A More Efficient Recursive Definition

M. Randall Holmes

initial posting of 2/25/2016: more systematic account of effects of restriction of domain on $\rho$ functions

1 Version remarks

2/24/2016 (initially posted): This is different from barebones.tex in the following ways.

I focussed on the importance of the point that function codes and argument list types depend on each other as combinatorial objects but actually do not depend on atom codes, near-litter codes or set codes at all (their uses have everything to do with the latter, but their set theoretical representations do not, and they can be defined first). This makes it a lot easier to see that only a set of objects are in play.

There are minor differences in the ways components of the atom codes, near-litter codes, and set codes are indexed, and I actually treat near-litter codes as sets of atom codes. Also of course, I actually call them atom codes, near-litter codes, and set codes, but I never make any reference to the atoms, near-litters, or sets that they represent. The gentle reader may be able to divine my intentions but I do not say anything about them.

2/24/2016 (3:56 and 4:06 PM): Made some minor changes re anomalies of near-litter codes in the range of an argument list – the ones with the same parent as the near-litter code do not need to viewed as novel items in the argument list. Fixed some typos.

2/24/2016 4:57 PM: a basis condition supplied! It’s rather obvious, but the proof of soundness doesn’t go without it. The case where formal
elements \((3, f', L')\) and \((3, g', M')\) to be compared have \(f'\) and \(g'\) to which complexity is not assigned had been omitted.

**5:20 PM:** I am not quite certain what the best definition of anomaly of a near-litter code is. I have oscillated between elements of the symmetric difference of the near-litter code and the formal litter close to it and just the elements of the near-litter code which do not belong to the formal litter close to it: I think that in this document the latter is the best definition, but in later proofs the other sort of anomaly (the elements of the formal litter which do not belong to the near-litter code) will also be significant.

**11:08 PM:** an essential point made: when domains of function codes are restricted, some type signature component values have to be adjusted. 11:21 typo fixes

**2/25/2016:** corrected and improved account of effects of restriction of domain on \(\rho\) components of type signatures. adjusted definition of item in an argument list slightly.

**2/25/2016:** systematic modifications to correct error in the definition of extension of an argument list. Argument lists may contain pointers to information in argument lists in which they are properly embedded, in order to support a correct definition of extension, using type information. The difficulty came up because I have cut down the information in argument lists of formal elements too far: if I relaxed the restrictions on what types could appear in argument lists with a given output type, the definition of extension could go back to its earlier form, and recursive arguments on complexity of codes would again be maddening.
2 The construction

This is a more effective definition. It should be easier to see that the recursions work. It defines exactly the same structure [the precise implementations of some internal components are different, but the structure is the same in all functional respects].

We have the usual preliminaries.

the parameter \( \lambda \), clan indices and operations on clan indices: We fix a limit ordinal \( \lambda \). Finite subsets of \( \lambda \) are called clan indices. If \( A \) is an nonempty clan index, \( A_1 \) is defined as \( A \setminus \{ \min(A) \} \). \( A_0 \) is defined as \( A \); \( A_{n+1} \) is defined as \( (A_n)_1 \), when \( |A| \geq n + 1 \). We say that \( A \) downward extends \( B \) (\( A << B \)) when \( A \) is included in \( B \) and all elements of \( A \) not in \( B \) are less than all elements of \( B \).

the parameter \( \kappa \), small and large sets: We fix a regular uncountable cardinal \( \kappa \): we call sets of cardinality less than \( \kappa \) small and all other sets large.

We begin with the mutual recursive definitions of the notions of type signature and function code.

Definition (type signature, mutually recursive with following definition):
A type signature is a triple \((D, \tau, \rho)\) where \( D \) is a set of small ordinals, \( \tau \) is a function from \( D \) into \( \{1, 2\} \times [\lambda]^{<\omega} \) (each value \( \tau(\alpha) \) is either of the form \((1, A)\) or \((2, A)\) with \( A \) a clan index), and \( \rho \) is another function with domain \( D \) satisfying somewhat more complex conditions:

1. For each \( \alpha \in D \), if \( \tau(\alpha) = (1, A) \), then \( \rho(\alpha) = (1, \beta) \) for some \( \beta < \alpha \), such that if \( \beta \in D \) then \( \tau(\beta) = (2, A) \) (\( \beta \not\in D \) is permitted).

2. For each \( \alpha \in D \), if \( \tau(\alpha) = (2, A) \), then one of the following cases occurs:

   (a) \( A = \emptyset \) and \( \rho(\alpha) = (2, \beta) \) for some \( \beta < \alpha \).

   (b) Where \( A \) may be empty or otherwise, \( \rho(\alpha) = (3, f, U) \) where \( U \) is a subset of \( D \cap \alpha \), \( f \) is a function code, and the input type of \( f \) is \((U, \tau|U, \rho|U)\). The output type of \( f \) will be either \((A_1, 0)\) or \((B, |B| - |A| + 1)\) for some \( B << A \). Input and output types of function codes are defined just below.
(c) $A$ is nonempty and $\rho(\alpha) = (3, (1, (D', \tau', \rho'), \beta), D')$, where 
$(1, (D', \tau', \rho'), \beta)$ is a function code with output type $(A_1, 0)$, 
$D' \subseteq \alpha$, and $\beta \not\in D$. It will be the case that $\tau'$ and $\rho'$ agree 
with $\tau$ and $\rho$ on any common elements of their domain.

**Definition (function code, mutually recursive with preceding definition):**

A function code has an input type which is a type signature and an 
output type $(A, n)$ where $A$ is a clan index and $0 \leq n \leq |A| + 1$.

A function code of output type $(A, 0)$ has input type a type signature 
$(D, \tau, \rho)$ with $\tau$ taking the constant value $(1, A)$ and $\rho$ taking values 
$(1, \beta)$ with $\beta \not\in D$. The function code is of the form $(1, (D, \tau, \rho), \alpha)$ 
where $\alpha \in D$.

A function code of output type $(A, n+1)$ has input type a type signature 
$(D, \tau, \rho)$ with each value $\pi_2(\tau(\alpha)) \ll A_n$. The function code is of the form $(2, (D, \tau, \rho), U)$ where $U$ is a nonempty set of function codes with 
output type $(A, n)$ and the property that for each $g \in U$ with input 
type $(D', \tau', \rho')$, all elements of $D'$ not in $D$ are larger than all elements 
of $D$ and the functions $\tau'$ and $\rho'$ agree with $\tau$ and $\rho$ on $D \cap D'$.

We argue that the function codes make up a set. Clearly the function 
codes with output types $(A, 0)$ make up a set.

**Definition (complexity of a function code of positive index):** We de- 
fine the complexity of a function code with output type $(A, n+1)$ as 
the minimum element of $A_n$ or as $\lambda$ if $A_n$ is empty. Function codes 
with output type $(A, 0)$ are not assigned complexities.

**Observation about complexity of components of function codes:** Note 
that if $(D, \tau, \rho)$ is the input type of $f$ with output type $(A, n+1)$, then 
each $\rho(\alpha)$ of the form $(3, g, U)$ with the output type of $g$ not of the form 
$(A_1, 0)$ will have the output type of $g$ of the form $(C, |C| - |B| + 1)$ where 
$\tau(\alpha) = (1, B)$ and $B \ll A_n$. The complexity of $g$ will then be the minimum 
element of $C_{|C| - |B| + 1 - 1}$, that is the minimum element of $B$ (or 
possibly $\lambda$ if $B = A_n = \emptyset$), and so less than or equal to the complexity 
of $f$. Note that even if the complexity of $g$ is equal to the complexity of 
f, the first component of the input type of $g$ will have strictly smaller 
supremum than that of $f$, so it is simpler in a measurable way.
If \((A, n + 1)\) is the output type of \((2, (D, \tau, \rho), U)\), notice that each element of \(U\) has output type \((A, n)\) and so either no complexity (if \(n = 0\)) or complexity the minimum element of \(A_{n-1}\), which will be strictly less than the complexity of \((2, (D, \tau, \rho), U)\), which is either the minimum element of \(A_n\) or \(\lambda\).

Thus each function code \((2, (D, \tau, \rho), U)\) is determined by a set \(U\) of function codes of lower complexity (or no complexity if they are of output type with second component 0) and a small set of function codes appearing as components of elements of the range of \(\rho\) which have complexity no greater that that of \((2, (D, \tau, \rho), U)\) and the supremum of the first element of their input types strictly less than the supremum of \(D\). This is sufficient for an argument by induction on complexity that the function codes of each complexity and so of all complexities make up a set.

**Starting the recursive definition of the basic classes:** We now define certain sets called basic classes. Where \(A\) is a clan index, these are of the form \(C(A)\) (elements of \(C(A)\)’s may be called atom codes), \(N(A)\) (elements of \(N(A)\) may be called near-litter codes) and \(Q(A, n)\) with \(0 \leq n \leq |A| + 1\) (elements of \(Q(A, n)\)’s with \(n\) positive may be called set codes; we will see that the elements of \(Q(A, 0)\)’s reduce immediately to elements of the corresponding \(C(A)\)). At the same time, we define an equivalence relation on each basic class: we use the same word “equivalent” for each of these relations and the same symbol \(\sim\), which is safe because the classes are disjoint.

**the basic classes \(C(A)\) of atom codes:** An element of \(C(A)\) is of the form \((1, a, \alpha, A)\) where \(a\) may be an element of a \(Q(B, |B| - |A| + 1)\) with \(B \ll A\), or if \(A\) is empty \(a\) may be a small ordinal, or if \(A\) is nonempty \(a\) may be an element of \(C(A_1)\). The formal parent of \((1, a, \alpha, A)\) is defined as \(a\): notice that each \(C(A)\) has a large collection of elements with iterated formal parent a small ordinal.

Elements \((1, a, \alpha, A)\) and \((1, b, \beta, A)\) of \(C(A)\) satisfy \((1, a, \alpha, A) \sim (1, b, \beta, A)\) iff \(\alpha = \beta\) and \(a, b\) are either equal small ordinals or equivalent members of the same basic class.

**the basic classes \(N(A)\) of near-litter codes:** An element of \(N(A)\) is a small subset \(N\) of \(C(A)\) with small symmetric difference from a set
Λ(α,a) = \{ (1,a,α,A) : α < κ \}. Distinct elements of N cannot be equivalent to one another. Any element of N equivalent to an element of the Λ(α,a) from which it has small symmetric difference is equal to that element of that Λ(α,a). The elements of N \ Λ(α,a) are referred to as anomalies of N. The object a is referred to as the formal parent of N.

Elements N₁ and N₂ of the same N(A) are equivalent iff each element of N₁ is equivalent to an element of N₂ and vice versa.

Definition (argument list): An argument list is a function L with domain a set of small ordinals and range included in the union of the basic classes C(A) and N(A), which conforms to a type signature (D,τ,ρ) in the following sense:

1. The domain of L is D.
2. If τ(α) = (1,A), L(α) ∈ C(A). If τ(α) = (2,A), L(α) ∈ N(A).
3. For any α, β ∈ D, L(α) is equivalent to (and in fact equal to) an element of L(β) iff ρ(α) = (1,β). If ρ(α) = (2,β) then L(α) has small symmetric difference from Λ(β,∅). If ρ(α) = (3,f,U), then L(α) has small symmetric difference from Λ((3,f,U) ↓, π₂(τ(α)))) [the ↓ notation will be defined shortly] except in the special case where (3,f,U) ↓ is undefined [the case where f = (1,(D',τ',ρ'),β) has output type (A₁,0) and β ∉ D], when L(α) has small symmetric difference from some Λ(a,A) where a ∈ C(A₁) is not equivalent to any element of the range of L.

Further an argument list L must satisfy the conditions that distinct elements of the range of L belonging to the same set C(A) must be inequivalent and distinct elements of the range of L belonging to the same N(A) may not have equivalent elements.

Definition (elements of the classes Q(A,n)): An element of Q(A,n) is of the form (3,f,L) where f is a function code with output type (A,n) and L is an argument list conforming to the type signature π₂(f).

---

¹The definition of anomaly here does not precisely parallel the definition of anomaly for a near-litter, and this is intended: anomalies which actually belong to the object under consideration are being emphasized here. In later proofs about this structure, the other sort of anomaly will no doubt come into focus.
Definition (reduction of codes in classes \(Q(A, n)\)): We define \((3, 1, (D, \tau, \rho), \alpha), L) \downarrow\) as \(L(\alpha)\) and define \((3, f, L) \downarrow\) as \((3, f, L)\) otherwise.

Definition (equivalence of codes in sets \(Q(A, 0)\)): Elements \((3, f, L)\) and \((3, g, M)\) of \(Q(A, 0)\) are equivalent iff \((3, f, L) \downarrow\) is equivalent to \((3, g, M) \downarrow\).

Definition (extension of argument lists): We say that an argument list \(M\) extends an argument list \(L\) \((L \leq_T V, M)\) relative to type signatures \(T = (D_L, \tau_L, \rho_L)\) and \(V = (D_M, \tau_M, \rho_M)\), when \(L\) and \(M\) agree on the common part of their domain, all elements of the domain of \(M\) which are not in the domain of \(L\) exceed all elements of the domain of \(L\), and \(L \cup M\) conforms to the type signature \((D_L \cup D_M, \tau_L \cup \tau_M, \rho_L \cup \rho_M)\). Notice that for this to be true certainly \(L\) conforms to \(T\) and \(M\) conforms to \(V\) and in addition \(T\) and \(V\) must be consistent with one another in a strong sense.

Definition (formal element of a set code): We define a formal element of \((3, f, L) = (3, 2, (D, \tau, \rho), U), L)\) as a code \((3, g, M)\) such that \(g \in U\) and \(\pi_2(f) \leq \pi_2(g)\) \(M\) (noting that \(\pi_2(f) = (D, \tau, \rho)\) is a type to which \(L\) conforms, \(\pi_2(g)\) is a type to which \(M\) conforms, and \(\pi_2(f)\) and \(\pi_2(g)\) are required to be consistent with one another in the appropriate sense by the definition of function codes).

Definition (item in an argument list): We define an item in an argument list as an atom code in the range of \(L\), or an atom code appearing as the formal parent of or an element of a near-litter code in the range of \(L\).

Definition (novel item in an argument list): An item in \(L\) is called a novel item in \(L\) if it is an atom code in the range of \(L\) which does not belong to any near-litter code in \(L\), or an anomaly of a near-litter code in the range of \(L\), or an atom code not in the range of \(L\) which is the formal parent of a near-litter code in the range of \(L\).

Definition (atom code remote from a set): We say that an atom code is remote from a set \(X\) of atom codes if it has an iterated formal parent which is a small ordinal and it is not an element of \(X\) and it has no iterated parent in \(X\).
Definition (bounded formal element): Define an $M$-bounded formal element of $(3, f, L) \in Q(A, n)$, where $M$ is an argument list, as a formal element $(3, f', L')$ of $(3, f, L)$ such that each novel item in $L'$ is either an item in $L$, an item in $M$, or remote from the set of items in $L$ and $M$.

Definition of equivalence of set codes: Elements $(3, f, L)$ and $(3, g, M)$ of the same $Q(A, n + 1)$ satisfy $(3, f, L) \sim_1 (3, g, M)$ iff each $M$-bounded formal element of $(3, f, L)$ is equivalent to an $L$-bounded formal element of $(3, g, M)$ and vice versa. We define $(3, f, L) \sim (3, g, M)$ if there is a finite sequence $s$ of elements of $Q(A, n + 1)$ with $s(0) = (3, f, L)$, $s(n) = (3, g, M)$ and $s(i) \sim_1 s(i + 1)$ for each $i$ such that both $i$ and $i + 1$ are in the domain of $s$.

Justification of the definition of equivalence: The definition of equivalence is well-founded because each computation of equivalence depends on computations of equivalence which are simpler in a suitable sense.

Definition (function code complexity of elements of the basic classes): Define the function code complexity of an element of any $C(A)$ relative to a small set $S$ of atom codes as zero if it belongs to $S$ or has formal parent a small ordinal and otherwise the same as that of its formal parent. Define the function code complexity of an element of any $N(A)$ as the supremum of the function code complexities of its elements. Define the function code complexity of an element $(3, f, L)$ of $Q(A, 0)$ as the same as the function code complexity of $(3, f, L) \downarrow$. Define the function code complexity of $(3, f, L) \in Q(A, n + 1)$ as the maximum of the complexity of $f$ and the function code complexities of the elements of the range of $L$.

Observations on function code complexity and novel items in argument lists: We observe that the function code complexity of a set code $(3, f, L) \in Q(n + 1, A)$ relative to $S$ is precisely the maximum of the complexity of $f$ and the supremum of the function code complexities of the novel items in $L$ relative to $S$, because each object in the range of $L$ is either an atom code belonging to a near-litter code earlier in the list (and so contributing nothing new to its complexity) or an atom code belonging to no near-litter code in the
range of the list (a novel item) or a near-litter code with formal
parent a small ordinal (contributing to the complexity only via
its (novel) anomalies) or a near-litter code with parent an atom
code not appearing in the range of the list (so the parent is novel,
and of course so are the (novel) anomalies of the near-litter code),
or a near-litter code whose parent is of the form \((3, g, M)\) where
\(g\) is a code of complexity no greater than that of \(f\) and \(M\) is a
sublist of the restriction of \(L\) to earlier items in the list – making
no new contribution to the function code complexity except via
its (novel) anomalies, or a near-litter code whose parent is an item
earlier in the list, again contributing new complexity only via its
(novel) anomalies.

The argument for soundness of the definition of equivalence:
Our inductive hypothesis in computing a one step equivalence
\((3, f, L) \sim_1 (3, g, M)\) where both belong to \(Q(A, n + 1)\) is that
we already know how to compute equivalences between items in
\(L\) and \(M\) and between codes of lower function code complexity
than the maximum of the complexities of \(f\) and \(g\) relative to a
small set \(S\), whenever we already know how to compute equiva-
rence between elements of \(S\). We handle \(C(A)\), \(N(A)\) and \(Q(A, 0)\)
equivalences by presuming that we always already know how to
compute equivalences between items of no higher function code
complexity and lower set theoretical rank. At the basis, we do
know how to compute equivalences between elements of \(C(A)\)’s
with small ordinal parents.

Now observe that computation of equivalence between \((3, f, L)\)
and \((3, g, M)\) in \(Q(A, n + 1)\) reduces to computations of equiva-
lences between \(M\)-bounded formal elements \((3, f', L')\) of \((3, f, L)\)
and \(L\)-bounded formal elements \((3, g', M')\) of \((3, g, M)\). The com-
plexity of \(f'\) is less than the complexity of \(f\) and the complexity
of \(g'\) is less than the complexity of \(g\) (or they have no complexity).
If \(f'\) and \(g'\) have no complexity, we know how to compute the
equivalence of \((3, f', L')\) and \((3, g', M')\), because in this case \(f'\) and
\(g'\) are in effect projection operators and we know how to compute
equivalence of range elements in \(L'\) and \(M'\), because in this case
range elements of \(L'\) and \(M'\) not in the range of \(L\) and \(M\) are
simply atom codes, either items in \(L\), items in \(M\), or having small
ordinal iterated parents.

We can compute equivalences between range elements of $L'$ and $M'$, in case $f'$ and $g'$ have complexity, because any such range element has function code complexity no greater than that of one of $f'$ or $g'$ (and so less than the maximum complexity of $f$ and $g$) relative to the small collection of novel items in $L'$ and $M'$, on all of which we know how to compute equivalences (as they are either in $L$, in $M$, or have iterated parent a small ordinal). If $f'$ and $g'$ have complexity, we can compute the equivalence of $(3, f', L')$ and $(3, g', M')$, because this is an equivalence between items with complexity no greater than the maximum of the complexities of $f'$ and $g'$ relative to the small collection of range elements of $L'$ and $M'$, on which we know how to compute equivalences.