
\overline{d}\overline{m}:\overline{\zeta}^{(5)} \leftrightarrow (d\overline{m})\overline{\zeta}
\]

for each \( m \in \mathbb{N} \), and so \( d \) can be simulated throughout \( \lambda \).

On seeing this you may think: ‘I wonder where that comes from’, or make some similar and more pointed observation. Before you read any further I suggest you try to organize a proof of the reduction.

Whilst you’re doing that I will partake of a nice cup of tea.

It’s not easy, is it? But with a bit of effort it can be done, and so obtain the following.

### 17.3 Lemma

We have \( d \in \mathcal{L}(3) \).

**Proof.** As a particular case of Definition 17.2 we have a simulation

\[
\overline{d} : \mathcal{N}^{(5)} \rightarrow \mathcal{N}''
\]

which gives an imitation

\[
\hat{d} : \mathcal{N}^{(5)} \rightarrow \mathcal{N}
\]
The complexity of the simulation of Definition 17.2 seems a little strange to me. We know that $d$ is not an extended polynomial so it cannot be simulated by a term of type $\zeta'' \rightarrow \zeta'''$. But why do we need a jump to $\zeta'(5) \rightarrow \zeta''$? Is there a simulation throughout $\lambda$ of type $\zeta''' \rightarrow \zeta''$ or even $\zeta'(4) \rightarrow \zeta''$? I have asked several people about this, but nobody seems to know.

Let’s take another look at the simulation. I have deliberately made this simulating term a bit obscure to illustrate a point. Simulations such as these often appear in the literature, and seem to be presented as mere tricks that work. In fact, this is a very simple example of a general method.

We do start with a little trick. I have always thought of this as Kleene’s trick, but other people have told me that it is Bernays’ trick. (No doubt they, and other people, thought of it independently.)

Consider the function

$$D : \mathbb{N}^2 \rightarrow \mathbb{N}^2$$

where

$$D(x, y) = (x', x)$$

for $x, y \in \mathbb{N}$. This looks a bit silly, doesn’t it.

Any function to the target $\mathbb{N}^2$ can be viewed as two functions in parallel, one to each component. But this does not mean that each specification of such a function can be written as two specifications in parallel. The two components may interfere with each other. In $D$ this interference is rather silly or extreme depending on how you look at it. Like a newly hatched cuckoo the first component $x$ turfs out the second component $y$ and takes its place. What is the point of this? Well, that is what cuckoos do, and looking at the example we have

$$D(x, dx) = (x', dx')$$

as a particular case. Thus, by iteration

$$D^2(0, 0) = (x, dx)$$

and we obtain $d$ by extracting the right component.

This shows that the recursion giving $d$ can be replaced by by an iteration, provided we are prepared to carry extra information through the iteration. In Section 17 we will show that many other forms of recursion can be handled in the same way.

Unlike the assertion just after Definition 17.2, the following has an almost trivial proof.

**17.4 LEMMA.** Let $\lambda \mathbf{T}$ be any tiered system which has an iterator of type $\mathcal{M} \rightarrow (\mathcal{N} \times \mathcal{N})''$ where the atom $\mathcal{M}$ may or may not be $\mathcal{N}$.

Let

$$\langle D \rangle = \lambda w : \mathcal{N} \times \mathcal{N}. \text{Pair}(S(\text{Left}w))(\text{Left}w)$$

$$\langle d \rangle = \lambda x : \mathcal{M}. \text{Right}(1x\langle D \rangle(\text{Pair}00))$$

where $1$ is the assumed iterator. Then

$$\vdash \langle d \rangle : \mathcal{M} \rightarrow \mathcal{N}$$

and $\langle d \rangle$ names the predecessor function.
Proof. The derivation of 
\[ \vdash \overline{d} : \mathcal{M} \rightarrow \mathcal{N} \]
was dealt with in Example 2.11.

For the reduction we first check that 
\[ \overline{D} \overline{\text{Pair}} \overline{m} \overline{n} \ implies \ \overline{\text{Pair}} \overline{1 + m} \overline{n} \]
so that 
\[ \overline{D} \overline{\text{sm Pair}} \overline{0} \overline{0} \ implies \ \overline{\text{Pair}} \overline{m} \overline{dm} \]
for each \( m \in \mathbb{N} \). But now 
\[ \overline{d} \overline{m} \ implies \ \overline{\text{Right}}(\overline{D} \overline{\text{sm Pair}} \overline{0} \overline{0}) \ implies \ \overline{dm} \]
as required. \( \blacksquare \)

Since \( \mathcal{N} \times \mathcal{N} \in L(1) \) the system \( \lambda B(1) \) has an iterator of type \( \mathcal{M} \rightarrow (\mathcal{N} \times \mathcal{N})'' \), and hence Theorem 15.6 with Lemma 17.4 gives the following improvement of Lemma 17.3.

17.5 Lemma. We have \( d \in \mathcal{L}(1) - \mathcal{L}(0) \).

This representation seems to be quite natural. Once the Kleene/Bernays trick is understood everything else is straightforward. But what about the simulation of Definition 17.2? Where does that come from?

Consider the representation 
\[ \overline{d} : \mathcal{M} \rightarrow \mathcal{N} \]
in \( \lambda B(1) \). This can be implemented in \( \lambda A \). The representation uses the iterator
\[ 1 : \mathcal{M} \rightarrow (\mathcal{N} \times \mathcal{N})'' \]
just once and we have 
\[ |\mathcal{M}| = (\mathcal{N} \times \mathcal{N})'' \]
by Definition 14.10. Thus we obtain an imitation 
\[ \hat{d} : (\mathcal{N} \times \mathcal{N})'' \rightarrow \mathcal{N} \]
of \( f \) in \( \lambda A \).

We can translate this into a simulation in \( \lambda A \). Let \( \zeta \) be any type and consider the translation generated by \( \mathcal{N} \mapsto \zeta'' \). This gives a simulation 
\[ \overline{\hat{d}} : (\zeta'' \times \zeta'')'' \rightarrow \zeta'' \]
which is beginning to look a bit like that of Definition 17.2, or at least the type is.

The big difference is that this last simulation still uses product types and explicit pairing gadgets, whereas the earlier one doesn’t. However, some instances of these can be captured in \( \lambda \).
17.6 Definition. For an arbitrary type \( \zeta \) consider the terms

\[
\begin{align*}
P &= \lambda l, r, z : \zeta'' . y : \zeta' . x : \zeta . (\lambda u : \zeta . r y x)(l y x) \quad \zeta'' \rightarrow \zeta'' \rightarrow \zeta'' \\
L &= \lambda w : \zeta'' . w \overline{0}_\zeta \quad \zeta'' \rightarrow \zeta'' \\
R &= \lambda w : \zeta'' . w \overline{1}_\zeta \quad \zeta'' \rightarrow \zeta'' \\
S &= \lambda v : \zeta'' . y : \zeta' . x : \zeta . y(v y x) \quad \zeta'' \rightarrow \zeta''
\end{align*}
\]

where the type of each is displayed on the right. The term \( S \) is just the standard simulation of the successor function over \( \zeta \).

Observe that all these are part of \( \lambda \). We show that in some circumstances the three terms \( P, L, R \) can take the place of the pairing gadgets, where \( \zeta'''' \) takes the place of \( \zeta'' \times \zeta'' \).

17.7 Lemma. We have

\[
L(P \overline{m}_\zeta \overline{n}_\zeta) \nrightarrow \overline{m}_\zeta \quad R(P \overline{m}_\zeta \overline{n}_\zeta) \nrightarrow \overline{n}_\zeta
\]

for all \( m, n \in \mathbb{N} \).

Proof. We first observe that

\[
\begin{align*}
P \overline{m}_\zeta \overline{n}_\zeta \overline{0}_\zeta & \nrightarrow \lambda y : \zeta' . x : \zeta . \overline{0}_\zeta(-)(\overline{m}_\zeta y x) \\
& \nrightarrow \lambda y : \zeta' . x : \zeta . (-)^0(\overline{m}_\zeta y x) \\
& \nrightarrow \lambda y : \zeta' . x : \zeta . (\overline{m}_\zeta y x) \\
& \nrightarrow \lambda y : \zeta' . x : \zeta . y^{\overline{n}_\zeta} x = \overline{m}_\zeta \\
P \overline{m}_\zeta \overline{n}_\zeta \overline{1}_\zeta & \nrightarrow \lambda y : \zeta' . x : \zeta . \overline{1}_\zeta(\lambda u : \zeta . \overline{n}_\zeta y x)(-)
\end{align*}
\]

for each \( m, n \in \mathbb{N} \), and then the required reductions are immediate.

Note how these reductions use the internal structure of the simulated numerals. In particular, reductions

\[
L(Plr) \nrightarrow l \quad R(Plr) \nrightarrow r
\]

do not hold for some other terms \( l, r \).

For each \( m, n \in \mathbb{N} \) let

\[
(m, n)_\zeta = P \overline{m}_\zeta \overline{n}_\zeta
\]

to produce a term \((m, n)_\zeta : \zeta''''\). We have

\[
L(m, n)_\zeta \nrightarrow \overline{m}_\zeta \quad R(m, n)_\zeta \nrightarrow \overline{n}_\zeta
\]

by Lemma 17.7.

Now let

\[
D = \lambda w : \zeta(3) . P(S(Lw))(Lw) \quad d = \lambda x : \zeta(5) . R(xD(0, 0)_\zeta)
\]

to produce terms with types

\[
D : \zeta(4) \quad d : \zeta(5) \rightarrow \zeta''
\]

respectively. Notice the highest type level \( \zeta(5) \) involved here.
A simple calculation shows that
\[ D(m, n) \zeq (m', m) \]
holds for all \( m, n \in \mathbb{N} \), and hence
\[ D^m(0, 0) \zeq (m, dm) \]
holds for each \( m \in \mathbb{N} \). Thus we have
\[ dm \zeq (0, 0) \zeq \lambda x : \zeta^{(5)} . x D(0, 0) \zeq \lambda x : \zeta^{(5)} . x (-2) (-3) \]
which shows that \( d \) simulates \( d \).

That wasn’t too bad, was it? Notice how this simulation follows the informal calculation of the Kleene/Bernays trick, and is similar to the representation of Lemma 17.4.

What has this got to do with the simulation \( \overline{d} \) of Definition 17.2. Where does this come from? Notice that \( d \) is not in normal form, so perhaps it can be ‘improved’ by some judicious internal reductions.

Remembering how the various components are constructed and that \( \overline{0} = \overline{\lambda} \), we have
\[ (0, 0) \zeq \lambda z : \zeta'' . y : \zeta', x : \zeta . z(\lambda u : \zeta . \overline{0}yx)(\overline{0}yx) \]

\[ \zeq \lambda z : \zeta'' . y : \zeta', x : \zeta . z(\lambda u : \zeta . x)x = (-2-) \]

and similarly
\[ D = \lambda w : \zeta''' . P(S(Lw))(Lw) \]
\[ \zeq \lambda w : \zeta''' . z : \zeta'' , y : \zeta', x : \zeta . z(\lambda u : \zeta . Lwyx)(S(Lw)yx) \]
\[ \zeq \lambda w : \zeta''' . z : \zeta'' , y : \zeta', x : \zeta . z(\lambda u : \zeta . \overline{0}yx)(y(w \overline{0}yx)) \]
\[ = \lambda w : \zeta''' . z : \zeta'' , y : \zeta', x : \zeta . z(-3b-)(-3a-) = (-3-) \]

which shows where the various components come from. With this we have
\[ d \zeq \lambda x : \zeta^{(5)} . x D(0, 0) \zeq \lambda x : \zeta^{(5)} . x (-3) (-2) \mathbf{T} = \overline{d} \]
to show what is going on.

There is a serious point here. The term \( d \) is not normal, but has some internal structure so we can see how it works. The term \( \overline{d} \) is the normal form of \( d \) but it is not easy to fathom out. There is a belief in some quarters that by moving towards a normal form we ‘improve’ a term, and make the encoded algorithm more efficient. That idea is patent nonsense.

This section has been a bit of an experiment. There are things going on here that I don’t understand. The use of explicit pairing gadgets seems to reduce the complexity of the reduction. Is there something more general here? I do not know of any detailed investigation of this feature.

[\textit{Held in 165../05-version../017.. Last changed April 13, 2006}]
In Section 16 we used the Kleene/Bernays trick to capture a rather trivial form of recursion. In fact, the trick can be used to handle many other forms of recursion. To illustrate this we will consider three more examples. To help bring out the underlying method we split the details into short segments, and do the three examples in unison. If you prefer you can read through the simplest one, case (a), first. However, some of the more intricate details are given only for the most complicated example, case (c), so you will need to fill in these yourself.

Later we will use case (a) to show that $S(\lambda A)$ is closed under two rather strange forms of recursion.

18.1 RUNNING EXAMPLES. Consider the following three examples of a recursive specification of a function $f$.

(a) $f0p = gp$ $f x'p = h xtp$ where $t = f xp$

(b) $f0p = gp$ $f x'p = h xtp$ where $t = f xq$ where $q = dxp$

(c) $f0p = gp$ $f x'p = h xtp$ where $t = f xq$ where $q = dsp$ $s = fx b$ $b = e(f xp)$

This function $f$ is constructed out of given data functions $g, h$ and, when they occur, $d, e$. In each case the recursion variable $x$ ranges over $\mathbb{N}$ (and $x' = x + 1$). The other input $p$ is a parameter.

I have deliberately left some of the details here a bit vague. In particular, I have not said what the parameter $p$ should be. I will say more about this aspect shortly.

Notice that in the recursion clause the ‘previous’ function $fx$ is called in the form

$$t = fx(d(fx(e(f xp))p))$$

with triple nesting. In truth, I have never seen a recursion of this form used in polite company. I have put it in here merely to illustrate how some very wild recursion can be re-organized as iterations.

In fact, it is not immediately obvious that the specification makes sense. Is there a unique total function $f$ that satisfies the specification? There is, and what we do in the section can be modified to prove this. However, these kind of questions are best answered using different techniques (domain theory) that are too far away from the topic of these notes. These we tacitly accept these each of (a, b, c) does specify a unique total function.

Of course, in all cases the types of the various functions must match correctly, and this imposes various restrictions on the functions. Let’s set down these restrictions.

18.2 RUNNING EXAMPLES. Let $\mathbb{P}$ be the space of parameters. Each of the three specifications produces a function $f : \mathbb{N} \to \mathbb{P} \to \mathbb{N}$ and we find that

(a) $g : \mathbb{P} \to \mathbb{N}$ $h : \mathbb{N} \to \mathbb{N} \to \mathbb{P} \to \mathbb{N}$

(b) $g : \mathbb{P} \to \mathbb{N}$ $h : \mathbb{N} \to \mathbb{N} \to \mathbb{P} \to \mathbb{N}$ $d : \mathbb{N} \to \mathbb{P} \to \mathbb{P}$

(c) $g : \mathbb{P} \to \mathbb{N}$ $h : \mathbb{N} \to \mathbb{N} \to \mathbb{P} \to \mathbb{N}$ $d : \mathbb{N} \to \mathbb{P} \to \mathbb{P}$ $e : \mathbb{N} \to \mathbb{P}$
are the required typing restrictions. It can help if we abbreviate these types. Thus let
\[ G = (\mathbb{P} \to \mathbb{N}) \]
so that
\[ f : \mathbb{N} \to G \]
with
\[
\begin{align*}
(a) \quad g : G & \quad h : \mathbb{N} \to \mathbb{N} \to G \\
(b) \quad g : G & \quad h : \mathbb{N} \to \mathbb{N} \to G \quad d : \mathbb{N} \to \mathbb{P}' \\
(c) \quad g : G & \quad h : \mathbb{N} \to \mathbb{N} \to G \quad d : \mathbb{N} \to \mathbb{P}' \quad e : \mathbb{N} \to \mathbb{P}
\end{align*}
\]
as the required typing restrictions.

For much of what we do the space \( \mathbb{P} \) of parameters can be anything. It could be a rather high order function space, and then even case (a) is a quite powerful form of recursion. However, to keep things simple we will tacitly assume that \( \mathbb{P} \) is a power of \( \mathbb{N} \), that is the parameter is a list of natural numbers. With this tacit restriction case (a) is just primitive recursion, and (b) is a form of primitive recursion with variation of parameters. Case (c) is a more intricate form of recursion for it allows both variation of the parameter and a nesting of the target function \( f \) within the recursion.

It can be useful to think of \( f \) as a 1-placed function from \( \mathbb{N} \) which returns a higher order gadget. In this way the parameter is hidden. The crucial step is to replace each of these recursions by an iteration. To do that we extend the method Kleene/Bernays trick, and produce two functions in parallel, one of which merely counts the number of steps.

18.3 RUNNING EXAMPLES. Using the data function \( h \) consider
\[ H : \mathbb{N} \times G \to \mathbb{N} \times G \]
given by
\[ H(x, \phi) = (x', \psi) \]
for each \( x \in \mathbb{N} \) and \( \phi \in G \) where for each \( p \in \mathbb{P} \)
\[
\begin{align*}
(a) \quad \psi p &= h x t p & \text{where} & \quad t = \phi p \\
(b) \quad \psi p &= h x t p & \text{where} & \quad t = \phi q & \text{where} & \quad q = d x p \\
(c) \quad \psi p &= h x t p & \text{where} & \quad t = \phi q & \text{where} & \quad q = d s p & s = \phi b & b = e(\phi p)
\end{align*}
\]
gives the appropriate \( \psi \) for each case. In particular, when \( \phi = f x \) we find that
\[ H(x, f x) = (x, f x') \]
so that, by iteration, we have
\[ H^x(0, g) = (x, f x) \]
from which we can extract \( f \).
We now turn to a more delicate question. How can we capture the constructions of Running Examples within $\mathbf{\lambda A}$? We will use simulated numerals of various types, and we assume given simulation of the data functions. The problem is to produce a simulation of $f$.

This is the reason for using product types and pairing gadgets in $\mathbf{\lambda A}$. With these facilities many forms of recursion can be replaced by iteration, and this is easier to organize and analyse. Without the explicit pairing facilities we have to resort to mimicking them as best we can. Sometimes, because of the particular circumstances, this can be done, as Example 13.3. However, in general we can not get at the required facilities.

Ignoring the space of parameters $P$, look at the number of uses on $N$ in the various types. We find there are

(a) 6  (b) 7  (a) 8

in the three cases. We try to simulate these different occurrences using different types. Of course, we don’t have a complete free hand (because of the compositions involved), but we do have some leeway. After a bit of trial and error we find that for these case the following will do.

18.4 NOTATION. Let $\zeta, \eta, \theta$ be three types. Eventually we will find that $\theta$ must be related to $\zeta, \eta$ in a certain way. We use simulated numerals over each of these types. Using too many subscripts in a term can make it look cluttered, so for each $m \in N$ we will write

\[ \bar{m} : \zeta'' \quad \bar{m} : \eta'' \quad \bar{m} : \theta'' \]

for the simulated numeral over $\zeta, \eta, \theta$, respectively.

We also need to mimic the parameters. We assume given some type $\pi$ and write

\[ \bar{p} : \pi \]

for the notated parameter. \[\square\]

We don’t say exactly how the notation $\bar{p}$ is formed, for this will depend on the particular situation. However, notice that $\bar{p}$ inhabits $\pi$ and not $\pi''$. In a case where a parameter is just a natural number we could have $\pi = \chi''$ for some $\chi$, and $\bar{p} = \bar{p}_\chi$. In the case where a parameter is a list of natural numbers we could have

\[ \pi = \chi(l)'' \times \cdots \times \chi(1)'' \]

for some component types, and then

\[ \bar{p} = \langle \bar{p}(l)_{\chi(l)}, \ldots, \bar{p}(1)_{\chi(1)} \rangle \]

where $\langle \cdots \rangle$ indicates some tupling construction built using the pairing gadgets. Fortunately, we don’t have to get inside $\bar{p}$.

18.5 RUNNING EXAMPLES. Let $\zeta, \eta, \pi$ be types and let

\[ \gamma = \pi \rightarrow \zeta'' \]
(to form a function type). We assume the data functions are simulated by terms

\[(a) \quad \exists : \bar{\gamma} : \eta'' \to \zeta'' \to \gamma \]
\[(b) \quad \exists : \bar{\gamma} : \eta'' \to \zeta'' \to \gamma \quad \exists : \eta'' \to \pi' \]
\[(c) \quad \exists : \bar{\gamma} : \eta'' \to \zeta'' \to \gamma \quad \exists : \zeta'' \to \pi' \quad \exists : \zeta'' \to \pi \]

respectively. Notice that in cases (b, c) the term \(\exists\) has different types. Thus we have

\[\exists \bar{\gamma} \exists \bar{\eta} \exists \bar{\pi} \quad \exists \bar{\eta} \exists \bar{\gamma} \exists \bar{\pi} \]

and

\[(b) \quad \exists \bar{\eta} \exists \bar{\gamma} \exists \bar{\pi} \quad (c) \quad \exists \bar{\eta} \exists \bar{\gamma} \exists \bar{\pi} \quad \exists \bar{\gamma} \exists \bar{\eta} \exists \bar{\pi} \]

for all \(m, n \in \mathbb{N}\) and \(p \in \mathbb{P}\).

Notice the difference between the given reductions for \(\exists\). This reflects its different type for the two cases.

As we go through the details we have to keep track of various terms, how they reduce, and what value (if any) they correspond to. We use a decoration \(\circ\) to indicate a general term. For instance, we may have a term \(\tilde{q}\) which reduces to a notation \(\bar{q}\) for some (real world) parameter \(q\). Of course, we will not be entirely rigid with this convention.

18.6 RUNNING EXAMPLES. As in Examples 18.5 we assume given types \(\zeta, \eta, \pi\) with associated terms. As there we set \(\gamma = (\pi \to \zeta'')\), but now we also set

\(\theta = \eta'' \times \gamma\)

to produce the type for the third kind of simulated numeral \(\bar{m}\) (as given in Notation 18.4). We use the pairing constants

\[\text{Pair} : \eta'' \to \gamma \to \theta \quad \text{Left} : \theta \to \eta'' \quad \text{Right} : \theta \to \gamma\]

associated with \(\theta\).

For a variable \(v : \theta\) we form terms

\[\bar{\bar{\gamma}} = \text{Left} v \quad \bar{\gamma} = \text{Right} v \quad \bar{\eta} = \lambda p : \pi . \bar{\eta} \bar{\eta} p \]

where, for a variable \(p : \pi\), the term \(\bar{\eta}\) is formed by

\[(a) \quad \bar{\eta} = \bar{\gamma} \bar{p} \]
\[(b) \quad \bar{\eta} = \bar{\gamma} \bar{q} \quad \text{where} \quad \bar{q} = \bar{d} \bar{\eta} \bar{p} \]
\[(c) \quad \bar{\eta} = \bar{\gamma} \bar{q} \quad \text{where} \quad \bar{q} = \bar{d} \bar{\eta} \bar{p} \quad \bar{s} = \bar{\phi} \bar{b} \quad \bar{b} = \bar{c}(\bar{\phi} \bar{p})\]

in each case.

Using these let

\[G = \text{Pair} \bar{\eta} \bar{\gamma} \quad H = \lambda v : \theta . \text{Pair}(\bar{S} \bar{\eta}) \bar{v} \]

where \(\bar{S} : \eta''\) is the term that simulates the successor function over \(\eta\). Finally, we set

\[\bar{f} = \lambda w : \theta'' , p : \pi . \text{Right}(wHGp)\]

to obtain the simulation of \(f\).
There are, of course, several things to be verified here. We need to check that all the component terms are at least correctly formed, and that \( \tilde{f} \) has the required reduction properties. The correctness of the terms could be verified by exhibiting several derivations (or more precisely, derivations templates, since we have to work relative to the assumed simulations of the data functions). We don’t need to go that far. We can argue informally, for most of the typing results are straightforward.

18.7 RUNNING EXAMPLES. For variable \( v : \theta, p : \pi \) we have

\[
\tilde{l} : \eta'' \quad \tilde{\phi} : \gamma
\]

and

\[
\begin{align*}
(a) & \quad \tilde{t} : \zeta'' \\
(b) & \quad \tilde{t} : \zeta'' \text{ since } \tilde{q} : \pi \\
(c) & \quad \tilde{t} : \zeta'' \text{ since } \tilde{q} : \pi \quad \tilde{s} : \zeta'' \quad \tilde{b} : \pi
\end{align*}
\]

where for case (c) we need to work from right to left. Let us check case (c).

We have

\[
v : \theta \quad p : \pi \quad \tilde{\phi} : (\pi \to \zeta'') \quad \tilde{e} : \zeta'' \to \pi
\]

so that \( \tilde{\phi}p : \zeta'' \) and \( \tilde{b} : \pi \) to give \( s : \zeta'' \) and \( \tilde{q} : \pi \) (by observing the type of \( d \)), and hence \( \tilde{t} : \zeta'' \), as required.

From these we have

\[
\tilde{l} : \eta'' \quad \tilde{\phi} : (\pi \to \zeta'') \quad \tilde{h} \tilde{l} p : \zeta''
\]

so that \( r : \gamma \).

Continuing, since \( \overline{S} : \eta''' \), we have \( \overline{SL} : \eta'' \) and hence

\[
\text{Pair} \overline{U} \overline{F} : \theta \quad \text{Pair} (\overline{SL}) \tilde{r} : \theta'
\]

to give

\[
G : \theta \quad H : \theta'
\]

for the auxiliary components.

Finally, for a variable \( w : \theta'' \) we have

\[
wHG : \theta \quad \text{Right}(wHG) : \gamma \quad \text{Right}(wHG)p : \zeta''
\]

so that

\[
\tilde{f} : \theta'' \to \pi \to \zeta''
\]

as is required, since \( \gamma = (\pi \to \zeta'') \).

This shows that all the terms are correctly formed. A less straightforward matter is to show that \( \tilde{f} \) has the required reduction properties. To help with this we use a sequence of intermediate terms.

18.8 RUNNING EXAMPLES. We generate a sequence \( (\tilde{\phi}_m \mid m \leq \omega) \) of terms by

\[
\tilde{\phi}_0 = g \quad \tilde{\phi}_m' = \lambda \tilde{p} : \pi \cdot \overline{h} \overline{m} \tilde{t} \tilde{p}
\]
where

\[(a) \quad \hat{t} = \hat{\phi}_m p \]
\[(b) \quad \hat{t} = \hat{\phi}_m \hat{q} \quad \text{where} \quad \hat{q} = d \hat{m} \hat{p} \]
\[(c) \quad \hat{t} = \hat{\phi}_m \hat{q} \quad \text{where} \quad \hat{q} = \hat{d} \hat{s} \hat{p} \quad \hat{s} = \hat{\phi}_m \hat{b} \quad \hat{b} = \hat{c}(\hat{\phi}_m p)\]

for the three cases. By an induction over \(m\) we check that

\[\hat{\phi}_m : \gamma\]

holds. The manipulations involved are similar to the ones occurring in Examples 18.7. ■

The idea is that for each \(m \in \mathbb{N}\) the term \(\hat{\phi}_m\) simulates the function \(f m \in \mathbb{G}\). We need to check this, and to show that the status of \(m\) can be raised from that of an external index to that of an internal input.

We give the details of the proof of the following only for case (c), which is the most intricate. You should produce for yourself the details for cases (a, b).

18.9 LEMMA. Consider the three Running Examples 18.1–18.8. For each \(m \in \mathbb{N}\) we have

\[\hat{\phi}_m \hat{p} \triangleright= f m \hat{p} \quad H^m G \triangleright= \text{Pair} \hat{\hat{\hat{\phi}}}_m\]

where, in the left-hand reduction, the parameter \(p\) is arbitrary.

Proof. As indicated above we give the details only for case (c). We deal with the two reductions separately.

For the left-hand reduction we proceed by induction over \(m\) with allowable variation of the parameter \(p\). Recall that, from Example 18.5, we have

\[\overline{\mathcal{G}} \hat{p} \triangleright= \overline{\mathcal{G}} \hat{p} \quad \overline{\mathcal{H}} \hat{m} \hat{n} \hat{p} \triangleright= \overline{\mathcal{H}} \hat{m} \hat{n} \hat{p} \quad \overline{\mathcal{D}} \hat{m} \hat{p} \triangleright= \overline{\mathcal{D}} \hat{m} \hat{p} \quad \overline{\mathcal{E}} \hat{m} \triangleright= \overline{\mathcal{E}} \hat{m}\]

for each \(m, n \in \mathbb{N}\) and parameter \(p \in \mathbb{P}\).

For the base case, \(m = 0\), we have

\[\phi_0 \hat{p} = \overline{\mathcal{G}} \hat{p} \triangleright= \overline{\mathcal{G}} \hat{p} = \mathcal{F} \hat{0} \hat{p}\]

as required.

For the induction step, \(m \mapsto m'\), we produce various terms \(\hat{v}\) each of which reduces to a simulation \(\overline{\hat{v}}\) or \(\overline{\overline{\hat{v}}}\) and corresponds to a value \(v\) in \(\mathbb{N}\) of \(\mathbb{P}\). These terms track the construction (c) of Examples 18.8.

In detail we have the following.

<table>
<thead>
<tr>
<th>term</th>
<th>simulation</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\hat{a} = \hat{\phi}_m \hat{p})</td>
<td>(\overline{\mathcal{G}} \hat{a} \triangleright= \overline{\mathcal{G}} \hat{a})</td>
<td>where (a = f m p)</td>
</tr>
<tr>
<td>(\hat{b} = \hat{c} \hat{a})</td>
<td>(\overline{\mathcal{E}} \hat{b} \triangleright= \overline{\mathcal{E}} \hat{b})</td>
<td>where (b = e a = e(f m p))</td>
</tr>
<tr>
<td>(\hat{s} = \hat{\phi}_m \hat{b})</td>
<td>(\overline{\mathcal{G}} \hat{s} \triangleright= \overline{\mathcal{G}} \hat{s})</td>
<td>where (s = f m b)</td>
</tr>
<tr>
<td>(\hat{q} = \overline{\mathcal{D}} \hat{s} \hat{p})</td>
<td>(\overline{\mathcal{D}} \hat{q} \triangleright= \overline{\mathcal{D}} \hat{q})</td>
<td>where (q = d s p)</td>
</tr>
<tr>
<td>(\hat{t} = \hat{\phi}_m \hat{q})</td>
<td>(\overline{\mathcal{G}} \hat{t} \triangleright= \overline{\mathcal{G}} \hat{t})</td>
<td>where (t = f m q)</td>
</tr>
</tbody>
</table>
There are three uses of the induction hypothesis here: the reduction in the top row, the right-most reduction in the third row, and the right-most reduction in the bottom row. These involve three different parameters, namely $p, b, q$. Using these we have

$$\tilde{\phi}_{m'} \tilde{p} \triangleright \tilde{h} \tilde{m} \tilde{t} \tilde{p} \triangleright \tilde{h} \tilde{m} \tilde{t} \tilde{p} \triangleright \text{hmt } p = \tilde{f} m' p$$

as required.

For the right-hand reduction we again proceed by induction over $m$, but this time there are no parameters involved, only a variable $p : \pi$. However, in the induction step there will be several pairs of terms, $\tilde{q}$ and $\tilde{\phi}$ for instance, where the former reduces to the latter.

For the base case, $m = 0$, we have

$$H^0 G = G = \text{Pair } \tilde{0} \tilde{0} = \text{Pair } \tilde{0} \tilde{\phi}_0$$

since $\tilde{\phi}_0 = \tilde{g}$. This is why the reflexive reduction occurs in the statement of the result.

For the induction step, $m \mapsto m'$, we have

$$H^{m'} G = H(H^m G) \triangleright H \tilde{v} \text{ where } \tilde{v} = \text{Pair } \tilde{m} \tilde{\phi}_m$$

by the induction hypothesis. We now start to reduce and unravel the compound term $H \tilde{v}$. For this it is useful to have in front of you the construction of $\tilde{r}$ as given in case (c) of Examples 18.6.

Firstly, we have

$$\tilde{l} = \text{Left } \tilde{v} \triangleright \tilde{m} \quad \tilde{\phi} = \text{Right } \tilde{v} \triangleright \tilde{\phi}_m$$

and hence

$$\tilde{S} \tilde{l} \triangleright \tilde{m}$$

holds. Next we partly reduce four terms. These depend on a variable $p : \pi$.

$$\tilde{a} = \tilde{\phi} p \triangleright \tilde{\phi}_m p = \tilde{a} \quad \text{(say)}$$
$$\tilde{b} = \tilde{c} \tilde{a} \triangleright \tilde{c} \tilde{a} = \tilde{b} \quad \text{(say)}$$
$$\tilde{s} = \tilde{\phi} \tilde{b} \triangleright \tilde{\phi}_m \tilde{b} = \tilde{s} \quad \text{(say)}$$
$$\tilde{q} = d \tilde{s} p \triangleright d \tilde{s} p = \tilde{q} \quad \text{(say)}$$
$$\tilde{t} = \tilde{\phi} \tilde{q} \triangleright \tilde{\phi}_m \tilde{q} = \tilde{t} \quad \text{(say)}$$

so that

$$\tilde{r} = \lambda p : \pi. \tilde{l} \tilde{l} p \triangleright \lambda p : \pi. \tilde{m} \tilde{t} p = \tilde{\phi}_{m'}$$

by inspecting the construction of $\tilde{\phi}_{m'}$. With these we have

$$H \tilde{v} \triangleright \text{Pair } (\tilde{S} \tilde{l}) \tilde{r} \triangleright \text{Pair } \tilde{m'} \tilde{\phi}_{m'}$$

as required.

Finally, we can pull all this together to obtain the simulation result.
**18.10 Theorem.** Consider an instance of a recursion as in Examples 18.1. Suppose the data functions are simulated by terms as in Examples 18.5. Then the specified function $f$ is simulated by the term $\tilde{f}$ as in Examples 18.6.

**Proof.** Consider any $m \in \mathbb{N}$ with corresponding numeral $\overline{m} : \theta''$. Then

$$\tilde{f} \overline{m} \Rightarrow \text{Right}(\overline{m}HG) \Rightarrow \text{Right}(H''G) \Rightarrow \text{Right}(\text{Pair} \overline{\phi}_m) \Rightarrow \overline{\phi}_m$$

where the third reduction uses Lemma 18.9. Thus, for each parameter $p$ with notation $\overline{p}$ we have

$$\tilde{f} \overline{m} \overline{p} \Rightarrow \overline{\phi}_m \overline{p} \Rightarrow \overline{f}m \overline{p}$$

by a second use of Lemma 18.9, to give the required result. $\blacksquare$

This result seems to say that any reasonable, or even unreasonable, form of recursion can be mimicked using simulated numerals. If you write down any recursion and then reorganize it as an iteration, you will find there is a corresponding simulation result. There is, of course, a catch. The colours of certain numerals must match; certain types must be the same.

In the construction of proof of Theorem 18.10 the data function

$$h : \mathbb{N} \to \mathbb{N} \to \mathbb{P} \to \mathbb{N}$$

must be simulated by a term

$$\overline{h} : \eta'' \to \zeta' \to \pi \to \zeta''$$

where the two types $\zeta$ are the same. This compatibility is needed to produce the sequence $(\phi_m \mid m < \omega)$.

I suggest that you attempt to mimic the construction of the stacking function $\mathbb{L}$ as a primitive recursive function but using simulated numerals. After a couple of steps the required typing restrictions fail. It is these extra requirements that tiering is designed to highlight.

Classes of simulated functions can be closed under many forms of recursion provided some tiering-like restrictions are met. As far as I know there has been no comprehensive investigation of just what these restrictions entail, but much information can be found in the literature.

I said earlier that $\mathfrak{S}(\mathfrak{A})$ is closed under a couple of rather strange looking recursions. By that I meant bounded sums and products. It turns out that this is a consequence Theorem 18.10 using a recursion that fits into kind (a). Before we look at the details of that particular case, let’s look at a more general closure property.

Consider the following specification of a function $f$.

$$f0p = \text{constant} \quad fx'p = lt(kxp) \quad \text{where } t = fxp$$

As before the recursion input $x$ ranges over $\mathbb{N}$ and $p$ is a parameter. The two given data function $k, l$ have types

$$k : \mathbb{N} \to \mathbb{P} \to \mathbb{N} \quad l : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

145
so that
\[ f : \mathbb{N} \to \mathbb{P} \to \mathbb{N} \]
is the type of the required function \( f \). We assume that \( l \) is a fairly simple kind of function that is fixed. The aim of the construction is to transform the function \( k \) into \( f \). In general \( \mathbb{P} \) could be any space of parameters, but here we assume it is \( \mathbb{N}^r \) for some \( r \). Thus the parameter \( p \) is a list of natural of length \( r \).

To organize this transformation \( k \mapsto f \) consider the functions

\[
g : \mathbb{P} \to \mathbb{N} \quad h : \mathbb{N} \to N \to \mathbb{P} \to \mathbb{N}
\]
given by

\[
gp = \text{constant} \quad hxtp = lt(kxp)
\]
for \( x, t : \mathbb{N} \) and \( p : \mathbb{P} \). Here the ‘constant’ is the one appearing in the specification of \( f \). With this we have

\[
f0p = gp \quad fx'p = hxtp \quad \text{where } t = fxp
\]
which is a recursion of kind (a) of the Running Examples. We capture the transformation using an application of Theorem 18.10. Incidentally, you might be wondering why we have a constant function for \( g \), other than something more general. The reason is in the proof of the following.

18.11 THEOREM. For a 2-placed function \( l \in \mathcal{E} \) and a \((1+r)\)-placed function \( k \in \mathcal{S}(\lambda A) \) consider the \((1+r)\)-placed function \( f \) specified above, where the constant is fixed and \( p \) is an arbitrary list from \( \mathbb{N} \) of length \( r \). Then \( f \in \mathcal{S}(\lambda A) \).

Proof. Consider any type \( \zeta \). We must produce a simulation of \( f \) with target \( \zeta'' \). Since \( l \in \mathcal{E} \) and \( k \in \mathcal{S}(\lambda A) \), there are terms

\[
\overline{l} : \zeta'' \to \zeta'' \to \zeta'' \quad \overline{k} : \eta'' \to \chi(r)'' \to \cdots \to \chi(1)'' \to \zeta''
\]
which simulate \( k \) and \( l \), respectively. Here \( \chi(1), \ldots, \chi(l), \eta \) are some types which depend on \( \zeta \) and \( k \). In particular, we have little control over what they can be. It is convenient to collapse the parameters types \( \chi(1), \ldots, \chi(l) \) to one. Thus let

\[
\pi = \chi(l)'' \times \cdots \chi(1)''
\]
so, with a bit a licence, we have

\[
\overline{k} : \eta'' \to \pi \to \zeta''
\]
as the simulating term. As above we may rephrase the specification as

\[
f0p = gp \quad fx'p = hxtp \quad \text{where } t = fxp
\]
where \( g \) is the appropriate constant function and

\[
hxtp = lt(kxp)
\]
for all \( x, t \in \mathbb{N} \) and parameters \( p \).
We construct terms

\[ \mathcal{g} : \pi \to \zeta'' \quad \mathcal{h} : \eta'' \to \zeta'' \to \pi \to \zeta'' \]

which simulate the functions \( g \) and \( h \). The construction of the term \( \mathcal{g} \) is trivial, and that of \( \mathcal{h} \) using the terms \( \mathcal{I} \) and \( \mathcal{K} \). Thus we may invoke Theorem 18.10.

The important point here is that in the construction \( \mathcal{I} \mapsto \hat{f} \) only the type of the active argument is changed. The types of the parameters and the target are unchanged. Thus we have a syntactic transformation

\[ \mathcal{I} : \eta'' \to \pi \to \zeta'' \quad \mapsto \quad \hat{f} : \theta'' \to \pi \to \zeta'' \]

where

\[ \theta = \eta'' \times (\pi \to \zeta'') \]

is the new source type.

Bounded sums and products provide particular application of Theorem 18.11. Consider any function

\[ k : \mathbb{N} \to \mathbb{P} \to \mathbb{N} \]

where, in practice, \( \mathbb{P} \) is a power of \( \mathbb{N} \) (as in Theorem 18.11). The two functions \( \sum[k] \) and \( \prod[k] \), both of the same type as \( k \), are defined by

\[ \sum[k]xp = \sum_{i<x} kip \quad \prod[k]xp = \prod_{i<x} kip \]

for \( x \in \mathbb{N}, p \in \mathbb{P} \). Here the index \( i \) ranges over \( \mathbb{N} \). Thus

\[ \sum[k]0p = 0 \quad \sum[k]xp = \sum[k]xp + kxp \]
\[ \prod[k]0p = 1 \quad \prod[k]xp = \prod[k]xp \times kxp \]

are recursive descriptions of the new functions. Let

\[ h : \mathbb{N} \to \mathbb{N} \to \mathbb{P} \to \mathbb{N} \]

be obtained from \( k \) by

\[ hxyyp = \begin{cases} y + kxp & \text{sum-case} \\ y \times kxp & \text{prod-case} \end{cases} \]

(for \( x, y \in \mathbb{N}, p \in \mathbb{P} \)). Then

\[ f0p = \text{appropriate constant} \quad fxp = hx(fxp)p \]

is the new function. We can handle this using Theorem 18.11.

18.12 THEOREM. Suppose the function

\[ k : \mathbb{N} \to \mathbb{P} \to \mathbb{N} \]

has a simulation

\[ \mathcal{K} : \eta'' \to \gamma \]

where \( \gamma = \pi \to \zeta'' \). Then both the functions \( \sum[k] \) and \( \prod[k] \) have simulations with type

\[ \theta'' \to \gamma \]

where \( \theta = (\eta'' \times \gamma) \).
Proof. Let \( l : \mathbb{N}^{(2)} \) be addition or multiplication. We have a simulation
\[
\vdash \overline{l} : \zeta'' \rightarrow \zeta'' \rightarrow \zeta''
\]
over \( \zeta \). Let \( g : \mathbb{P} \rightarrow \mathbb{N} \) be the appropriate constant function, and set
\[
hxyp = ly(kxp)
\]
so that
\[
f0p = gp \quad fx'p = hx(fxp)p
\]
generates the function of interest. Using the given simulation \( \overline{k} \) let
\[
\overline{h} = \lambda x : \eta'', y : \zeta'', p : \pi . ly(kxp)
\]
so that
\[
\vdash \overline{h} : \eta'' \rightarrow \zeta'' \rightarrow \gamma
\]
and \( \overline{h} \) simulates \( h \). The construction of the proof of Theorem 18.11 produces a simulation
\[
\vdash \overline{f} : \theta'' \rightarrow \gamma
\]
of \( f \), as required. \( \blacksquare \)

This gives us the ‘unusual’ closure property of simulation.

18.13 COROLLARY. For each tiered system \( \lambda T \) the clone \( \mathcal{S}(\lambda T) \) is closed under bounded sums and products.

We will return to this closure property in Section 21.

[Held in 165../0../019.. Last changed April 13, 2006]

19 Global upper bounds

Consider any tiered system \( \lambda T \) as described in Section 2 and illustrated in Section 6. This system has a monochrome atom \( \mathcal{N} \) and several coloured atoms \( \mathcal{N}[i] \). It also has several constants each with some colour coding. For instance, it may have an iterator
\[
1 : \mathcal{S} \rightarrow \mathcal{T}''
\]
where \( \mathcal{S} \) and \( \mathcal{T} \) are different coloured atoms.

Suppose we now switch off the colour, that is we replace each coloured atom \( \mathcal{N}[i] \) by \( \mathcal{N} \). This will cause some changes in the housing types of the constants. For instance, the iterator above becomes
\[
1 : \mathcal{N} \rightarrow \mathcal{N}''
\]
the iterator of \( \lambda F \). In this way, by switching off the colour, we convert \( \lambda T \) into a subsystem of \( \lambda G \). For instance, this process converts \( \lambda B \) into \( \lambda F \), and both \( \lambda C \) and \( \lambda L \) into \( \lambda G \). By the process each derivation in \( \lambda T \) becomes a derivation in \( \lambda G \). Thus
\[
\mathcal{R}(\lambda T) \subseteq \mathcal{R}(\lambda G)
\]
so that $\lambda G$ provides an upper bound for for anything we can do with this kind of tiered system.

What is this upper bound? Does the removal of the colour coding have any discernible increase in the strength of the system?

In this section we consider the first of these two questions. We show how to calibrate the strength of $\lambda G$ and certain of its subsystems. Essentially we look at a variant of the long Grzegorczyk hierarchy

$$\mathcal{E}_0 \subseteq \mathcal{E}_1 \subseteq \cdots \subseteq \mathcal{E}_\alpha \subseteq \cdots \subseteq \mathcal{E}_\varepsilon$$

indexed by the ordinals $\alpha \leq \varepsilon_0$. The strength of $\lambda G$ is measured by $\varepsilon_0$. Roughly speaking, the strength of each canonical subsystem is measured by some term of a natural fundamental sequence $\varepsilon[\cdot]$ for $\varepsilon_0$. We look at more precise details later in this section.

We need to remind ourselves how hierarchies of sets of functions can be organized. I will give a brief recap here. Further details can be found in [5], especially Chapters 7, 8, and 9. A wealth of information about this and related topics can be found in [1, 2].

We start with some ground function $g : \mathbb{N}'$ which is strictly inflationary and monotone, that is

$$x + 1 \leq g(x) \quad x \leq y \implies g(x) \leq g(y)$$

for $x, y \in \mathbb{N}$. It is convenient to refer to such a function as a snake. Eventually we take $g$ to be the successor function, but for the time being it is we work with $g$ as a parameter.

Given two snakes $g$ and $h$ we write $g \leq h$ for the pointwise comparison between $g$ and $h$. This comparison is a measure of the comparative rates of growth of the two snakes.

Trivially, the composite of two snakes is a snake. Thus, by iteration each snake $g$ generates an ascending chain

$$g \leq g^2 \leq g^3 \leq \cdots \leq g^r \leq \cdots (r < \omega)$$

of faster and faster snakes. In a moment we see how to produce a snake which is faster than all of these.

Each snake $g$ gives an upper bound for the rate of growth of a whole set of many placed function $\phi$. We use the notion of $\phi$ being eventually dominated by $g$. I need not give the precise definition of this since we hardly need it here, but we will look at the details in Section ??.

19.1 DEFINITION. For each snake $g$ let $\mathfrak{B}(g)$ be the set of functions each of which is (eventually) dominated by some iterate $g^{r+1}$.

It is convenient to refer to a set $\mathfrak{B}(g)$ as a basket of functions. Notice that each such basket is closed under composition. The word ‘eventually’ need not be included in the definition since any function that is eventually dominated by $g$ is totally dominated by some iterate of $g$.

We think of $g$ as a measure of the rate of growth of the members of $\mathfrak{B}(g)$. Of course, some members of $\mathfrak{B}(g)$ may have a slower rate of growth, but $g$ and all of its iterates are in $\mathfrak{B}(g)$. Observe that

$$g \leq h \implies \mathfrak{B}(g) \subseteq \mathfrak{B}(h)$$

for snakes $g, h$. In this way we organize all the ophidians into layers of rates of growth. Of course, different snakes $g, h$ can give the same basket, $\mathfrak{B}(g) = \mathfrak{B}(h)$. Since it is the
baskets we are interested in. We need not worry too much about the fine details of any particular snake.

Given a snake \( g \) how can we find a snake with a faster rate of growth, that is how can we jump out of \( \mathcal{B}(g) \)? Naturally, we use a jump operator. Here are the common examples of these.

19.2 DEFINITION. Let

\[
\begin{align*}
Ack & \quad Rob & \quad Grz \\
\end{align*}
\]

be the higher order functions of type \( \mathbb{N}'' \) given by

\[
\begin{align*}
Ack f x &= f^{x+1} x \\
Rob f x &= f^{x+1} 1 \\
Grz f x &= f^x 2
\end{align*}
\]

for each \( f : \mathbb{N}' \) and \( x \in \mathbb{N} \).

These are the jump operators devised by Ackermann, Robinson and Péter (separately), and Grzegorczyk.

Let \( G : \mathbb{N}'' \) be any such jump operator. It is not too hard to show that if \( g \) is a snake then so is \( Gg \) and

\[
\mathcal{B}(g) \subsetneq \mathcal{B}(Gg)
\]

holds. Thus each jump operator enables us to jump out of any basket, hence the name.

Thus starting from any ground snake \( g \) and a jump operator \( G \) we can produce an ascending chain

\[
\mathcal{B}(g) \subsetneq \mathcal{B}(Gg) \subsetneq \mathcal{B}(G^2g) \subsetneq \cdots \subsetneq \mathcal{B}(G^i g) \subsetneq \cdots \quad (i < \omega)
\]

of larger and larger baskets. We collect these into one big basket.

19.3 DEFINITION. For each snake \( g \) and jump operator \( G \) we set

\[
\mathcal{B}(G^* g) = \bigcup \{ \mathcal{B}(G^i g) \mid i < \omega \}
\]

to obtain a veritable bushel of functions.

The notation \( \mathcal{B}(G^* g) \) may look a bit awkward, but it will not hamper our analysis.

The obvious question ‘How do we jump out of \( \mathcal{B}(G^* g) \)?’ has an obvious answer. We find a more powerful jump operator. More precisely, we find a super-jump operator which jumps the jump. After that we find a super-duper jump operator, and so on.

19.4 EXAMPLE. Consider the increasingly higher order functions

\[
\begin{align*}
Ack_0 : \mathbb{N}^{(3)}, & \quad Ack_1 : \mathbb{N}^{(4)} \quad Ack_2 : \mathbb{N}^{(5)} \\
\end{align*}
\]

given by

\[
\begin{align*}
Ack_0 Gg x &= G^{x+1} g x \\
Ack_1 \phi Gg x &= \phi^{x+1} Gg x \\
Ack_2 \Phi Gg x &= \Phi^{x+1} \phi Gg x
\end{align*}
\]

for each \( g : \mathbb{N}', G : \mathbb{N}'', \phi : \mathbb{N}^{(3)}, \Phi : \mathbb{N}^{(4)} \).
It can be checked that for a snake $g$ and jump operator $G$

\[ Ack_0 G g \text{ jumps out of } \mathcal{B}(G^* g) = \bigcup \{ \mathcal{B}(G^i g) \mid i < \omega \} \]
\[ Ack_1 Ack_0 G g \text{ jumps out of } \mathcal{B}(Ack_0^* G g) = \bigcup \{ \mathcal{B}(Ack_0^i G g) \mid i < \omega \} \]
\[ Ack_2 Ack_1 Ack_0 G g \text{ jumps out of } \mathcal{B}(Ack_1^* Ack_0 G g) = \bigcup \{ \mathcal{B}(Ack_1^i Ack_0 G g) \mid i < \omega \} \]
together with the obvious extension of these leaps.

When I say ‘it can be checked that’ I am being a little glib. I know of no account of this kind of material where the details are set down in a neat and tidy fashion. This is something that ought to be done sometime.

Continuing with the notation of Example 19.4 we see that we have a multi-indexed hierarchy

\[ (((Ack_2^2 Ack_1)^{(1)} Ack_0)^{(0)} G)^i g \mid i(2), i(1), i(0), i < \omega \]

of snakes. This enables us to calibrate the rate of growth of many functions. Furthermore, there is an obvious way to extend this idea, but for that we need a more systematic notation. We also introduce another parameter which you may recognize.

19.5 DEFINITION. Let $\omega[x] : N'$. For each $s < \omega$ let

\[ \Omega_s : N^{(s+3)} \]

be given by

\[ \Omega_s F f f_s \cdots f_1 x = F^{\omega[x]} f f_s \cdots f_1 x \]

for each $F : N^{(s+2)}, f : N^{(s+1)}, f_s : N^{(s)}, \ldots, f : N', x \in N$.

Observe that the functions $f_s, \ldots, f_1$ play a passive role in this definition. Thus we may write

\[ \Omega_s F f f_s \cdots f_1 x = F^{\omega[x]} f f x \]

where $f$ abbreviates this list of functions. Notice that for $s = 0$ this list $f$ is empty.

As an example, with $\omega[x] = x + 1$ we have

\[ \Omega_0 = Ack_0, \Omega_1 = Ack_1, \Omega_2 = Ack_3 \]

and then $\Omega_3, \Omega_4, \ldots$ are the obvious extensions to these particular examples.

Using these functions $\Omega_s$ we can produce a much larger family of snakes. For instance we have

\[ (((\cdots (\Omega_s^{(i(s))} \Omega_{s-1}^{(i(s-1))} \cdots \Omega_0^{(i(0))})^i g \mid i(2), i(1), i(0), i \in N \]

for each snake $g$, jump operator $G$, and $i(s), \ldots, i(0), i \in N$. These will enable us to calibrate the rates of growth of a much larger family of functions. However, there are a few of observations we should make.

Notice that the base snake $g$ and the jump operator $G$ don’t get too involved in the process. Essentially what we do is produce many leap operators

\[ (\cdots (\Omega_s^{(i(s))} \Omega_{s-1}^{(i(s-1))} \cdots \Omega_0^{(i(0))})^i \]

which can be used to accelerate any jump operator $G$. It is an analysis of these leap functions that is important, and for that we don’t need to know what $G$ is.
Starting from an arbitrary function $\omega[\cdot] : \mathbb{N}'$ we have set up a family of functions

$$\Omega_s : \mathbb{N}^{(s+2)}$$

one for each $s < \omega$. Various legal combinations of these produce a whole battery of leap operators

$$A : \mathbb{N}^{(s+2)}$$
on various level. As above, these leap operators are closed under legal composition and application. They are also closed under another, slightly unusual, construction.

19.6 DEFINITION. For each $s \lt \omega$ and operators

$$B, C : \mathbb{N}^{(s+2)}$$
the sum

$$C \oplus B : \mathbb{N}^{(s+2)}$$
is the operator given by

$$(C \oplus B)F = CF \circ BF$$
for each $F : \mathbb{N}^{(s+2)}$.

Let $\omega[\cdot] : \mathbb{N}'$. The family of leap operators induced by $\omega[\cdot]$ is the least family which contains $\Omega_s$ for each $s$ and is closed under composition, applications and summation.

This unusual compound $C \oplus B$ will be explained later in this section.

Suppose we have one of these leap operators

$$A : \mathbb{N}'''$$
where, of course, this may be built up using leap operators at higher levels. Then for each base snake $g : \mathbb{N}'$ and a jump function $G : \mathbb{N}''$ we have a snake

$$AGg$$
and an ascending chain

$$\mathcal{B}(g) \subseteq \mathcal{B}(AGg) \subseteq \mathcal{B}((AG)^2g) \subseteq \cdots \subseteq \mathcal{B}((AG)^ig) \subseteq \cdots \subseteq \mathcal{B}((AG)^*g) \quad (i < \omega)$$
of baskets. These generated snakes are the test functions against which we may calibrate the rate of growth of a large family of functions by determining in which of the baskets a given function lives.

There is, however, a problem with this. The comparisons within the battery of leap operators seem to be rather intricate. Wouldn’t it be much nicer it was linearly ordered.

We are going to show that a cofinal part of this battery can be linearly ordered by matching it against an initial section of the ordinals.

To do this we need a brief refresher of the ordinals.

Each ordinal is either 0, a successor ordinal $\beta + 1$, or a limit ordinal $\mu$. In particular $\omega$ is the least infinite ordinal and the least limit ordinal (unless you are one of those silly people who believe that 0 is a limit ordinal). We have operations

$$\gamma, \beta \rightarrow \gamma + \beta \quad \gamma, \beta \rightarrow \gamma \cdot \beta \quad \beta \rightarrow \omega^\beta$$
of addition, multiplication, and exponentiation to base \( \omega \). Using this last operation let

\[
\epsilon[0] = \omega \quad \epsilon[r + 1] = \omega^\epsilon[r]
\]

for each \( r < \omega \) to generate an ascending chain

\[
\epsilon[0] = \omega, \quad \epsilon[1] = \omega^\omega, \quad \epsilon[2] = \omega^{\omega^\omega}, \quad \ldots
\]

of limit ordinals. Let

\[
\epsilon_0 = \bigvee \{\epsilon[r] \mid r < \omega\}
\]

be the limit of this chain. This is the ordinal we mentioned above. It is the least critical ordinal. In these notes we never need to go beyond \( \epsilon_0 \).

In general we have

\[
\theta \leq \omega^\theta
\]

for each ordinal \( \theta \), and this comparison is strict for ‘most’ ordinals. The ordinal \( \epsilon_0 \) is the least ordinal \( \theta \) with \( \theta = \omega^\theta \). Thus we have

\[
\alpha < \omega^\alpha < \epsilon_0
\]

for each ordinal \( \alpha < \epsilon_0 \).

Each such (non-zero) ordinal has a unique decomposition

\[
\alpha = \omega^{\alpha(a)} + \cdots + \omega^{\alpha(0)}
\]

with

\[
\alpha > \alpha(a) \geq \cdots \geq \alpha(0)
\]

where the first comparison is strict but the other need not be. (It is more common to collect together those components with the same exponent, but here it is convenient not to do this.) By repeated use of this decomposition, that is by decomposing the exponents that are generated, we find that each \( \alpha < \epsilon_0 \) is a particular kind of finite rooted tree.

Each limit ordinal \( \mu \) is the supremum of all those \( \beta < \mu \). To handle \( \mu \) we select a fundamental sequence, a function

\[
\mu[\cdot] : \mathbb{N} \rightarrow [0, \mu)
\]

which approaches \( \mu \) from below.

There are several ways to select such a fundamental sequence, but here we will consider just one method. However, it should be remembered that some results are quite sensitive to the way fundamental sequences are selected.

Each limit ordinal below \( \epsilon_0 \) has one of three shapes

\[
\omega^{\alpha+1} \quad \omega^\mu \quad \zeta + \eta
\]

for some strictly smaller ordinal \( \alpha \), strictly smaller limit ordinal \( \mu \), or strictly smaller ordinals \( \zeta, \eta \) with \( \eta = \omega^\beta \) for some \( \beta \). We use these cases to generate fundamental sequences.
19.7 DEFINITION. Let \( \omega[: N] \) be some given function. The canonical system of fundamental sequences induced by \( \omega[:] \) is generated by

\[
\omega^{\alpha+1}[x] = \omega^\alpha \cdot \omega[x] \quad \omega^{\mu}[x] = \omega^{\mu[x]} \\
(\zeta + \eta)[x] = \zeta + \eta[x]
\]

for each ordinal \( \alpha \), limit ordinal \( \mu \), appropriate limit ordinals \( \zeta, \eta \), and each \( x \in \mathbb{N} \). ■

The particular case \( \alpha = 0 \) of the first clause shows that the given function \( \omega[:] \) is selected as the fundamental sequence of \( \omega \) (hence the notation). This means that we should require \( \omega[:] \) to have certain properties, such as increasing without limit. Again we need not worry about these details here. The most common choice for \( \omega[:] \) is the successor function, but there are others. In particular, \( 2^* \) has its uses. Naturally, the complexity of \( \omega[:] \) will have an effect on the complexity of any gadget generated using these fundamental sequences.

You should note what we have done here. We are not using the raw ordinals below \( \epsilon_0 \), but some system of notations for these ordinals. A change in the way these notations are generated can have a dramatic effect on the complexity of any gadget we produce. I won’t labour this point. If you know about these things, then you will know what to do. If you don’t know about these things, then it is something you should learned about in due course.

With this fixed selection of fundamental sequences we convert each ordinal (below \( \epsilon_0 \)) into a whole family of functions.

19.8 DEFINITION. For each \( s < \omega \) the ordinal indexed family of functions

\[
\{ | - | \}_s : \mathbb{N}^{(s+3)}
\]

is generated by

\[
\{0\}_s F f = f \quad \{\alpha + 1\}_s F f = F(\{\alpha\}_s F f) \quad \{\mu\}_s F f x = \{\mu[x]\}_s F f x
\]

for each \( F : \mathbb{N}^{(s+2)}, f : \mathbb{N}^{(s+1)}, f^s : \mathbb{N}^{(s)}, \ldots, f_1 : \mathbb{N}^{'}, x \in \mathbb{N} \), ordinal \( \alpha < \epsilon_0 \), and limit ordinal \( \mu < \epsilon_0 \) with \( \mu[:] \) as its selected fundamental sequence. As earlier, the list \( f_s, \ldots, f_1 \) of passive functions has been abbreviated to \( f \). ■

For instance, we find that

\[
\{m\}_s F = F^m
\]

for each \( m \in \mathbb{N} \). Thus we have

\[
\{\omega\}_s F f x = \{\omega[x]\}_s F f x = F^{\omega[x]} f f x = \Omega_s F f x
\]

to give the following

19.9 LEMMA. We have

\[
\{\omega\}_s = \Omega_s
\]

for each \( s < \omega \).
We need to know how these functions fit together. A proof of the following is not too hard.

19.10 LEMMA. For each \( s < \omega \) we have

\[
\begin{align*}
(i) & \quad \{ \beta + \gamma \}_s F = \{ \gamma \}_s F \circ \{ \beta \}_s F \\
(ii) & \quad \{ \omega^\gamma \cdot m \}_s F = (\{ \omega^\alpha \}_s F)^m \\
(iii) & \quad \{ \omega^{\beta+1} \}_s = \{ \omega \}_s \circ \{ \omega^\beta \}_s \\
(iv) & \quad \{ \omega^\beta \}_s = \{ \beta \}^s \{ \omega \}_s \\
\end{align*}
\]

for each ordinal \( \beta \), limit ordinal \( \mu \), \( m \in \mathbb{N} \), and each \( F : \mathbb{N}^{(s+2)} \).

Strictly speaking the part (i) of this result is not does not hold in this generality. The two ordinal \( \beta \) and \( \gamma \) are required to mesh correctly. If you know what this means then you will know how to handle the identity. If you don’t know what this means it doesn’t matter too much.

Part (iv) of this result gives

\[
\{ \omega^\beta \}_s = \{ \beta \}^s \Omega_s
\]

and this enables us to handle all legal combinations of the \( \Omega_s \) leap functions.

19.11 EXAMPLE. Consider leap function

\[
(((\Omega^{l}_{s+3}\Omega_{s+2})^k\Omega_{s+1})^j\Omega_s)^i
\]

for \( i, j, k, l \in \mathbb{N} \), and \( s < \omega \).

For each \( m \in \mathbb{N} \) and \( s < \omega \) we have

\[
\Omega^m_{s+1}\Omega_s = \{ \omega \}_s \{ \omega \}_s = \{ \omega \cdot m \}_s = \{ \omega \}_s
\]

so that

\[
((\Omega^{l}_{s+3}\Omega_{s+2})^k\Omega_{s+1})^j\Omega_s)^i = (\{ \omega^\omega^{\omega^{\omega^{\cdot \cdot \cdot}} \cdot l} \}_s k\Omega_{s+1})^j\Omega_s)^i
\]

\[
= (\{ \omega^\omega^{\omega^{\omega^{\cdot \cdot \cdot}} \cdot k} \}_s \Omega_{s+1})^j\Omega_s)^i
\]

\[
= (\{ \omega^\omega^{\omega^{\omega^{\cdot \cdot \cdot}} \cdot k \cdot j} \}_s \Omega_{s+1})^i = \{ \alpha \}_s
\]

where

\[
\alpha = \omega^\omega^{\omega^{\omega^{\cdot \cdot \cdot}} \cdot k \cdot j \cdot i}
\]

is the final value. ■

This example indicates that the leap operators built up be means of composition and application corresponding to those ordinals built up by iterated exponentiation. There are two missing compounds, one on each side.

19.12 EXAMPLE. For each \( s < \omega \) suppose have a pair of leap operators

\[
B = \{ \beta \}_s \quad C = \{ \gamma \}_s
\]
in $\mathbb{N}^{(s+2)}$. Then for each $F : \mathbb{N}^{(s+2)}$ we have

$$(C \oplus B)F = CF \circ BF = \{\gamma\}_s F \circ \{\beta\}_s F = \{\beta + \gamma\}_s F$$

using Lemma 19.10(i), and hence

$$(C \oplus B) = \{\beta + \gamma\}_s$$

holds, to explain the $\oplus$ operation. \[\square\]

As with Lemma 19.10(i) itself, this example has to be treated with some care. This is because the meshing properties in the sum $\gamma + \beta$. For instance, suppose

$$B = \{\omega\}_s \quad C = \{\omega^2\}_s$$

so that the example appears to show

$$(C \oplus B) = \{\omega + \omega^2\}_s = \{\omega^2\}_s = C$$

since ordinal arithmetic gives $\omega + \omega^2 = \omega^2$. This ordinal arithmetic is correct, but $C \oplus B$ is definitely not just $B$.

19.13 EXAMPLE. We show that

$$\{\omega^2\}_0 \oplus \{\omega\}_0 \neq \{\omega^2\}_0$$

by a particular example.

Consider that case where $\omega[x] = x$, and let $G = Grz$, Then for each snake $g$ and $m \in \mathbb{N}$ we have

$$\{\omega\}_0 Ggm = \{\omega[m]\}_0 Ggm = \{m\}_0 Ggm = G^m gm$$

with

$$\{\omega\}_0 Gg1 = Gg1 = g2$$

as a particular case. With this we have

$$\{\omega^2\}_0 Gg1 = \{\omega^2[1]\}_0 Gg1 = \{\omega \cdot 1\}_0 Gg1 = \{\omega\}_0 Gg1 = g2$$

with

$$\{\omega^2\}_0 GS1 = S2 = 3$$

as a particular case. These give

$$(\{\omega^2\}_0 \oplus \{\omega\}_0)GS1 = (\{\omega^2\}_0 G \circ \{\omega\}_0 G)S1$$

$$= \{\omega^2\}_0 G(\{\omega\}_0 GS)1$$

$$= \{\omega\}_0 GS2$$

$$= G^2 S2$$

$$= G(GS)2$$

$$= (GS)^2$$

$$= GS(GS2)$$

$$= GS(S^2)$$

$$= GS4$$

$$= S^4 2 = 6$$
so that 
\[(\omega^2)_0 \oplus (\omega)_0)S1 = 6 \neq 3 = (\omega^2)_0GS1\]
to illustrate the difference. Bigger inputs bring about much larger discrepancies. You
might like to amuse yourself for an hour or two by computing \((\omega^2)_0 \oplus (\omega)_0)S2\) and
\((\omega^2)_0GS2\).

The explanation of this quandry is that we are not dealing with ordinals here, but with
notations for ordinals, and one ordinal can have several different notations. Furthermore,
we are dealing with notations for gadgets that are more general than ordinals, namely
iterations. However, we do not need to go into that here, for the ordinals do form a cofinal
collection of the more general gadgets.

The ideas of Definition 19.8 can be set up in a different way.

19.14 DEFINITION. For each \(s < \omega\) and \(F : \mathbb{N}^{(s+1)}\) the ordinal iterates

\[ F^\alpha\ (\alpha < \epsilon_0) \]
are generated by

\[ F^0 f = f \quad F^{\alpha+1} f = F(F^\alpha f) \quad F^\mu ff x = F^\mu [f] ff x \]
for each ordinal \(\alpha < \epsilon_0\), and limit ordinal \(\mu < \epsilon_0\) with \(\mu[\cdot]\) as its selected fundamental
sequence, and each \(f : \mathbb{N}^{(s+1)}, f^s : \mathbb{N}^s, \ldots, f_1 : \mathbb{N}', x \in \mathbb{N}\), where \(f\) abbreviates the list
\(f_s, \ldots, f_1\) of passive functions. ■

In other words we have

\[ F^\alpha = \{\alpha\}SF \]
for each \(s < \omega, \alpha < \epsilon_0\) and \(F : \mathbb{N}^{(s+1)}\). The notation on the left is the more common
one, but the notation on the right is better suited for what we do here. Of course, as we
have seen, the phrase 'ordinal iterate' is wrong, for the construct \(F^\alpha\) is sensitive to the
notation for \(\alpha\).

We also have to be careful with the arithmetic of these iterates. The equality

\[ F^\gamma \circ F^\beta = F^{\beta+\gamma} \]
holds only when \(\gamma\) and \(\beta\) mesh together in an appropriate fashion.

In general, for all reasonable base snakes \(g\) and jump operators \(G\) we have

\[ \mathfrak{B}(G^i g) \subseteq \mathfrak{B}(G^\omega g) \]

for each \(i < \omega\) and hence

\[ \mathfrak{B}(G^* g) \subseteq \mathfrak{B}(G^\omega g) \]
holds. However, in general these two baskets are different, since \(G^\omega g\) is not dominated by
any \(G^i g\) for \(i < \omega\).

Let us now turn to the system \(\Lambda G\) and its selected subsystems.
19.15 DEFINITION. For $r < \omega$ let $\mathbf{F}(r)$ be the monochrome system with iterators $I_s : \mathcal{N} \rightarrow \mathcal{N}^{(s+2)}$ for all $s \leq r$ and no other iterators.

For $r < \omega$ let $\mathbf{G}(r)$ be the monochrome system with iterators $I_{\tau} : \mathcal{N} \rightarrow \tau''$ for all $\tau \in L(r)$ and no other iterators.

Since $L(0) = \{\mathcal{N}\}$ we have $\mathbf{G}(0) = \mathbf{F}(0) = \mathbf{G}(0)$ at the bottom level. Since $\mathcal{N}^{(s)} \in L(s)$ for each $s < \omega$ we have $\mathbf{F}(r) \subseteq \mathbf{G}(r)$ for each $r < \omega$ with equality only for $r = 0$. These construction give us two ascending chains

$$\mathbf{G}(0) \subseteq \mathbf{G}(1) \subseteq \cdots \subseteq \mathbf{G}(r) \subseteq \cdots \quad (r < \omega)$$

with

$$\mathbf{G}(\omega) = \bigcup \{\mathbf{G}(r) \mid r < \omega\} = \mathbf{G}$$

and where

$$\mathbf{F}(\omega) = \bigcup \{\mathbf{F}(r) \mid r < \omega\}$$

seems to be a substantial part of $\mathbf{G}$.

We will show how these two ascending chains correspond in a certain way to the fundamental sequence $\epsilon[\cdot]$ of $\epsilon_0$.

Before we get to the array of leap operators, we need to consider the base snake and the jump operator.

Usually we take the base snake $g$ to be the successor functions. This is named in $\mathbf{A}$ by the constant $S$. Sometimes, for technical reasons, we require $g$ to be something like $2^*$. This can not be named in $\mathbf{A}$, but it can in $\mathbf{F}(0)$. For what we do here it doesn’t really matter what the base snake is, provided it is not too complicated.

19.16 ASSUMPTION. We assume the base snake $g$ is named in a ‘relatively small’ part of $\mathbf{F}$.

As we will see, the system $\mathbf{F}$ can name snakes which grow far too quickly to be used as a base snake. We want to exclude these. Under normal circumstances a base snake should belong to the class $\Sigma(1)$ of 1-latent functions.

What about the jump operators $G$? The three canonical examples are typical.

19.17 LEMMA. Let $l = l_0$. The terms

$$\text{Ack} \quad \lambda f : \mathcal{N}', x : \mathcal{N}, l(Sx)f \ x$$

$$\text{Rob} \quad \lambda f : \mathcal{N}', x : \mathcal{N}, l(Sx)f \ 1$$

$$\text{Grz} \quad \lambda f : \mathcal{N}', x : \mathcal{N}, l \ x \ f \ 2$$

name Ack, Rob, and Grz in $\mathbf{F}(0)$. 

158
Notice that again we use a rather small part of $\lambda F(0)$. To construct names for the operators

$$\{[\alpha]\}_s : \mathbb{N}^{(s+3)}$$

we proceed by recursion over the canonical decomposition

$$\alpha = \omega^{\alpha(a)} + \cdots + \omega^{\alpha(0)}$$

of $\alpha$. The exponents are strictly smaller which allows the recursion to go through. However each exponent lives at a 'higher level', which is way we need the parameter $s$. As the recursion proceeds we allow the parameter $s$ to vary.

To begin we must name each base operator

$$\{[\omega]\}_s = \Omega_s$$

which lives in $\mathbb{N}^{(s+3)}$. These operators make use of a selected fundamental sequence $\omega[\cdot]$ of $\omega$. We need a name for that function.

19.18 ASSUMPTION. We assume the function $\omega[\cdot] : \mathbb{N}'$ is named by a term $\omega : \mathcal{N}'$ is a 'relatively small' part of $\lambda F$. [proved]

Again I won’t say precisely what ‘relatively small’ means, but we don’t want the function $\omega[\cdot]$ to be too complicated.

With this term $\omega$ the operators $\{[\alpha]\}_s$ are easy to name.

19.19 LEMMA. For each $s < \omega$, the term

$$\omega_s = \lambda F : \mathcal{N}^{(s+2)}, f : \mathcal{N}^{(s+1)}, f_s : \mathcal{N}^s, \ldots, f_1 : \mathcal{N}', x : \mathcal{N} \cdot l_{s+1}(\omega x) F f f x$$

does a good job at naming the function $\{[\omega]\}_s$ in $\lambda F(s+1)$.

In the body of the term I have taken the liberty of abbreviating the sequence $f_s \cdots f_1$ to $f$.

With this we can convert the notation for each $\alpha < \epsilon_0$ into names for $\{[\alpha]\}_s$ for each $s$.

19.20 THEOREM. For each $r, s < \omega$, and each $\alpha < \epsilon[r]$, the operator $\{[\alpha]\}_s$ is named in $\lambda F(r+s)$.

Proof. We proceed by induction over $r$ with variation of $s$.

For the base case, $r = 0$, each $\alpha < \epsilon[0] = \omega$ is a natural number $m$. Thus

$$m_s = \lambda F : \mathcal{N}^{(s+2)}, f : \mathcal{N}^{(s+1)}, f_s : \mathcal{N}^s, \ldots, f_1 : \mathcal{N}', x : \mathcal{N} \cdot F^m f f x$$

is an appropriate name. Notice that this is a term of $\lambda A$.

For the induction step, $r \mapsto r + 1$, suppose

$$\alpha < \epsilon[r + 1] = \omega^{\epsilon[r]}$$

so that

$$\alpha = \omega^{\alpha(a)} + \cdots + \omega^{\alpha(0)}$$

is a good job at naming the function $\{[\alpha]\}_s$ in $\lambda F(r+s)$.
where each \( \alpha(i) < \epsilon[r] \). We build up a name for \( \{\alpha\}_s \) bit by bit.

Consider an ordinal \( \omega^\beta \) where \( \beta < \epsilon[r] \). We have

\[
\{\omega^\beta\}_s = \{\beta\}_s\{\omega\}_s
\]

by Lemma 19.10(iv). By the induction hypothesis we have a name

\[ \beta_{s+1} \]

in \( \lambda F(r + s + 1) \) for \( \{\beta\}_{s+1} \). Also, by Lemma 19.19, we have a name

\[ \omega_s \]

in \( \lambda F(s + 1) \) for \( \{\omega\}_s \). Thus the compound

\[ \beta_{s+1}\omega_s \]

is a name in \( \lambda F(r + s + 1) \) for \( \{\omega^\beta\}_s \).

It remains to deal with the sum

\[ \alpha = \beta + \gamma \]

of two ordinals where these do mesh in the appropriate fashion. We may suppose we have already constructed names \( \beta_s, \gamma_s \) in some system of the two components. Using Lemma 19.10(i) we see that

\[
\lambda F : \mathcal{N}(s+2), f : \mathcal{N}(s+1) \cdot \gamma_s F(\beta_s Ff)
\]

denames \( \{\alpha\}_s \) in the same system.

Each of these terms \( \alpha_s \) van be reduced to a normal form

\[
\lambda F : \mathcal{N}(s+2), f : \mathcal{N}(s+1) \cdot \langle \alpha \rangle_s
\]

where the abstractions in \( \langle \alpha \rangle_s \) can be highly nested. In some ways this nesting reflects this complexity of the notation used for \( \alpha \).

This result shows why the system \( \lambda F(r) \) are useful. For \( r \neq 0 \) the system \( \lambda F(r) \) has just enough iterators to name \( \{\alpha\}_0 \) for all \( \alpha < \epsilon[r] \). There is a slight hiccough at the start, since each natural number can be named in \( \lambda A \) which is smaller than \( \lambda F(0) \).

What about the systems \( \lambda G(r) \)? I believe the following is true.

19.21 THEOREM. We have

\[
\mathcal{M}(\lambda G(r)) \subset \mathcal{B}(G^{\epsilon[r]}S)
\]

for each \( r < \omega \). Here \( G \) is any one of the standard jump operators.

Notice that the inclusion is strict. In particular, we see that the function \( G^{\epsilon[r]}S \) is not named in \( \lambda G(r) \).

19.22 COROLLARY. The function \( \Uparrow : \mathbb{N} \) given by

\[
\Uparrow(x) = G^x Sx
\]

can not be named in \( \lambda G \).

This function \( \Uparrow \) eventually dominates every function we can name in \( \lambda G \). Note, however, that \( \Uparrow \) has a tremendous rate of growth. With \( \omega[^x] = S \) we find that \( \Uparrow(0) \leq 3 \), but it is impossible to write down the decimal expansion of even \( \Uparrow(1) \).
In Section 3 we used an informal version of the orthodox semantics of a tiered system (and, more generally, of an applied λ-calculus). But, as we saw there, when we start to get into the nitty-gritty of this informal version, we find that some things are not quite right. In this section I give an account that addresses those problems (without being the final word on the subject).

A full account of the orthodox semantics would be rather dreary but, judging by some of the stuff out there, it ought to be done properly sometime. I don’t claim that this is a full account, but it may help to clear up some of the confusion.

One of the main problems of the informal account is that it doesn’t handle the empty context very well, it fudges the issue. (Perhaps this is because it is difficult to handle something that isn’t there.)

Each derivation

\((\nabla)\quad \Gamma \vdash t : \tau\)

should give a function

\([\Gamma] \xrightarrow{[\nabla]} [\tau] \)

where the source and target come from the context and predicate. In particular, a derivation

\((\emptyset)\quad \vdash t : \tau\)

in the empty context should give a function

\(\text{??} \xrightarrow{[\emptyset]} [\tau] \)

from some source. The informal version of the semantics views \([\emptyset]\) as a member of \([\tau]\), not a function to \([\tau]\). How can we reconcile these different views?

Consider any set \(T\) (thought of as \([\tau]\)). Let

\(1 = \{\varnothing\}\)

be a 1-element set. It doesn’t matter what \(\varnothing\) is, but we don’t want to confuse it with anything else. We give \(\varnothing\) a name and call it the tag. It is a trivial observation that the two sets

\[
\begin{array}{c}
T \\
\hline
1 \\
\hline
T
\end{array}
\]
are in bijective correspondence. Each \( t \in T \) corresponds to the function
\[
t^*: \mathbb{1} \longrightarrow T
\]
where
\[
t^*(\emptyset) = t
\]
is the only value. Every function
\[
f: \mathbb{1} \longrightarrow T
\]
as the form \( f = t^* \) for a unique \( t \in T \), namely \( t = f(\emptyset) \).

To sort out this problem we need an explicit name for the empty context. We use \( \emptyset \) for the empty context (and, of course, this is not the same as \( \emptyset \), the empty set).

Each derivation
\[(\emptyset, \emptyset \vdash t : \tau)\]
in the empty context should give a function
\[
[\emptyset] \xrightarrow{[\emptyset]} [\tau]
\]
so we must decide what \([\emptyset]\) is.

**23.1 DEFINITION.** The meaning of a context is generated by recursion over its length, by
\[
[\emptyset] = \mathbb{1} = \{\emptyset\} \quad [\Gamma, x : \sigma] = [\Gamma] \times [\sigma]
\]
where \( \emptyset \) is the empty context and \( \emptyset \) is the tag. ■

Consider a context
\[
\Gamma = x_l : \sigma, \ldots, x_1 : \sigma_l
\]
of length \( l \). Let
\[
S = [\sigma_i]
\]
for each \( 1 \leq i \leq l \). Then we have
\[
[\Gamma] = (\cdots (\mathbb{1} \times S_l) \times \cdots \times S_1)
\]
according to Definition 23.1 whereas we had
\[
[\Gamma] = S_l \times \cdots \times S_1
\]
in the informal semantics. It is not the use of punctuation brackets that matter here, but the use of \( \mathbb{1} \) to hang the product on. In fact, there is no harm in saying
\[
[\Gamma] = \mathbb{1} \times S_l \times \cdots \times S_1
\]
is the official version of the semantics.

The two function spaces
\[
S_l \times \cdots \times S_1 \longrightarrow T \quad \mathbb{1} \times S_l \times \cdots \times S_1 \longrightarrow T
\]
are in bijective correspondence. Each function

\[ f : S_l \times \cdots \times S_1 \to T \]
corresponds to the function

\[ f^\wedge : 1 \times S_l \times \cdots \times S_1 \to T \]
given by

\[ f^\wedge (\varnothing, s_l, \ldots, s_1) = f(s_l, \ldots, s_1) \]
for \( s_1 \in S_1, \ldots, s_l \in S_l \). Every function of type

\[ 1 \times S_l \times \cdots \times S_1 \to T \]
is of the form \( f^\wedge \) for a unique function \( f \). When \( l = 0 \) we retrieve the simple case described above.

Carrying around this tag \( \varnothing \) looks a bit of a nuisance, but it does come in useful in extreme circumstances.

Each derivation

\[ (\emptyset) \quad \emptyset \vdash t : \tau \]
in the empty context gives a function

\[ 1 \xrightarrow{[\emptyset]} [\tau] \]
and this has the form

\[ [\emptyset] = t^\wedge \]
for some unique \( t \in [\tau] \). This is the element that we used in the informal version of the semantics.

How do we set up the orthodox semantics, the intended meaning, of a derivation?

\[ (\nabla) \quad \Gamma \vdash t : \tau \]
As in the informal version, we start at the leaves of \( \nabla \), interpret each of these, and then pass down the tree until we reach the root. As in the informal version, this produces a function

\[ [\Gamma] \xrightarrow{[\nabla]} [\tau] \]
except now the source space \([\Gamma]\) has an extra component \(1\), and so the function \([\nabla]\) requires \(\varnothing\) as its only left-most input. This tag \(\varnothing\) might look a bit of a nuisance, and it is most of the time, but at times it’s just what we need.

Let’s look at some of the details.

Consider first an Axiom

\[ (\heartsuit) \quad \Gamma \vdash k : \kappa \]
based on an axiom \( k : \kappa \). We already have a meaning \([k] = [k]\) of the predicate type, and when we set up the system we have a meaning \(k = [k] \in [\kappa]\) of the constant. For instance, consider

\[ 0 : \mathcal{N} \quad S : \mathcal{N}' \quad 1 : \mathcal{N} \to \mathcal{N}' \]
the monochrome numeric gadgets and an iterator. We have

\[ [\mathcal{N}] = \mathbb{N} \quad [\mathcal{N}'] = \mathbb{N}' \quad [\tau] = \mathbb{T} \]

for some space \( \mathbb{T} \), and then

\[ [0] = 0 \quad [S] = S \quad [I] = I \]

the number zero \( 0 \in \mathbb{N} \) and the two functions

\[ S : \mathbb{N} \to \mathbb{N} \quad I : \mathbb{N} \to \tau' \]

where

\[ Sx = x + 1 \quad lxf = f^x t \]

for each \( x \in \mathbb{N} \), each function \( f : \mathbb{T} \to \mathbb{T} \), and each input \( t \in \mathbb{T} \).

For convenience let us write

\[ [\Gamma] = 1 \times S \]

where \( S \) is the ‘important’ part. Then the meaning

\[ 1 \times S \xrightarrow{[\varheartsuit]} \mathbb{K} \]

of the Axiom \( \varheartsuit \) is the function given by

\[ [\varheartsuit](\varnothing, s) = k \]

for each \( s \in S \). This input \( s \) is lost at the output. For instance, the meaning of each Axiom

\[
(Zero) \quad \Gamma \vdash 0 : \mathcal{N} \\
(Next) \quad \Gamma \vdash S : \mathcal{N}' \\
(Run) \quad \Gamma \vdash I : \mathcal{N} \to \tau''
\]

is, respectively, the function

\[ Z : 1 \times S \to \mathbb{N} \quad N : 1 \times S \to (\mathbb{N} \to \mathbb{N}) \quad R : 1 \times S \to (\mathbb{N} \to \mathbb{T}') \]

given by

\[ Z(\varnothing, s) = 0 \quad N(\varnothing, s)x = x + 1 \quad R(\varnothing, s)xft = f^x t \]

for each \( s \in S, x \in \mathbb{N}, f : \mathbb{T} \to \mathbb{T} \) and \( t \in \mathbb{T} \). Notice the mixture of curried and uncurried aspects here.

As a particular case of this, the meaning of an Axiom in the empty context

\[ (\varheartsuit) \quad \varnothing \vdash k : \kappa \]

is the function

\[ 1 \xrightarrow{[\varheartsuit]} \mathbb{K} \]

which merely selects the member \( k \in \mathbb{K} \) we first thought of.

This use of \( \varnothing \) looks a bit silly, but we begin to see its benefits when we start to pass the semantic function \([\cdot]\) down through a derivation.
How do we pass the semantic function \([\cdot]\) across a use of Elimination?

\[
\begin{align*}
\Gamma \vdash q : \pi \to \tau & \quad \Gamma \vdash p : \pi \\
\hline
\Gamma \vdash (qp) : \tau
\end{align*}
\]

Consider such a use. Let

\[
G = [\Gamma] \quad P = [\pi] \quad T = [\tau]
\]

be the meanings of the context and types. The two numerators of the rule give us a pair of functions

\[
\begin{array}{c}
G \quad q \\
\hline
(P \quad T)
\end{array} \quad \begin{array}{c}
G \quad p \\
\hline
P
\end{array}
\]

\[
G \quad q \ast p \\
\hline
T
\]

where I have taken the liberty of writing ‘\(q\)’ for the left hand term and function, and ‘\(p\)’ for the right hand term and function. We require some function \(q \ast p\) which is the \(G\)-indexed composition of \(q\) and \(p\).

For each \(\star \in G\) we have

\[
q(\star) : P \rightarrow T \quad p(\star) \in P
\]
a function and a compatible input. Thus we set

\[
(q \ast p)(\star) = q(\star)(p(\star))
\]

that is we evaluate \(q(x)\) at \(p(\star)\). (If you know about these things, this is just a use of the \(S\)-combinator.)

As an example of this let’s use part of the top derivation of Table 4. This that the term \(\text{Add}\) is well-formed (in the empty context). We explicitly put in the two declarations of the context to give the following derivation.

\[
\begin{align*}
\mathcal{M}, \mathcal{N} & \vdash 1 : \mathcal{M} \rightarrow \mathcal{N}'' \\
\mathcal{M}, \mathcal{N} & \vdash u : \mathcal{M} \rightarrow u : \mathcal{M} \\
\mathcal{M}, \mathcal{N} & \vdash u : \mathcal{M}, v : \mathcal{N} \rightarrow S : \mathcal{N}' \\
\mathcal{M}, \mathcal{N} & \vdash u : \mathcal{M}, v : \mathcal{N} \rightarrow v : \mathcal{N} \\
\mathcal{M}, \mathcal{N} & \vdash u : \mathcal{M}, v : \mathcal{N} \rightarrow uSv : \mathcal{N}
\end{align*}
\]

Next we remember that \([\mathcal{M}] = [\mathcal{N}] = \mathbb{N}\) and track through the derivation to get

\[
\begin{align*}
\mathbb{N} \times \mathbb{N} & \quad \overset{a}{\longrightarrow} \quad (\mathbb{N} \rightarrow \mathbb{N}'') \\
\mathbb{N} \times \mathbb{N} & \quad \overset{b}{\longrightarrow} \quad \mathbb{N} \\
\mathbb{N} \times \mathbb{N} & \quad \overset{c}{\longrightarrow} \quad \mathbb{N}'' \\
\mathbb{N} \times \mathbb{N} & \quad \overset{d}{\longrightarrow} \quad \mathbb{N}' \\
\mathbb{N} \times \mathbb{N} & \quad \overset{e}{\longrightarrow} \quad \mathbb{N}' \\
\mathbb{N} \times \mathbb{N} & \quad \overset{f}{\longrightarrow} \quad \mathbb{N} \\
\mathbb{N} \times \mathbb{N} & \quad \overset{g}{\longrightarrow} \quad \mathbb{N}'
\end{align*}
\]
where

\begin{align*}
a(v, u) &= I \\
b(v, u) &= u \\
c(v, u) &= (a \ast b)(v, u) = a(v, u)b(v, u) = Iu \\
d(v, u) &= S \\
e(v, u) &= (c \ast d)(v, u) = c(v, u)d(v, u) = IuS \\
f(v, u) &= v \\
g(v, u) &= (e \ast f)(v, u) = e(v, u)f(v, u) = IuSv
\end{align*}

are the various components. In other words

\[ g(v, u) = v + u \]

for \( u, v \in \mathbb{N} \).

Of course, strictly speaking this is not correct, for the tag \( \emptyset \) has gone missing. The final result should be the 3-placed function

\[ 1 \times \mathbb{N} \times \mathbb{N} \xrightarrow{g} \mathbb{N}' \]

given by

\[ g(\emptyset, v, u) = v + u \]

for \( u, v \in \mathbb{N} \).

Now consider the last bit of the derivation

\[ u : M, v : N \vdash IuSv : N' \]
\[ u : M \vdash -- : N'' \]
\[ \emptyset \vdash \text{Add} : M \rightarrow N'' \]

given by two uses of Introduction. Following through the semantics we see that the meaning of this is the function

\[ 1 \longrightarrow N \longrightarrow N' \]
\[ \emptyset \longrightarrow \text{Add} \]

which simply pick out the addition operation.

References


[7] Below is an old set of references -- sort out as needed


Index

\(\lambda\)-redex, 26
1-step reduction, 27

alphabetic variant, 30
arity of a function, 65
atom, 4
  coloured, 4
  monochrome, 4
authentic
  input, 68
  numeral, 11
Axiom, 8
  versus axiom, 8
axiom
  versus Axiom, 8

basket, 148
bounded
  product, 146
  sum, 146
bracketting conventions, 14
captured in a system
  imitated
    in the broad sense, 94
    in the narrow sense, 64
  named, 37
  represented, 38
  simulated throughout, 43
  uniformly simulated throughout, 43
category
  versus syntactic category, 5
close, 60
Coherence, 31, 32
coloured atom, 4
composition
  of functions, 57
  of meek functions, 70
Confluence, 30, 31
constant, 6
  Iteration, 10
Numeric, 10
Pairing, 10
context, 7
  legal, 7
control type
  of a translation, 102
decision function, 67
declaration, 7
denominator
  of a rule, 8
derivation, 7
derivation system, 6
descends to, 84
descent relation for types, 84
Elimination, 8
exhaustive search method, examples of
  Theorem 12.3, 99
  Theorem 12.4, 101
  Theorem 7.1, 46
  Theorem 7.4, 51
  Theorem 10.10, 79
  Theorem 16.1, 122
extended polynomial, 62
finite rooted tree, 8
formal iterate, 11
function and product hierarchy, 3
hierarchy
  function and product, 3
housing space, 3
humble function, 65

identifier, 6
imitated
  in the broad sense, 94
  in the narrow sense, 64
immediate reduct, 27
immediately descends to, 84
Influence, 31, 32
input
  authentic, 68
  simulated, 68
intended interpretation
  of constants, 17
  of types, 5
Introduction, 8
Iteration constants, 10
type, 4
uniformly simulated throughout . . . , 43
Weakening, 8
well-formed term, 7