Unrestrained beta reduction

October 9, 2012

Abstract A major argument for syntactic reconstruction is based on the well-known fact that semantic reconstruction by β-reduction is possible only if the term to be substituted for a variable does not contain any variable that would become bound as a result of substitution: e.g. the expression $\lambda x_2 \forall x_1 P(x_1, x_2)(x_1)$ cannot be β-reduced to $\forall x_1 P(x_1, \underline{x_1})$, since the underlined occurrence of x_1 would become bound. However, syntactic reconstruction is not without its problems, simply because the surface form and the reconstructed form may still differ with respect to other syntactic, semantic, and information theoretic properties.

In this paper we propose to overcome the limitation imposed by β-reduction on semantic reconstruction, by introducing a translation **T** of expressions of a predicate logic L_0 with λ-abstraction into expressions of a higher-order language L_1, with the desirable property that e.g. $\mathbf{T}(\lambda x_2 \forall x_1 P(x_1, x_2)(x_1)) = \mathbf{T}(\forall x_1 P(x_1, x_1))$. By way of indirect interpretation, **T** provides for a novel (non-standard) semantics for lambda abstraction and quantification that allows for unrestricted β-reduction. We will give a formal proof showing that unrestricted β-reduction and our non-standard semantic interpretation of L_0 are logically equivalent.

Keywords beta-reduction, lambda-conversion, semantic reconstruction, syntactic reconstruction, lambda calculus

1 Introduction

In semantics of natural language, lambda abstraction is omnipresent and fundamental; ever since the work of Richard Montague it resides at the core of compositionality. For example, quantifier raising or quantifying-in crucially relies on lambda abstraction; quantifier raising in turn is essential for the the binding of pronouns, and binding itself presupposes a way of identifying variables that is expressed by coindexation. It is this connection between lambda abstraction and coindexation that is at issue in our paper; as will be illustrated in this introduction, coindexation poses a problem for the semantics of lambda abstraction in the context of so-called reconstruction.

According to received wisdom, the sentences in (1) get a different interpretation depending on whether or not the pronoun is coindexed with the quantifier:

(1) a. noone$_k$ doubts that he$_k$ is smart
b. noone$_k$ doubts that he$_j$ is smart

Given that quantifying expressions are interpreted as generalized quantifiers, the representations in (2) differ, depending on whether the variable is interpreted as bound or as free. The relevant binder cannot be the generalized quantifier itself, which only expresses a relation between sets, but it must be the lambda operator:

(2) a. noone$_k$ λx_k x_k doubts that x_k is smart
b. noone$_k$ λx_k x_k doubts that x_j is smart

Bound and free variables also play a crucial role in the interpretation of movement. Consider standard examples of topicalization, as in (3):

(3) a. That pizza$_i$, I won't eat t$_i$
b. Such examples$_i$, I thought you said that Tom believes the explanation needs t$_i$

Again, lambda abstraction provides for a straightforward semantic interpretation of (3) by converting the trace into a bound variable:

(4) a. That pizza λx_i I won't eat x_i
b. Such examples λx_i I thought you said that Tom believes the explanation needs x_i

However, this immediate connection between binding, coindexation and movement is undermined by topicalized sentences like (5):

(5) [His mother$_i$]$_j$ noone$_i$ would eat t$_j$[1]

The problem is that the pronoun in (5) has left the syntactic domain of its binder. Perhaps more natural examples in English are topicalized sentences. Consider the following scenario. A psychopath has a quite different view of the world. He knows that what he does is violent, but he considers it justified by the circumstances...

(6) That he is alone in his interpretation, no psychopath realises.

Likewise:

(7) a. Quite how socially privileged he$_i$ is, no student$_i$ realizes

[1] Unfortunately, the example seems to be rather marked in English, and even the reconstructed form seems to be marked, compare (i-a) with the more natural (i-b):

(i) a. ?noone$_i$ hates his$_i$ mother
b. noone$_i$ hates their$_i$/one$_i$'s mother

But in a language like German, both (ii-a) and (ii-b) are perfectly natural and unmarked:

(ii) a. Niemand$_i$ hasst seine$_i$ Mutter
b. Seine$_i$ Mutter hasst niemand$_i$

b. That they$_i$ are handicapped, few students$_i$ realize
c. That he$_i$ is handicapped, no autistic individual$_i$ realizes

Again, the obvious problem with these constructions is that the pronoun has been moved out of the scope of the binding expression. The intended meaning is of course still represented by coindexation, but it seems that this meaning cannot be derived by interpreting the surface structure. Rather, it has been assumed that for this purpose one has to syntactically reconstruct the moved material back into its deep structure position, where the pronoun is c-commanded by its binder.

The reason for this failure of surface compositionality is rooted in a restriction embodied in the lambda calculus. To illustrate the general case, assuming that a term T originates in X as shown in (8),

(8) $[_X$ ... T ...]

movement of T in front of X forms a lambda abstract, i.e. a function that can be applied to T, as shown in (9):

(9) T $\lambda y[_X ... y ...]$

Assuming that functional application is from right to left, this normally works fine with all sorts of Ts in that it still yields the desired truth conditions. Unfortunately, it does not work for the problematic cases discussed above, because here T itself contains a pronoun to be interpreted as a bound variable—but one that has been moved out of the scope of its binder:

(10) $[_T \ldots x \ldots] \lambda y[_X \ldots \lambda x[\ldots y \ldots] \ldots]$

According to the standard semantics of the lambda calculus, β-reduction of λy in (10) is blocked, due to the fact that x occurs free in (10), whereas after β-reduction, it would be bound, as shown in (11):

(11) $[_X \ldots \lambda x \ldots [_T \ldots x \ldots] \ldots]$

In other words, the lambda calculus does not allow a transformation, namely β-reduction, that converts (10) into (11), because the formulas are not semantically equivalent. This prohibition against β-reduction is motivated formally by demonstrating that an expression like

(12) $\lambda x \lambda y.P(y,x)(y)$

cannot be equivalent to $\lambda y.P(y,y)$ because this would "change the meaning" of the relation P.

Although the argument is perfectly sound, there is still something unsatisfactory about it, because in some sense, this change of meaning is precisely what pronouns can achieve, in particular if the pronoun is a reflexive pronoun applying to a verb like *hate*, resulting in the meaning of *hating oneself*. Moreover, our semantic intuitions tell us that topicalization does not affect truth conditions, so a topicalized sentence should

be truth functionally equivalent to its source (although they differ in information structure), contrary to what our translations would predict. In other words, our indexing conventions try to express something that seems to be inexpressible from the perspective of logic, namely that coindexation should still express binding in the reconstruction context under discussion.

The overall picture has been that there are two potential processes of reconstruction, namely syntactic movement at LF that moves T into its original base position (syntactic reconstruction), or β-reduction that applies in semantics as the result of the logical equivalence of (8) and (9) (semantic reconstruction). But as we have seen this equivalence does not always hold, hence β-reduction and semantic reconstruction are claimed not to work for natural language so that we must resort to syntactic reconstruction.

However, syntactic reconstruction is not without its problems, simply because the surface form and the reconstructed form may still differ with respect to other syntactic and semantic properties. In a framework like Minimalism, this is a contradiction because a potential difference cannot even be formulated in that theory (because surface form is not a level of representation). For example, it has been shown that syntactic reconstruction predicts to elicit Condition (C) effects that are not attested in the data, therefore syntactic reconstruction is not a solution for the bound variable problem (cf. e.g. Salzmann (2006)). Moreover, there are contexts that do not permit a reconstructed reading because some blocking material intervenes between the topicalized item and the trace. As shown by Heycock (2011), the nature of these interveners must be semantic rather than syntactic. We will not discuss such arguments here, in particular, because it is not clear how these intervention effects can be implemented in semantics. But the overall conclusion so far is that syntactic reconstruction is on the wrong track and that a semantic approach to reconstruction (and potential semantic restrictions) is called for.

This leaves us with the problem that surface compositional semantics lacks a mechanism that allows us to interpret movement in such a way that the indexing of pronouns still reflects the fact that some sort of binding can be going on. More generally, we wish to be able to apply β-reduction in a way that goes against the traditional rules of logic but in accord with linguistic indexing devices. In other words, the goal of this paper is to reconcile logic with natural language semantics by showing that it is possible to interpret ordinary predicate logic with lambda abstraction in such a way that β-reduction works without restriction. In consequence, we aim to interpret lambda expressions in such a way that the two formulas $\lambda x \lambda y.P(y,x)(y)$ and $\lambda y.P(y,y)$ receive equivalent interpretations such that the common device of co-indexation employed in linguistic examples yields the correct result, without syntactic reconstruction and even without syntactic binding (i.e. c-command) at surface structure.

Our strategy will be that of indirect interpretation (direct interpretation would also be possible, but less comprehensible). That is, we will provide for an interpretation of a lambda expression by specifying a translation function **T** that translates first order logic plus lambda expressions into a more complex higher order logic. This logic receives its traditional interpretation. We will then show that

(13) $\mathbf{T}(\lambda x \lambda y.P(y,x)(y)) = \mathbf{T}(\lambda y.P(y,y))$

is a special case of a general equivalence between β-reduced and unreduced formulas.

We proceed as follows. Section 2 introduces assignment functions into the formal language of a sorted and typed predicate logic L_1 with lambda abstraction, and we show that each formula of first order predicate logic can be translated into a type shifted formula of L_1. We demonstrate how this first step can account for most linguistic problems of variable binding by reconstruction. Section 3 attacks the key problem for lambda abstraction, namely the case illustrated in (13), and presents the solution in a piecemeal fashion. First we deal with iterated abstraction and application, then with quantification, and finally with an asymmetry between binding by quantifiers and binding by lambda abstraction. In Section 4 we formally prove that the system defined in Section 3 allows for unrestricted semantic reconstruction.

2 Indices and assignment functions

Our point of departure is Bennett (1979), who intended to design a denotation for questions as open formulas, but as an open formula standardly denotes a truth value, such ordinary denotations are unusable for that purpose. However, if open formulas are instead represented as sets of assignment functions, they have enough internal structure to be useful. Assignment functions themselves are, as usual, construed as functions from variables to entities of the usual sort. But now, these variables must denote entities of the model, and for this purpose Bennett simply took integers to be the representatives of variables, namely those integers that normally appear as the subscripts of variables in $x_1, x_2, \ldots x_n$. The correspondence between open formulas of L_0 and sets of assignment functions of L_1 illustrated in (14) is straightforward:

(14) a. L_0: $P(x_1, x_6, x_7)$
b. L_1: $\lambda g.P(g(1), g(6), g(7))$

In (14-b), the variable g is a function from an index to an individual. Indices (or pointers, sometimes also called discourse markers) are constants of type n, therefore g has type $\langle n, e\rangle$.[2] To start with, we assume that (14-b) is the translation of (14-a) into the target language L_1. Likewise, the translation of variables x_i is $\lambda g.g(i)$ which will also be the translation of pronouns of Natural Language. Such an expression is called a pseudo-variable, for the obvious reason that it does not contain any free variable.

At this point we can already grasp the basic intuition that will enable us to interpret β-reduction in the desired way: the relevant feature of the translation of variables and open formulas is that none of the translations in L_1 will contain any free variables whatsoever. This is the crucial feature of the system: as there are no free variables in the translated formulas, the problematic cases for β-reduction simply do not arise in the target language. Moreover, the interpretation/translation is in an obvious sense

[2]The functions g are subsets of $\mathbb{N} \times D$ (where D is the domain of entities), whereas the assignment functions a of standard predicate logic are subsets of $VAR \times D$ (where VAR is the set of variables).

equivalent to its source. Let $[\![.]\!]_g$ be the usual interpretation function for L_0. Let $\mathbf{T}(\alpha)$ be the translation of a formula or term into L_1. Given the result of the translation procedure (to be specified precisely further below), namely that $\mathbf{T}(\alpha)$ never contains a free variable, the interpretation function for the resulting formulas of L_1 does not depend on an assignment for variables; this will simply be the function $[\![.]\!]$. In order to compare the standard interpretation of L_0 with its new interpretation via L_1, let both $[\![.]\!]$ and $[\![.]\!]_g$ depend on the same model for constants of L_0 (but this additional index $\mathscr{M}$ is omitted in what follows). The equivalence can then be expressed as in (15):

(15) $[\![\alpha]\!]_g = [\![\mathbf{T}(\alpha)]\!](g) = [\![\mathbf{T}(\alpha)(g)]\!]$ for any assignment function g.[3]

To get the complete picture we must deal with quantifiers. Bennett's analysis is simply a restatement of the usual truth conditions for quantification of L_0 now expressed in L_1 rather than in the meta-language of L_0. Accordingly, the first thing to do is express modified assignments in L_1:

(16) *Modified assignments:* $g[i/y] := (\iota f)(f(i) = y \wedge \forall n(n \neq i \rightarrow f(n) = g(n)))$

These are needed for stating universal quantification as shown in (17):

(17) *Universal Quantification:* $\mathbf{T}(\forall x_i \phi) = \lambda g \forall y_i \mathbf{T}(\phi)(g[i/y_i]$[4]

Note that hitherto we only translated meta language into object language; the only new device needed to do so is to shift indices (pointers, discourse referents) from the meta language into the language of L_1. The remaining clauses for deriving full-fledged predicate logic are given in (18):

(18) a. $\mathbf{T}(\neg\phi) = \lambda g \neg \mathbf{T}(\phi)(g)$
b. $\mathbf{T}((\phi \wedge \chi) = \lambda g(\mathbf{T}(\phi)(g) \wedge \mathbf{T}(\chi)(g))$

It is obvious that up to now nothing has changed in the semantics of logical expressions.

As the reader may verify, the new format already solves most reconstruction problems. Let us return to (7) repeated as (19):

(19) That he$_i$ is smart, noone$_i$ doubts

Assume that *doubts* translates as the formula in (20) that contains a free variable p of type $\langle\langle n, e\rangle, \langle s, t\rangle\rangle$ for propositions; this variable corresponds to the trace left by the movement operation in (19):

[3] Note that the first g is an expression of the meta language of L_0, the second g belongs to the meta language of L_1 and the third is an expression of L_1.

[4] Since the bound variables in the target language need to be distinguished both from the bound variables in the source language and from other bound variables in the target language, we shall use x for universally quantified variables in the source language and y for universally quantified variables in the target language. Moreover, different indices in the source language must be mapped onto different indices in the target language. For ease of readability we map x_i to y_i (a mapping from x_i to y_{2i} would also do the job).

(20) $\lambda g.\mathbf{doubt}(g(i), p_j(g))$

Adding the (negative) quantifier that binds the subject position and applying (17) yields (21):

(21) $\lambda g' \neg \exists y_i [\lambda g.\mathbf{doubt}(g(i), p_j(g))](g'[i/y_i])$
$=$
$\lambda g' \neg \exists y_i \mathbf{doubt}(g'[i/y_i](i), p_j(g'[i/y_i]))$
$=$
$\lambda g' \neg \exists y_i \mathbf{doubt}(y_i, p_j(g'[i/y_i]))$
$=$
$\lambda g \neg \exists y_i \mathbf{doubt}(y_i, p_j(g[i/y_i]))$

The important assumption here is that the syntax has to tell us that it is the subject that binds the argument with index i; this is a matter (also called theta role assignment) that will be ignored in what follows. The next steps are straightforward: As usual, the effect of movement is captured by lambda abstraction over the open proposition:

(22) $\lambda p_j \lambda g \neg \exists y_i \mathbf{doubt}(y_i, p_j(g[i/y_i]))$

And finally the topicalized clause *that he$_i$ is smart* translates as the open proposition (23).[5]

(23) $\lambda g'.\mathbf{smart}(g'(i))$

(23) will be the argument of (22). By intensional functional application we get

(24) $\lambda p_j \lambda g \neg \exists y_i \mathbf{doubt}(y_i, p_j(g[i/y_i]))(\lambda g'.\hat{}\mathbf{smart}(g'(i)))$
$=$
$\lambda g \neg \exists y_i \mathbf{doubt}(y_i, \lambda g'.\hat{}\mathbf{smart}(g'(i))(g[i/y_i]))$
$=$
$\lambda g \neg \exists y_i \mathbf{doubt}(y_i, \hat{}\mathbf{smart}(g[i/y_i](i)))$
$=$
$\lambda g \neg \exists y_i \mathbf{doubt}(y_i, \hat{}\mathbf{smart}(y_i))$

This is exactly what we aimed for. More applications of the system just described and further discussion can be found in Sternefeld (2001, 2013).

As should be obvious, intensionality is irrelevant for the problem under discussion, hence we will ignore intensions and dismiss with the semantic type s. Accordingly, propositional variables have the simplified type $\langle\langle n, e\rangle, t\rangle$ and the logic to be developed below is extensional.

[5]Note that the choice of variables g or g' is made for mnemotechnical reasons only.

3 Unrestricted semantic reconstruction

The goal we are attempting to reach in this paper is more ambitious than the examples discussed above would suggest. What we want to develop is a system that not only works for the reconstruction of open propositions but for β-reduction in general. This aim is much more difficult to attain. The problem so far is that for examples like (5) there is simply no open proposition that could reconstruct; what is needed is the semantic reconstruction of a variable (or more generally, a term) simpliciter.

As the system we are going to develop is quite complex, we will try to motivate each step by showing what goes wrong in a simpler system, developing the translation in a piecemeal fashion. First we discuss iterated abstraction and functional application. Secondly, we show that given these translations of abstraction and application we need continuations in order to account for quantification. Thirdly, we show that delayed binding via quantification differs from delayed binding via abstraction, and discuss how to account for this asymmetry, namely by introducing an index set which keeps track of all the indices quantified over.

3.1 Abstraction and application

Assuming that $P(x_1, x_2)$ translates as $\lambda g.P(g(1), g(2))$ we first ask how to define abstraction and application so that e.g. the formula $(\lambda x_2 \lambda x_1 P(x_1, x_2))(x_1)$ gets the same translation as $\lambda x_1 P(x_1, x_1)$. To start with, assume the following types for (meta)variables and constants:

(25) a. Variables of L_1:
$\tau(g), \tau(g'), \ldots = \langle n, e\rangle$
$\tau(\Psi), \tau(\Psi'), \ldots = \langle\langle n, e\rangle, e\rangle$

b. Constants of L_1:
all constants of L_0
$\tau(1), \tau(2), \ldots = n$
$\tau(A) = \langle\tau(\Psi), \langle n, \langle\tau(g), \tau(g)\rangle\rangle\rangle$

c. Metavariables:
$\tau(i), \tau(j) = n$

Given a formula $\alpha \in L_0$ of type t, we define:

(26) **Translation of abstraction (first version):**
$\mathbf{T}(\lambda x_i \alpha) = \lambda\Psi\lambda g[\mathbf{T}(\alpha)(A(\Psi)(i)(g))]$,
where $\mathbf{T}(\alpha)$ is the translation of α and A is a constant function defined as

$$A(\Psi)(i)(g)(j) = \begin{cases} \Psi(g), & \text{if } i = j \\ g(j), & \text{else} \end{cases}$$

We then have:

(27) $\mathbf{T}(\lambda x_1 P(x_1, x_2))$

= (translation of abstraction)
$\lambda\Psi\lambda g[\mathbf{T}(P(x_1,x_2))(A(\Psi)(1)(g))]$
= (translation of atomic formulas)
$\lambda\Psi\lambda g[\lambda g' P(g'(1),g'(2))(A(\Psi)(1)(g))]$
= (conversion of g')
$\lambda\Psi\lambda g[P(A(\Psi)(1)(g)(1),A(\Psi)(1)(g)(2))]$
= (definition of A)
$\lambda\Psi\lambda g[P(\Psi(g),g(2))]$

Assume further that functional application is translated as follows:

(28) **Translation of functional application (first version):**
$\mathbf{T}(\alpha(x_i)) = \mathbf{T}(\alpha)(\mathbf{T}(x_i))$

To illustrate functional application, we show that $\mathbf{T}(\lambda x_1 P(x_1,x_2)(x_3)) = \mathbf{T}(P(x_3,x_2))$:

(29) $\mathbf{T}(\lambda x_1 P(x_1,x_2)(x_3))$
= (by translation of functional application)
$\mathbf{T}(\lambda x_1 P(x_1,x_2))(\mathbf{T}(x_3))$
= (by translation of x_3)
$\mathbf{T}(\lambda x_1 P(x_1,x_2))(\lambda g'.g'(3))$
= (by translation of abstraction, see derivation above)
$\lambda\Psi\lambda g[P(\Psi(g),g(2))](\lambda g'.g'(3))$
= (conversion of Ψ)
$\lambda g[P(\lambda g'.g'(3)(g),g(2))]$
= (conversion of g')
$\lambda g[P(g(3),g(2))]$
= (translation of atomic formulas)
$\mathbf{T}(P(x_3,x_2))$

Let us turn next to iterated abstraction. Recall that

(30) $\mathbf{T}(\lambda x_1 P(x_1,x_2)) = \lambda\Psi\lambda g P(A(\Psi)(1)(g)(1),A(\Psi)(1)(g)(2))$

What we want as a translation of $\lambda x_2\lambda x_1 P(x_1,x_2)$ is:

(31) $\mathbf{T}(\lambda x_2\lambda x_1 P(x_1,x_2)) = \lambda\Psi'\lambda\Psi\lambda g P(A(\Psi')(2)(A(\Psi)(1)(g))(1),A(\Psi')(2)(A(\Psi)(1)(g))(2))$

Applying this term to $\lambda g'.g'(1)$, the translation of x_1, we get:

(32) $\lambda\Psi'\lambda\Psi\lambda g P(A(\Psi')(2)(A(\Psi)(1)(g))(1),A(\Psi')(2)(A(\Psi)(1)(g))(2))(\lambda g' g'(1))$
= (definition of A)
$\lambda\Psi'\lambda\Psi\lambda g P(A(\Psi)(1)(g)(1),\Psi'(A(\Psi)(1)(g)))(\lambda g' g'(1))$
= (definition of A)
$\lambda\Psi'\lambda\Psi\lambda g P(\Psi(g),\Psi'(A(\Psi)(1)(g)))(\lambda g' g'(1))$
= (conversion of Ψ')

$\lambda\Psi\lambda g P(\Psi(g), \lambda g'.g'(1)(A(\Psi)(1)(g)))$
= (conversion of g')
$\lambda\Psi\lambda g P(\Psi(g), A(\Psi)(1)(g)(1))$
= (definition of A)
$\lambda\Psi\lambda g P(A(\Psi)(1)(g)(1), A(\Psi)(1)(g)(1))$
= (conversion of g' below)
$\lambda\Psi\lambda g[\lambda g' P(g'(1), g'(1))(A(\Psi)(1)(g))]$
= (definition of atomic formulas)
$\lambda\Psi\lambda g[\mathbf{T}(P(x_1, x_1))(A(\Psi)(1)(g))]$
= (definition of **T**, abstraction)
$\mathbf{T}(\lambda x_1 P(x_1, x_1))$

Focusing just on the terms x_1 and x_2, we see that their translation in an atomic formula is $g(1)$ and $g(2)$. After the first abstraction over x_1, the corresponding terms are $A(\Psi)(1)(g)(1)$ and $A(\Psi)(1)(g)(2)$, respectively. After the second abstraction over x_2, what we want to get are the terms $A(\Psi')(2)(A(\Psi)(1)(g))(1)$ (which by definition of A is identical to $A(\Psi)(1)(g)(1)$) and $A(\Psi')(2)(A(\Psi)(1)(g))(2)$ (which by definition of A is identical to $\Psi'(A(\Psi)(1)(g))$, respectively, which after application to $\lambda g' g'(1)$ both turn to $A(\Psi)(1)(g)(1) = \Psi(g)$.

So far, so good. But now the crucial question is how to arrive at (31) in a systematic (recursive) way on the basis of (30). According to our preliminary definition of abstraction and application, the only terms we can substitute in $A(\Psi)(1)(g)(1)$ are Ψ and g, so by abstracting over x_2 we need to get from the term $A(\Psi)(1)(g)(1)$ to the term $A(\Psi')(2)(A(\Psi)(1)(g))(1)$ just by substituting Ψ and g. As it happens this is not feasible. What we can do instead is introduce variables h for $A(\Psi)$ and h' for $A(\Psi')$, so that our task can be reformulated as getting from $h(1)(g)(1)$ to $h'(2)(h(1)(g))(1)$ by replacing h and/or g.[6] This can be achieved by substituting h in $h(1)(g)(1)$ with $\lambda u\lambda v.h'(2)(h(u)(v))$, where u is a variable of type n and v is a variable of type $\langle n, e\rangle$. To see this, note that:

(33) $(\lambda u\lambda v.h'(2)(h(u)(v)))(1)(g)(1)$
= (conversion of u)
$(\lambda v.h'(2)(h(1)(v)))(g)(1)$
= (by conversion of v and omission of outer brackets)
$h'(2)(h(1)(g))(1)$

To make this work we need to first adjust the translation of application. Instead of stipulating that $\mathbf{T}(\alpha(x_i)) = \mathbf{T}(\alpha)(\mathbf{T}(x_i))$ we say that $\mathbf{T}(\alpha(x_i)) = \mathbf{T}(\alpha)(A(\mathbf{T}(x_i)))$. Secondly, we need to adjust the translation of abstraction in case α is of type *t*. Instead of saying that $\mathbf{T}(\lambda x_i \alpha) = \lambda\Psi\lambda g.[\mathbf{T}(\alpha)(A(\Psi)(i)(g))]$ we say that $\mathbf{T}(\lambda x_i \alpha) = \lambda h\lambda g.[\mathbf{T}(\alpha)(h(i)(g))]$. Finally, for all other cases of abstraction (i.e. α not of type t) we stipulate that

(34) $\mathbf{T}(\lambda x_i \alpha) = \lambda h'\lambda h.[\mathbf{T}(\alpha)(\lambda u\lambda v.h'(i)(h(u)(v)))]$

[6] Since A has type $\langle\tau(\Psi), \langle n, \langle\tau(g), \tau(g)\rangle\rangle\rangle$ it follows that $A(\Psi)$ is of type $\langle n, \langle\tau(g), \tau(g)\rangle\rangle$, so the variable h is of type $\langle n, \langle\tau(g), \tau(g)\rangle\rangle$, too.

(35) **Translation of abstraction (second version):**

$$\mathbf{T}(\lambda x_i \alpha) = \begin{cases} \lambda h \lambda g[\mathbf{T}(\alpha)(h(i)(g))], & \text{if } \alpha \text{ has type } t \\ \lambda h \lambda h'[\mathbf{T}(\alpha)(\lambda u \lambda v.h(i)(h'(u)(v)))], & \text{else} \end{cases}$$

(36) **Translation of functional application (second version):**

$\mathbf{T}(\phi(x_i)) = \mathbf{T}(\phi)(A(\mathbf{T}(x_i)))$

where $A(\mathbf{T}(x_i))(j)(g)(k) = \begin{cases} \mathbf{T}(x_i)(g), & \text{if } j = k \\ g(k), & \text{else} \end{cases}$

To illustrate these translations in full detail, we show that $\mathbf{T}((\lambda x_2 \lambda x_1 P(x_1, x_2))(x_1)) = \mathbf{T}(\lambda x_1 P(x_1, x_1))$:

(37) $\mathbf{T}((\lambda x_2 \lambda x_1 P(x_1, x_2))(x_1))$
= (translation of application)
$\mathbf{T}(\lambda x_2 \lambda x_1 P(x_1, x_2))(A(\mathbf{T}(x_1)))$
= (translation of $\lambda x_i \alpha$ if α is not of type t)
$\lambda h' \lambda h[\mathbf{T}(\lambda x_1 P(x_1, x_2))(\lambda u \lambda v.h'(2)(h(u)(v)))](A(\mathbf{T}(x_1)))$
= (conversion of h')
$\lambda h[\mathbf{T}(\lambda x_1 P(x_1, x_2))(\lambda u \lambda v.A(\mathbf{T}(x_1))(2)(h(u)(v)))]$
= (translation of $\lambda x_i \alpha$ if α is of type t)
$\lambda h[\lambda h' \lambda g'[\mathbf{T}(P(x_1, x_2))(h'(1)(g'))](\lambda u \lambda v.A(\mathbf{T}(x_1))(2)(h(u)(v)))]$
= (conversion of h')
$\lambda h[\lambda g'[\mathbf{T}(P(x_1, x_2))(\lambda u \lambda v.A(\mathbf{T}(x_1))(2)(h(u)(v))(1)(g'))]]$
= (conversion of u)
$\lambda h[\lambda g'[\mathbf{T}(P(x_1, x_2))(\lambda v.A(\mathbf{T}(x_1))(2)(h(1)(v))(g'))]]$
= (conversion of v)
$\lambda h[\lambda g'[\mathbf{T}(P(x_1, x_2))(A(\mathbf{T}(x_1))(2)(h(1)(g')))]]$
= (translation of atomic formulas)
$\lambda h[\lambda g'[\lambda g'' P(g''(1), g''(2))(A(\mathbf{T}(x_1))(2)(h(1)(g')))]]$
= (conversion of g'')
$\lambda h[\lambda g'[P(A(\mathbf{T}(x_1))(2)(h(1)(g'))(1), A(\mathbf{T}(x_1))(2)(h(1)(g'))(2))]]$
= (definition of A)
$\lambda h[\lambda g'[P(h(1)(g')(1), \mathbf{T}(x_1)(h(1)(g')))]]$
= (translation of x_1)
$\lambda h[\lambda g'[P(h(1)(g')(1), \lambda g.g(1)(h(1)(g')))]]$
= (conversion of g)
$\lambda h \lambda g'[P(h(1)(g')(1), h(1)(g')(1))]$
= (conversion of g below)
$\lambda h \lambda g'[\lambda g.P(g(1), g(1))(h(1)(g'))]$
= (translation of atomic formulas)
$\lambda h \lambda g'[\mathbf{T}(P(x_1, x_1))(h(1)(g'))]$
= (translation of abstraction)
$\mathbf{T}(\lambda x_1 P(x_1, x_1))$

Note that our interpretation of lambda abstraction is non-standard as it does not satisfy alpha equivalence. For example, (38-a) and (38-b)

(38) a. $\lambda x_2 \lambda x_1 . P(x_1, x_2)$
b. $\lambda x_2 \lambda x_3 . P(x_3, x_2)$

are equivalent in L_0, but this cannot hold for the respective translations in L_1. If this were the case the results of applying (38-a) and (38-b) to x_1 should be identical, but as we have argued above this outcome is unwarranted. This difference of interpretation also implies that depending on the choice of α it does not always hold that $\mathbf{T}(\lambda x_i \ldots(\alpha)) = \mathbf{T}(\lambda x_i \ldots)(\mathbf{T}(\alpha))$. It follows that the system is not alphabetically invariant when it comes to binding by lambda operators.[7]

3.2 Quantification via continuations

As the truth conditions for quantification and lambda abstraction will become more complex, we will give a list of type assignments τ for constants and variables introduced so far. As usual, e is for entities, t for truth values, n for integers.

(39) a. Variables of L_1:
$\tau(u) = n$
$\tau(g), \tau(g'), \ldots = \tau(v) = \langle n, e \rangle$
$\tau(\Psi), \tau(\Psi'), \ldots = \langle \langle n, e \rangle, e \rangle$
$\tau(h), \tau(h'), \ldots = \langle n, \langle \tau(g), \tau(g) \rangle \rangle$
b. Constants of L_1:
all constants of L_0
$\tau(1), \tau(2), \ldots = n$
$\tau(A) = \langle \tau(\Psi), \langle n, \langle \tau(g), \tau(g) \rangle \rangle \rangle$
c. Metavariables:
$\tau(i), \tau(j) = n$

Next we turn to universally quantified formulas, and ask ourselves how to translate them so that e.g. the formula $\lambda x_2 \forall x_1 P(x_1, x_2)(x_1)$ gets the same translation as the formula $\forall x_1 P(x_1, x_1)$. Let us begin by assuming that:

(40) **Translation of quantification (first version):**
$\mathbf{T}(\forall x_i \alpha) = \lambda g \forall y_i [\mathbf{T}(\alpha)(g[i/y_i])]$[8]

To illustrate, consider the translation of $\forall x_1 P(x_1, x_2)$:

(41) $\mathbf{T}(\forall x_1 P(x_1, x_2))$

[7] In particular, the attempt to assimilate the format of lambda abstraction of L_1 to that of L_0 by saying that a set of individuals (or the characteristic function thereof) in L_0 should correspond to a set pseudo variables defined by something like $\lambda \psi \mathbf{T}(\alpha)(A(\psi))$ would not make much sense as this similarity disappears when it comes to functional application.

[8] By definition, $g[i/y_i] := \iota f(f(i) = y_i \wedge \forall j(j \neq i \rightarrow f(j) g(j)))$.

= (by this translation of quantified formulas)
$\lambda g \forall y_1[\mathbf{T}(P(x_1,x_2))(g[1/y_1])]$
= (by translation of atomic formulas)
$\lambda g \forall y_1[\lambda g' P(g'(1),g'(2))(g[1/y_1])]$
= (by conversion of g')
$\lambda g \forall y_1[P(g[1/y_1](1),g[1/y_1](2))]$
= (by definition of modification $g[i/x]$ of g)
$\lambda g \forall y_1[P(y_1,g[1/y_1](2))]$

However, this translation of quantified formulas does not yield the desired equivalence of $\mathbf{T}(\lambda x_2 \forall x_1 P(x_1,x_2)(x_1))$ and $\mathbf{T}(\forall x_1 P(x_1,x_1))$. Understanding exactly why this translation of quantified formulas fails will point to the solution, namely the introduction of continuations. So let us see why this translation does not work:

(42) $\mathbf{T}(\lambda x_2 \forall x_1 P(x_1,x_2)(x_1))$
= (translation of functional application)
$\mathbf{T}(\lambda x_2 \forall x_1 P(x_1,x_2))(A(\mathbf{T}(x_1)))$
= (translation of abstraction if α is of type t)
$\lambda h \lambda g.[\mathbf{T}(\forall x_1 P(x_1,x_2))(h(2)(g))](A(\mathbf{T}(x_1)))$
= (translation of quantified formulas)
$\lambda h \lambda g.[\lambda g' \forall y_1[\mathbf{T}(P(x_1,x_2))(g'[1/y_1])](h(2)(g))](A(\mathbf{T}(x_1)))$
= (translation of atomic formulas)
$\lambda h \lambda g.[\lambda g' \forall y_1[\lambda g'' P(g''(1),g''(2))(g'[1/y_1])](h(2)(g))](A(\mathbf{T}(x_1)))$
= (by conversion of g'')
$\lambda h \lambda g.[\lambda g' \forall y_1[P(g'[1/y_1](1),g'[1/y_1](2))](h(2)(g))](A(\mathbf{T}(x_1)))$
= (by conversion of g')
$\lambda h \lambda g.[\forall y_1[P(h(2)(g)[1/y_1](1),h(2)(g)[1/y_1](2))]](A(\mathbf{T}(x_1)))$
= (by definition of modification: $h(2)(g)[1/y_1]$ is the function which when applied to 1 yields y_1, and if applied to any integer x other than 1 yields $h(2)(g)(x)$)
$\lambda h \lambda g.[\forall y_1[P(y_1,h(2)(g)[1/y_1](2))]](A(\mathbf{T}(x_1)))$
= (by definition of modification)
$\lambda h \lambda g.[\forall y_1[P(y_1,h(2)(g)(2))]](A(\mathbf{T}(x_1)))$
= (by conversion of h)
$\lambda g.[\forall y_1[P(y_1,A(\mathbf{T}(x_1))(2)(g)(2))]]$
= (by definition of A)
$\lambda g.[\forall y_1[P(y_1,\mathbf{T}(x_1)(g))]]$
= (by translation of x_1)
$\lambda g.[\forall y_1[P(y_1,\lambda g'.g'(1)(g))]]$
= (by conversion of g')
$\lambda g.[\forall y_1[P(y_1,g(1))]]$

Note that this translation differs from the translation of $\forall x_1 P(x_1,x_1)$:

(43) $\mathbf{T}(\forall x_1 P(x_1,x_1))$

= (by translation of quantified formulas)
$\lambda g \forall y_1[\mathbf{T}(P(x_1,x_1))(g[1/y_1])]$
= (by translation of atomic formulas)
$\lambda g \forall y_1[\lambda g' P(g'(1),g'(1))(g[1/y_1])]$
= (by conversion of g')
$\lambda g \forall y_1[P(g[1/y_1](1),g[1/y_1](1))]$
= (by definition of modification)
$\lambda g \forall y_1[P(y_1,y_1)]$

So why exactly does the translation of quantification fail? Consider again the step in which $\lambda g'$ gets converted. During this step the term $g'[1/y_1](2)$ is transformed into $h(2)(g)[1/y_1](2)$, which by the definition of modification is identical with the term $h(2)(g)(2)$ (given that $h(2)(g)[1/y_1]$ is that function which when applied to 1 yields y_1, and when applied to any other integer x yields $h(2)(g)(x)$). The crucial point leading to the failure of this preliminary translation of quantification is that at this conversion step we loose the information about the quantification of x_1 – the modified assignment $g'[1/y_1]$ in the term $g'[1/y_1](2)$ does not survive abstraction of x_2!

To overcome this shortcoming we introduce modifiers of assignment functions, represented by variables $c, c', c'', \ldots$. The variable c has type $\tau(c) = \langle\langle n,e\rangle,\langle n,e\rangle\rangle$, and represents a place for future modifications of g. So instead of assuming that the translation of $P(x_1,x_2)$ is $\lambda g P(g(1),g(2))$, we now propose that $\mathbf{T}(P(x_1,x_2)) = \lambda c \lambda g P(c(g)(1)),c(g)(2))$. By analogy to other types of in-built future oriented fortune-telling (cf. Barker (2002)), c is called a continuation of g. As in continuation semantics, the equivalence stated in (15) now comes about by applying $\mathbf{T}(\alpha)$ to the identity function $\lambda g.g$:

(44) $[\![\alpha]\!]_g = \mathbf{T}(\alpha)(\lambda g'.g')(g)$ for any assignment function g.

Assuming that F is a constant function of type $\langle n,\langle\tau(c),\tau(c)\rangle\rangle$, quantified formulas will be translated as follows:

(45) **Translation of quantification (second version):**
$\mathbf{T}(\forall x_i \alpha) = \lambda c \lambda g \forall y_i[T(\alpha)(F(i)(c))(g[i/y_i])]$
where
$$F(i)(c)(g)(j) = \begin{cases} g(j), & \text{if } i = j \\ c(g)(j), & \text{else} \end{cases}$$

To illustrate this definition, consider the translation of $\forall x_1 P(x_1,x_2)$:

(46) $\mathbf{T}(\forall x_1 P(x_1,x_2))$
= (translation of quantified formulas)
$\lambda c \lambda g[\mathbf{T}(P(x_1,x_2))(F(1)(c))(g[1/y_1])]$
= (translation of atomic formulas)
$\lambda c \lambda g[\lambda c' \lambda g' P(c'(g')(1),c'(g')(2))(F(1)(c))(g[1/y_1])]$
= (conversion of $\lambda c'$)
$\lambda c \lambda g[\lambda g' P(F(1)(c)(g')(1),F(1)(c)(g')(2))(g[1/y_1])]$
= (conversion of $\lambda g'$)

$\lambda c \lambda g[P(F(1)(c)(g[1/y_1])(1), F(1)(c)(g[1/y_1])(2))]$
= (definition of F)
$\lambda c \lambda g[P(g[1/y_1](1), F(1)(c)(g[1/y_1])(2))]$
= (definition of modification)
$\lambda c \lambda g[P(y_1, F(1)(c)(g[1/y_1])(2))]$
= (definition of F, omission of square brackets)
$\lambda c \lambda g.P(y_1, c(g[1/y_1])(2))$

The introduction of continuation variables c necessitates the adjustment of the translation of abstraction and application:

(47) **Translation of abstraction (third and final version):**
$\mathbf{T}(\lambda x_i \alpha) =$
$= \begin{cases} \lambda h \lambda c[\mathbf{T}(\alpha)(h(i)(c))], & \text{if } \alpha \text{ has type } t \\ \lambda h \lambda h'[\mathbf{T}(\alpha)(\lambda u \lambda v h(i)(h'(u)(v)))], & \text{else} \end{cases}$

(48) **Translation of functional application (third version):**
$\mathbf{T}(\alpha(x_i)) = \mathbf{T}(\alpha)(A(\mathbf{T}(x_i)))$
where $A(\mathbf{T}(x_i))(j)(c)(g)(k) = \begin{cases} \mathbf{T}(x_i)(g), & \text{if } j = k \\ c(g)(k), & \text{else} \end{cases}$

With these adjusted definitions of translation we can show that the formula $\lambda x_2 \forall x_1 P(x_1, x_2)(x_1)$ gets the same translation as the formula $\forall x_1 P(x_1, x_1)$:

(49) $\mathbf{T}(\lambda x_2 \forall x_1 P(x_1, x_2)(x_1))$
= (translation of functional application)
$\mathbf{T}(\lambda x_2 \forall x_1 P(x_1, x_2))(A(\mathbf{T}(x_1)))$
= (translation of abstraction if α is of type t)
$\lambda h \lambda c[\mathbf{T}(\forall x_1 P(x_1, x_2))(h(2)(c))](A(\mathbf{T}(x_1)))$
= (translation of $\forall x_1 P(x_1, x_2)$)
$\lambda h \lambda c[\lambda c' \lambda g' \forall y_1 P(y_1, c'(g'[1/y_1])(2))(h(2)(c))](A(\mathbf{T}(x_1)))$
= (by conversion of c')
$\lambda h \lambda c[\lambda g' \forall y_1 P(y_1, \underline{h(2)(c)(g'[1/y_1])(2))}](A(\mathbf{T}(x_1)))$
= (by conversion of h, omission of square brackets)
$\lambda c \lambda g' \forall y_1 P(y_1, A(\mathbf{T}(x_1))(2)(c)(g'[1/y_1])(2))$
= (by definition of A)
$\lambda c \lambda g' \forall y_1 P(y_1, \mathbf{T}(x_1)(g'[1/y_1]))$
= (by translation of x_1)
$\lambda c \lambda g' \forall y_1 P(y_1, \lambda g''.g''(1)(g'[1/y_1]))$
= (by conversion of g'')
$\lambda c \lambda g' \forall y_1 P(y_1, g'[1/y_1](1))$
= (by definition of modification)
$\lambda c \lambda g' \forall y_1 P(y_1, y_1)$
= (by definition of modification)
$\lambda c \lambda g' \forall y_1 P(g'[1/y_1](1), g'[1/y_1](1))$

= (definition of F)
$\lambda c \lambda g' \forall y_1 [P(F(1)(c)(g'[1/y_1])(1), F(1)(c)(g'[1/y_1])(1))]$
= (conversion of g'' below)
$\lambda c \lambda g' \forall y_1 [\lambda g'' P(F(1)(c)(g'')(1), F(1)(c)(g'')(1))(g'[1/y_1])]$
= (conversion of c'' below)
$\lambda c \lambda g' \forall y_1 [\lambda c'' \lambda g'' P(c''(g'')(1), c''(g'')(1))(F(1)(c))(g'[1/y_1])]$
= (translation of atomic formulas)
$\lambda c \lambda g' \forall y_1 [\mathbf{T}(P(x_1, x_1))(F(1)(c))(g'[1/y_1])]$
= (translation of quantified formulas)
$\mathbf{T}(\forall x_1 P(x_1, x_1))$

The crucial difference to the previous translation definitions occurs at the conversion of g'. Abstraction over x_2 converts the term $c'(g'[1/y_1])(2)$ into $h(2)(c)(g'[1/y_1])(2)$ (see underlined term in derivation). Note, importantly, that the information about the way in which quantification modified the assignment is preserved. After substitution of h by $A(\mathbf{T}(x_1))$ this term is transformed into $A(\mathbf{T}(x_1))(2)(c)(g'[1/y_1])(2)$, which by definition of A is identical with $\mathbf{T}(x_1)(g'[1/y_1])$. After substitution of $\mathbf{T}(x_1)$ by $\lambda g.g(1)$ we get $\lambda g.g(1)(g'[1/y_1])$, which after conversion of λg results in $g'[1/y_1](1)$. Finally by the definition of modification this term is identical with y_1.

In contrast, previously the term $g'[1/y_1](2)$ was transformed into $h(2)(g)[1/y_1](2)$ (which by definition of modification is the term $h(2)(g)(2)$), leading to a loss of information about the way in which the assignments were modified by quantification. To sum up, the introduction of continuations allows for the preservation of modified assignments during abstraction.

Summing up the type of the variables and constants used so far we have:

(50) a. Variables of L_1:
$\tau(y_1), \tau(y_2), \ldots = e$
$\tau(u) = n$
$\tau(g), \tau(g'), \ldots \langle n, e \rangle$
$\tau(c), \tau(c'), \ldots = \tau(v) = \langle \tau(g), \tau(g) \rangle$
$\tau(h), \tau(h'), \ldots = \langle n, \langle \tau(c), \tau(c) \rangle \rangle$
$\tau(\Psi), \tau(\Psi'), \ldots = \langle \langle n, e \rangle, e \rangle$

b. Constants of L_1:
all constants of L_0
$\tau(1), \tau(2), \ldots = n$
$\tau(A) =$
$= \langle \tau(\psi), \tau(h) \rangle$
$= \langle \tau(\psi), \langle n, \langle \tau(c), \tau(c) \rangle \rangle \rangle$
$= \langle \tau(\psi), \langle n, \langle \tau(c), \langle \tau(g), \tau(g) \rangle \rangle \rangle \rangle$
$\tau(F) =$
$= \langle n, \langle \tau(c), \tau(c) \rangle \rangle$
$= \langle n, \langle \tau(c), \langle \tau(g), \tau(g) \rangle \rangle \rangle$
$= \langle n, \langle \tau(c), \langle \tau(g), \langle n, e \rangle \rangle \rangle \rangle$

c. Metavariables:
$\tau(i), \tau(j) = n$

3.3 Asymmetry of delayed quantification and abstraction binding

The last step in the formulation of the translation definition is to account for an asymmetry in delayed binding. To illustrate the issue, we would like the translation of the term $\lambda x_2 P(x_5, x_2)(x_1)$ to be $\lambda c \lambda g' P(c(g')(5), c(g')(1))$, but what we actually get instead is $\lambda c \lambda g' P(c(g')(5), g'(1))$. Again, understanding why will point to the solution:

(51) $\mathbf{T}(\lambda x_2 P(x_5, x_2)(x_1))$
= (translation of application)
$\mathbf{T}(\lambda x_2 P(x_5, x_2))(A(\mathbf{T}(x_1)))$
= (translation of abstraction for α of type t)
$\lambda h \lambda c[\mathbf{T}(P(x_5, x_2))(h(2)(c))](A(\mathbf{T}(x_1)))$
= (translation of atomic formulas)
$\lambda h \lambda c[\lambda c' \lambda g' P(c'(g')(5), c'(g')(2))(h(2)(c))](A(\mathbf{T}(x_1)))$
= (conversion of c')
$\lambda h \lambda c[\lambda g' P(h(2)(c)(g')(5), h(2)(c)(g')(2))](A(\mathbf{T}(x_1)))$
= (conversion of h)
$\lambda c[\lambda g' P(A(\mathbf{T}(x_1))(2)(c)(g')(5), A(\mathbf{T}(x_1))(2)(c)(g')(2))]$
= (definition of A)
$\lambda c[\lambda g' P(c(g')(5), \mathbf{T}(x_1)(g'))]$
= (translation of x_1)
$\lambda c[\lambda g' P(c(g')(5), \lambda g.g(1)(g'))]$
= (conversion of g)
$\lambda c[\lambda g' P(c(g')(5), g'(1))]$

The reason we get the term $g'(1)$ and not as desired the term $c(g')(1)$ is that the constant function A is defined such that if the two indices i and j are identical, then the value is $A(\Psi)(i)(c)(g)(j) = \Psi(g)$. The motivation for this was so that in case of e.g. delayed binding via the universal quantifier $\forall x_1$, the result is the term in (52):

(52) $\Psi(g[1/y_1]) = \lambda g' g'(1)(g[1/y_1]) = g[1/y1](1) = y_1.$

And consequently we would have:

(53) $\mathbf{T}([\lambda x_2 \forall x_1 P(x_2)(x_1)) = \lambda c \lambda g \forall y_1 P(g[1/y_1](1)) = \lambda c \lambda g \forall y_1 P(y_1)$

The derivation of the term in (53) is:

(54) $c(g)(2)$
= (quantification over x_1)
$F(1)(c)(g[1/y_1])(2)$
= (definition of F)

$c(g[1/y_1])(2)$
= (abstraction over x_2)
$h(2)(c)(g[1/y_1])(2)$
= (application to x_1)
$A(T(x_1))(2)(c)(g[1/y_1])(2)$
= (definition of A)
$T(x_1)(g[1/y_1])$
= (transl. of x_1)
$\lambda g' g'(1)(g[1/y_1])$
= (conversion of g')
$g[1/y_1](1)$
= (definition of modification)
y_1

In case of delayed binding via abstraction, however, we do not want the resulting term to be $g[1/y1](1) = y_1$, but $c(g[1/y1])(1)$. But precisely here we encounter the problem. To see why, we focus again on the derivation of the term of the translation of the formula (55), illustrated in (56):

(55) $\mathbf{T}(\lambda x_2 \lambda x_1 P(x_2)(x_1)) \neq \lambda h \lambda c \lambda g P(c(g)(1))$

(56) $c(g)(2)$
= (abstraction over x_1)
$h(1)(c)(g)(2)$
= (abstraction over x_2)
$h'(2)(h(1)(c))(g)(2)$
= (application to x_1)
$A(\lambda g' g'(1))(2)(h(1)(c))(g)(2)$
= (definition of A)
$\lambda g'.g'(1)(g)$
=
$g(1)$

To account for this asymmetry between delayed binding via quantification and delayed binding via abstraction we need to make the constant function A sensitive to whether or not the last index is to be bound by a quantifier or by an abstractor. If it is to be bound by a quantifier, then we eventually want the term to be of the form $g(i)$, but if the index is to be bound by a λ-abstractor, then the resulting term should be of the form $c(g)(i)$.

To make this work we first introduce a way of storing all indices that are quantified over. To do this we translate atomic predicates of the form $P(x_i)$ into $\lambda c \lambda g.P(c(\varnothing)(g)(i))$, with the empty set indicating that no index has been quantified over yet. The type of c is changes from $\langle \tau(g), \tau(g) \rangle$ to $\tau(c) = \langle \tau(M), \langle \tau(g), \tau(g) \rangle \rangle$, where $\tau(M) = \langle n, t \rangle$, i.e. M is a function from indices to truth-values. Next we modify the translation of quantification so that it adds the index quantified over to the index set M. Below we use $M + i$ as a

conventional notation for set theoretic addition $M \cup \{i\}$:

(57) **Translation of quantification (third and final version):**
$\mathbf{T}(\forall x_i \alpha) = \lambda c \lambda g \forall y_i [\mathbf{T}(\alpha)(\lambda M.F(c)(M + i))(g[i/y_i])]$
where

$$F(c)(M)(g)(j) = \begin{cases} g(j), & \text{if } j \in M \\ c(M)(g)(j), & \text{else} \end{cases}$$

The type of F is now $\tau(F) = \langle \tau(c), \tau(c) \rangle = \langle \tau(c), \langle \tau(M), \langle \tau(g), \tau(g) \rangle \rangle \rangle$. To illustrate briefly the new translation of quantified formulas, consider the translation of $\forall x_1 P(x_1, x_2)$:

(58) $\mathbf{T}(\forall x_1 P(x_1, x_2))$
= (translation of quantification)
$\lambda c \lambda g \forall y_1 [\mathbf{T}(P(x_1, x_2))(\lambda M.F(c)(M + 1))(g[1/y_1])]$
= (translation of atomic formulas)
$\lambda c \lambda g \forall y_1 [\lambda c' \lambda g' P(c'(\varnothing)(g')(1), c'(\varnothing)(g')(2))(\lambda M.F(c)(M + 1))(g[1/y_1])]$
= (conversion of c')
$\lambda c \lambda g \forall y_1 [\lambda g' P(\lambda M.F(c)(M + 1)(\varnothing)(g')(1), \lambda M.F(c)(M + 1)(\varnothing)(g')(2))$
$(g[1/y_1])]$
= (conversion of M, twice)
$\lambda c \lambda g \forall y_1 [\lambda g' P(F(c)(\{1\})(g')(1), F(c)(\{1\})(g')(2))(g[1/y_1])]$
= (conversion of g')
$\lambda c \lambda g \forall y_1 [P(F(c)(\{1\})(g[1/y_1])(1), F(c)(\{1\})(g[1/y_1])(2))]$
= (definition of F)
$\lambda c \lambda g \forall y_1 [P(g[1/y_1](1), F(c)(\{1\})(g[1/y_1])(2))]$
= (definition of modification)
$\lambda c \lambda g \forall y_1 [P(y_1, F(c)(\{1\})(g[1/y_1])(2))]$
= (definition of F)
$\lambda c \lambda g \forall y_1 [P(y_1, c(\{1\})(g[1/y_1])(2))]$

The second step towards accounting for the asymmetry in delayed binding is to redefine functional application:

(59) **Translation of functional application (fourth and final version):**
$\mathbf{T}(\alpha(x_i)) = \mathbf{T}(\alpha)(A(\mathbf{T}(x_i)))$, where:

$$A(\psi)(i)(c)(M)(g)(j) = \begin{cases} \psi(F(c)(M)(g)), & \text{if } i = j \\ c(M)(g)(j), \text{else} \end{cases}$$

Crucially, the constant function A has been redefined such that if $i = j$ then the resulting value is not $\Psi(g)$, but $\Psi(F(c)(M)(g))$. To illustrate, we calculate the translation of $\lambda x_2 P(x_5, x_2)(x_1)$ again, showing that with the modified translations we get the right result, namely $\lambda c \lambda g P(c(g)(5), c(g)(1))$:

(60) $\mathbf{T}(\lambda x_2 P(x_5, x_2)(x_1))$
= (translation of application)
$\mathbf{T}(\lambda x_2 P(x_5, x_2))(A(\mathbf{T}(x_1)))$

= (translation of abstraction for α of type t)
$\lambda h \lambda c[\mathbf{T}(P(x_5, x_2))(h(2)(c))](A(\mathbf{T}(x_1)))$
= (translation of atomic formulas)
$\lambda h \lambda c[\lambda c' \lambda g' P(c'(\varnothing)(g')(5), c'(\varnothing)(g')(2))(h(2)(c))](A(\mathbf{T}(x_1)))$
= (conversion of c')
$\lambda h \lambda c[\lambda g' P(h(2)(c)(\varnothing)(g')(5), h(2)(c)(\varnothing)(g')(2))](A(\mathbf{T}(x_1)))$
= (conversion of h)
$\lambda c[\lambda g' P(A(\mathbf{T}(x_1))(2)(c)(\varnothing)(g')(5), A(\mathbf{T}(x_1))(2)(c)(\varnothing)(g')(2))]$
= (definition of A)
$\lambda c[\lambda g' P(c(\varnothing)(g')(5), \mathbf{T}(x_1)(F(c)(\varnothing)(g')))]$
= (translation of x_1)
$\lambda c[\lambda g' P(c(\varnothing)(g')(5), \lambda g.g(1)(F(c)(\varnothing)(g')))]$
= (conversion of g)
$\lambda c[\lambda g' P(c(\varnothing)(g')(5), F(c)(\varnothing)(g')(1))]$
= (definition of F)
$\lambda c[\lambda g' P(c(\varnothing)(g')(5), c(\varnothing)(g')(1))]$

Finally, we illustrate delayed binding by quantification, by showing that:

$$\mathbf{T}(\lambda x_2 \forall x_1 P(x_1, x_2)(x_1)) = \mathbf{T}(\forall x_1 P(x_1, x_1)) = \lambda c \lambda g \forall y_1 P(y_1, y_1)$$

(61) $\mathbf{T}(\lambda x_2 \forall x_1 P(x_1, x_2)(x_1))$
= (translation of application)
$\mathbf{T}(\lambda x_2 \forall x_1 P(x_1, x_2))(A(\mathbf{T}(x_1)))$
= (translation of abstraction for α of type t)
$\lambda h \lambda c[\mathbf{T}(\forall x_1 P(x_1, x_2))(h(2)(c))](A(\mathbf{T}(x_1)))$
= (translation of quantification)
$\lambda h \lambda c[\lambda c' \lambda g' \forall y_1[\mathbf{T}(P(x_1, x_2))(\lambda M.F(c')(M+1))(g'[1/y_1])](h(2)(c))](A(\mathbf{T}(x_1)))$
= (translation of atomic formulas)
$\lambda h \lambda c[\lambda c' \lambda g' \forall y_1[\lambda c'' \lambda g'' P(c''(\varnothing)(g'')(1), c''(\varnothing)(g'')(2))(\lambda M.F(c')(M+1))$
$(g'[1/y_1])](h(2)(c))](A(\mathbf{T}(x_1)))$
= (conversion of c'')
$\lambda h \lambda c[\lambda c' \lambda g' \forall y_1[\lambda g'' P(\lambda M.F(c')(M+1)(\varnothing)(g'')(1), \lambda M.F(c')(M+1)(\varnothing)(g'')(2))$
$(g'[1/y_1])](h(2)(c))](A(\mathbf{T}(x_1)))$
= (conversion of M, twice)
$\lambda h \lambda c[\lambda c' \lambda g' \forall y_1[\lambda g'' P(F(c')(\{1\})(g'')(1), F(c')(\{1\})(g'')(2))(g'[1/y_1])](h(2)(c))](A(\mathbf{T}(x_1)))$
= (conversion of g'', twice)
$\lambda h \lambda c[\lambda c' \lambda g' \forall y_1[P(F(c')(\{1\})(g'[1/y_1])(1), F(c')(\{1\})(g'[1/y_1])(2))](h(2)(c))](A(\mathbf{T}(x_1)))$
= (conversion of c')
$\lambda h \lambda c[\lambda g' \forall y_1[P(F(h(2)(c))(\{1\})(g'[1/y_1])(1), F(h(2)(c))(\{1\})(g'[1/y_1])(2))]](A(\mathbf{T}(x_1)))$
= (definition of F)
$\lambda h \lambda c[\lambda g' \forall y_1 P(g'[1/y_1](1), h(2)(c)(\{1\})(g'[1/y_1])(2))](A(\mathbf{T}(x_1)))$
= (definition of modification)
$\lambda h \lambda c[\lambda g' \forall y_1 P(y_1, h(2)(c)(\{1\})(g'[1/y_1])(2))](A(\mathbf{T}(x_1)))$

= (conversion of h)
$\lambda c[\lambda g' \forall y_1 P(y_1, A(\mathbf{T}(x_1))(2)(c)(\{1\})(g'[1/y_1])(2))]$
= (definition of A)
$\lambda c[\lambda g' \forall y_1 P(y_1, \mathbf{T}(x_1)(F(c)(\{1\})(g'[1/y_1])))]$
= (translation of x_1)
$\lambda c[\lambda g' \forall y_1 P(y_1, \lambda g.g(1)(F(c)(\{1\})(g'[1/y_1])))]$
= (conversion of g)
$\lambda c[\lambda g' \forall y_1 P(y_1, F(c)(\{1\})(g'[1/y_1])(1))]$
= (definition of F)
$\lambda c \lambda g' \forall y_1 P(y_1, g'[1/y_1](1))$
= (definition of modification)
$\lambda c \lambda g' \forall y_1 P(y_1, y_1)$

Summing up the (final) types of variables and constants, we have:

(62) a. Variables of L_1:
$\tau(y_1), \tau(y_2), \ldots = e$
$\tau(u) = n$
$\tau(g), \tau(g'), \ldots = \langle n, e\rangle$
$\tau(M) = \langle n, t\rangle$
$\tau(c), \tau(c'), \ldots = \tau(v) = \langle \tau(M), \langle \tau(g), \tau(g)\rangle\rangle$
$\tau(h), \tau(h'), \ldots = \langle n, \langle \tau(c), \tau(c)\rangle\rangle$
$\tau(\Psi), \tau(\Psi'), \ldots = \langle\langle n, e\rangle, e\rangle$

b. Constants of L_1:
all constants of L_0
$\tau(1), \tau(2), \ldots = n$
$\tau(A) =$
$= \langle \tau(\Psi), \tau(h)\rangle$
$= \langle \tau(\Psi), \langle n, \langle \tau(c), \tau(c)\rangle\rangle\rangle$
$= \langle \tau(\Psi), \langle n, \langle \tau(c), \langle \tau(M), \langle \tau(g), \langle n, e\rangle\rangle\rangle\rangle\rangle\rangle$
$\tau(F) =$
$= \langle \tau(c), \tau(c)\rangle$
$= \langle \tau(c), \langle \tau(M), \langle \tau(g), \langle n, e\rangle\rangle\rangle\rangle$

c. Metavariables:
$\tau(i), \tau(j) = n$

The final versions of the translations are:

(63) **Translation of atomic formulas:**
$\mathbf{T}(P(t_1, \ldots, t_n)) = \lambda c \lambda g P(t'_1, \ldots, t'_n)$,
where for all i with $1 \leq i \leq n$

$$t_i = \begin{cases} t_i, \text{if } t_i \text{ is a constant} \\ c(\varnothing)(g)(j), \text{if } t_i = x_j \text{ for some integer } j \end{cases}$$

(64) **Translation of quantification:**
$\mathbf{T}(\forall x_i \alpha) = \lambda c \lambda g \forall y_i [T(\alpha)(\lambda M.F(c)(M+i))(g[i/y_i])]$, where

$$F(c)(M)(g)(j) = \begin{cases} g(j), & \text{if } j \in M \\ c(M)(g)(j), & \text{else} \end{cases}$$

(65) **Translation of abstraction:**
$$\mathbf{T}(\lambda x_i \alpha) = \begin{cases} \lambda h \lambda c[\mathbf{T}(\alpha)(h(i)(c))], & \text{if } \alpha \text{ has type } t \\ \lambda h \lambda h'[\mathbf{T}(\alpha)(\lambda u \lambda v h(i)(h'(u)(v)))], & \text{else} \end{cases}$$

(66) **Translation of functional application:**
$\mathbf{T}(\alpha(x_i)) = \mathbf{T}(\alpha)(A(\mathbf{T}(x_i)))$, where:

$$A(\psi)(i)(c)(M)(g)(j) = \begin{cases} \psi(F(c)(M)(g)), & \text{if } i = j \\ c(M)(g)(j), & \text{else} \end{cases}$$

Note that constants like F and A in (66) are logical constants without descriptive content. They can equally well be eliminated and replaced be definite descriptions. For example, F can be defined as

(67) $(\iota f)(\forall c \forall M \forall g \forall j((j \in M \rightarrow f(c)(M)(g)(j) = g(j)) \wedge (j \notin M \rightarrow f(c)(M)(g)(j) = c(M)(g)(j)))$.

4 A formal proof of unrestricted semantic reconstruction

The translation function **T** is defined as follows:

(68) **Definition (translation function T):**

a. If $\alpha = P(t_1, \ldots, t_n)$ is an atomic L_0-formula (with P an n-ary relation symbol, and $t_1, \ldots, t_n$ terms[9], then
$\mathbf{T}(\alpha) = \lambda c \lambda g P(t'_1, \ldots, t'_n)$, where for all t'_i with $1 \leq i \leq n$:
$$t'_i = \begin{cases} c(\emptyset)(g)(k), & \text{if } t_i = x_k \\ t_i, & \text{if } t_i \text{ is an individual constant} \end{cases}$$

b. If $\alpha = \neg\beta$ is an L_0-formula, then
$\mathbf{T}(\alpha) = \lambda c \lambda g \neg[\mathbf{T}(\beta)(c)(g)]$

c. If $\alpha = \beta \wedge \gamma$ is an L_0-formula (of type t), then
$\mathbf{T}(\alpha) = \lambda c \lambda g[\mathbf{T}(\beta)(c)(g) \wedge \mathbf{T}(\gamma)(c)(g)]$

d. If $\alpha = \forall x_i \beta$ is an L_0-formula (of type t), then
$\mathbf{T}(\forall x_i \alpha) = O^i_\forall \mathbf{T}(\alpha)$, with
$O^i_\forall \mathbf{T}(\alpha) = \lambda c \lambda g \forall y_i [\mathbf{T}(\alpha)(\lambda M.F(c)(M+i))(g[i/y_i])]$ with
$F(c)(N)(g)(j) = g(j)$ if $j \in N$ and $c(N)(g)(j)$ else.

[9] Variables and individual constants are terms.

e. If $\alpha = \lambda x_i \beta$ is an L_0-formula, $\mathbf{T}(\beta) = \lambda h_{a_1} \ldots \lambda h_{a_n} \lambda c \lambda g \beta'$, $a_i > 0$ for all $i, 1 \leq i \leq n$, and u, v integers different from $a_1, \ldots, a_n$, then
$\mathbf{T}(\alpha) = O^i_\lambda \mathbf{T}(\alpha)$, with
$$O^i_\lambda \mathbf{T}(\alpha) = \begin{cases} \lambda h_u \lambda c[\mathbf{T}(\beta)(h_u(i)(c))], & \text{if } \beta \text{ is of type } t \\ \lambda h_u \lambda h_v \mathbf{T}(\beta)(\lambda j \lambda f . h_u(i)(h_v(j)(f))), & \text{else} \end{cases}$$

f. If $\alpha = \beta(x_i)$ is an L_0-formula, then
$\mathbf{T}(\alpha) = O^i_A \mathbf{T}(\alpha) = \mathbf{T}(\beta)(A(\mathbf{T}(x_i)))$, where:
$$A(\psi)(i)(c)(M)(g)(j) = \begin{cases} \psi(F(c)(M)(g)), & \text{if } i = j \\ c(M)(g)(j), & \text{else} \end{cases}$$

(69) **Definition (unrestricted substitution** $[x//y]\alpha$**)**:

a. If $\alpha = P(t_1, \ldots, t_n)$ is an atomic L_0-formula (with P an n-ary relation symbol, and $t_1, \ldots, t_n$ terms, then
$[x//y]P(t_1, \ldots, t_n) = P(t'_1, \ldots, t'_n)$, where for all t'_i with $1 \leq i \leq n$:
$$t'_i = \begin{cases} y, & \text{if } t_i = x \\ t_i, & \text{else} \end{cases}$$
b. $[x//y]\neg\alpha = \neg[x//y]\alpha$
c. $[x//y](\alpha \wedge \beta) = [x//y]\alpha \wedge [x//y]\beta$
d. $$[x//y]\forall x_i \alpha = \begin{cases} \forall x_i \alpha, & \text{if } x = x_i \\ \forall x_i [x//y]\alpha, & \text{else} \end{cases}$$
e. $$[x//y]\lambda x_i \alpha = \begin{cases} \lambda x_i \alpha, & \text{if } x = x_i \\ \lambda x_i [x//y]\alpha, & \text{else} \end{cases}$$
f. $[x//y](\alpha(z)) = [x//y]\alpha([x//y]z)$

(70) **Definition (unrestricted reduction r)**:

a. if α is atomic L_0-formula,
then $\mathbf{r}(\alpha) = \alpha$
b. if $\alpha = \neg\alpha'$ is L_0-formula,
then $\mathbf{r}(\alpha) = \neg\mathbf{r}(\alpha')$
c. if $\alpha = \beta \wedge \gamma$ is L_0-formula,
then $\mathbf{r}(\alpha) = \mathbf{r}(\beta) \wedge \mathbf{r}(\gamma)$
d. $\mathbf{r}(\forall x_i \alpha) = \forall x_i \mathbf{r}(\alpha)$
e. $\mathbf{r}(\lambda x_i \alpha) = \lambda x_i \mathbf{r}(\alpha)$
f. $\mathbf{r}(\lambda x_i \alpha(x_z)) = [x_i//x_z]\mathbf{r}(\alpha)$

Examples:

(71) $\mathbf{r}(\lambda x_3(\lambda x_3(P(x_3))(x_5))(x_4)) =$
= (70-f)
$[x_3//x_4]\mathbf{r}(\lambda x_3(P(x_3))(x_5))$
= (70-f)
$[x_3//x_4][x_3//x_5]\mathbf{r}(P(x_3))$
= (70-a)
$[x_3//x_4][x_3//x_5]P(x_3)$
= (definition of substitution)

$[x_3//x_4]P(x_5)$
= (definition of substitution)
$P(x_5)$

(72) $\mathbf{r}(\lambda x_2 \lambda x_3 \forall x_2 P(x_2, x_3)(x_2)(x_3))$
= (definition of **r**, clause f)
$[x_2//x_3]\mathbf{r}(\lambda x_3 \forall x_2 P(x_2, x_3)(x_2))$
= (definition of **r**, clause f)
$[x_2//x_3][x_3//x_2]\mathbf{r}(\forall x_2 P(x_2, x_3))$
= (definition of **r**, clause d)
$[x_2//x_3][x_3//x_2]\forall x_2 \mathbf{r}(P(x_2, x_3))$
= (definition of **r**, clause a)
$[x_2//x_3][x_3//x_2]\forall x_2 P(x_2, x_3)$
= (definition of substitution)
$[x_2//x_3]\forall x_2 [x_3//x_2]P(x_2, x_3)$
= (definition of substitution)
$[x_2//x_3]\forall x_2 P(x_2, x_2)$
= (definition of substitution)
$\forall x_2 P(x_2, x_2)$

(73) **Lemma (reduction)**:
Let R be the smallest set of formulas of L_0 such that:

a. if α is an atomic formula of L_0, then $\alpha \in R$
b. if $\alpha \in R$, then $\neg\alpha \in R$
c. if $\alpha \in R$ and $\beta \in R$, then $\alpha \wedge \beta \in R$
d. if $\alpha \in R$, then $\lambda x_i \alpha \in R$ (for any x_i)
e. if $\alpha \in R$, then $\forall x_i \alpha \in R$ (for any x_i)

Then for all $\alpha \in L_0$ it holds that $\mathbf{r}(\alpha) \in R$.

(74) Proof of reduction lemma: by induction on the structure of α.

a. Base case: if α is an atomic formula, then $\mathbf{r}(\alpha) = \alpha$ (by definition of **r**), and therefore $\alpha \in R$ (by definition of the set R)
b. Let $\mathbf{r}(\alpha) \in R$. Then $\mathbf{r}(\neg\alpha) = \neg\mathbf{r}(\alpha)$ (definition **r**), and since by ind. hypothesis $\mathbf{r}(\alpha) \in R$ we also have by definition of R that $\neg\mathbf{r}(\alpha) \in R$.
c. Let $\mathbf{r}(\alpha) \in R$ and $\mathbf{r}(\beta) \in R$. Then by definition of **r** we have $\mathbf{r}(\alpha \wedge \beta) = \mathbf{r}(\alpha) \wedge \mathbf{r}(\beta)$, and since by ind. hypothesis $\mathbf{r}(\alpha) \in R$ and $\mathbf{r}(\beta) \in R$ we also have by definition of R that $\mathbf{r}(\alpha) \wedge \mathbf{r}(\beta) \in R$.
d. Let $\mathbf{r}(\alpha) \in R$. Then $\mathbf{r}(\lambda x_i \alpha) = \lambda x_i \mathbf{r}(\alpha)$ (definition **r**), and since by ind. hypothesis $\mathbf{r}(\alpha) \in R$ we also have by definition of R that $\lambda x_i \mathbf{r}(\alpha) \in R$.
e. Let $\mathbf{r}(\alpha) \in R$. Then $\mathbf{r}(\forall x_i \alpha) = \forall x_i \mathbf{r}(\alpha)$ (definition **r**), and since by ind. hypothesis $\mathbf{r}(\alpha) \in R$ we also have by definition of R that $\forall x_i \mathbf{r}(\alpha) \in R$.
f. Let $\mathbf{r}(\lambda x_i \alpha) \in R$. Then by definition of **r** we have $\lambda x_i \mathbf{r}(\alpha) \in R$. Therefore, $\mathbf{r}(\alpha) \in R$. Since substitution does not change membership in R, it follows further that for any x, y: $[x//y]\mathbf{r}(\alpha) \in R$, and so also for x_i, t, showing that $[x_i//t]\mathbf{r}(\alpha) \in R$, and by definition of **r** we have $[\mathbf{r}(\lambda x_i \alpha(t)) \in R$.

What this essentially says is that every reduced formula $\mathbf{r}(\alpha)$ is built from atomic formulas using negation, conjunction, quantification and abstraction (but no application). This is important in the proof of the next theorem.

We now turn to the formulation of the central theorem, showing that the translation $\mathbf{T}(\alpha)$ of an arbitrary L_0-formula α is beta-equivalent to the translation $\mathbf{T}(\mathbf{r}(\alpha))$ of the reduced formula $\mathbf{r}(\alpha)$:

(75) **Theorem**:
Let α be an arbitrary L_0-formula. Then: $\mathbf{T}(\alpha) \equiv_\beta \mathbf{T}(\mathbf{r}(\alpha))$

(76) Proof: by induction on the structure of α.

a. Base case:
Let α be atomic formula. Then $\mathbf{r}(\alpha) = \alpha$, and therefore $\mathbf{T}(\alpha) = \mathbf{T}(\mathbf{r}(\alpha))$.

b. Negation: Assume that $\mathbf{T}(\alpha) \equiv_\beta \mathbf{T}(\mathbf{r}(\alpha))$. We show that $\mathbf{T}(\neg\alpha) \equiv_\beta \mathbf{T}(\mathbf{r}(\neg\alpha))$:
$\mathbf{T}(\neg\alpha)$
= (definition **T**, negation)
$\lambda c \lambda g \neg[\mathbf{T}(\alpha)(c)(g)]$
= (ind. hypothesis)
$\lambda c \lambda g \neg[\mathbf{T}(\mathbf{r}(\alpha))(c)(g)]$
= (definition **T**)
$\mathbf{T}(\neg\mathbf{r}(\alpha))$
= (definition **r**)
$\mathbf{T}(\mathbf{r}(\neg\alpha))$

c. Conjunction: Assume that $\mathbf{T}(\alpha) = \mathbf{T}(\mathbf{r}(\alpha))$ and that $\mathbf{T}(\beta) = \mathbf{T}(\mathbf{r}(\beta))$. We show that $\mathbf{T}(\alpha \wedge \beta) = \mathbf{T}(\mathbf{r}(\alpha \wedge \beta))$:
$\mathbf{T}(\alpha \wedge \beta)$
= (definition of **T**)
$\lambda c \lambda g[\mathbf{T}(\alpha)(c)(g) \wedge \mathbf{T}(\beta)(c)(g)]$
= (ind. hypothesis)
$\lambda c \lambda g[\mathbf{T}(\mathbf{r}(\alpha))(c)(g) \wedge \mathbf{T}(\mathbf{r}(\beta))(c)(g)]$
= (definition of **T**)
$\mathbf{T}(\mathbf{r}(\alpha) \wedge \mathbf{r}(\beta))$
= (definition of **r**)
$\mathbf{T}(\mathbf{r}(\alpha \wedge \beta))$

d. Quantification:
Let α be such that $\mathbf{T}(\alpha) = \mathbf{T}(\mathbf{r}(\alpha))$. Let x_i be an arbitrary variable. We show that:
$\mathbf{T}(\forall x_i \alpha) \equiv_\beta \mathbf{T}(\mathbf{r}(\forall x_i \alpha))$
$\mathbf{T}(\forall x_i \alpha)$
= (definition of **T**)
$\lambda c \lambda g \forall y_i[\mathbf{T}(\alpha)(\lambda M.F(c)(M + i))(g[i/y_i])]$
= (induction hypothesis)
$\lambda c \lambda g \forall y_i[\mathbf{T}(\mathbf{r}(\alpha))(\lambda M.F(c)(M + i))(g[i/y_i])]$
= (definition of **T**)
$\mathbf{T}(\forall x_i \mathbf{r}(\alpha))$

= (definition of **r**)
$\mathbf{T}(\mathbf{r}(\forall x_i \alpha))$

e. Abstraction: Assume that $\mathbf{T}(\alpha) = \mathbf{T}(\mathbf{r}(\alpha))$. We show that $\mathbf{T}(\lambda x_i \alpha) = \mathbf{T}(\mathbf{r}(\lambda x_i \alpha))$:
First case: α is of type t:
$\mathbf{T}(\lambda x_i \alpha)$
= (definition **T**)
$\lambda h_u \lambda c[\mathbf{T}(\alpha)(h_u(i)(c))]$
= (ind. hypothesis)
$\lambda h_u \lambda c[\mathbf{T}(\mathbf{r}(\alpha))(h_u(i)(c))]$
= (definition of **T**)
$\mathbf{T}(\lambda x_i \mathbf{r}(\alpha))$
= (definition of **r**)
$\mathbf{T}(\mathbf{r}(\lambda x_i \alpha))$

Second case: α is a λ-term:
$\mathbf{T}(\lambda x_i \alpha)$
= (definition **T**)
$\lambda h_u \lambda h_v \mathbf{T}(\alpha)(\lambda j \lambda f.h_u(i)(h_v(j)(f)))$
= (induc. hypothesis)
$\lambda h_u \lambda h_v \mathbf{T}(\mathbf{r}(\alpha))(\lambda j \lambda f.h_u(i)(h_v(j)(f)))$
= (definition **T**)
= $\mathbf{T}(\lambda x_i \mathbf{r}(\alpha))$
= (definition **r**)
$\mathbf{T}(\mathbf{r}(\lambda x_i \alpha))$

f. Application: Assume that $\mathbf{T}(\lambda x_i \alpha) = \mathbf{T}(\mathbf{r}(\lambda x_i \alpha))$, for arbitrary x_i and α. We show that for arbitrary x_z it holds that $\mathbf{T}(\lambda x_i \alpha(x_z)) = \mathbf{T}(\mathbf{r}(\lambda x_i \alpha(x_z)))$
$\mathbf{T}(\lambda x_i \alpha(x_z))$
= (definition **T**)
$\mathbf{T}(\lambda x_i \alpha)(A(\mathbf{T}(x_z)))$
= (ind. hypothesis)
$\mathbf{T}(\mathbf{r}(\lambda x_i \alpha))(A(\mathbf{T}(x_z)))$
= (definition **T**)
$\mathbf{T}(\mathbf{r}(\lambda x_i \alpha)(x_z))$
= (definition **r**)
$\mathbf{T}(\lambda x_i \mathbf{r}(\alpha)(x_z))$
= (lemma (77))
$\mathbf{T}([x_i//x_z]\mathbf{r}(\alpha))$
= (definition of **r**)
$\mathbf{T}(\mathbf{r}(\lambda x_i \alpha(x_z)))$

(77) **Lemma**:
For arbitrary x_i, x_z, α it holds that $\mathbf{T}(\lambda x_i \mathbf{r}(\alpha)(x_z)) = \mathbf{T}([x_i//x_z]\mathbf{r}(\alpha))$

(78) Proof: By the reduction lemma (73) we know that $\mathbf{r}(\alpha) \in R$, i.e. it is built by applying negation, conjunction, quantification and abstraction (but not application) to atomic formulas. It therefore suffices for the proof of this lemma to

show that (i) it holds for atomic α, and (ii) that if it holds for α and β, it also holds for the negation, conjunction, quantification and abstraction of α and β. This then guarantees that the lemma holds for $\mathbf{r}(\alpha)$ for arbitrary $\alpha \in L_0$.

a. Base case: we show that $\mathbf{T}(\lambda x_i \mathbf{r}(\alpha)(x_z)) = \mathbf{T}([x_i//x_z]\mathbf{r}(\alpha))$ for atomic α.
Let $\alpha = P(t_1, \ldots, t_n)$, without loss of generalization. Then by the definition of $\mathbf{T}$ we have:
$\mathbf{T}(\alpha) = \lambda c \lambda g P(t'_1, \ldots, t'_n)$, where for all $1 \leq j \leq n$:

$$t'_j = \begin{cases} t_j, & \text{if } t_j \text{ is a constant} \\ c(\varnothing)(g)(k), & \text{if } t_j = x_k \end{cases}$$

Then:
$\mathbf{T}(\lambda x_i \mathbf{r}(\alpha)(x_z))$
= (definition of $\mathbf{r}$)
$\mathbf{T}(\lambda x_i \alpha(x_z))$
= (definition of $\mathbf{T}$)
$\mathbf{T}(\lambda x_i \alpha)(A(\mathbf{T}(x_z)))$
= (definition of $\mathbf{T}$)
$\lambda h \lambda c[\mathbf{T}(\alpha)(h(i)(c))](A(\mathbf{T}(x_z)))$
= (α is an atomic formula, no loss of generalization)
$\lambda h \lambda c[\lambda c \lambda g P(t'_1, \ldots, t'_n)(h(i)(c))](A(\mathbf{T}(x_z)))$
= (conversion of h)
$\lambda c[\lambda c \lambda g P(t'_1, \ldots, t'_n)(A(\mathbf{T}(x_z))(i)(c))]$
= (conversion of inner c)
$\lambda c[\lambda g P(t''_1, \ldots, t''_n)]$, where for all $1 \leq j \leq n$:

$$t''_j = \begin{cases} t_j, & \text{if } t_j \text{ is an ind. constant} \\ A(\mathbf{T}(x_z)(i)(c)(\varnothing)(g)(k)), & \text{if } t_j = x_k \end{cases}$$

$$= \begin{cases} t_j, & \text{if } t_j \text{ is an ind. constant} \\ F(c)(\varnothing)(g)(z) = c(\varnothing)(g)(z), & \text{if } t_j = x_k \wedge k = i \\ c(\varnothing)(g)(k), & \text{if } t_j = x_k \wedge k \neq i \end{cases}$$

To conclude the proof of this clause, we now show that $\mathbf{T}([x_i//x_z]\mathbf{r}(\alpha)) = \lambda c[\lambda g P(t''_1, \ldots, t''_n)]$, where for all $1 \leq j \leq n$:

$$t''_j = \begin{cases} t_j, & \text{if } t'_j \text{ is an ind. constant} \\ c(\varnothing)(g)(z), & \text{if } t'_j = c(\varnothing)(g)(k) \text{ and } k = i \\ c(\varnothing)(g)(k), & \text{if } t'_j = c(\varnothing)(g)(k) \text{ and } k \neq i \end{cases}$$

To begin with, $\mathbf{r}(\alpha) = \alpha$ since α is atomic, so $\mathbf{r}(\alpha) = P(t_1, \ldots, t_n)$. By the definition of substitution we have $[x_i//x_z]\mathbf{r}(\alpha) = P(q_1, \ldots, q_n)$, where for all $1 \leq j \leq n$:

$$q_j = \begin{cases} t_j, & \text{if } t_j \text{ is a constant} \\ x_z, & \text{if } t_j = x_k \wedge k = i \\ x_k, & \text{if } t_j = x_k \wedge k \neq i \end{cases}$$

Then by the definition of $\mathbf{T}$ we have $\mathbf{T}([x_i//x_z]\mathbf{r}(\alpha)) = \lambda c\lambda g P(q'_1,\ldots,q'_n)$, where:

$$q'_j = \begin{cases} t_j, & \text{if } t'_j \text{ is an ind. constant} \\ c(\varnothing)(g)(z), & \text{if } q_j = x_k \text{ and } k = i \\ c(\varnothing)(g)(k), & \text{if } q_j = x_k \text{ and } k \neq i \end{cases}$$

showing that for every $j, 1 \leq j \leq n$ we have $t''_j = q'_j$.

b. Negation: analogous to conjunction

c. Conjunction:
Assume $\mathbf{T}(\lambda x_i \alpha(x_z)) = \mathbf{T}([x_i//x_z]\alpha)$ and $\mathbf{T}(\lambda x_i \beta(x_z)) = \mathbf{T}([x_i//x_z]\beta)$. We show that $\mathbf{T}(\lambda x_i(\alpha \wedge \beta)(x_z)) = \mathbf{T}([x_i//x_z](\alpha \wedge \beta))$.
$\mathbf{T}(\lambda x_i(\alpha \wedge \beta)(x_z))$
= (translation of application)
$\mathbf{T}(\lambda x_i(\alpha \wedge \beta))(A(\mathbf{T}(x_z))))$
= (translation of abstraction)
$\lambda h\lambda c[\mathbf{T}(\alpha \wedge \beta)(h(i)(c))](A(\mathbf{T}(x_z))))$
= (translation of conjunction)
$\lambda h\lambda c[\lambda c\lambda g[\mathbf{T}(\alpha)(c)(g) \wedge \mathbf{T}(\beta)(c)(g)](h(i)(c))](A(\mathbf{T}(x_z))))$
= (conversion of inner c)
$\lambda h\lambda c\lambda g[\mathbf{T}(\alpha)(h(i)(c))(g) \wedge \mathbf{T}(\beta)(h(i)(c))(g)](A(\mathbf{T}(x_z))))$
= (conversion of h)
$\lambda c\lambda g[\mathbf{T}(\alpha)(A(\mathbf{T}(x_z)))(i)(c))(g) \wedge \mathbf{T}(\beta)(A(\mathbf{T}(x_z)))(i)(c))(g)]$
= (abstraction over c and application to c in both conjuncts)
$\lambda c\lambda g[\lambda c[\mathbf{T}(\alpha)(A(\mathbf{T}(x_z)))(i)(c))](c)(g) \wedge \lambda c[\mathbf{T}(\beta)(A(\mathbf{T}(x_z)))(i)(c))](c)(g)]$
= (conversion of h below)
$\lambda c\lambda g[\lambda h\lambda c[\mathbf{T}(\alpha)(h(i)(c))](A(\mathbf{T}(x_z)))(c)(g) \wedge \lambda h\lambda c[\mathbf{T}(\beta)(h(i)(c))](A(\mathbf{T}(x_z)))(c)(g)]$
= (definition of $\mathbf{T}$)
$\lambda c\lambda g[\mathbf{T}(\lambda x_i \alpha(x_z))(c)(g) \wedge \mathbf{T}(\lambda x_i \beta(x_z))(c)(g)]$
= (induction hypothesis)
$\lambda c\lambda g[\mathbf{T}([x_i//x_z]\alpha)(c)(g) \wedge \mathbf{T}([x_i//x_z]\beta)(c)(g)]$
= (definition $\mathbf{T}$)
$\mathbf{T}([x_i//x_z]\alpha \wedge [x_i//x_z]\beta)$
= (definition of $\mathbf{r}$)
$\mathbf{T}([x_i//x_z](\alpha \wedge \beta))$

d. Quantification:
Assume that $\mathbf{T}(\lambda x_i \alpha(x_z)) = \mathbf{T}([x_i//x_z]\alpha)$ for arbitrary integers i, z and for an arbitrary type t formula $\alpha \in R$. We have to show that $\mathbf{T}(\lambda x_i \forall x_j \alpha(x_z)) = \mathbf{T}([x_i//x_z]\forall x_j \alpha)$.
We distinguish two cases, depending on whether or not $i = j$.
First case: $i \neq j$.
$\mathbf{T}(\lambda x_i \forall x_j \alpha(x_z))$
= (O-notation)
$O^z_A O^i_\lambda O^j_\forall \mathbf{T}(\alpha)$

= (see below)
$O^j_\forall O^z_A O^i_\lambda \mathbf{T}(\alpha)$
= (O notation)
$O^j_\forall \mathbf{T}(\lambda x_i \alpha(x_z))$
= (induction hypothesis)
$O^j_\forall \mathbf{T}([x_i//x_z]\alpha)$
= (O-notation)
$\mathbf{T}(\forall x_j [x_i//x_z]\alpha)$
= (by definition of substitution, since $i \neq j$)
$\mathbf{T}([x_i//x_z]\forall x_j \alpha)$

What remains to be shown for this case is that

$$O^z_A O^i_\lambda O^j_\forall \mathbf{T}(\alpha) \equiv O^j_\forall O^z_A O^i_\lambda \mathbf{T}(\alpha)$$

Since α is of type t, its terms are either constants, variables bound by some universal quantifier, or free variables. Note that since α is a reduced formula type t, its variables cannot be bound by λ.
Assume that $\mathbf{T}(\alpha) \equiv \lambda c \lambda g \beta$. Then:
$O^j_\forall \mathbf{T}(\alpha)$
=
$\lambda c' \lambda g' \forall y_j [\mathbf{T}(\alpha)(\lambda M.F(c)(M+j))(g'[j/y_j])]$
=
$\lambda c' \lambda g' \forall y_j [\lambda c \lambda g \beta(\lambda M.F(c)(M+j))(g'[j/y_j])]$
= (conversion of c)
$\lambda c' \lambda g' \forall y_j [\lambda g \beta[c/(\lambda M.F(c)(M+j))](g'[j/y_j])]$
= (conversion of g)
$\lambda c' \lambda g' \forall y_j [\beta[c/(\lambda M.F(c)(M+j))][g/(g'[j/y_j])]]$
Next we apply O^i_λ:
$O^i_\lambda(\lambda c' \lambda g' \forall y_j [\beta[c/(\lambda M.F(c)(M+j))][g/(g'[j/y_j])]])$
=
$\lambda h'' \lambda c'' [\lambda c' \lambda g' \forall y_j [\beta[c/(\lambda M.F(c)(M+j))][g/(g'[j/y_j])]](h''(i)(c''))]$
= (conversion of c')
$\lambda h'' \lambda c'' [\lambda g' \forall y_j [\beta[c/(\lambda M.F(c)(M+j))][g/(g'[j/y_j])]][c'/(h''(i)(c''))]]$
Finally we apply O^z_A:
$O^z_A(\lambda h'' \lambda c'' [\lambda g' \forall y_j [\beta[c/(\lambda M.F(c)(M+j))][g/(g'[j/y_j])]][c'/(h''(i)(c''))]])$
=
$\lambda h'' \lambda c'' [\lambda g' \forall y_j [\beta[c/(\lambda M.F(c)(M+j))][g/(g'[j/y_j])]][c'/(h''(i)(c''))]](A(\mathbf{T}(x_z)))$
= (conversion of h'')
$\lambda c'' \lambda g' \forall y_j \beta[c/(\lambda M.F(c)(M+j))][g/(g'[j/y_j])][c'/(h''(i)(c''))][h''/(A(\mathbf{T}(x_z)))]$
Let t be a term of $\mathbf{T}(\alpha)$. Then the corresponding term of $O^z_A O^i_\lambda O^j_\forall \mathbf{T}(\alpha)$ is

$$t[c/(\lambda M.F(c)(M+j))][g/(g'[j/y_j])][c'/(h''(i)(c''))][h''/(A(\mathbf{T}(x_z)))]$$

If t is a constant, then:

$$t[c/(\lambda M.F(c)(M+j))][g/(g'[j/y_j])][c'/(h''(i)(c''))][h''/(A(\mathbf{T}(x_z)))] = t$$

If $t = y_k$ for some k (this is the case if the corresponding term in α is bound by $\forall x_k$), then:

$$t[c/(\lambda M.F(c)(M+j))][g/(g'[j/y_j])][c'/(h''(i)(c''))][h''/(A(\mathbf{T}(x_z)))] = y_k$$

If $t = c(M)(g[\ldots](k))$ for some M, k such that $k \notin M$ (this is the case if the corresponding term of α is a free variable), then:
$c(M)(g[\ldots])(k)[c/(\lambda M.F(c')(M+j))][g/(g'[j/y_j])][c'/(h''(i)(c''))][h''/(A(\mathbf{T}(x_z)))]$
= (substitution of c)
$F(c')(M+j)(g[\ldots])(k)[g/(g'[j/y_j])][c'/(h''(i)(c''))][h''/(A(\mathbf{T}(x_z)))]$
= (substitution of g)
$F(c')(M+j)(g'[j/y_j][\ldots])(k)[c'/(h''(i)(c''))][h''/(A(\mathbf{T}(x_z)))]$
= (substitution of c')
$F(h''(i)(c''))(M+j)(g'[j/y_j][\ldots])(k)[h''/(A(\mathbf{T}(x_z)))]$
= (substitution of h'')
$F(A(\mathbf{T}(x_z))(i)(c''))(M+j)(g'[j/y_j][\ldots])(k) =$

$$= \begin{cases} A(\mathbf{T}(x_z))(i)(c'')(M+j)(g'[j/y_j][\ldots])(k), & \text{if } k \notin M+j \\ y_k, & \text{if } k \in M+j \end{cases}$$

=

$$\begin{cases} \mathbf{T}(x_z)(F(c'')(M+j)(g'[j/y_j][\ldots])), & \text{if } k = i \wedge k \notin M+j \\ c''(M+j)(g'[j/y_j][\ldots])(k), & \text{if } k \neq i \wedge k \notin M+j \\ y_k, & \text{if } k \in M+j \end{cases}$$

=

$$\begin{cases} F(c'')(M+j)(g'[j/y_j][\ldots])(z), & \text{if } k = i \wedge k \notin M+j \\ c''(M+j)(g'[j/y_j][\ldots])(k), & \text{if } k \neq i \wedge k \notin M+j \\ y_k, & \text{if } k \in M+j \end{cases}$$

=

$$\begin{cases} y_z, & \text{if } z \in M+j \wedge k = i \wedge k \notin M+j \\ c''(M+j)(g'[j/y_j][\ldots])(z), & \text{if } z \notin M+j \wedge k = i \wedge k \notin M+j \\ c''(M+j)(g'[j/y_j][\ldots])(k), & \text{if } k \neq i \wedge k \notin M+j \\ y_k, & \text{if } k \neq i \wedge k \in M+j \end{cases}$$

= (if $k = i$ and $i \neq j$ it follows that $k \neq j$; and since also $k \notin M$ it follows that $k \notin M+j$, so this cond. can be omitted)

$$\begin{cases} y_z, & \text{if } k = i \wedge z \in M+j \\ c''(M+j)(g'[j/y_j][\ldots])(z), & \text{if } k = i \wedge z \notin M+j \\ y_k, & \text{if } k \neq i \wedge k \in M+j \\ c''(M+j)(g'[j/y_j][\ldots])(k), & \text{if } k \neq i \wedge k \notin M+j \end{cases}$$

Analogously we can show that:

$$\lambda c''\lambda g'\forall y_j\beta[c/(h'(i)(c'))][h'/(A(\mathbf{T}(x_z)))][c'/(\lambda M.F(c'')(M+j))][g/(g'[j/y_j])]$$

so that if $t = c(M)(g[\ldots](k))$ for some M, k such that $k \notin Mt$ is a term of $\mathbf{T}(\alpha)$, then the corresponding term in $O^j_\forall O^z_A O^i_\lambda \mathbf{T}(\alpha)$ is

$c(M)(g[\ldots])(k)[c/(h'(i)(c'))][h'/(A(\mathbf{T}(x_z)))][c'/(\lambda M.F(c'')(M+j))][g/(g'[j/y_j])]$
= (conversion of c)
$h'(i)(c')(M)(g[\ldots])(k)[h'/(A(\mathbf{T}(x_z)))][c'/(\lambda M.F(c'')(M+j))][g/(g'[j/y_j])]$
= (conversion of h')
$A(\mathbf{T}(x_z))(i)(c')(M)(g[\ldots])(k)[c'/(\lambda M.F(c'')(M+j))][g/(g'[j/y_j])]$
= (conversion of c')
$A(\mathbf{T}(x_z))(i)(\lambda M.F(c'')(M+j))(M)(g[\ldots])(k)[g/(g'[j/y_j])]$
= (conversion of g)
$A(\mathbf{T}(x_z))(i)(\lambda M.F(c'')(M+j))(M)(g'[j/y_j][\ldots])(k)$

$$= \begin{cases} \mathbf{T}(x_z)(F(\lambda M.F(c'')(M+j))(M)(g'[j/y_j][\ldots])) =, & \text{if } k = i \\ F(\lambda M.F(c'')(M+j))(M)(g'[j/y_j][\ldots])(z) & \\ F(c'')(M+j)(g'[j/y_j][\ldots])(k), & \text{if } k \neq i \end{cases}$$

$$= \begin{cases} y_z, & \text{if } k = i \wedge z \in M+j \\ c''(M+j)(g'[j/y_j][\ldots])(z), & \text{if } k = i \wedge z \notin M+j \\ y_k, & \text{if } k \neq i \wedge k \in M+j \\ c''(M+j)(g'[j/y_j][\ldots])(k), & \text{if } k \neq i \wedge k \notin M+j \end{cases}$$

Second case: $i = j$. Analogous to first case

e. Abstraction:
Assume that $\mathbf{T}(\lambda x_i \alpha(x_z)) = \mathbf{T}([x_i//x_z]\alpha)$ for arbitrary integers i, z and for an arbitrary type t formula $\alpha \in R$. We have to show that $\mathbf{T}(\lambda x_i \lambda x_j \alpha(x_z)) = \mathbf{T}([x_i//x_z]\lambda x_j \alpha)$.
We distinguish two cases, depending on whether or not $i = j$.
First case: $i \neq j$.
$\mathbf{T}(\lambda x_i \lambda x_j \alpha(x_z))$
= (O-notation)
$O_A^z O_\lambda^i O_\lambda^j \mathbf{T}(\alpha)$
= (see below)
$O_\lambda^j O_A^z O_\lambda^i \mathbf{T}(\alpha)$
= (O notation)
$O_\lambda^j \mathbf{T}(\lambda x_i \alpha(x_z))$
= (induction hypothesis)
$O_\lambda^j \mathbf{T}([x_i//x_z]\alpha)$
= (O-notation)
$\mathbf{T}(\lambda x_j [x_i//x_z]\alpha)$
= (by definition of substitution, since $i \neq j$)
$\mathbf{T}([x_i//x_z]\lambda x_j \alpha)$

What remains to be shown for this case is that

$$O_A^z O_\lambda^i O_\lambda^j \mathbf{T}(\alpha) \equiv O_\lambda^j O_A^z O_\lambda^i \mathbf{T}(\alpha)$$

Since α is of type t, its terms are either constants, variables bound either by $\forall$ or λ, or free variables. We show that if t is a variable of α bound by λ, then the terms of $O_A^z O_\lambda^i O_\lambda^j \mathbf{T}(\alpha)$ and $O_\lambda^j O_A^z O_\lambda^i \mathbf{T}(\alpha)$ which correspond to t are

equivalent. The same can be shown if t is a constant, a free variable or bound by $\forall$.

Assume that the term t of α is bound by λx_k. Then the corresponding term of $\mathbf{T}(\alpha)$ is of the form $h_l(l)(\ldots)(M)(g[\ldots])(k)$, with $k \notin M$ (not bound by $\forall$). Note that the abstracted index k does not have to be the 'highest' index. (For example, the term of $\mathbf{T}(\lambda x_l \lambda x_k.P(x_k))$ corresponding to x_k is $h_l(l)(h_k(k)(c))(M)(g[\ldots])(k)$. The index l on h_l is chosen for ease of readability only.)

O_λ^j substitutes $h_l(l)(\ldots)$ with $h_j(j)(h_l(l)(\ldots))$, and thus modifies the term $h_l(l)(\ldots)(M)(g[\ldots])(k)$ to $h_j(j)(h_l(l)(\ldots))(M)(g[\ldots])(k)$.

O_λ^i substitutes $h_j(j)(h_l(l)(\ldots))$ with $h_i(i)(h_j(j)(h_l(l)(\ldots)))$, and thus modifies the term $h_j(j)(h_l(l)(\ldots))(M)(g[\ldots])(k)$ into $h_i(i)(h_j(j)(h_l(l)(\ldots)))(M)(g[\ldots])(k)$.

O_A^z substitutes h_i for $A(\mathbf{T}(x_z))$, and thus modifies the term $h_i(i)(h_j(j)(h_l(l)(\ldots)))(M)(g[\ldots])(k)$ into
$A(\mathbf{T}(x_z))(i)(h_j(j)(h_l(l)(\ldots)))(M)(g[\ldots])(k)$
=
$$\begin{cases} \mathbf{T}(x_z)(F(h_j(j)(h_l(l)(\ldots)))(M)(g[\ldots])), & \text{if } k = i \\ h_j(j)(h_l(l)(\ldots))(M)(g[\ldots])(k), & \text{if } k \neq i \end{cases}$$
= (definition of $\mathbf{T}$; conversion)
$$\begin{cases} F(h_j(j)(h_l(l)(\ldots)))(M)(g[\ldots]))(z), & \text{if } k = i \\ h_j(j)(h_l(l)(\ldots))(M)(g[\ldots])(k), & \text{if } k \neq i \end{cases}$$
= (definition of F)
$$\begin{cases} y_z, & \text{if } k = i \wedge z \in M \\ h_j(j)(h_l(l)(\ldots))(M)(g[\ldots]))(z), & \text{if } k = i \wedge z \notin M \\ h_j(j)(h_l(l)(\ldots))(M)(g[\ldots])(k), & \text{if } k \neq i \end{cases}$$

We now show that the corresponding term in $O_\lambda^j O_A^z O_\lambda^i \mathbf{T}(\alpha)$ is equivalent.

First, O_λ^i modifies the term $h_l(l)(\ldots)(M)(g[\ldots])(k)$ into $h_i(i)(h_l(l)(\ldots))(M)(g[\ldots])(k)$

Next, O_A^z modifies the term $h_i(i)(h_l(l)(\ldots))(M)(g[\ldots])(k)$ into $A(\mathbf{T}(x_z))(i)(h_l(l)(\ldots))(M)(g[\ldots])(k)$
= (definition of A; $\mathbf{T}(x_z)$; conversion)
$$\begin{cases} F(h_l(l)(\ldots))(M)(g[\ldots])(z), & \text{if } k = i \\ h_l(l)(\ldots))(M)(g[\ldots])(k), & \text{if } k \neq i \end{cases}$$
= (definition of F)
$$\begin{cases} y_z, & \text{if } k = i \wedge z \in M \\ h_l(l)(\ldots)(M)(g[\ldots])(z), & \text{if } k = i \wedge z \notin M \\ h_l(l)(\ldots)(M)(g[\ldots])(k), & \text{if } k \neq i \end{cases}$$

Finally, O_λ^j modifies this term by substituting $h_l(l)(\ldots)$ with $h_j(h_l(l)(\ldots))$, yielding the terms:
$$\begin{cases} y_z, & \text{if } k = i \wedge z \in M \\ h_j(h_l(l)(\ldots))(M)(g[\ldots])(z), & \text{if } k = i \wedge z \notin M \\ h_j(h_l(l)(\ldots))(M)(g[\ldots])(k), & \text{if } k \neq i \end{cases}$$

Second case: $i = j$: analogous to first case.

5 Conclusion

Semantic reconstruction via β-reduction inherits (from the definition of β-reduction of the λ-calculus) the restriction that a term t can be substituted for a variable x only if t contains no variable that would become bound as a result of substitution. Given that the alternative approach via syntactic reconstruction is not without its own problems, we conclude that it is desirable to somehow overcome this restriction on semantic reconstruction, in order to allow for semantic reconstruction even in cases where a bound pronoun occurs outside the scope of its binder, e.g. when it is part of a topicalized noun phrase (a phenomenon we dubbed delayed quantification).

In this paper we proposed a way of doing so by translating each expression α of the language L_0 of predicate logic (with λ-abstraction) into an expression $\mathbf{T}(\alpha)$ of a new language L_1. Crucially, the translation $\mathbf{T}$ is set up such that the formulas $\mathbf{T}(\alpha)$ contain no free variables. In particular, a variable x_i of L_0 is translated as $\lambda g.g(i)$ with g a function from entities of type n (i.e. integers) to entities of type e. Since the term $\lambda g.g(i)$ contains no free variables it can be substituted for any variable without restriction. The main difficulty was in coming up with a novel (non-standard) semantics for abstraction, application and quantification which accounts for delayed abstraction as well as delayed quantification, and thus allows for pronouns to be bound even if they occur outside the syntactic scope of the binder. In the final section we introduce the notion of unrestricted reduction $\mathbf{r}$ (e.g. $\mathbf{r}(\lambda x_2 \forall x_1 P(x_1, x_2)(x_1)) = \forall x_1 P(x_1, x_1)$) and show that the translation of a formula $\alpha \in L_0$ is equivalent with the translation of its unrestricted reduction $\mathbf{r}(\alpha)$.

References

Barker, Chris (2002): 'Continuations and the Nature of Quantification', *Natural Language Semantics* **10**, 211–242.

Bennett, Michael (1979): Questions in Montague Grammar. mimeo. Indiana University Linguistics Club.

Heycock, Caroline (2011): Relative Reconstructions. Presented at the ZAS Reconstruction Workshop July 2011.

Salzmann, Martin (2006): 'Resumptive Prolepsis: A Study in Indirect A'-Dependencies', *LOT Dissertation Series* **136**.

Sternefeld, Wolfgang (2001): Semantic vs. Syntactic Reconstruction. *In:* C. Rohrer, A. Roßdeutscher and H. Kamp, eds, *Linguistic Form and its Computation*. CSLI Publications, Stanford, CA, pp. 145–182.

Sternefeld, Wolfgang (2013): Telescoping by Delayed Binding. *In:* M. Schenner, ed., *Proceedings of the ZAS Workshop on Head Internal Relative Clauses*. Akademie Verlag, Berlin.