An Illative Lambda-Calculus

Roger Bishop Jones

Abstract

This is an approach to illative lambda-calculi via construction of an infinitary calculus in a well-founded set theory.

Created 2010/09/07

Last Change Date: 2013/01/03 17:12:44

Id: t041.doc,v 1.18 2013/01/03 17:12:44 rbj Exp

http://www.rbjones.com/rbjpub/pp/doc/t041.pdf

Contents

1 Prelude

This document is one of a series in which I explore approaches to non-well-founded foundations for mathematics. The immediately preceding document in this series is [7] which considers an idea for the construction of non-well-founded interpretations of set theory. It was my intention in that document ultimately to apply the results in giving a semantics to an illative lambda-calculus. When I last considered resuming the work in [7] I concluded that it might be better to go directly to the desired lambda-calculus omitting the non-well-founded set theory, by adapting the methods of that paper to a functional rather than a set theoretic ontology. This had not been possible in the first place, because I did not have clear enough idea of how it could be done. The methods I am employing are more easily understood in the first instance through set theory (at least, it was in that context that I came to them). With the benefit of having explored the techniques in the context of set theory, I now am able to see how they might be applied directly to a lambda-calculus, and in this document I make the attempt.

Discussion of what might become of this document in the future may be found the postscript (Section 6).

In this document, phrases in coloured text are hyperlinks, like on a web page, which will usually get you to another part of this document (the blue parts, the contents list, page numbers in the Index) but sometimes take you (the red bits) somewhere altogether different (if you happen to be online), e.g.: the online copy of this document.

[8]

2 Introduction

The idea is to obtain an exotic model for illative lambda-calculi by methods analogous to those employed for set theory in [7], and then to use that as a semantic foundation for the definition of one or more finitary illative lambda-calculi with systems of type assignment.

The work is conducted by conservative extension in the context of a higher order set theory established axiomatically in HOL, which I will refer to as HOL/GSU, or just GSU[4]. The end result is intended to be a semantic embedding of an illative lambda-calculus into HOL/GSU. This end is to be realised in stages as follows:

- Establish an infinitary illative combinatory logic.
  - Define the infinitary syntax by transfinite recursion in GSU.
  - Define the semantics, by determining a partial equivalence relation from a notion of reducibility over the syntax. This involves taking a least fixed point of a recursive definition of the relation and proving a result analogous to the Church-Rosser theorem to establish coherence and non-triviality of the results.
- Using this as a semantic domain, a new type is then introduced, which will be the domain over which the operations in the illative lambda-calculus will be defined. The theory is then developed by defining further operations as necessary and proving results which establish the required type assignment and inference rules for the embedded calculus.

2.1 Type Assignments

The systems of type assignments I am interested in have the following features:

- A universal type.
- Dependent product and sum (function and pair) types.
- A heirarchy of 'Universes' (but since we have an absolute universe I will probably follow GSU in calling them galaxies), not all well-founded, each closed under dependent type constructions (this has the effect of power set and replacement).
- A well-founded part in which classical mathematics can be conducted in the normal way (i.e. getting the theories isomorphic to the standard classical versions), and which admits arbitrarily risky axiomatic extensions analogous to (possibly even identical to) large cardinal axioms (with no greater risk here than in ZFC). The well founded part is the union of iterated galaxy construction over the empty set.

The difficult question with such a shopping list, is whether it can be consistently fulfilled, and the principal purpose of this document is to explore an idea about how to construct an interpretation which might allow the equiconsistency of the systems considered here with ZFC (plus large cardinals where appropriate) to be established. Having said that, the procedure is *semantics first*, so I aim to construct the interpretation first, and then to derive logical systems of which it is an interpretation. This method guarantees that the resulting system is consistent relative to my set theory (which is 'A Higher Order Theory of Well-Founded Sets (with Urelements)' (GSU), see [4]). [1]

In the talk which follows about syntax and semantics there are two levels involved, that of the infinitary calculus (which is the interpretation constructed in GSU) and that of the finitary calculus which will be based on it.

2.2 Consistency

A few more words about consistency may be in order.

The approach here is to ensure consistency in the desired logic by working from semantics to proof rules. The logic determined by proof rules is bound to be consistent because the proof rules are either theorems proven about the chosen semantic domain or else derived rules implemented in our metalanguage for reasoning in the new language. This is a natural outcome of the method which consists in implementing the desired language using a 'shallow embedding' in a higher-order set theory.

Adoption of this method does not in itself solve the consistency problem, the problem appears in the necessary semantic preliminaries for the embedding. To succeed it is necessary to understand how the consistency problem appears and have some idea of how it can be resolved.

In this case, the consistency problem for an illative lambda-calculus is to be resolved by providing a syntactic model in an infinitary illative combinatory logic. The problem is then to obtain a satisfactory account of the infinitary combinatory system. Ideally the elements of the model would be equivalence classes of terms under convertibility, but one of the primitive combinators is to be

[1] This is a modification of the pure set theory GS to admit urelements, which are irrelevant to the theory here, but may improve the smoothness of its integration into HOL. GS actually started with urlements, but then I removed them, and now I have put them back in. They don't make it significantly more complicated, at least not in the areas I have covered.

read as that very equivalence relation and the effect is that the relation of convertability cannot be a total relation over the combinators (this is how the consistency problem reappears). This effect arises because the combinator must be defined by recursion, and the higher order equation of which a fixed point is obtained from the definition will not have a total fixed point (this is the manifestation of the various paradoxes such as Russell's or more directly relevant here, Curry's paradox). Consistency is obtained by accepting a partial fixed point, this is equivalent on the one hand to regarding the Russell set as a "partial set", of which some things (including itself) are neither members nor non-members, neither definitely in nor out. In the case of the Curry paradox, we find that implication does not always have a truth value (it is total over the truth values, just not over lambda terms in general). So we have to find a way of working with a partial equivalence relation, which will be the primitive illative combinator in terms of which all the others are defined.

In the context of a combinatory logic this can be realised by limiting the possible reductions of equivalence terms, which is not a pure combinator. Such an equivalence will always reduce to 'T' if its two arguments are convertible, but in the case that they are not, it may not always be reduced to 'F', in some cases it will not reduce. It therefore is not 'on the nail', it is in effect an *approximation* to convertability. The key point in the semantics is therefore the definition of this relation of convertibility.

The definition of this is recursive and is not well-founded so the definition involves taking a fixed point of a monotone functor over partial relations which approximate to convertability. A total equality combinator would reduce to T or F (suitably chose combinators) when applied to any two combinatory terms, according to whether the two terms are convertible. A partial equality combinator will not always reduce, there will be some pairs of terms for which one cannot have a definite view as to their convertibility. One reason for this might be that if no reduction of the equation were possible then they would not be convertible, but once the reduction of the equation to F is added it then becomes possible to convert the two terms. Such a pair of terms could be constructed corresponding to the liar paradox, and making the equality combinator partial protects against the derivation of contradiction in that way, by allowing that the liar sentence be neither true nor false.

Allowing this equivalence to be partially defined, in particular taking it to be the least fixed point of the defining equations under an appropriate ordering, is what gets us out of the paradox and should yield a good semantics. This device is to be combined with steps intended to make this semantics as rich as a strong first order set theory (one in which inaccessible cardinals can be proven to exist). Whether this will also support a system of type assignments along the lines desired remains to be seen, I have not articulated in this context my reasons for hoping that will be the case. The rationale is more fully thought through for the analogous methods which I was previously applying to models of non-well-founded set theories, and for the preceeding work on 'polysets'(see [6, 5, 10, 9, 3].

The equivalence relation is compounded from several components of which only one is 'illative' and contentious (though the two combinators which handle infinitely many arguments may also be thought contentious!). There is a question how to organise these separate components and how much to do prior to devising the functor of which the least fixed point gives the semantics of equivalence.

The proposed method, thus outlined, requires no further fundamental innovations to deliver a model, though there will be many opportunities to get details wrong. There is however, no guarantee that the resulting system will support the kind of type-assignment system which I have in mind, this is highly speculative. The greatest difficulty is in the complexity of the formal derivations required, which may prove beyond me.

3 The Infinitary Interpretation

I have had quite a few (lost count) iterations in my conception of the syntax of the infinitary system which I use for the semantics of the target illative lambda-calculus.

My first attempt was an infinitary illative lambda calculus in which there was infinitary abstraction (abstraction over infinitely many variables at once) and application (application of an infinitary abstraction to an infinite collection of arguments). There was something unsatisfactory about this approach (a muddle about the scope of the variables, which I used to tag the arguments in an application) and by the time I had sorted it out the infinitary abstraction had been dumped (in favour of infinitary function displays). Unfortunately the new version left me with five syntactic constructors, and, because the (infinitary) abstract syntax is hand cranked, I have a strong incentive to keep its complexity to an absolute minimum. So I decided to ditch abstraction, and with it, variables.

Several attempts on I now have an infinitary illative combinatory logic with just one syntactic constructor.

I call this syntax infinitary because the syntax has large (inaccessible) cardinality. It isn't actually a deductive system, but it is a language, for which we have syntax and semantics. We have a notion of truth and can prove the truth of expressions of our infinitary language in the metalanguage (which is HOL/GSU, a higher order set theory).

The semantic intent explains the infinitary nature of the system considered, this is so that we have underlying our ultimate language (which will be a finitary untyped illative lambda-calculus with a system of type assignments) a basis for the development of exactly the same ("classical") applicable mathematics as is more often considered in the context of ZFC or HOL. For this purpose there are intended to be, within the ontology, structures which mirror or mimic the hierarchy of well-founded sets, and particularly, full function spaces over well founded subsets of the universe.

The idea is to define a kind of syntactic interpretation for the lambda-calculus and to give a semantics to it. Lambda terms in the target language will denote equivalence classes of infinitary combinators in the underlying language. The equivalence classes over the combinators are generated by the defining equations for the combinators except for the sole illative combinator, which will be equivalence (aka convertibility). The tricky bit is to determine the semantics of this equivalence relation, since the formal definition follows the previous sentence in being recursive, and the recursion will probably not be well-founded. This is done by treating it as a partial equivalence, giving the recursive definition in the form of a functor over partial approximations to convertibility, and then taking the least fixed point of this functor, and praying that there will be enough in there.

An important departure from the method adopted for non-well-founded set theory in [7] is the acceptance that the domain of discourse will be full of junk (e.g. non-normalising terms), and that the application of a lambda-calculus based on this interpretation will depend on reasoning almost exlusively within the confines of well-behaved subdomains which are delimited by an appropriate system of type-assignment (even though the terms themselves will be un-typed). This is intended to yield something which will look a bit like a typed lambda-calculus with subtyping and a universal type (plus some other exotic features to make it at least as strong as ZFC) but which differs from it in a manner similar to the difference between the simply-typed lambda-calculus and a system of type assignments to the pure lambda-calculus.

SML

```
open_theory "misc3";
force_new_theory "icomb";
force_new_pc "'icomb";
```

```
merge_pcs ["'savedthm_cs_∃_proof"] "'icomb";
new_parent "equiv";
set_merge_pcs ["misc31", "'icomb"];
```

3.1 Infinitary Syntax

The syntax of an infinitary illative combinatory logic is to be encoded as sets in a higher order set theory.

Since I will probably have to prove something like the Church-Rosser theorem and the syntax can be made very simple, this will be a bare bones treatment of the syntax, by contrast with my previous similar enterprise for non-well-founded set theory in [7].

There will be just one constructor, which is the ordered pair constructor constrained to apply a single constant name to sequences of combinators of any cardinality. We can therefore go immediately to specification of the required closure condition for the syntax.

The closure condition is:

HOL Constant

$$\boldsymbol{CrepClosed}: 'a\ GSU\ SET \to BOOL$$

$$\forall\ s\bullet\ CrepClosed\ s \Leftrightarrow (\forall c\ i\bullet\ Fun_u\ i \wedge Ordinal_u\ (Dom_u\ i) \wedge X_u\ (Ran_u\ i) \subseteq s \Rightarrow c \mapsto_u i \in s)$$

The well-formed syntax is then the smallest set which is *CrepClosed*.

HOL Constant

$$\boldsymbol{Csyntax} : 'a\ GSU\ SET$$

$$Csyntax = \bigcap\{x \mid CrepClosed\ x\}$$

$$\boldsymbol{crepclosed\_csyntax\_lemma} = \vdash CrepClosed\ Csyntax$$

$$\boldsymbol{crepclosed\_csyntax\_thm} = \vdash \forall\ c\ i\bullet\ Fun_u\ i \wedge Ordinal_u\ (Dom_u\ i) \wedge (\forall\ x\bullet\ x \in X_u\ (Ran_u\ i) \Rightarrow x \in Csyntax) \Rightarrow c \mapsto_u i \in Csyntax$$

$$\boldsymbol{crepclosed\_csyntax\_thm2} = \vdash \forall\ c\ i\bullet\ Fun_u\ i \wedge Ordinal_u\ (Dom_u\ i) \wedge (\forall\ x\bullet\ x \in_u Ran_u\ i \Rightarrow x \in Csyntax) \Rightarrow c \mapsto_u i \in Csyntax$$

$$\boldsymbol{crepclosed\_csyntax\_thm3} = \vdash \forall\ c\ i\bullet\ Seq_u\ i \wedge (\forall\ x\bullet\ x \in_u Ran_u\ i \Rightarrow x \in Csyntax) \Rightarrow c \mapsto_u i \in Csyntax$$

crepclosed_csyntax_lemma1 =

$\vdash \forall\ s\bullet\ CrepClosed\ s \Rightarrow Csyntax \subseteq s$

crepclosed_csyntax_lemma2 =

$\vdash \forall\ p\bullet\ CrepClosed\ \{x|p\ x\} \Rightarrow (\forall\ x\bullet\ x \in Csyntax \Rightarrow p\ x)$

3.1.1 Recursion and Induction Principles and Rules

We need to be able to define functions by recursion over this syntax. To do that we need to prove that the syntax is well-founded. This is the case relative to the transitive closure of the membership relation, but to get a convenient basis for reasoning inductively over the syntax and for defining functions by recursion over the syntax it is best to define an ordering in terms of the syntactic constructors for the syntax.

This could be done strictly over the well-formed syntactic constructs, but this would involve more complexity both in the definitions and in subsequent proofs than by defining it in terms of the syntactic constructors whatever they are applied to.

HOL Constant

CscPrec : $'a\ GSU\ REL$

$\forall\alpha\ \gamma\bullet\ CscPrec\ \alpha\ \gamma \Leftrightarrow \exists c\ i\bullet\ \alpha \in_u (Ran_u\ i) \wedge \gamma = c \mapsto_u i$

CscPrec_tc_∈_thm =

$\vdash \forall\ x\ y\bullet\ CscPrec\ x\ y \Rightarrow tc\ \$\in_u\ x\ y$

well_founded_CscPrec_thm =

$\vdash well\_founded\ CscPrec$

well_founded_tcCscPrec_thm =

$\vdash well\_founded\ (tc\ CscPrec)$

Using the well-foundedness theorems we can define tactics for inductive proofs.

SML

val **CSC_INDUCTION_T** = *WFCV_INDUCTION_T well_founded_CscPrec_thm*;
val **csc_induction_tac** = *wfcv_induction_tac well_founded_CscPrec_thm*;

csc_fc_thm =

$\vdash \forall\ x\bullet\ x \in Csyntax \Rightarrow$
$(\exists c\ i\bullet\ Fun_u\ i \wedge Ordinal_u(Dom_u\ i)$
$\wedge (\forall\ y\bullet\ y \in_u Ran_u\ i \Rightarrow y \in Csyntax)$
$\wedge\ x = c \mapsto_u i)$

$\neg\varnothing_u\_\in\_csyntax\_lemma =$
$\vdash \neg\ \varnothing_u \in Csyntax$

$\neg\varnothing_u\_\in\_csyntax\_lemma2 =$
$\vdash \forall\ x\bullet\ x \in Csyntax \Rightarrow \neg\ x = \varnothing_u$

$\neg\varnothing_u\_\in\_csyntax\_lemma3 =$
$\vdash \forall\ V\ x\bullet\ x \in V \wedge V \subseteq Csyntax \Rightarrow \neg\ x = \varnothing_u$

$csc\_fc\_thm2 =$
$\vdash \forall c\ i\bullet\ c \mapsto_u i \in Csyntax \Rightarrow Fun_u\ i \wedge Ordinal_u\ (Dom_u\ i)$
$\qquad \wedge\ (\forall\ x\bullet\ x \in_u (Ran_u\ i) \Rightarrow x \in Csyntax)$

$cscprec\_fc\_thm =$
$\vdash \forall\ c\ i\ x\bullet\ x \in_u Ran_u\ i \Rightarrow CscPrec\ x\ (c \mapsto_u i)$

Inductive proofs using the well-foundedness of ScPrec are fiddly. The following induction principle simplifies the proofs.

$csyn\_induction\_thm =$
$\vdash\ (\forall c\ i\bullet\ Fun_u\ i \wedge Ordinal_u(Dom_u\ i)$
$\qquad \wedge\ (\forall x\bullet\ x \in_u (Ran_u\ i) \Rightarrow x \in Csyntax \wedge p\ x)$
$\qquad \Rightarrow p\ (c \mapsto_u i))$
$\quad \Rightarrow (\forall\ x\bullet\ x \in Csyntax \Rightarrow p\ x)$

Using this induction principle an induction tactic is defined as follows:

SML
```
fun icomb_induction_tac t (a,c) = (
    let val l1 = mk_app (mk_λ (t,c), t)
        and l2 = mk_app (mk_app (mk_const ("∈", ⌜:'a GSU → 'a GSU SET → BOOL⌝), t),
                         mk_const ("Csyntax", ⌜:'a GSU SET⌝))
    in  let val l3 = mk_∀ (t, mk_⇒ (l2, l1))
    in  LEMMA_T l1 (rewrite_thm_tac o rewrite_rule[])
    THEN DROP_ASM_T l2 ante_tac
    THEN LEMMA_T l3 (rewrite_thm_tac o rewrite_rule[])
    THEN bc_tac [csyn_induction_thm]
    THEN rewrite_tac[]
    THEN strip_tac
    end end) (a,c);
```

This tactic expects an argument t of type $TERM$ which is a free variable of type $\ulcorner : ('a)GSU \urcorner$ whose sole occurrence in the assumptions is in an assumption $\ulcorner {}_{\mathrm{ML}}\ulcorner t \urcorner \in Csyntax \urcorner$, and results in a single subgoal. The conclusion of the subgoal is as in the original except that free occurrences of $\ulcorner t \urcorner$ have been replaced by $\ulcorner c \mapsto {}_u i \urcorner$. The assumption $\ulcorner {}_{\mathrm{ML}}\ulcorner t \urcorner \in Csyntax \urcorner$ is replaced by three new assumptions asserting that $\ulcorner i \urcorner$ is a function, that its domain is an ordinal, and that everything in its range is a member of $Csyntax$ for which the property expressed in the conclusion holds.

A slight variant on that is:

```
csyn_induction_thm2 =
  ⊢ (∀c i• Seq_u i
          ∧ (∀x• x ∈_u (Ran_u i) ⇒ x ∈ Csyntax ∧ p x)
          ⇒ p (c ↦_u i))
      ⇒ (∀ x• x ∈ Csyntax ⇒ p x)
```

SML

```
fun icomb_induction_tac2 t (a,c) = (
      let val l1 = mk_app (mk_λ (t,c), t)
          and l2 = mk_app (mk_app (mk_const ("∈", ⌜:'a GSU → 'a GSU SET → BOOL⌝), t),
                           mk_const ("Csyntax", ⌜:'a GSU SET⌝))
      in  let val l3 = mk_∀ (t, mk_⇒ (l2, l1))
      in  LEMMA_T l1 (rewrite_thm_tac o rewrite_rule[])
      THEN DROP_ASM_T l2 ante_tac
      THEN LEMMA_T l3 (rewrite_thm_tac o rewrite_rule[])
      THEN bc_tac [csyn_induction_thm2]
      THEN rewrite_tac[]
      THEN strip_tac
      end end) (a,c);
```

This tactic expects an argument t of type $TERM$ which is a free variable of type $\ulcorner : ('a)GSU \urcorner$ whose sole occurrence in the assumptions is in an assumption $\ulcorner {}_{\mathrm{ML}}\ulcorner t \urcorner \in Csyntax \urcorner$, and results in a single subgoal. The conclusion of the subgoal is as in the original except that free occurrences of $\ulcorner t \urcorner$ have been replaced by $\ulcorner c \mapsto {}_u i \urcorner$. The assumption $\ulcorner {}_{\mathrm{ML}}\ulcorner t \urcorner \in Csyntax \urcorner$ is replaced by three new assumptions asserting that $\ulcorner i \urcorner$ is a sequence, and that everything in its range is a member of $Csyntax$ for which the property expressed in the conclusion holds.

```
csyntax_pair_thm = ⊢ ∀ t• t ∈ Csyntax ⇒ (∃ c ts• t = c ↦_u ts)
```

The following function provides domain restriction of a function over $\ulcorner : 'a\ GSU \urcorner$. Since this is an operation on total functions, the effect is achieved by delivering a function which returns the same value for all arguments outside the restricted domain. The purpose is to constrain recursion to be well founded, so the possibility of returning a function completely unconstrained in what it does off the restricted domain does not suffice (we would not be able to prove that the domain restriction had done anything at all). The domain is a set, but the constraint is to its transitive closure.

SML

```
declare_infix(310, "◁_ue");
```

HOL Constant

```
$◁_ue : 'a GSU → ('a GSU → 'b) → ('a GSU → 'b)
```

```
∀s f• s ◁_ue f = λt• if t ∈_u^+ s then f t else εx•T
```

A recursion lemma suitable for consistency proofs of primitive recursive definitions over our syntax can now be proven, this supports definition by primitive recursion of functions over the syntax.

$$\mathbf{csc\_recursion\_lemma} =$$
$$\vdash \forall af \bullet \exists f \bullet \forall c\ i \bullet f\ (c \mapsto_u i) = af\ (i \lhd_{ue} f)\ c\ i$$

This is (when proven) plugged into proof context *'icomb* for use in consistency proofs.

SML
```
add_∃_cd_thms [csc_recursion_lemma] "'icomb";
set_merge_pcs ["misc31", "'icomb"];
```

This is just to test the recursion theorem.

HOL Constant
$$\mathbf{CscRank} : 'a\ GSU \to 'a\ GSU$$

$$\forall c\ i \bullet CscRank\ (c \mapsto_u i) = \bigcup_u\ (Imagep_u\ Suc_u\ ((Imagep_u\ (i \lhd_{ue} CscRank))\ (Ran_u\ i)))$$

3.1.2 More Closure Properties

In order to prove that various expressions yield elements of Csyntax further definitions and theorems are required.

The construction of terms is accomplished primarily by the construction of sequences of terms. For concision in expressing closure conditions of operators over such sequences we give a name for a sequence of terms in our syntax.

HOL Constant
$$\mathbf{CscSeq} : 'a\ GSU \to BOOL$$

$$\forall s \bullet CscSeq\ s \Leftrightarrow Seq_u\ s \wedge \forall t \bullet t \in_u Ran_u\ s \Rightarrow t \in Csyntax$$

3.1.3 Finitary Constant Combinators

Our syntax does not include the standard finitary application, to do this you have to append the argument to the sequence of arguments already present on the function. It's handy to define a function in HOL which does this operation (i.e. which constructs the required combinatory term). For this we will use an infix bare subscript c.

SML
```
declare_infix (350, "_c");
```

HOL Constant
$$\$_{\mathbf{c}} : 'a\ GSU \to 'a\ GSU \to 'a\ GSU$$

$$\forall f\ a \bullet f\ _c\ a = let\ fc = Fst_u\ f \quad and\ fa = Snd_u\ f \quad in\ fc \mapsto_u (fa\ @_u\ (Unit_u\ (\varnothing_u \mapsto_u a)))$$

We also define a function corresponding to a constant constructor. This requires the constant 'name' as argument and returns a combinator with that name at the head and an empty list of arguments.

HOL Constant

$$\$\boldsymbol{MkCcon} : 'a\ GSU \to 'a\ GSU$$

$$\forall n \bullet\ MkCcon\ n = n \mapsto_u \varnothing_u$$

$$\boldsymbol{MkCcon\_\in\_Csyntax\_thm} = \vdash\ \forall\ x \bullet\ MkCcon\ x \in Csyntax$$

Using which we name the primitive combinators.

We use an initial segment of the natural numbers as the names of combinators in the infinitary object langauge, but we give more intuitive names in our metalanguage (HOL) for the resulting combinatory term.

First two finitary pure combinators:

HOL Constant

$$\boldsymbol{S_c} : 'a\ GSU$$

$$S_c = MkCcon\ (Nat_u\ 0)$$

HOL Constant

$$\boldsymbol{K_c} : 'a\ GSU$$

$$K_c = MkCcon\ (Nat_u\ 1)$$

Then the sole illative combinator, which is equality or equivalence.

HOL Constant

$$\$\equiv_{\boldsymbol{c}} : 'a\ GSU$$

$$\$\equiv_c = MkCcon\ (Nat_u\ 2)$$

Various useful combinatorial expressions may now be named.

$$I = \lambda x \bullet x$$

HOL Constant

$$\$\boldsymbol{I_c} : 'a\ GSU$$

$$I_c = (S_c\ _c\ K_c)\ _c\ K_c$$

The truth values may be represented as two projections from the argument list.

$$T = \lambda x\ y \bullet\ x$$

HOL Constant

$\$\boldsymbol{T_c} : 'a\ GSU$

$T_c = K_c$

$F = \lambda x\ y\bullet\ y$

HOL Constant

$\$\boldsymbol{F_c} : 'a\ GSU$

$F_c = K_c\ _c\ I_c$

Conditionals may then be represented by applying the condition to the two alternatives:

$if\ x\ then\ y\ else\ z = xyz$

HOL Constant

$\$\boldsymbol{If_c} : 'a\ GSU \to 'a\ GSU \to 'a\ GSU \to 'a\ GSU$

$\forall x\ y\ z\bullet\ If_c\ x\ y\ z = (x\ _c\ y)\ _c\ z$

Natural numbers may be represented as iterators (though this is unlikely to be how they are represented in the target illative lambda calculus):

Thus, zero is the identity function.

HOL Constant

$\$\boldsymbol{0_c} : 'a\ GSU$

$0_c = I_c$

$Suc\ n = \lambda f\ x\bullet\ f((n\ f)x)$
$= \lambda f\bullet\ \lambda x\bullet\ f((n\ f)x)$
$= \lambda f\bullet\ S\ (K\ f)\ (n\ f)$
$= S\ (\lambda f\bullet\ S\ (K\ f))\ n$
$= S\ (S\ (K\ S)\ K)\ n$
$= \lambda f\bullet\ ((S\ (K\ S)\ K)\ f)\ (n\ f)$
$= \lambda f\bullet\ ((\lambda x\bullet\ ((K\ S)\ x)\ (K\ x))\ f)\ (n\ f)$
$= \lambda f\bullet\ ((\lambda x\bullet\ S\ (K\ x))\ f)\ (n\ f)$
$= \lambda f\bullet\ S\ (K\ f)\ (n\ f)$
$= \lambda f\ x\bullet\ ((K\ f\ x)\ ((n\ f)\ x))$
$= \lambda f\ x\bullet\ f\ ((n\ f)\ x)$

HOL Constant

$\$\boldsymbol{Suc_c} : 'a\ GSU$

$Suc_c = S_c\ _c\ (S_c\ _c\ (K_c\ _c\ S_c)\ _c\ K_c)$

However, we require a representation of transfinite ordinals. It is not clear how this could be obtained as iterators.

It is probable that arithmetic will be arrived at in a manner more similar to that adopted in a well-founded set theory. Such a set theory could be derived within the target illative lambda-calculus as a theory of characteristic functions, provided that we have a sufficiency of such functions, which the infinitary combinators are intended to ensure.

3.1.4 Infinitary Combinators

The idea is to get a system which is ontologically equivalent to the standard interpretations of well-founded set theory with large cardinal axioms. By equivalent here I mean something like mutually interpretable, but the interpretations at stake here are correspondences between the ontologies, not interpretability of theories. However, this is the case without the infinitary combinators, so what I am looking for is a bit more.

I want to ensure that the functions available as combinators are all functions with small graphs (and a decent collection of those with large graphs, but they are not our present concern). The infinitary combinators are introduced to permit a function to be defined in extension, so long as its graph is "small" (i.e. smaller than the universe of discourse).

It is not entirely straightforward to do this. The present proposal is to use three infinitary combinators.

First an inert combinator which is used to construct lists or sequences.

HOL Constant

$$\$\mathbf{\Phi_c} : 'a\ GSU \to 'a\ GSU$$

$$\forall s \bullet\ \Phi_c\ s = (Nat_u\ 3) \mapsto_u s$$

Such combinators are irreducible and injective, and it is therefore possible in principle to extract the components, for which we supply the following projection combinator.

The projection combinator takes two sequences, the first effectively determining an element of the second to be extracted. Normally this would be supplied with two sequences of equal length and the element to be extracted from the second would be the one which corresponds to the first element of the first sequence which reduces to 'T', provided that all previous values reduce to 'F' (otherwise no element is selected, i.e. no reduction is possible).

HOL Constant

$$\$\mathbf{\Omega_c} : 'a\ GSU$$

$$\Omega_c = MkCcon\ (Nat_u\ 4)$$

One may think of the first sequence supplied as an argument to Ω_c as an ordinal α and of $\Omega_{c\ c}\ \alpha_{\ c}\ \beta$ as selecting the α^{th} element of β, but when we do eventually come to the theory of ordinals we will not use this representation. Furthermore note that this way of indicating the element to be selected is determined primarily because it is convenient for constructing infinitary 'case' constructions, and hence for the definition of arbitrary functions.

To achieve this effect we need one further infinitary combinator, a mapping combinator.

This combinator expects its second argument to be a sequence, and maps its first argument over the elements of the sequence, returning a sequence which consists of the elements of the original sequence

transformed by the function supplied as the first argument. Of course, the intended "transformation" will only take place as the combinators in the new sequence are themselves reduced, the reduction arising from this combinator is just to apply the function to each element giving a sequence of applications.

HOL Constant

$$\$\boldsymbol{\Psi_c} : 'a\ GSU$$

$$\Psi_c\ =\ MkCcon\ (Nat_u\ 5)$$

These three combinators together may be used to define a function as a graph in the following way. Form two corresponding sequences, the first of values in the domain of the desired function, the second having at each position the value of the function at that point. The required function takes some value $\ulcorner x \urcorner$, and maps $\ulcorner \equiv_c x \urcorner$ over the first argument, and then uses that list to determine which value to select from the second list. The effect of the mapping operation is to create a sequence in which every element is the truth value of the claim that it is equal to the argument $\ulcorner x \urcorner$, which, provided that all terms normalise will be the a representation of the position in the list of the argument, which can then be used to select the required value from the other sequence.

An infinitary case combinator might therefore be obtained from these three combinators as follows.

$$Gfun_c \qquad = \lambda x\ y\ z\bullet\ \Omega_c\ (\Psi_c\ (\equiv_c\ z)_c\ x)_c\ y$$

It is not expected that this will ever be done, it is important only that it *could* be done (if only by some inaccessibly infinite deity), and this is intended to ensure that we can do 'classical' mathematics in pretty much the normal way within our target calculus, which will probably be demonstrated by reconstructing a strong set theory within it. The ability to represent arbitrary functions on infinite domains (e.g. over the reals) by such means depends upon the axiom of choice, which we do have at our disposal. It is intended also that the target illative lambda calculus will benefit (or perhaps in some eyes be blighted by) a choice principle.

$$\begin{aligned}&\boldsymbol{MkCcons\_\in\_Csyntax\_clauses} = \vdash S_c \in Csyntax\\ &\qquad \wedge\ K_c \in Csyntax\\ &\qquad \wedge\ \$\equiv_c \in Csyntax\\ &\qquad \wedge\ T_c \in Csyntax\\ &\qquad \wedge\ \Omega_c \in Csyntax\\ &\qquad \wedge\ \Psi_c \in Csyntax\end{aligned}$$

3.2 Semantics

The following semantics is a first crude cut. Conceivably it might be 'correct' but certainly it is not very nicely structured. A good structure is only likely to emerge as I get a handle on the proofs I need to do with it.

The semantics which follows must be in effect a monotone functor, so that we can obtain a least fixed point. In my previous attempts for a non well-founded set theory I used truth functions into a suitably ordered four-valued set of truth values, but in this case a simpler approach seems possible, in which we use a functor over standard bi-valent relations, ordered by inclusion (of the set of ordered pairs of which the relation is the characteristic function). The relation in question is that of *convertibility*.

To understand how this works it is best to think of it in other terms, as a *partial* equivalence relation. A *partial* equivalence relation is one for which some pairs are equivalent and some pairs are not equivalent, and the status of others is unknown. The relation we work with will be the positive part of this partial equivalence relation. However, it will contain coded within it also the negative part.

This encoding occurs through the conversions for applications of the equivalence combinator. The equivalence combinator is intended to be the relation of convertibility, but for reasons connected with the paradoxes it cannot be total. This is because the notion of convertibility at stake involves the conversions for equivalence expressions, and so the dependency is recursive. Defining convertibility as the least fixed point of our functor gives is recursive definition which will fail to be well-founded for some pairs of combinators. For these pairs, equivalence will not reduce, and it will therefore be a partial equivalence relation.

The functor is defined in two parts, respectively for the positive and the negative parts of the convertibility relation. The two are then put together.

The positive part is an equivalence relation which ultimately corresponds exactly to convertibility.

The negative part is an approximation to distinctness, but there will be some pairs of terms which are not convertible which never appear in the negative part (possibly because if they did, they would then appear also in the positive part engendering contradiction, this is what we would expect to happen with the assertion that the Russell set is a member of itself, if it comes out true, then it will also come out false).

The negative part plays a role in the definition of the positive part, because an application of an equivalence combinator reduces to F_c only if the two arguments are related under this negative part of the partial equivalence relation.

Of course, the negative part is not itself an equivalence relation. The requirement that the whole (both the positive and the negative parts) is a partial equivalence relation is simply the usual conditions (reflexive, symmetric and transitive), on the positive part. We also require that the two parts don't disagree!

We consider the definition in two parts, the positive part and the negative part.

The positive part of the functor is a functor which takes a convertibility relation, closes it up under all the direct conversions for combinators other than the equivalence combinator. The negative part then computes a set of inequalities from this convertibility relation. The postive and negative parts are then incorporated into the relation as conversions of equations.

positive part The positive part is defined in three stages.

First *direct reduction* is defined for the *Pure* combinators . These are combined into a closure operator which obtains from any relation the smallest equivalence relation closed under direct reduction subject to a rule of extensionality (equivalent functions applied to equivalent arguments give equivalent combinators). In this context the pure combinators are those for which direct reduction is definable independently of the other combinators.

The second stage is the definition of direct reducibility for the illative equivalence combinator, which is done *relative to* some given partial equivalence relation.

The third stage is to combine the two previous definitions into a partial functor, which takes a partial equivalence relation and delivers the positive part of a partial equivalence relation. It does this by applying the definition of direct reducibility of equivalence to the partial equivalence to get a first version of the positive part, and then closing this under the closure operator obtained from

the definitions of the other combinators.

negative part There are two parts to this.

Firstly a fixed part under which all constants are known to be distinct.

Secondly we infer that two combinatorial expressions are distinct if when applied to equal arguments they yield distinct results. To obtain such results you need to have both positive and negative results, so this part has to be defined as a function over a partial equivalence relation.

I now see that the negative part is derivable from the positive part, in which it is incorporated as reductions of equations to F_c. Its not obvious whether it is better to extract the information in that way or to maintain it separately as I first set out to do. I will hedge my bets a bit until I get a stronger sense of which is the best way to go.

3.2.1 The Combinators

We have five combinatory constants, S, K, $\equiv$ (convertability), Ω (projection), and Ψ (map), of which the last two are infinitary, the first three being much the same as you would expect in a finitary illative combinatory logic in which the illative primitive is equality.

The rationale for this is that the infinitary aspect is of marginal significance, and may be thought of as supplying strength, just as in the role of large cardinals in set theory, the strength is bound up with ontological plenitude. It should be like an afterthough, the main features of the system should arise in the finitary case from the first three combinators, and the principle innovation is in the approach to giving meaning to $\equiv$.

The definitions of the pure combinators (all except $\equiv$) are given as conversion rules, whose reflexive symmetric transitive closure give a first approximation to the meaning of $\equiv$. If we assume given some meaning for $\equiv$ and obtain from it a second version by combining it with the conversion rules for the other combinators, then we have a functor over possible meanings for $\equiv$, which will be monotone. The least fixed point will then be used to determine the semantics of the infinitary combinatory logic.

3.2.2 Direct Conversions for the Pure Combinators

These are defined as relations over *Csyntax*.

SML

$$declare\_type\_abbrev(\texttt{"}\boldsymbol{RED}\texttt{"},\ [\texttt{"}'a\texttt{"}],\ \ulcorner :'a\ GSU \to 'a\ GSU \to BOOL\urcorner);$$

HOL Constant

$$\boldsymbol{Kred} : 'a\ RED$$

$$\forall s\ t\bullet\ Kred\ s\ t \Leftrightarrow \exists x\ y\bullet\ s = (K_c\ _c\ x)\ _c\ y \wedge t = x$$

HOL Constant

$$\boldsymbol{Sred} : 'a\ RED$$

$$\forall s\ t\bullet\ Sred\ s\ t \Leftrightarrow \exists x\ y\ z\bullet\ s = ((S_c\ _c\ x)\ _c\ y)\ _c\ z \wedge t = (x\ _c\ z)\ _c\ (y\ _c\ z)$$

The projection combinator is more complicated.

HOL Constant

$$\mathbf{\Omega_{red}} : 'a\ RED$$

$$\begin{aligned}
&\forall s\ t\bullet\ \Omega_{red}\ s\ t \Leftrightarrow (\exists k\ l\ m\ n\bullet\ Dom_u\ k = Dom_u\ m \\
&\qquad \wedge\ Ordinal_u\ (Dom_u\ k) \wedge Ran_u\ k = Unit_u\ F_c \\
&\qquad \wedge\ s = \Omega_c\ _c\ (\Phi_c\ (k\ @_u\ l))\ _c\ (\Phi_c\ (m\ @_u\ n)) \\
&\qquad \wedge\ t = \Omega_c\ _c\ (\Phi_c\ l)\ _c\ (\Phi_c\ n)) \\
&\quad \vee\ (\exists k\ m\bullet\ \varnothing_u \mapsto_u T_c \in_u k \\
&\qquad \wedge\ \varnothing_u \mapsto_u t \in_u m \\
&\qquad \wedge\ s = \Omega_c\ _c\ (\Phi_c\ k)\ _c\ (\Phi_c\ m))
\end{aligned}$$

As is the infinitary "map".

HOL Constant

$$\mathbf{\Psi_{red}} : 'a\ RED$$

$$\begin{aligned}
&\forall s\ t\bullet\ \Psi_{red}\ s\ t \Leftrightarrow \exists f\ l\ m\bullet\ Dom_u\ l = Dom_u\ m \\
&\qquad \wedge\ Ordinal_u\ (Dom_u\ l) \\
&\qquad \wedge\ m = (\lambda_u\ x\bullet\ f\ _c\ x)(Dom_u\ l) \\
&\qquad \wedge\ s = (\Psi_c\ _c\ f)\ _c\ (\Phi_c\ l) \\
&\qquad \wedge\ t = \Phi_c\ m
\end{aligned}$$

Direct combinatorial reducibility is the union of these relationships. Because the infinitary combinators are so much more complex than the finitary combinators in ways which are probably not pertinent to the method of defining equivalence, I propose to separate out the two kinds of reduction, and undertake the development in the first instance using only the finitary part.

The finitary combinators yield this notion of reducibility:

HOL Constant

$$\mathbf{DComRed} : 'a\ RED$$

$$\forall s\ t\bullet\ DComRed\ s\ t \Leftrightarrow Kred\ s\ t \vee Sred\ s\ t$$

With the following for the infinitary combinators, which we will consider no further until we have demonstrated the viability of our method using only the finitary combinators.

HOL Constant

$$\mathbf{DiComRed} : 'a\ RED$$

$$\forall s\ t\bullet\ DiComRed\ s\ t \Leftrightarrow \Omega_{red}\ s\ t \vee \Psi_{red}\ s\ t$$

HOL Constant

$$\mathbf{DbComRed} : 'a\ RED$$

$$\forall s\ t\bullet\ DbComRed\ s\ t \Leftrightarrow Kred\ s\ t \vee Sred\ s\ t \vee \Omega_{red}\ s\ t \vee \Psi_{red}\ s\ t$$

3.2.3 Extraction of Negative Part

Since the negative part of the partial equivalence relation (and the positive part for that matter) is encoded in the positive part through reductions of equivalences, it is possible to recover it from the positive part.

This is how it is done:

HOL Constant

$$\boldsymbol{Np} : (' a\ RED) \to (' a\ RED)$$

$$\forall r\bullet\ Np\ r\ =\ \lambda x\ y\bullet\ r\ ((\$\equiv_c\ _c\ x)\ _c\ y)\ F_c$$

This just says that two terms are distinct if the equivalence between them converts to F_c.

3.2.4 Equality

We now consider the question, when can we know that two combinators are equal or not equal.

The first obvious principle is that if two combinators are convertible then they are equal. We cannot adopt the converse principle, that two combinators which are not convertible are distinct, because the equality conditions feed into the conversion rules through the equality combinator. So we have to have some definite criteria for distinctness which we are confident will not include any combinator pairs which might ever be found to be convertible.

There are three parts to these criteria for distinctness.

The first part is that all the constants are to be considered distinct. This is consistent with the reduction rules for those constants to which we assign a definite meaning.

This provides an initial value for the inequality relation.

HOL Constant

$$\boldsymbol{Ineq0} : ' a\ RED$$

$$Ineq0\ =\ \lambda x\ y\bullet\ \ \exists v\ w\bullet\ x\ =\ MkCcon\ v\ \wedge\ y\ =\ MkCcon\ w\ \wedge\ \neg\ v\ =\ w$$

The second part is that if two combinators when applied to equal lists of arguments reduce to combinators which are known to be distinct, then they are themselves distinct.

We need a closure operation which will take a notion of inequality and add to it any further inequalities which are immediately apparent from that inequality relation using the second rule just cited.

HOL Constant

$$\boldsymbol{EqSeq} : (' a\ RED) \to (' a\ RED)$$

$$\forall eq\bullet\ EqSeq\ eq\ =\ \lambda v\ w\bullet\ Fun_u\ v\ \wedge\ Fun_u\ w\ \wedge\ \ Dom_u\ v\ =\ Dom_u\ w\ \wedge\ Ordinal_u\ (Dom_u\ v)$$
$$\wedge\ \forall x\bullet\ x\ \in_u\ Dom_u\ v\ \Rightarrow\ eq\ (v\ _u\ x)\ (w\ _u\ x)$$

HOL Constant

$$\textbf{\textit{IneqStep}} : ('a\ RED) \to ('a\ RED)$$

$$\forall eq\bullet\ IneqStep\ eq = \lambda x\ y\bullet\ Ineq0\ x\ y \vee Np\ eq\ x\ y$$
$$\vee\ \exists v\ w\bullet\ EqSeq\ eq\ v\ w \wedge Np\ eq\ (x\ _c\ v)\ (y\ _c\ w)$$

3.2.5 Illative Reduction

Reduction for the illative combinator $\equiv_c$ is intended to encapsulate reducibility as a whole. We could define such a notion here in terms of closure of the reducibility relation for the pure combinators, but this would fall short of the entire reducibility relation by not recognising the reductions it introduces. The definition needs to be recursive, reducibility needs to be defined in terms of itself.

Such a recursive definition can be realised by taking a fixed point of a functor which defines reducibility given some supposed definition of reducibility.

In constructing such a functor we need to specify the reductions which take place on applications of the equality combinator, which will be done using the supplied equivalence relation. This will be a partial relation in the form of two disjoint relations, one positive and one negative. If the positive relation holds over the arguments the equation reduces to T_c if the negative relation holds the equation reduces to F_c.

HOL Constant

$$\textbf{\textit{EqStep}} : ('a\ RED) \to ('a\ RED)$$

$$\forall eq\bullet\ EqStep\ eq = \lambda v\ w\bullet\ \exists c\ i1\ i2\bullet\ c \mapsto_u i1 \in Csyntax \wedge c \mapsto_u i2 \in Csyntax$$
$$\wedge\ Dom_u\ i1 = Dom_u\ i2 \wedge (\forall x\bullet\ x \in_u Dom_u\ i1 \Rightarrow eq\ (i1\ _c\ x)\ (i2\ _c\ x))$$

HOL Constant

$$\equiv_{\textbf{\textit{red}}} : ('a\ RED) \to ('a\ RED)$$

$$\forall eq\bullet\ \equiv_{red}\ eq = \lambda x\ y\bullet\ \exists l\ m\bullet\ x = ((\$\equiv_c\ c)\ _c\ l)\ _c\ m)$$
$$\wedge\ (EqStep\ eq\ x\ y \wedge y = T_c \vee IneqStep\ eq\ x\ y \wedge y = F_c)$$

3.2.6 Closure

There must be a name for this.

I have two versions, not sure yet which to use.

HOL Constant

$$\textbf{\textit{RedClosed1}} : ('a\ RED) \to BOOL$$

$$\forall r\bullet\ RedClosed1\ r = \forall c\ i1\ i2\bullet\ c \mapsto_u i1 \in Csyntax \wedge c \mapsto_u i2 \in Csyntax$$
$$\wedge\ Dom_u\ i1 = Dom_u\ i2 \wedge (\forall x\bullet\ x \in_u Dom_u\ i1 \Rightarrow r\ (i1\ _c\ x)\ (i2\ _c\ x))$$
$$\Rightarrow\ r\ (c \mapsto_u i1)\ (c \mapsto_u i2)$$

HOL Constant

$$\boldsymbol{RedClosed2} : (\prime a\ RED) \to BOOL$$

$$\forall r \bullet\ RedClosed2\ r = \forall c\ d\ e\ f\ g\ h \bullet\ c \mapsto_u e \in Csyntax \wedge d \mapsto_u f \in Csyntax \wedge r\ (c \mapsto_u e)\ (d \mapsto_u f) \wedge EqSeq\ r\ g\ h \Rightarrow r\ (c \mapsto_u e\ @_u\ g)\ (c \mapsto_u f\ @_u\ h)$$

I'll use them both pro-tem.

HOL Constant

$$\boldsymbol{RedClosure} : (\prime a\ RED) \to (\prime a\ RED)$$

$$\forall r \bullet\ RedClosure\ r = Snd\ (EquivClosure\ (Csyntax, \lambda x\ y \bullet \forall r1 \bullet\ RedClosed1\ r1 \wedge RedClosed2\ r1 \wedge (Csyntax, r) \subseteq (Csyntax, r1) \Rightarrow r1\ x\ y))$$

3.2.7 The Semantic Functor

The functor with which we define equivalence must a monotone functor over partial equivalence relations. It requires, or rather consists of a positive and a negative closure operator, each of which has access to the partial equivalence relation and delivers one half of the resulting partial equivalence relation.

HOL Constant

$$\boldsymbol{ILamFunct} : (\prime a\ RED) \to (\prime a\ RED)$$

$$\forall r \bullet\ ILamFunct\ r = RedClosure\ (\equiv_{red}\ r)$$

To take a fixed point we define a version of this as an operator over sets.

HOL Constant

$$\boldsymbol{ILamFunctS} : (\prime a\ GSU \times \prime a\ GSU)SET \to (\prime a\ GSU \times \prime a\ GSU)SET$$

$$\forall s \bullet\ ILamFunctS\ s = \{x \mid ILamFunct(\lambda x\ y \bullet (x,y) \in s)\ (Fst\ x)(Snd\ x)\}$$

3.2.8 Equivalence

Pure infinitary combinatorial equivalence is then:

HOL Constant

$$\boldsymbol{PIComEq} : \prime a\ GSU\ SET \times (\prime a\ GSU \to \prime a\ GSU \to BOOL)$$

$$PIComEq = EquivClosure(Csyntax, \lambda x\ y \bullet (x,y) \in Lfp\ ILamFunctS)$$

$$\boldsymbol{Equiv\_PIComEq\_thm} = \vdash Equiv\ PIComEq$$
$$\boldsymbol{Fst\_PIComEq\_thm} = \vdash Fst\ PIComEq = Csyntax$$
$$\boldsymbol{CSyntax\_PIComEq\_thm} = \vdash \forall\ x \bullet\ x \in Csyntax \Rightarrow x \in EquivClass\ PIComEq\ x$$

We now define a set of equivalence classes which will form a new type of combinators.

HOL Constant

$$\boldsymbol{PureInfComb} : {}'a\ GSU\ SET \to BOOL$$

$$\forall x\bullet\ PureInfComb\ x \Leftrightarrow x \in QuotientSet\ Csyntax\ (Snd\ PIComEq)$$

$$\boldsymbol{PureInfComb\_\exists\_lemma} =$$
$$\vdash\ \forall\ x\bullet\ x \in Csyntax \Rightarrow EquivClass\ PIComEq\ x \in PureInfComb$$
$$\exists\_\boldsymbol{PureInfComb\_lemma} =$$
$$\vdash\ \exists\ x\bullet\ x \in PureInfComb$$

3.3 Consistency

At this point it is desirable to prove something like the Church-Rosser theorem for this system, so that we know that not all combinators are equivalent. Since this is probably hard I am exploring a bit more before digging in to the proof.

The most doubtful part of the enterprise is not the infinitary combinators. It is the illative combinator. So its probably worth working through the handling of the illative combinator in a system with no infinitary combinators, and adding in the infinitary combinators (whose purpose is just to add strength), when the treatment of the illative combinator has been proven in the simpler context.

HOL Constant

$$\boldsymbol{ChurchRosser} : ({}'a \to {}'a \to BOOL) \to BOOL$$

$$\forall r\bullet\ ChurchRosser\ r \Leftrightarrow \forall w\ x\ y\bullet\ r\ w\ x \wedge r\ w\ y \Rightarrow \exists z\bullet\ r\ x\ z \wedge r\ y\ z$$

I approach the Church-Rosser proofs (loosely) following a method I took from Barendregt [1] (which he attributes to W. Tait and Per Martin-Löf). This consists in proving that the transitive closure of a Church-Rosser relation is Church-Rosser, and then finding a Church-Rosser relation whose transitive closure is the desired relation.

The first theorem is:

$$\boldsymbol{CrTc\_thm} = \vdash\ \forall\ r\bullet\ ChurchRosser\ r \Rightarrow ChurchRosser\ (tc\ r)$$

Typically a relation of direct reduction (or its reflexive closure) will not be Church-Rosser. This is because a reduction of a redex may replicate some other redex (contained within it). To reduce that other redex after the first reduction will require more than one reduction. To get a Church-Rosser relation we need to allow all these 'residuals' to be reduced in one step.

I therefore define an operator which takes a relation of direct reduction and makes from it a parallel reduction relation which is more likely to be Church-Rosser. The idea is that any set of redexes in the term can be reduced in a single step, but that no redex which was not in that original term can be reduced in that step.

There are (at least!) two ways of defining this relationship.

The first (which is the kind of approach used in Barendregt) is to define it as undertaking a single depth-first scan of the term reducing any of the redexes.

The second is to devise a representation of a set of redexes and to define the reduction parameterised by such a set of redexes. The desired relation is then obtained by existentially quantifying over sets of redexes.

I started out with the second approach, but this was rapidly getting too complicated, so I am trying out the first approach.

They are presented here (so far as they go) in reverse order.

With either of these approaches it is possible to identify a maximal direct reduction of any term. One can argue that the relation is Church-Rosser because whatever reduction is undertaken on a term it can be converted by just one more reduction to the maximal reduct of the original term. Thus for each term, there is some other term (its maximal reduct), which could serve as the bottom of the diamond whatever two other reducts were considered. One might hope that this observation would permit a simplification of the Church-Rosser proof, but so far I have not found a way to realise that simplification.

I shall keep my eye on this as I work through the preliminary stages of the proof.

In the following section I make some steps towards a proof based on coordinates, in the subsequent section I mentally put this aside and try an approach more closely following one presented by Barendregt.

3.3.1 Approach based on Coordinates

Redexes (or subterms in general) in a combinatorial term can be identified by coordinates, finite sequences of (not necessarily finite) ordinals suffice. We do need to insist that the coordinates are coordinates of redexes.

I define first the operation of extracting from a combinator the substructure identified by a set of coordinates. I use the type LIST for finite sequences. When it comes to selecting a subterm, a term is an ordered pair of which the second element is a sequence of terms (the first is a constant name). The sequence is a function whose domain is an ordinal, so we just apply this sequence to the relevant coordinate (which is here the head h of the list of coordinates).

HOL Constant

$$\boldsymbol{SubTerm} : 'a\ GSU\ LIST \to 'a\ GSU \to 'a\ GSU$$

$$\forall h\ t\ tl \bullet \quad SubTerm\ []\ t = t$$
$$\wedge \quad SubTerm\ (Cons\ h\ tl)\ t = SubTerm\ tl\ ((Snd_u\ t)\ _u\ h)$$

Since this function is total (and therefore will yield junk if there is no subterm addressed by the coordinates) I also need a test so I know whether a coordinate list does actually point to a subterm. This test assumes that its second argument is a combinatory term, and tests whether the coordinate list points to a subterm.

HOL Constant

$$\boldsymbol{is\_SubTerm} : 'a\ GSU\ LIST \to 'a\ GSU \to BOOL$$

$$\forall h\ t\ tl \bullet \quad (is\_SubTerm\ []\ t \Leftrightarrow T)$$
$$\wedge \quad (is\_SubTerm\ (Cons\ h\ tl)\ t \Leftrightarrow$$
$$h \in_u Dom_u\ (Snd_u\ t) \wedge is\_SubTerm\ tl\ ((Snd_u\ t)\ _u\ h))$$

SubTerm_in_Csyntax_lemma =

$\vdash \forall\ t\bullet\ t \in Csyntax \Rightarrow (\forall\ cs\bullet\ is\_SubTerm\ cs\ t \Rightarrow SubTerm\ cs\ t \in Csyntax)$

This isn't quite a sufficient check. It is important that nothing is accepted as a redex unless it really is one, particularly not if it might be turned into one by some other reduction (this is because we don't want the description of the reductions required to give different results after some other reduction than before), so the best thing is to use the direct reduction relation to test whether the identified subterms are redexes, i.e. we just check whether the subterm is in the domain of the relationship.

The parameters here are:

r the direct reduction relation

t the combinatory term

cls a set of coordinate lists

and the value is true if all the identified subterms are in the domain of the reduction relation.

HOL Constant

is_redex_set : $'a\ RED \to 'a\ GSU \to ('a\ GSU\ LIST)SET \to BOOL$

$\forall(r{:}'a\ RED)\ t\ cls\bullet\ is\_redex\_set\ r\ t\ cls \Leftrightarrow$
$\quad \forall cs\bullet\ cs \in cls \Rightarrow is\_SubTerm\ cs\ t \wedge (\exists y\bullet\ r\ (SubTerm\ cs\ t)\ y)$

We now need to define the effects of applying some such set of reductions. There are two effects of interest. The first of course is the resulting term.

The second is a residual map, which tells us how to find the instances in the new term of the subterms of the old term. To compute the residual map we assume that a residual map is available for the direct reduction relation, and use that in computing the residual map function for the derived parallel reduction relation.

The best way to compute the new term is with a depth first traversal. This ensures that redexes for reduction are are not replicated until they have been reduced.

The definition is inductive over the structure of the combinatory terms. At each stage in the recursion the set of coordinate sequences is filtered by selecting only those which point within the selected subterm and chopping the head of those sequences. The reduction of the term takes place after the reduction of its subterms. We assume that the second parameter is a syntactically valid combinatory term and that the set of coordinate sequences passes the *is_redex_set* test.

The following function restricts a set of coordinates to the relative coordinates of subterms of a give top-level sub-term, identified by its coordinate.

HOL Constant

cls_restrict : $('a\ GSU\ LIST)SET \to 'a\ GSU \to ('a\ GSU\ LIST)SET$

$\forall cls\ (t{:}'a\ GSU)\bullet\ cls\_restrict\ cls\ t = \{v{:}\ 'a\ GSU\ LIST \mid Cons\ t\ v \in cls\}$

We need direct reduction to be deterministic, i.e. to be a many-one relation. Assuming that it is we can use the following function to perform a reduction.

HOL Constant

$$\textbf{\textit{reduce}} : 'a\ RED \to 'a\ GSU \to 'a\ GSU$$

$$\forall (r{:}'a\ RED)\ (t{:}'a\ GSU)\bullet\ reduce\ r\ t =$$
$$\epsilon nt\bullet\ r\ t\ nt \vee ((\neg\ \exists nt2\bullet\ r\ t\ nt2) \wedge nt = t)$$

We want to map certain operations over terms to be applied at certain locations specified by coordinates. Some general combinators might be the best way to achieve this.

We need to obtain from a parallel reduction not only the resulting term but also a map which tells us the location of any residuals. The function to be mapped over the term must therefore not simply be a function over combinators, but a function which transforms a combinator and delivers a residual map (which is a kind of map over coordinate lists).

The type of such a function is therefore:

SML

$$declare\_type\_abbrev\ (\texttt{"}CT\texttt{"},\ [\texttt{"}'a\texttt{"}],\ \ulcorner{:}'a\ GSU \to ('a\ GSU \times ('a\ GSU\ LIST \to 'a\ GSU\ LIST))\urcorner);$$

We could build into the mapping function the means of composing the resulting coordinate transformers obtained in the traversal, but it is better to supply that as a parameter. The type of such a parameter would be:

SML

$$declare\_type\_abbrev\ (\texttt{"}CC\texttt{"},\ [\texttt{"}'a\texttt{"}],\ \ulcorner{:}('a\ GSU \times ('a\ GSU\ LIST \to 'a\ GSU\ LIST))\ LIST$$
$$\to 'a\ GSU\ LIST \times ('a\ GSU\ LIST \to 'a\ GSU\ LIST)\urcorner);$$

HOL Constant

$$\textbf{\textit{depth\_map\_aux}} : ('a\ GSU \to 'a\ GSU) \to 'a\ GSU \to ('a\ GSU\ LIST)\ SET \to 'a\ GSU$$

$$\forall f\ c\ ts\bullet\ depth\_map\_aux\ f\ (c \mapsto_u ts) = \lambda cls\bullet$$
$$let\ newts =$$
$$Image ep_u\ (\lambda op\bullet\ Fst_u\ op \mapsto_u (ts \lhd_{ue}\ depth\_map\_aux\ f)$$
$$(Snd_u\ op)\ (cls\_restrict\ cls\ (Fst_u\ op)))\ ts$$
$$in\ if\ [] \in cls\ then\ f\ (c \mapsto_u newts)\ else\ c \mapsto_u newts$$

HOL Constant

$$\textbf{\textit{depth\_map}} : ('a\ GSU \to 'a\ GSU) \to ('a\ GSU\ LIST)SET \to 'a\ GSU \to 'a\ GSU$$

$$\forall f\ cls\ t\bullet\ depth\_map\ f\ cls\ t = depth\_map\_aux\ f\ t\ cls$$

This function defines the term which results from a parallel reduction.

HOL Constant

$$\textbf{\textit{par\_reduce}} : 'a\ RED \to ('a\ GSU\ LIST)SET \to 'a\ GSU \to 'a\ GSU$$

$$\forall r\ c\ ts\ cls\bullet\ par\_reduce\ r\ cls\ (c \mapsto_u ts) =$$
$$depth\_map\ (reduce\ r)\ cls\ (c \mapsto_u ts)$$

We need to be able to compute residual maps which show where to find the residuals of each redex after a (parallel) reduction. It is probably easier just to map the movement of all combinations irrespective of whether they are redexes. I think functions from coordinate sets to coordinate sets will probably do the trick. We suppose that the direct reductions are specified both as a relation and as a residual map. We can then compute a residual map for a parallel reduction by composing these maps in a manner which corresponds to the depth first traversal which defines the parallel reduction derived from some direct reducibility relation.

We then prove for each direct reduction a commutativity result, which says that performing some reduction before that direct reduction is equivalent to applying that reduction to each of the residuals after the direct reduction. Next I prove that combining direct reductions preserves this property, and finally that lifting to parallel reduction gives a similar commutativity for the resulting parallel reduction.

From this to the diamond property should be straightforward, which then applies to the transitive closure which is close to the desired reduction relation. This approach gives Church-Rosser for all by the equivalence combinator.

I then define a functor which takes a value for a direct equivalence reduction satisfying the relevant commutativity property, and yields a Church-Rosser reduction relation incorporating the other combinators, and a new notion of equivalence hopefully also commutative. The functor will be monotone, and its least fixed point will be the final reduction relation, which will be a limit of Church-Rosser relations. So then I need to prove that the functor some relevant continuity property so that its least fixed point is also Church-Rosser.

3.3.2 More closely following Barendregt

The previous approach looks too complex. Eventually I returned to Barendregt for better ideas. My main reason for not following him more closely had been that he works with systems which contain variables, whereas I am working with a system which lacks variables. However, it now occurs to me that I can do with "inert" combinators the kinds of thing which are normally done with free variables, and so it is plausible that following Barendregt more closely will be possible. The signficance of this is that variables provide a simple way to track residuals, and the main complexity arising from my previous proof ideas arose from tracking residuals so that a deferred reduction can be completed at all the instances.

Barendregt leaves the Church-Rosser proof for combinatory logic as an exercise (exercise 7.13), so it ought to be manageable, even perhaps with the added complications here. Before noting the complications I consider the clues available in Barendregt on how to do the exercise.

There is an earlier proof that beta reduction in the lambda-calculus is Church-Rosser in Chapter 3 (Theorem 3.2.8), which can be simplified to give the proof for combinatory logic. The approach is to define a restricted notion of beta reduction whose transitive closure is CR and to prove that this more restrictive notion is CR. The particular reduction used for the lambda calculus cannot be used for combinatory logic, but Barendregt suggests instead the reduction which allows parallel reduction of any disjoint set of redexes.

The plan here is to do a version of that adapted to our infinitary combinators, and follow through in the spirit of the proof for beta reduction in Chapter 3.

In addition to allowing for infinitary combinators, we have to parameterise the proof to work with any notion of direct reduction which satisfies certain conditions, so that we can then apply it to systems involving illatory combinators. Exactly what these conditions should be is not yet clear, but I propose to start by trying something analogous to Barendregt's Lemma 3.2.4 which may be

paraphrase as "reduction commutes with substitution".

So we start with the definition of disjoint parallel reduction, which is given as an operator over relations.

HOL Constant

$$\boldsymbol{disj\_par\_reduce} : 'a\ RED \to 'a\ RED$$

$$\begin{array}{l}
\forall r\ c\ ts\ nt\bullet\ disj_par_reduce\ r\ (c \mapsto_u ts)\ nt \Leftrightarrow \\
\qquad r\ (c \mapsto_u ts)\ nt \\
\quad \vee \quad \exists ts'\bullet\ (\forall i\ v\bullet\ i \mapsto_u v \in_u ts \\
\qquad\qquad \Leftrightarrow \exists v'\bullet\ i \mapsto_u v' \in_u ts' \\
\qquad\qquad\quad \wedge\ ((ts \lhd_{ue} (disj_par_reduce\ r))\ v\ v' \vee v' = v)) \\
\qquad\qquad \wedge\ (\exists i\ v\ v'\bullet\ i \mapsto_u v \in_u ts \wedge i \mapsto_u v' \in_u ts' \\
\qquad\qquad\quad \wedge\ ((ts \lhd_{ue} (disj_par_reduce\ r))\ v\ v')) \\
\qquad\qquad \wedge\ nt = (c \mapsto_u ts')
\end{array}$$

The first result I need to prove is that disjoint parallel reduction commutes with substitution, provided that direct reduction does. So I need to define substitution. We have no variables so the substitution in question is substitution for a combinator, which with these infinitary combinators is not quite as straightforward as with the traditional combinators, it invoves concatenating the list of arguments in the two expressions involved.

In the intended applications substitution is for inert combinators only, but we do not need to take that into consideration in the definition of substitution. In the following:-

- the first argument is the name of the combinator for which the substitution is being made (not actually a combinatory term, just the name)
- the second is the combinatory term which is being substituted for that combinator, and
- the third is the combinatory term into which substitution takes place.

HOL Constant

$$\boldsymbol{subs} : 'a\ GSU \to 'a\ GSU \to 'a\ GSU \to 'a\ GSU$$

$$\begin{array}{l}
\forall\ cn\ c\ ts\ s\bullet\ subs\ cn\ s\ (c \mapsto_u ts) = \\
\qquad if\ c = cn \\
\qquad then\ (Fst_u\ s \mapsto_u ((Snd_u\ s) @_u (RanMap_u\ (ts \lhd_{ue} (subs\ cn\ s))\ ts))) \\
\qquad else\ (c \mapsto_u (RanMap_u\ (ts \lhd_{ue} (subs\ cn\ s))\ ts))
\end{array}$$

$$\boldsymbol{r\_disj\_par\_thm} = \vdash \forall\ r\ x\ y\bullet\ x \in Csyntax \wedge r\ x\ y \Rightarrow disj\_par\_reduce\ r\ x\ y$$

We do now need the notion of an inert combinator, relative to some notion of reduction. This takes a reduction and returns a property of combinator names which is satisfied only be combinators which are never reduced by the reduction.

HOL Constant

$$\boldsymbol{inert} : 'a\ RED \to 'a\ GSU \to BOOL$$

$$\forall\ r\ c\bullet\ inert\ r\ c \Leftrightarrow \neg\ \exists\ ts\ c'\ ts'\bullet\ r\ (c \mapsto_u ts)\ (c' \mapsto_u ts') \wedge \neg\ c = c'$$

$$\textbf{\textit{inert\_disj\_par\_thm}} = \vdash \forall\ r\ cn\bullet\ inert\ (disj\_par\_reduce\ r)\ cn \Leftrightarrow inert\ r\ cn$$

The property of reductions that they commute with substitution is now definable.

HOL Constant

$$\textbf{\textit{comm\_subs}} : 'a\ RED \to BOOL$$

$$\forall\ r\bullet\ comm\_subs\ r \Leftrightarrow \forall cn\bullet\ inert\ r\ cn \Rightarrow \forall\ s\ t\ t2\bullet\ s \in Csyntax\ \wedge\ t \in Csyntax \Rightarrow r\ t\ t2 \Rightarrow r\ (subs\ cn\ s\ t)\ (subs\ cn\ s\ t2)$$

The following theorem states that substitution commutes with disjoint parallel reduction if it commutes with direct reduction.

3.3.3 Proof Strategy

The strategy is, first prove a *Church/Rosser* theorem for the pure part of the infinitary combinatory logic, i.e. the bit without the equivalence combinator reductions. Then do the rest, which we won't think about yet (much).

A Church/Rosser theorem was first proved for the λ-calculus by Church and Rosser[2]. The result we require here should be simpler to produce because we have only a combinatory logic. However, the adoption of an infinitary system naturally complicates matters, though possibly not greatly.

The following proof strategem is new, so far as I am aware. The need to completely formalise the proof forces one to be more definite about certain things, which at first glance may seem to impose additional difficulties, but ultimately may assist in finding a nice proof.

The required result is rather like a commutativity result for combinatory reduction, like saying that the order of reduction is not very significant. To state exactly what that means in this context is not so straightforward as it is to state the commutativity of addition in arithmetic. The result obtains not because of any peculiarities of the specific reductions under consideration, but simply because they are combinatory, or some such relatively simple characteristic which they share. It would hold if they were purely combinatory, as S and K are, but we also have combinators which are not quite so pure, the projection combinator and the equivalence combinator. For these, being 'nearly pure combinators' in some sense to be defined, will have to suffice.

The respect in which the map and projection combinators deviate from being purely combinatory is that they are not entirely careless as to their arguments. A pure combinator, in the sense in which S and K and anything composed from them is a pure combinator, apply irrespective of the values of their arguments, just so long as there are enough of them, and they deliver results in which their arguments are replicated whole (or not replicated at all) without being in any way modified. The projection combinator is fussy about the value of its first argument, which serves to stipulate an element of its second arguments which is to be extracted. The map combinator is not fussy about the values of its arguments, but it mangles the first rather than merely replicating it. The feature which saves these is that they nevertheless 'respect reducibility', but unfortunately that is a difficult characteristic to define in this context and so some simpler sufficient condition for their admissibility is desirable. Perhaps that they are sensitive only to a part of their argumenst which is in normal form, and something else about how the map combinator forms its result.

The relevance to the Church Rosser theorem is that performing reductions on operands will have

very limited effects on the permissible reductions of the whole, the same reductions will be possible and the only effect is on the copies of that operand in the resulting structure.

With the projection combinator, an initial segment of the operands must normalise before reduction can take place, and the reduction is sensitive to the value of the selector. The sensitivity is therefore safe, insofar as the relevant values cannot possibly change if the reduction is deferred. We could therefore stipulate that a reduction has a pattern (or more than one) which is a combinatory structure in which the only parts which are not variables are normal terms. However, the notion of normal term depends upon all the reductions so this is not an easy thing to include in the criteria, perhaps it would be better just to allow T and F. That would be OK for the projection combinator, but not for equivalence.

Equivalence does not reduce without inspecting the operands, and does not look for T or F, or for normality. Its saving grace is that if, while reduction is deferred, one of two identical operands is reduced, then it will be possible to reduce the other to make the two operands equal again and effect the reduction of the equality at a later date.

In order to be able to express the precise sense in which these reductions commute, it is necessary to say what a reduction *is*, what it operates *on* and what it *does* in manner which makes it possible to talk about effecting the same reduction either before or after some other transformations. This means that we have to include sufficient information when undertaking a transformation to know how to apply the same reduction either before or after. The most important element of this is keeping track of so called 'residuals', i.e. keeping track of what happens to redexes of interest when they are moved about by some reduction.

This is achieved using a system of coordinates. A coordinate determines a place in a combinatory term and a constituent combinator which occurs within the first combinator. When a reduction takes place, it does so by reduction of an occurrence which may be identified by such a coordinate. Every constituent of the combinator which is being reduced may then appear in zero, one, or more than one place in the result of the reduction. The effect of a reduction on the location of the constituent combinators may be recorded as a map from locations (as coordinates) to sets of locations. If coordinates are relative then this map can be rendered using relative coordinates.

3.3.4 Representation of Reductions

The natural choice for our calculus, both for absolute and for relative coordinates, is a sequence of ordinals. This is such a simple notion that one is immediately tempted to go back and start again using certain maps over sequences of ordinals to represent the combinators. Without prejudice to whether this should be done, it seems likely to be a good idea to use this representation for combinatory terms in the proof, which will then be straightforward to transfer back to the present representation.

We need representative for the following kinds of entity:

- Terms and Coordinates
- Reduction Patterns and applicability
- Reduction effects or transformations

In this calculus, give a suitable notion of coordinate, a combinatory term is determined completely by a map from coordinates to constant names (or possibly variables, for metatheoretic purposes). A reduction pattern is a combinatory term containing some variables (and subject possibly to other

constraints), and its applicability is determined by a simple matching algorithm. The local effect of a reduction is a new term and a residual map, which maps each argument number in the original pattern to the set of relative coordinates at which it appears in the result of the transformation. From the local effect of a reduction, we obtain its global affect, which is the application to a subterm of some larger combinatory term, and the conversion of the residual map from local to global coordinates.

The representation I propose is a map from coordinates to combinator names or variables (I'm not sure whether I need the variables yet, but we'll have them in just in case). A combinator is just a value of type $\ulcorner : 'aGSU \urcorner$, for the variables ordinals of type $\ulcorner : 'aGSU \urcorner$ will do. To code the alternative we enclose a combinator name in two singleton sets so that it cannot be mistaken for an ordinal.

So a 'combinator map' may be defined as follows.

1. it is a function
2. values in the domain of the function are sequences of ordinals
3. all initial segments of values in the domain are also in the domain
4. if some sequence of ordinals y is in the domain and x is some other sequence of ordinalys which has the same length as y and at every point is not greater than y, then it also is in the domain.

HOL Constant

$$\boldsymbol{is\_cmap} : 'a\ GSU \rightarrow BOOL$$

$$\begin{aligned}
&\forall m\bullet\ is_cmap\ m \Leftrightarrow Fun_u\ m\\
&\wedge\ (\forall x\bullet\ x \in_u Dom_u\ m \Rightarrow Seq_u\ x \wedge \forall y\bullet\ y \in_u Ran_u\ x \Rightarrow Ordinal_u\ y)\\
&\wedge\ (\forall x\ y\bullet\ y \in_u Dom_u\ m \wedge Dom_u\ x \subseteq_u Dom_u\ y \Rightarrow x \in_u Dom_u\ m)\\
&\wedge\ (\forall x\ y\bullet\ y \in_u Dom_u\ m \wedge Dom_u\ x = Dom_u\ y \wedge (\forall z\bullet\ z \in_u Dom_u\ x \Rightarrow x\ _u\ z \subseteq_u y\ _u\ z)\\
&\qquad\qquad \Rightarrow x \in_u Dom_u\ m)
\end{aligned}$$

3.3.5 The Reduction Graph

This is a fresh start, in the exposition. Only after I have written it will I know what to do with the preceeding material.

The idea is that a combinatory term together with a set of reduction rules determines a directed graph with labelled nodes and arcs. The nodes and the arcs are labelled by distinct ordinals, together with certain other information.

The other information on a node is a combinatory term. The node labelled zero is special and the combinatory term is the starting point for the reductions represented by the graph. A location is a position in one of the combinatory terms which appear on nodes of the graph. It consists of the number of the node and a coordinate within that term. A location map is a map which assignes to each term affected by a reduction the new locations.

The arcs are *direct reductions*, the nodes are combinatory terms together with certain other information. A direct reduction consists of a direct reduction *rule* and a location. The other information on the nodes is a location map, which shows where constituents of 'earlier' terms appear in the present term, these are sometimes called 'residuals'.

3.4 Coordinate Set Equivalence

This is a third attempt. It has been too early to write about this, because most of the action is taking place in my head, and the evolution of ideas continually invalidates anything I may previously have written.

In this version coordinates play a larger role so I begin with discussing coordinates.

3.4.1 Coordinates

To refer to specific parts of an infinitary combinatory term we use coordinate systems. Because of the infinitary nature, a set of coordinates is a sequence of ordinals. I think that because of the way the combinatory terms are organised this will always be a finite sequence of ordinals, though the ordinals will not in general be finite. (This reflects the well-foundedness of these terms, there could only be infinite sequences if there were infinite descending membership chains in the underlying set theoretic representation, the infinitary nature appears, as it were, horizontally.)

A sequence of ordinals may be used to refer to parts of combinatory terms in any of the following ways:

1. In its primary significance the coordinates refer to a sub-term which is the whole of or a part of the term within which the coordinate system is being used. The empty sequence refers to the whole term. Give a term t referred to by some coordinates c, the coordinates of the $\alpha_t h$ element in its argument list consist of the sequence obtained by appending α to c.
2. The coordinates may also be used to refer to a combinatory constant, which would be the one at the head of the sub-term which is referred to by the coordinates.
3. Coordinates may be use to refer to an application, bearing in mind that in this system the correspondence between subterms and applications is broken. In this case you take the sub-term which is referred to by the coordinates, and the application in question is that of which the sub-term is the argument. Bear in mind that an infinitary combinatory term is in the sense of application used here, a possibly infinite sequence of applications.

3.4.2 Residuals and Node Formation

In order to be able to talk about effecting the same reduction under differing circumstances (different points in some sequence of reductions), we need a way of describing a reduction which is independent of the details which vary. The signficant changes are changes in location of the redex, taking into account that it may be replicated or eliminated.

The replicas of some term after a reduction of sequence of reductions are called its residuals.

If we identify a reduction by the location of its redex, then to be able to talk about the same reduction being applied before and after some other sequence of reductions it is necessary to treat a redex before some sequence of reductions as equivalent to the complete set of its residuals after the reductions. We are therefore looking for an equivalence relation between sets of sets of coordinate sets. That's not quite right, the coordinate sets

3.4.3 Reductions

Given a combinatory term and a set of reduction rules, a reduction, which is a sequence of applications of the rules, could be represented by a (possibly transfinite) sequence of coordinate sequences, if the

rules are deterministic (which our will be). This will not suffice for an approach to the Church-Rosser theorem based on a conception of reduction which is commutative, because the principle way in which 'the same' reduction might differ when its application is deferred is in the coordinates at which the reduction is applied.

3.4.4 Combinatory Direct Reductions

We now define the effects of the primitive pure combinators in two ways. The first is as a transformation on a term represented as a *cmap*. The second is as a residual map, which shows the new locations of the subterms of a redex.

The following function assumes that the argument *cmap* is a K redex, i.e. that its head is K and its argument list has a length greater than 1.

HOL Constant

$$\boldsymbol{K\_cmap} : {'a}\ GSU \to {'a}\ GSU$$

$$\begin{array}{l}
\forall m{:}('a)\,GSU\bullet\ K_cmap\ m = (\lambda_u\ cs\bullet \\
\quad let\ h = Head_u\ cs \\
\quad and\ x = m\ _u\ Unit_u\ (\varnothing_u \mapsto_u \varnothing_u) \\
\quad in\quad m\ _u \\
\qquad\qquad (if\ (Nat_u\ 2) \leqslant_u h \\
\qquad\qquad then\ \ let\ h2 = \quad if\ natural_number_u\ h \\
\qquad\qquad\qquad\qquad\qquad\quad then\ h\ --_u\ (Nat_u\ 1) \\
\qquad\qquad\qquad\qquad\qquad\quad else\ h \\
\qquad\qquad\qquad\quad in\ SeqCons_u\ h2\ (Tail_u\ cs) \\
\qquad\qquad else\ \ if\ h = Nat_u\ 1 \\
\qquad\qquad\qquad\quad then\ Tail_u\ cs \\
\qquad\qquad\qquad\quad else\ cs))\ (Gx_u\ m)
\end{array}$$

HOL Constant

$$\boldsymbol{K\_rmap} : {'a}\ GSU \to {'a}\ GSU \to {'a}\ GSU$$

$$\begin{array}{l}
\forall m\bullet\ K_rmap\ m = \lambda\ cs\bullet \\
\quad let\ h = Head_u\ cs \\
\quad and\ x = m\ _u\ Unit_u\ (\varnothing_u \mapsto_u \varnothing_u) \\
\quad in\quad (if\ (Nat_u\ 2) \leqslant_u h \\
\qquad then\ \ let\ h2 = \quad if\ natural_number_u\ h \\
\qquad\qquad\qquad\qquad then\ h\ --_u\ (Nat_u\ 1) \\
\qquad\qquad\qquad\qquad else\ h \\
\qquad\qquad in\ SeqCons_u\ h2\ (Tail_u\ cs) \\
\qquad else\ \ if\ h = Nat_u\ 1 \\
\qquad\qquad then\ Tail_u\ cs \\
\qquad\qquad else\ cs)
\end{array}$$

4 Primitives of Illative Lambda Calculus

In this section we define the primitive notions of the Illative Lambda Calculus in terms of the underlying model of terms as equivalence classes of combinators (represented as sets in GSU).

In the next section the Illative Lambda Calculus will then be developed in a separate theory abstracted away from the underlying model.

At this point it is not clear what the primitives should be, so this is exploratory material.

4.1 The Type of Lambda-Terms

SML

```
val LT_def = new_type_defn (["LT"], "LT", ["'a"], ∃_PureInfComb_lemma);
```

$$\vert \boldsymbol{LT\_def} = \vdash \exists\, f\bullet\ TypeDefn\ PureInfComb\ f$$

We will have a new theory for the finitary system, which I will try to confine to results in the finitary system. The definitions and proofs will involve materials in the underlying theory, which may well be quite substantial. These materials are supplied here, before we start the new theory.

First we need to define functions to take us between the new abstract entities and their representatives as equivalence classes of combinators.

HOL Constant

$$\begin{array}{l}\vert\ \boldsymbol{abs\_LT} : 'a\ GSU\ SET \to 'a\ LT;\\ \vert\ \boldsymbol{rep\_LT} : 'a\ LT \to 'a\ GSU\ SET\\ \vert\ \hline\\ \vert\qquad (\forall\ a\bullet\ abs\_LT\ (rep\_LT\ a) = a)\\ \vert\quad \wedge\ (\forall\ r\bullet\ PureInfComb\ r \Leftrightarrow rep\_LT\ (abs\_LT\ r) = r)\\ \vert\quad \wedge\ OneOne\ rep\_LT\end{array}$$

$$\vert \boldsymbol{LT\_clauses} = \vdash (\forall\ a\bullet\ abs\_LT\ (rep\_LT\ a) = a) \wedge (\forall\ a\ b\bullet\ rep\_LT\ a = rep\_LT\ b \Leftrightarrow a = b)$$
$$\vert \boldsymbol{LT\_fc\_thm1} = \vdash \forall\ r\bullet\ PureInfComb\ r \Rightarrow rep\_LT\ (abs\_LT\ r) = r$$
$$\vert \boldsymbol{LT\_fc\_thm2} = \vdash \forall\ x\ y\bullet\ PureInfComb\ x \wedge PureInfComb\ y \Rightarrow (abs\_LT\ x = abs\_LT\ y \Leftrightarrow x = y)$$

4.2 Judgements

A proposition is an object of type LT. There is just one true proposition, which is the equivalence class containing T_c. We will call this T_l. To assert a proposition is to identify it with T_l.

Something like a sequent calculus might be best.

SML

```
declare_infix (0, "⊨ₗ");
declare_infix (0, "⊨");
```

HOL Constant

$\$\models_{\boldsymbol{l}} : 'a\ LT\ LIST \to 'a\ LT \to BOOL$

$\forall pl\ (p{:}'a\ LT)\bullet\ (pl \models_l p) \Leftrightarrow \forall_L\ (Map\ (\lambda p\bullet\ T_c \in (rep\_LT\ p))\ pl) \Rightarrow T_c \in (rep\_LT\ p)$

SML

declare_alias ("⊨", ⌜$⊨$l⌝);

4.3 Primitive Combinators

HOL Constant

$\boldsymbol{Lift2l} : 'a\ GSU \to 'a\ LT$

$\forall c\bullet\ Lift2l\ c = abs\_LT\ (EquivClass\ PIComEq\ c)$

The combinators S, K and ≡ are required, the others may not be made visible directly.

HOL Constant

$\boldsymbol{S_l} : 'a\ LT$

$S_l = Lift2l\ S_c$

HOL Constant

$\boldsymbol{K_l} : 'a\ LT$

$K_l = Lift2l\ K_c$

HOL Constant

$\$\equiv_{\boldsymbol{l}} : 'a\ LT$

$\$\equiv_l = Lift2l\ \$\equiv_c$

SML

declare_alias ("S", ⌜S_l⌝);
declare_alias ("K", ⌜K_l⌝);

4.4 Application and Abstraction

To reason about this we are going to need to show that choice of representative does not matter.

SML

```
declare_infix (250, "l");
```

HOL Constant

```
$l : 'a LT → 'a LT → 'a LT
```

```
∀f a• f l a = Lift2l ((ϵx• x ∈ rep_LT f) c (ϵx• x ∈ rep_LT a))
```

SML

```
declare_alias(".", ⌜$l⌝);
declare_infix (250, ".");
```

5 Illative Lambda-Calculus

SML

```
open_theory "icomb";
force_new_theory "ilamb";
force_new_pc "'ilamb";
merge_pcs ["'savedthm_cs_∃_proof"] "'ilamb";
new_parent "equiv";
set_merge_pcs ["icomb", "'ilamb"];
```

```
I = λx•x
```

HOL Constant

```
$Il : 'a LT
```

```
Il = (Sl l Kl) l Kl
```

```
T = λx y• x
```

HOL Constant

```
$Tl : 'a LT
```

```
Tl = Kl
```

```
F = λx y• y
```

HOL Constant

$\$\boldsymbol{F_l} : 'a\ LT$

$F_l\ =\ K_{l\ l}\ I_l$

SML

declare_alias ("I", ⌜I_l⌝);
declare_alias ("T", ⌜T_l⌝);
declare_alias ("F", ⌜F_l⌝);

$\boldsymbol{T_l\_thm} = \vdash [] \models_l T_l$

5.1 Primitive Equality

I expect that non-primitive equalities will be normally used, but these will be defined in terms of the primitive equality.

SML

declare_infix(*210*, "$\equiv_n$");

HOL Constant

$\$\equiv_{\boldsymbol{n}} : 'a\ LT \to 'a\ LT \to 'a\ LT$

$\forall x\ y\bullet\ x \equiv_n y\ =\ \$\equiv_{l\ l}\ x\ _l\ y$

SML

declare_alias ("$\equiv$", ⌜$\$\equiv_n$⌝);

5.2 The System of Type Assignment

SML

set_flag ("*subgoal_package_quiet*", *false*);
set_flag ("*pp_use_alias*", *true*);

6 Postscript

After iterating over the abstract syntax for the infinitary system many times I have now managed to come up with a first cut at the semantics. This was not particularly easy for me, and the result is a bit of a mess, so even if it miraculously turns out actually to be correct, I will still in all probability have to massage it for a while longer to get it in a better state for reasoning with.

The main requirement is something like a consistency proof, which in this context comes down to proving that true and false are not equivalent. I am not rushing into that.

It is now possible, even in default of this consistency result, to start moving forward on the illative lambda calculus by thinking about and defining the primitives, and it may be possible to get some results in that theory (inconsistency, on the face of it, does not make it harder to get results!). So I will explore that a while before even starting on the consistency result, and then perhaps progress the two in tandem.

A Theory Listings

A.1 The Theory icomb

Parents

equiv misc3

Children

ilamb

Constants

$\boldsymbol{CrepClosed}$	$'a\ GSU\ \mathbb{P} \to BOOL$
$\boldsymbol{Csyntax}$	$'a\ GSU\ \mathbb{P}$
$\boldsymbol{CscPrec}$	$'a\ RED$
$\$\lhd_{\boldsymbol{ue}}$	$'a\ GSU \to ('a\ GSU \to 'b) \to 'a\ GSU \to 'b$
$\boldsymbol{CscRank}$	$'a\ GSU \to 'a\ GSU$
$\boldsymbol{CscSeq}$	$'a\ GSU \to BOOL$
$\$_{\boldsymbol{c}}$	$('a\ GSU, 'a\ GSU)\ BR$
$\boldsymbol{MkCcon}$	$'a\ GSU \to 'a\ GSU$
$\boldsymbol{S_c}$	$'a\ GSU$
$\boldsymbol{K_c}$	$'a\ GSU$
$\$\equiv_{\boldsymbol{c}}$	$'a\ GSU$
$\boldsymbol{I_c}$	$'a\ GSU$
$\boldsymbol{T_c}$	$'a\ GSU$
$\boldsymbol{F_c}$	$'a\ GSU$
$\boldsymbol{If_c}$	$('a\ GSU, 'a\ GSU \to 'a\ GSU)\ BR$
$\mathbf{0_c}$	$'a\ GSU$
$\boldsymbol{Suc_c}$	$'a\ GSU$
$\boldsymbol{\Phi_c}$	$'a\ GSU \to 'a\ GSU$
$\boldsymbol{\Omega_c}$	$'a\ GSU$
$\boldsymbol{\Psi_c}$	$'a\ GSU$
$\boldsymbol{Kred}$	$'a\ RED$
$\boldsymbol{Sred}$	$'a\ RED$
$\boldsymbol{\Omega_{red}}$	$'a\ RED$
$\boldsymbol{\Psi_{red}}$	$'a\ RED$
$\boldsymbol{DComRed}$	$'a\ RED$
$\boldsymbol{DiComRed}$	$'a\ RED$
$\boldsymbol{DbComRed}$	$'a\ RED$
$\boldsymbol{Np}$	$'a\ RED \to 'a\ RED$
$\boldsymbol{Ineq0}$	$'a\ RED$
$\boldsymbol{EqSeq}$	$'a\ RED \to 'a\ RED$
$\boldsymbol{IneqStep}$	$'a\ RED \to 'a\ RED$
$\boldsymbol{EqStep}$	$'a\ RED \to 'a\ RED$
$\equiv_{\boldsymbol{red}}$	$'a\ RED \to 'a\ RED$
$\boldsymbol{RedClosed1}$	$'a\ RED \to BOOL$
$\boldsymbol{RedClosed2}$	$'a\ RED \to BOOL$
$\boldsymbol{RedClosure}$	$'a\ RED \to 'a\ RED$
$\boldsymbol{ILamFunct}$	$'a\ RED \to 'a\ RED$
$\boldsymbol{ILamFunctS}$	$'a\ GSU \leftrightarrow 'a\ GSU \to 'a\ GSU \leftrightarrow 'a\ GSU$
$\boldsymbol{PIComEq}$	$'a\ GSU\ \mathbb{P} \times 'a\ RED$
$\boldsymbol{PureInfComb}$	$'a\ GSU\ \mathbb{P} \to BOOL$
$\boldsymbol{ChurchRosser}$	$('a, BOOL)\ BR \to BOOL$
$\boldsymbol{SubTerm}$	$'a\ GSU\ LIST \to 'a\ GSU \to 'a\ GSU$
$\boldsymbol{is\_SubTerm}$	$'a\ GSU\ LIST \to 'a\ GSU \to BOOL$
$\boldsymbol{is\_redex\_set}$	$'a\ RED \to 'a\ GSU \to 'a\ GSU\ LIST\ \mathbb{P} \to BOOL$
$\boldsymbol{cls\_restrict}$	$'a\ GSU\ LIST\ \mathbb{P} \to 'a\ GSU \to 'a\ GSU\ LIST\ \mathbb{P}$
$\boldsymbol{reduce}$	$'a\ RED \to 'a\ GSU \to 'a\ GSU$
$\boldsymbol{depth\_map\_aux}$	$('a\ GSU \to 'a\ GSU) \to 'a\ GSU \to 'a\ GSU\ LIST\ \mathbb{P} \to 'a\ GSU$
$\boldsymbol{depth\_map}$	$('a\ GSU \to 'a\ GSU) \to 'a\ GSU\ LIST\ \mathbb{P} \to 'a\ GSU \to 'a\ GSU$
$\boldsymbol{par\_reduce}$	$'a\ RED \to 'a\ GSU\ LIST\ \mathbb{P} \to 'a\ GSU \to 'a\ GSU$

disj_par_reduce	
	$'a\ RED \to 'a\ RED$
subs	$('a\ GSU, 'a\ GSU \to 'a\ GSU)\ BR$
inert	$'a\ RED \to 'a\ GSU \to BOOL$
comm_subs	$'a\ RED \to BOOL$
is_cmap	$'a\ GSU \to BOOL$
K_cmap	$'a\ GSU \to 'a\ GSU$
K_rmap	$('a\ GSU, 'a\ GSU)\ BR$
rep_LT	$'a\ LT \to 'a\ GSU\ \mathbb{P}$
abs_LT	$'a\ GSU\ \mathbb{P} \to 'a\ LT$
$\$\models_{\boldsymbol{l}}$	$'a\ LT\ LIST \to 'a\ LT \to BOOL$
Lift2l	$'a\ GSU \to 'a\ LT$
$\boldsymbol{S_l}$	$'a\ LT$
$\boldsymbol{K_l}$	$'a\ LT$
$\$\equiv_{\boldsymbol{l}}$	$'a\ LT$
$\$_{\boldsymbol{l}}$	$('a\ LT, 'a\ LT)\ BR$

Aliases

$\models$	$\$\models_l : 'a\ LT\ LIST \to 'a\ LT \to BOOL$
S	$S_l : 'a\ LT$
K	$K_l : 'a\ LT$
.	$\$_l : ('a\ LT, 'a\ LT)\ BR$

Types

$'1$ ***LT***

Type Abbreviations

$'a$ ***RED***	$'a\ RED$
$'a$ ***CT***	$'a\ CT$
$'a$ ***CC***	$'a\ CC$

Fixity

Right Infix 0:	$\models$ $\models_{\boldsymbol{l}}$
Right Infix 250:	
	. $_{\boldsymbol{l}}$
Right Infix 310:	
	$\lhd_{\boldsymbol{ue}}$
Right Infix 350:	
	$_{\boldsymbol{c}}$

Definitions

$$\vdash \forall\ s \bullet\ CrepClosed\ s \Leftrightarrow (\forall\ c\ i \bullet\ Fun_u\ i \wedge Ordinal_u\ (Dom_u\ i) \wedge X_u\ (Ran_u\ i) \subseteq s \Rightarrow c \mapsto_u i \in s)$$ **CrepClosed**

$$\vdash Csyntax = \bigcap \{x | CrepClosed\ x\}$$ **Csyntax**

$$\vdash \forall\ \alpha\ \gamma \bullet\ CscPrec\ \alpha\ \gamma \Leftrightarrow (\exists\ c\ i \bullet\ \alpha \in_u Ran_u\ i \wedge \gamma = c \mapsto_u i)$$ **CscPrec**

$$\vdash \forall\ s\ f \bullet\ s \lhd_{ue} f = (\lambda\ t \bullet\ if\ t \in_u{}^+\ s\ then\ f\ t\ else\ \epsilon\ x \bullet\ T)$$ $\lhd_{ue}$

$$\vdash \forall\ c\ i \bullet\ CscRank\ (c \mapsto_u i) = \bigcup_u (Imagep_u\ Suc_u\ (Imagep_u\ (i \lhd_{ue} CscRank)\ (Ran_u\ i)))$$ **CscRank**

$$\vdash \forall\ s \bullet\ CscSeq\ s \Leftrightarrow Seq_u\ s \wedge (\forall\ t \bullet\ t \in_u Ran_u\ s \Rightarrow t \in Csyntax)$$ **CscSeq**

$$\vdash \forall\ f\ a \bullet\ f\ {}_c\ a = (let\ fc = Fst_u\ f\ and\ fa = Snd_u\ f\ in\ fc \mapsto_u fa\ @_u\ Unit_u\ (\varnothing_u \mapsto_u a))$$ $_c$

$$\vdash \forall\ n \bullet\ MkCcon\ n = n \mapsto_u \varnothing_u$$ **MkCcon**

$$\vdash S_c = MkCcon\ (Nat_u\ 0)$$ $\boldsymbol{S_c}$

$$\vdash K_c = MkCcon\ (Nat_u\ 1)$$ $\boldsymbol{K_c}$

$$\vdash \$\equiv_c = MkCcon\ (Nat_u\ 2)$$ $\equiv_c$

$$\vdash I_c = (S_c\ {}_c\ K_c)\ {}_c\ K_c$$ $\boldsymbol{I_c}$

$$\vdash T_c = K_c$$ $\boldsymbol{T_c}$

$$\vdash F_c = K_c\ {}_c\ I_c$$ $\boldsymbol{F_c}$

$$\vdash \forall\ x\ y\ z \bullet\ If_c\ x\ y\ z = (x\ {}_c\ y)\ {}_c\ z$$ $\boldsymbol{If_c}$

$$\vdash 0_c = I_c$$ $\boldsymbol{0_c}$

$$\vdash Suc_c = S_c\ {}_c\ S_c\ {}_c\ (K_c\ {}_c\ S_c)\ {}_c\ K_c$$ $\boldsymbol{Suc_c}$

$$\vdash \forall\ s \bullet\ \Phi_c\ s = Nat_u\ 3 \mapsto_u s$$ $\boldsymbol{\Phi_c}$

$$\vdash \Omega_c = MkCcon\ (Nat_u\ 4)$$ $\boldsymbol{\Omega_c}$

$$\vdash \Psi_c = MkCcon\ (Nat_u\ 5)$$ $\boldsymbol{\Psi_c}$

$$\vdash \forall\ s\ t \bullet\ Kred\ s\ t \Leftrightarrow (\exists\ x\ y \bullet\ s = (K_c\ {}_c\ x)\ {}_c\ y \wedge t = x)$$ **Kred**

$$\vdash \forall\ s\ t \bullet\ Sred\ s\ t \Leftrightarrow (\exists\ x\ y\ z \bullet\ s = ((S_c\ {}_c\ x)\ {}_c\ y)\ {}_c\ z \wedge t = (x\ {}_c\ z)\ {}_c\ y\ {}_c\ z)$$ **Sred**

$$\vdash \forall\ s\ t \bullet\ \Omega_{red}\ s\ t \Leftrightarrow (\exists\ k\ l\ m\ n \bullet\ Dom_u\ k = Dom_u\ m$$ $\boldsymbol{\Omega_{red}}$

$\wedge\ Ordinal_u\ (Dom_u\ k)$
$\wedge\ Ran_u\ k = Unit_u\ F_c$
$\wedge\ s = \Omega_{c\ c}\ \Phi_c\ (k\ @_u\ l)\ _c\ \Phi_c\ (m\ @_u\ n)$
$\wedge\ t = \Omega_{c\ c}\ \Phi_c\ l\ _c\ \Phi_c\ n)$
$\vee\ (\exists\ k\ m$
$\bullet\ \varnothing_u \mapsto_u T_c \in_u k$
$\wedge\ \varnothing_u \mapsto_u t \in_u m$
$\wedge\ s = \Omega_{c\ c}\ \Phi_c\ k\ _c\ \Phi_c\ m)$

Ψ_{red} $\vdash \forall\ s\ t$
$\bullet\ \Psi_{red}\ s\ t$
$\Leftrightarrow (\exists\ f\ l\ m$
$\bullet\ Dom_u\ l = Dom_u\ m$
$\wedge\ Ordinal_u\ (Dom_u\ l)$
$\wedge\ m = (\lambda_u\ x\bullet\ f\ _c\ x)\ (Dom_u\ l)$
$\wedge\ s = (\Psi_{c\ c}\ f)\ _c\ \Phi_c\ l$
$\wedge\ t = \Phi_c\ m)$

DComRed $\vdash \forall\ s\ t\bullet\ DComRed\ s\ t \Leftrightarrow Kred\ s\ t \vee Sred\ s\ t$

DiComRed $\vdash \forall\ s\ t\bullet\ DiComRed\ s\ t \Leftrightarrow \Omega_{red}\ s\ t \vee \Psi_{red}\ s\ t$

DbComRed $\vdash \forall\ s\ t$
$\bullet\ DbComRed\ s\ t$
$\Leftrightarrow Kred\ s\ t \vee Sred\ s\ t \vee \Omega_{red}\ s\ t \vee \Psi_{red}\ s\ t$

Np $\vdash \forall\ r\bullet\ Np\ r = (\lambda\ x\ y\bullet\ r\ ((\$\equiv_{c\ c}\ x)\ _c\ y)\ F_c)$

Ineq0 $\vdash Ineq0$
$= (\lambda\ x\ y$
$\bullet\ \exists\ v\ w\bullet\ x = MkCcon\ v \wedge y = MkCcon\ w \wedge \neg\ v = w)$

EqSeq $\vdash \forall\ eq$
$\bullet\ EqSeq\ eq$
$= (\lambda\ v\ w$
$\bullet\ Fun_u\ v$
$\wedge\ Fun_u\ w$
$\wedge\ Dom_u\ v = Dom_u\ w$
$\wedge\ Ordinal_u\ (Dom_u\ v)$
$\wedge\ (\forall\ x$
$\bullet\ x \in_u Dom_u\ v \Rightarrow eq\ (v\ _u\ x)\ (w\ _u\ x)))$

IneqStep $\vdash \forall\ eq$
$\bullet\ IneqStep\ eq$
$= (\lambda\ x\ y$
$\bullet\ Ineq0\ x\ y$
$\vee\ Np\ eq\ x\ y$
$\vee\ (\exists\ v\ w$
$\bullet\ EqSeq\ eq\ v\ w \wedge Np\ eq\ (x\ _c\ v)\ (y\ _c\ w)))$

EqStep $\vdash \forall\ eq$
$\bullet\ EqStep\ eq$
$= (\lambda\ v\ w$
$\bullet\ \exists\ c\ i1\ i2$
$\bullet\ c \mapsto_u i1 \in Csyntax$
$\wedge\ c \mapsto_u i2 \in Csyntax$
$\wedge\ Dom_u\ i1 = Dom_u\ i2$
$\wedge\ (\forall\ x$
$\bullet\ x \in_u Dom_u\ i1 \Rightarrow eq\ (i1\ _c\ x)\ (i2\ _c\ x)))$

$\equiv_{red}$ $\vdash \forall\ eq$

$\bullet \equiv_{red} eq$
$= (\lambda\ x\ y$
$\bullet \exists\ l\ m$
$\bullet\ x = (\$\equiv_{c\ c}\ l)_{\ c}\ m$
$\wedge\ (EqStep\ eq\ x\ y \wedge y = T_c$
$\vee\ IneqStep\ eq\ x\ y \wedge y = F_c))$

RedClosed1 $\vdash \forall\ r$
$\bullet\ RedClosed1\ r$
$\Leftrightarrow (\forall\ c\ i1\ i2$
$\bullet\ c \mapsto_u i1 \in Csyntax$
$\wedge\ c \mapsto_u i2 \in Csyntax$
$\wedge\ Dom_u\ i1 = Dom_u\ i2$
$\wedge\ (\forall\ x$
$\bullet\ x \in_u Dom_u\ i1 \Rightarrow r\ (i1\ _c\ x)\ (i2\ _c\ x))$
$\Rightarrow r\ (c \mapsto_u i1)\ (c \mapsto_u i2))$

RedClosed2 $\vdash \forall\ r$
$\bullet\ RedClosed2\ r$
$\Leftrightarrow (\forall\ c\ d\ e\ f\ g\ h$
$\bullet\ c \mapsto_u e \in Csyntax$
$\wedge\ d \mapsto_u f \in Csyntax$
$\wedge\ r\ (c \mapsto_u e)\ (d \mapsto_u f)$
$\wedge\ EqSeq\ r\ g\ h$
$\Rightarrow r\ (c \mapsto_u e\ @_u\ g)\ (c \mapsto_u f\ @_u\ h))$

RedClosure $\vdash \forall\ r$
$\bullet\ RedClosure\ r$
$= Snd$
$(EquivClosure$
$(Csyntax,$
$(\lambda\ x\ y$
$\bullet \forall\ r1$
$\bullet\ RedClosed1\ r1$
$\wedge\ RedClosed2\ r1$
$\wedge\ (Csyntax,\ r)$
$\subseteq (Csyntax,\ r1)$
$\Rightarrow r1\ x\ y)))$

ILamFunct $\vdash \forall\ r \bullet\ ILamFunct\ r = RedClosure\ (\equiv_{red} r)$

ILamFunctS $\vdash \forall\ s$
$\bullet\ ILamFunctS\ s$
$= \{x$
$|ILamFunct\ (\lambda\ x\ y \bullet\ (x,\ y) \in s)\ (Fst\ x)\ (Snd\ x)\}$

PIComEq $\vdash PIComEq$
$= EquivClosure$
$(Csyntax,\ (\lambda\ x\ y \bullet\ (x,\ y) \in Lfp\ ILamFunctS))$

PureInfComb $\vdash \forall\ x \bullet\ PureInfComb\ x \Leftrightarrow x \in Csyntax\ /\ Snd\ PIComEq$

ChurchRosser $\vdash \forall\ r$
$\bullet\ ChurchRosser\ r$
$\Leftrightarrow (\forall\ w\ x\ y \bullet\ r\ w\ x \wedge r\ w\ y \Rightarrow (\exists\ z \bullet\ r\ x\ z \wedge r\ y\ z))$

SubTerm $\vdash \forall\ h\ t\ tl$
$\bullet\ SubTerm\ []\ t = t$
$\wedge\ SubTerm\ (Cons\ h\ tl)\ t$
$= SubTerm\ tl\ (Snd_u\ t\ _u\ h)$

is_SubTerm $\vdash \forall\ h\ t\ tl$
$\bullet\ (is\_SubTerm\ []\ t \Leftrightarrow T)$
$\wedge\ (is\_SubTerm\ (Cons\ h\ tl)\ t$
$\Leftrightarrow h \in_u Dom_u\ (Snd_u\ t)$
$\wedge\ is\_SubTerm\ tl\ (Snd_u\ t\ _u\ h))$

is_redex_set $\vdash \forall\ r\ t\ cls$
$\bullet\ is\_redex\_set\ r\ t\ cls$
$\Leftrightarrow (\forall\ cs$
$\bullet\ cs \in cls$
$\Rightarrow is\_SubTerm\ cs\ t$
$\wedge\ (\exists\ y\bullet\ r\ (SubTerm\ cs\ t)\ y))$

cls_restrict $\vdash \forall\ cls\ t\bullet\ cls\_restrict\ cls\ t = \{v|Cons\ t\ v \in cls\}$

reduce $\vdash \forall\ r\ t$
$\bullet\ reduce\ r\ t$
$= (\epsilon\ nt\bullet\ r\ t\ nt \vee \neg\ (\exists\ nt2\bullet\ r\ t\ nt2) \wedge nt = t)$

depth_map_aux
$\vdash \forall\ f\ c\ ts$
$\bullet\ depth\_map\_aux\ f\ (c \mapsto_u ts)$
$= (\lambda\ cls$
$\bullet\ (let\ newts$
$= Imagep_u$
$(\lambda\ op$
$\bullet\ Fst_u\ op$
$\mapsto_u (ts \lhd_{ue} depth\_map\_aux\ f)$
$(Snd_u\ op)$
$(cls\_restrict\ cls\ (Fst_u\ op)))$
ts
$in\ if\ [] \in cls$
$then\ f\ (c \mapsto_u newts)$
$else\ c \mapsto_u newts))$

depth_map $\vdash \forall\ f\ cls\ t\bullet\ depth\_map\ f\ cls\ t = depth\_map\_aux\ f\ t\ cls$

par_reduce $\vdash \forall\ r\ c\ ts\ cls$
$\bullet\ par\_reduce\ r\ cls\ (c \mapsto_u ts)$
$= depth\_map\ (reduce\ r)\ cls\ (c \mapsto_u ts)$

disj_par_reduce
$\vdash \forall\ r\ c\ ts\ nt$
$\bullet\ disj\_par\_reduce\ r\ (c \mapsto_u ts)\ nt$
$\Leftrightarrow r\ (c \mapsto_u ts)\ nt$
$\vee\ (\exists\ ts'$
$\bullet\ (\forall\ i\ v$
$\bullet\ i \mapsto_u v \in_u ts$
$\Leftrightarrow (\exists\ v'$
$\bullet\ i \mapsto_u v' \in_u ts'$
$\wedge\ ((ts \lhd_{ue} disj\_par\_reduce\ r)$
v
v'
$\vee\ v' = v)))$
$\wedge\ (\exists\ i\ v\ v'$
$\bullet\ i \mapsto_u v \in_u ts$
$\wedge\ i \mapsto_u v' \in_u ts'$
$\wedge\ (ts \lhd_{ue} disj\_par\_reduce\ r)\ v\ v')$

$$\wedge\ nt = c \mapsto_u ts')$$

subs $\vdash \forall\ cn\ c\ ts\ s$

$\bullet\ subs\ cn\ s\ (c \mapsto_u ts)$

$= (if\ c = cn$

$then$

$Fst_u\ s$

$\mapsto_u Snd_u\ s$

$@_u\ RanMap_u\ (ts \lhd_{ue} subs\ cn\ s)\ ts$

$else\ c \mapsto_u RanMap_u\ (ts \lhd_{ue} subs\ cn\ s)\ ts)$

inert $\vdash \forall\ r\ c$

$\bullet\ inert\ r\ c$

$\Leftrightarrow \neg\ (\exists\ ts\ c'\ ts'$

$\bullet\ r\ (c \mapsto_u ts)\ (c' \mapsto_u ts') \wedge \neg\ c = c')$

comm_subs $\vdash \forall\ r$

$\bullet\ comm\_subs\ r$

$\Leftrightarrow (\forall\ cn$

$\bullet\ inert\ r\ cn$

$\Rightarrow (\forall\ s\ t\ t2$

$\bullet\ s \in Csyntax \wedge t \in Csyntax$

$\Rightarrow r\ t\ t2$

$\Rightarrow r\ (subs\ cn\ s\ t)\ (subs\ cn\ s\ t2)))$

is_cmap $\vdash \forall\ m$

$\bullet\ is\_cmap\ m$

$\Leftrightarrow Fun_u\ m$

$\wedge (\forall\ x$

$\bullet\ x \in_u Dom_u\ m$

$\Rightarrow Seq_u\ x$

$\wedge (\forall\ y\bullet\ y \in_u Ran_u\ x \Rightarrow Ordinal_u\ y))$

$\wedge (\forall\ x\ y$

$\bullet\ y \in_u Dom_u\ m \wedge Dom_u\ x \subseteq_u Dom_u\ y$

$\Rightarrow x \in_u Dom_u\ m)$

$\wedge (\forall\ x\ y$

$\bullet\ y \in_u Dom_u\ m$

$\wedge\ Dom_u\ x = Dom_u\ y$

$\wedge (\forall\ z$

$\bullet\ z \in_u Dom_u\ x \Rightarrow x\ _u\ z \subseteq_u y\ _u\ z)$

$\Rightarrow x \in_u Dom_u\ m)$

K_cmap $\vdash \forall\ m$

$\bullet\ K\_cmap\ m$

$= (\lambda_u\ cs$

$\bullet\ (let\ h = Head_u\ cs$

$and\ x = m\ _u\ Unit_u\ (\varnothing_u \mapsto_u \varnothing_u)$

$in\ m$

$_u\ (if\ Nat_u\ 2 \leqslant_u h$

$then$

$let\ h2$

$= (if\ natural\_number_u\ h$

$then\ h\ --_u\ Nat_u\ 1$

$else\ h)$

$in\ SeqCons_u\ h2\ (Tail_u\ cs)$

$else\ if\ h = Nat_u\ 1$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\quad \mathit{then}\ \mathit{Tail}_u\ \mathit{cs} \\
\qquad\qquad\qquad\qquad\quad \mathit{else}\ \mathit{cs}))) \\
\qquad\qquad\qquad\quad (\mathit{Gx}_u\ m)
\end{array}
$$

K_rmap $\vdash \forall\ m$

$$
\begin{array}{l}
\bullet\ \mathit{K\_rmap}\ m \\
\qquad = (\lambda\ \mathit{cs} \\
\qquad \bullet\ (\mathit{let}\ h = \mathit{Head}_u\ \mathit{cs} \\
\qquad\qquad \mathit{and}\ x = m\ {}_u\ \mathit{Unit}_u\ (\varnothing_u \mapsto_u \varnothing_u) \\
\qquad\qquad \mathit{in}\ \mathit{if}\ \mathit{Nat}_u\ 2 \leqslant_u h \\
\qquad\qquad \mathit{then} \\
\qquad\qquad\quad \mathit{let}\ h2 \\
\qquad\qquad\qquad\quad = (\mathit{if}\ \mathit{natural\_number}_u\ h \\
\qquad\qquad\qquad\quad\ \mathit{then}\ h\ {--}_u\ \mathit{Nat}_u\ 1 \\
\qquad\qquad\qquad\quad\ \mathit{else}\ h)\ \mathit{in}\ \mathit{SeqCons}_u\ h2\ (\mathit{Tail}_u\ \mathit{cs}) \\
\qquad\qquad \mathit{else}\ \mathit{if}\ h = \mathit{Nat}_u\ 1 \\
\qquad\qquad \mathit{then}\ \mathit{Tail}_u\ \mathit{cs} \\
\qquad\qquad \mathit{else}\ \mathit{cs}))
\end{array}
$$

LT $\vdash \exists\ f \bullet\ \mathit{TypeDefn}\ \mathit{PureInfComb}\ f$

abs_LT

rep_LT $\vdash (\forall\ a \bullet\ \mathit{abs\_LT}\ (\mathit{rep\_LT}\ a) = a)$

$$
\begin{array}{l}
\qquad \wedge\ (\forall\ r \bullet\ \mathit{PureInfComb}\ r \Leftrightarrow \mathit{rep\_LT}\ (\mathit{abs\_LT}\ r) = r) \\
\qquad \wedge\ \mathit{OneOne}\ \mathit{rep\_LT}
\end{array}
$$

$\models_{\boldsymbol{l}}$ $\vdash \forall\ \mathit{pl}\ p$

$$
\begin{array}{l}
\bullet\ (\mathit{pl} \models p) \\
\qquad \Leftrightarrow \forall_L\ (\mathit{Map}\ (\lambda\ p \bullet\ T_c \in \mathit{rep\_LT}\ p)\ \mathit{pl}) \\
\qquad\quad \Rightarrow T_c \in \mathit{rep\_LT}\ p
\end{array}
$$

Lift2l $\vdash \forall\ c \bullet\ \mathit{Lift2l}\ c = \mathit{abs\_LT}\ (\mathit{EquivClass}\ \mathit{PIComEq}\ c)$

$\boldsymbol{S_l}$ $\vdash S = \mathit{Lift2l}\ S_c$

$\boldsymbol{K_l}$ $\vdash K = \mathit{Lift2l}\ K_c$

$\equiv_{\boldsymbol{l}}$ $\vdash \$\equiv_l = \mathit{Lift2l}\ \$\equiv_c$

$\boldsymbol{\imath}$ $\vdash \forall\ f\ a$

$$
\begin{array}{l}
\bullet\ f\ .\ a \\
\qquad = \mathit{Lift2l} \\
\qquad\quad ((\epsilon\ x \bullet\ x \in \mathit{rep\_LT}\ f)\ {}_c\ (\epsilon\ x \bullet\ x \in \mathit{rep\_LT}\ a))
\end{array}
$$

Theorems

crepclosed_csyntax_thm

$\vdash \forall\ c\ i$

$$
\begin{array}{l}
\bullet\ \mathit{Fun}_u\ i \\
\qquad \wedge\ \mathit{Ordinal}_u\ (\mathit{Dom}_u\ i) \\
\qquad \wedge\ (\forall\ x \bullet\ x \in X_u\ (\mathit{Ran}_u\ i) \Rightarrow x \in \mathit{Csyntax}) \\
\qquad \Rightarrow c \mapsto_u i \in \mathit{Csyntax}
\end{array}
$$

crepclosed_csyntax_thm2

$\vdash \forall\ c\ i$

$$
\begin{array}{l}
\bullet\ \mathit{Fun}_u\ i \\
\qquad \wedge\ \mathit{Ordinal}_u\ (\mathit{Dom}_u\ i) \\
\qquad \wedge\ (\forall\ x \bullet\ x \in_u \mathit{Ran}_u\ i \Rightarrow x \in \mathit{Csyntax}) \\
\qquad \Rightarrow c \mapsto_u i \in \mathit{Csyntax}
\end{array}
$$

crepclosed_csyntax_thm3

$\vdash \forall\ c\ i$

$\bullet\ \mathit{Seq}_u\ i \wedge (\forall\ x \bullet\ x \in_u \mathit{Ran}_u\ i \Rightarrow x \in \mathit{Csyntax})$

$\Rightarrow\ c \mapsto_u i \in Csyntax$

crepclosed_csyntax_lemma1

$\vdash\ \forall\ s\bullet\ CrepClosed\ s \Rightarrow Csyntax \subseteq s$

crepclosed_csyntax_lemma2

$\vdash\ \forall\ p\bullet\ CrepClosed\ \{x|p\ x\} \Rightarrow (\forall\ x\bullet\ x \in Csyntax \Rightarrow p\ x)$

well_founded_CscPrec_thm

$\vdash\ well\_founded\ CscPrec$

well_founded_tcCscPrec_thm

$\vdash\ well\_founded\ (tc\ CscPrec)$

csc_fc_thm

$\vdash\ \forall\ x$
$\bullet\ x \in Csyntax$
$\Rightarrow (\exists\ c\ i$
$\bullet\ Fun_u\ i$
$\wedge\ Ordinal_u\ (Dom_u\ i)$
$\wedge\ (\forall\ y\bullet\ y \in_u Ran_u\ i \Rightarrow y \in Csyntax)$
$\wedge\ x = c \mapsto_u i)$

¬∅u_∈_csyntax_lemma

$\vdash\ \neg\ \varnothing_u \in Csyntax$

¬∅u_∈_csyntax_lemma2

$\vdash\ \forall\ x\bullet\ x \in Csyntax \Rightarrow \neg\ x = \varnothing_u$

¬∅u_∈_csyntax_lemma3

$\vdash\ \forall\ V\ x\bullet\ x \in V\ \wedge\ V \subseteq Csyntax \Rightarrow \neg\ x = \varnothing_u$

csc_fc_thm2

$\vdash\ \forall\ c\ i$
$\bullet\ c \mapsto_u i \in Csyntax$
$\Rightarrow Fun_u\ i$
$\wedge\ Ordinal_u\ (Dom_u\ i)$
$\wedge\ (\forall\ x\bullet\ x \in_u Ran_u\ i \Rightarrow x \in Csyntax)$

cscprec_fc_thm

$\vdash\ \forall\ c\ i\ x\bullet\ x \in_u Ran_u\ i \Rightarrow CscPrec\ x\ (c \mapsto_u i)$

csyn_induction_thm

$\vdash\ (\forall\ c\ i$
$\bullet\ Fun_u\ i$
$\wedge\ Ordinal_u\ (Dom_u\ i)$
$\wedge\ (\forall\ x\bullet\ x \in_u Ran_u\ i \Rightarrow x \in Csyntax\ \wedge\ p\ x)$
$\Rightarrow p\ (c \mapsto_u i))$
$\Rightarrow (\forall\ x\bullet\ x \in Csyntax \Rightarrow p\ x)$

csyn_induction_thm2

$\vdash\ (\forall\ c\ i$
$\bullet\ Seq_u\ i$
$\wedge\ (\forall\ x\bullet\ x \in_u Ran_u\ i \Rightarrow x \in Csyntax\ \wedge\ p\ x)$
$\Rightarrow p\ (c \mapsto_u i))$
$\Rightarrow (\forall\ x\bullet\ x \in Csyntax \Rightarrow p\ x)$

csyntax_pair_thm

$\vdash\ \forall\ t\bullet\ t \in Csyntax \Rightarrow (\exists\ c\ ts\bullet\ t = c \mapsto_u ts)$

csc_recursion_lemma

$\vdash\ \forall\ af\bullet\ \exists\ f\bullet\ \forall\ c\ i\bullet\ f\ (c \mapsto_u i) = af\ (i \lhd_{ue} f)\ c\ i$

MkCcon_∈_Csyntax_thm

$\vdash\ \forall\ x\bullet\ MkCcon\ x \in Csyntax$

Equiv_PIComEq_thm

$\vdash\ Equiv\ PIComEq$

CSyntax_PIComEq_thm

$\vdash \forall\ x \bullet\ x \in Csyntax \Rightarrow x \in EquivClass\ PIComEq\ x$

CrTc_thm $\vdash \forall\ r \bullet\ ChurchRosser\ r \Rightarrow ChurchRosser\ (tc\ r)$

SubTerm_in_Csyntax_lemma

$\vdash \forall\ t$
$\quad \bullet\ t \in Csyntax$
$\quad\quad \Rightarrow (\forall\ cs$
$\quad\quad\quad \bullet\ is\_SubTerm\ cs\ t \Rightarrow SubTerm\ cs\ t \in Csyntax)$

r_disj_par_thm

$\vdash \forall\ r\ x\ y \bullet\ x \in Csyntax \wedge r\ x\ y \Rightarrow disj\_par\_reduce\ r\ x\ y$

inert_disj_par_thm

$\vdash \forall\ r\ cn \bullet\ inert\ (disj\_par\_reduce\ r)\ cn \Leftrightarrow inert\ r\ cn$

LT_fc_thm1 $\vdash \forall\ r \bullet\ PureInfComb\ r \Rightarrow rep\_LT\ (abs\_LT\ r) = r$

LT_fc_thm2 $\vdash \forall\ x\ y$
$\quad \bullet\ PureInfComb\ x \wedge PureInfComb\ y$
$\quad\quad \Rightarrow (abs\_LT\ x = abs\_LT\ y \Leftrightarrow x = y)$

Lift2l_thm $\vdash \forall\ x \bullet\ x \in Csyntax \Rightarrow x \in rep\_LT\ (Lift2l\ x)$

A.2 The Theory ilamb

Parents

equiv icomb

Constants

$\boldsymbol{I_l}$	$'a\ LT$
$\boldsymbol{T_l}$	$'a\ LT$
$\boldsymbol{F_l}$	$'a\ LT$
$\$\equiv_{\boldsymbol{n}}$	$('a\ LT, 'a\ LT)\ BR$

Aliases

$\boldsymbol{I}$	$I_l : 'a\ LT$
$\boldsymbol{T}$	$T_l : 'a\ LT$
$\boldsymbol{F}$	$F_l : 'a\ LT$
$\equiv$	$\$\equiv_n : ('a\ LT, 'a\ LT)\ BR$

Fixity

Right Infix 210:

$\equiv_{\boldsymbol{n}}$

Definitions

$\boldsymbol{I_l}$	$\vdash\ I = (S\ .\ K)\ .\ K$
$\boldsymbol{T_l}$	$\vdash\ T = K$
$\boldsymbol{F_l}$	$\vdash\ F = K\ .\ I$
$\equiv_{\boldsymbol{n}}$	$\vdash\ \forall\ x\ y\bullet\ x \equiv y = \$\equiv_l\ .\ x\ .\ y$

Theorems

$\boldsymbol{T_l\_thm}$	$\vdash\ [] \models T$

Bibliography

[1] H.P. Barendregt. *The Lambda-Calculus - Its Syntax and Semantics*. North Holland, 2 edition. 1984.

[2] Alonzo Church. *The calculi of Lambda-conversion*. Princeton University Press. 1941.

[3] Roger Bishop Jones. PolySets. 2007. http://www.rbjones.com/rbjpub/www/papers/p011.pdf.

[4] Roger Bishop Jones. A Higher Order Theory of Well-Founded Sets (with Urelements). *RBJones.com*, 2010. http://www.rbjones.com/rbjpub/pp/doc/t042.pdf.

[5] Roger Bishop Jones. Infinitarily Definable Non-Well-Founded Sets. *RBJones.com*, 2010. http://www.rbjones.com/rbjpub/pp/doc/t024.pdf.

[6] Roger Bishop Jones. Infinitarily Definable Sets. *RBJones.com*, 2010. http://www.rbjones.com/rbjpub/pp/doc/t026.pdf.

[7] Roger Bishop Jones. Infinitary First Order Set Theory. *RBJones.com*, 2010. http://www.rbjones.com/rbjpub/pp/doc/t027.pdf.

[8] Roger Bishop Jones. Introduction to Work in Progress. *RBJones.com*, 2010. http://www.rbjones.com/rbjpub/pp/doc/t000.pdf.

[9] Roger Bishop Jones. PolySet Theory. *RBJones.com*, 2010. http://www.rbjones.com/rbjpub/pp/doc/t020.pdf.

[10] Roger Bishop Jones. Set Theory as Consistent Infinitary Comprehension. *RBJones.com*, 2010. http://www.rbjones.com/rbjpub/pp/doc/t021.pdf.

Index