A system at the cross-roads of functional and logic programming

Science of Computer Programming 19, 239-279, 1992.

Fairouz Kamareddine
Technical University of Eindhoven
Department of Mathematics and Computing Science
Den Dolech 2, P.O.Box 513, Eindhoven,
The Netherlands

November 30, 1996

Abstract

The type free λ-calculus is powerful enough to contain all the polymorphic and higher order nature of functional programming and furthermore types could be constructed inside it. However, mixing the type free λ-calculus with logic is not very straightforward (see [Aczel 80] and [Scott 75]). In this paper, a system that combines polymorphism and higher order functions with logic is presented. The system is suitable for both the functional and the logical paradigms of programming as from the functional paradigms point of view, the system enables one to have all the polymorphism and higher order that exist in functional languages and much more. In fact even the fixed point operator Y which is defined as $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ can be type checked to $((\alpha \to \alpha) \to \alpha))$ where α is a variable type. $(\lambda x.xx)(\lambda x.xx)$ can be type checked too, something not allowed in functional languages. From the point of view of theorem proving, the system is expressive enough to allow self referential sentences and those sentences that lead to Russell's and Curry's paradoxes. However, the paradoxes do not hold due to the notion of circular types which contain the type of propositions. In fact both sentences $\lambda x.\neg xx$ and $\lambda x.xx \to \bot$ are ill typed according to the system, because their resulting types are circular. Hence the application of either sentence to itself will not result in a proposition. The system is implemented in Milner's ML and can be seen as extending ML in two important ways. First, it extends the part related to the functional paradigm in that it can type terms that could not be typed in ML; namely, those are the terms that contain self application such as the Y term above. Second, our system extends ML by adding logic to it in a consistent way.

1 Introduction

1.1 Type freeness and logic

It is well known that mixing type freeness and logic leads to contradictions. For example, by taking the following syntax of terms:

$E := x \mid E'E" \mid \lambda x.E' \mid \neg E" \mid E' \to E"$

and applying the term $\lambda x.\neg xx$ to itself one gets a contradiction (known as Russell's paradox)[1]. Church was aware of the problem when he started the λ-calculus which he intended to be a theory of functions and logic. But his first theory of the λ-calculus was type free and so was inconsistent. The paradox could be described as follows:

The system had the following three concepts:

- Modus Ponens (MP): From $E \to E'$ and E, deduce E'.
- Deduction Theorem (DT): If Γ is a context, and $\Gamma \cup \{E\} \vdash E'$ then $\Gamma \vdash E \to E'$.
- β-conversion (β): $(\lambda x.E)E' = E[x := E']$.

Now we will show that we can derive E for every term E. Let E be a term and let $a = \lambda x.(xx \to E)$, then, from (MP), (DT), and (β), we could derive Curry's paradox:

1	$aa = aa \to E$	(β)
2	$aa \vdash aa$	Definition of $\vdash$
3	$aa \vdash E$	$MP + 1 + 2$
4	$\vdash aa \to E$	$DT + 3$
5	$\vdash aa$	1
6	$\vdash E$	$MP + 4 + 5$

This of course, is a contradiction because we can prove anything in the system. The presence of these foundational difficulties led to the creation of two routes of research. The first route placed a big emphasis on logic and deduction systems, but avoided the difficulty by restricting the language used to first or higher order without allowing any self-reference or polymorphism. The second route placed the emphasis on the expressiveness of the language and the richness of functional application and self reference, but at the expense of including logic in the language except if restrictions are made (such as using non-classical logics). Church, for example, followed Russell and introduced the simply typed λ-calculus. However, it became obvious that the theory had many unattractive features. Of these features we mention that at each level we should have a natural number system, such that the numbers at each level n say, are different from those at level $n + 1$. Moreover, polymorphic functions (that is functions which take arguments from many levels such as the polymorphic identity function) do not exist. Church and others then decided to enrich the syntax and the language but to avoid or restrict logic, hence the type free λ-calculus.

These two routes resulted in a gap between well worked out logics (where we have a sophisticated body of axioms and rules) and fully expressive languages (which allow the presence of a rich variety of terms including the self-referential ones). The need to remove the gap created various theories such as Martin-Löf's type theory and Feferman's T_0 which were polymorphic, allowed self reference and contained a big fragment of logic ([Martin-Löf 73] and [Feferman 79]).

While the polymorphically typed languages which contained logic (such as Martin-Löf's and Feferman's) were being developed (we call this route 3 in the history of foundation), two disciplines in programming were already doing well producing implemented systems based

[1] Of course here it might be questionned whether this is actually a contradiction. In fact, in the type free λ-calculus, every term has a fixed point. In particular, the term $\lambda x.\neg x$ has a fixed point E such that $E = \neg E$. Once we allow propositions to be a part of our terms however, we have to explain this phenomena of $E = \neg E$. We may run to three valued logic, but if we wanted to keep to two valued logic, we have to find a persuasive explanation that there is no paradox.

on routes 1 and 2 above. The first discipline, logic programming, concentrated on theorem proving and prolog, where the foundation was taken from route 1 but in the least courageous way by using the bare minimum language (first order) which assures safety from the paradoxes. The second discipline, functional programming, concentrated on implementing polymorphism and self reference where the foundation was taken from route 2, but at the expense of logic and deductions.

The above history does not include the semantics of type free theories which combine expressiveness and logic. In fact, the models of the type free λ-calculus alone were not obvious and it was in an attempt to prove their non existence, that Scott managed to construct such a model. Since then a variety of such models were constructed. These models however cannot model the addition of logic to the type free λ-caculus. The reason for this is that even though $\neg, \vee, \forall$ are continuous, the presence of $\forall$ will trivialise the model. For we would get that $(\forall d \in D)([[F]]_{g[d/x]} = 1) \Leftrightarrow [[F]]_{g[u/x]} = 1$ where u is the bottom element of the domain. In other words, the ordering relation on Scott domains makes predication trivial. For, a predicate P is true of all the objects in the model iff it is true of the bottom element. Both semanticians and computing scientists, however, share an interest in quantification and hence this problem of predication that faced Turner (in [Turner 84]) is a major issue for those interested in the semantics of either computer or natural languages and who base their work on Scott domains. The problem can be described as follows: Assume a language which has both *objects* and *functions* and assume that wffs are built out of other ones using $\wedge, \vee, \forall, \exists, \ldots$. If the model is a Scott domain E_∞ then there is no problem interpreting anything which is not a quantified sentence, as the interpretations of all such things are continuous functions and hence belong to the model. The interpretation of the quantifiers however will be problematic. This is because if we take the following interpretation for the quantifiers $\forall$ and $\exists$

$$[[\forall x \phi]]_g = \begin{cases} 1 & \text{if for each } d \text{ in } D, [[\phi]]_{g[d/x]} = 1 \\ 0 & \text{if for some } d \text{ in } D, [[\phi]]_{g[d/x]} = 0 \\ \perp & \text{otherwise} \end{cases}$$

$$[[\exists x \phi]]_g = \begin{cases} 1 & \text{if for some } d \text{ in } D, [[\phi]]_{g[d/x]} = 1 \\ 0 & \text{if for each } d \text{ in } D, [[\phi]]_{g[d/x]} = 0 \\ \perp & \text{otherwise} \end{cases}$$

Then the following is a proof of the continuity of the quantifier clause for $\forall$:

Assume by induction that we have $[[\phi]]$ is continuous where ϕ does not involve quantifiers. To prove the continuity of $[[\forall x \phi]]$ (i.e. to prove it in $[ASG \longrightarrow E_\infty]$ where ASG is the collection of assignment functions), we prove it continuous separately in each of its arguments, according to a theorem related to semantic domains.

Let us prove the continuity of $[[\forall x \phi]]$ for g in ASG. Take an ω-sequence $(g_n)_n$ and prove that: $[[\forall x \phi]]_{\cup g_n} = \cup [[\forall x \phi]]_{g_n}$.

- Assume $[[\forall x \phi]]_{\cup g_n} = 0 \Longleftrightarrow$ by definition,

  $(\exists d \in D)([[\phi]]_{\cup g_n [d/x]} = 0) \Longleftrightarrow$ by induction,

  $(\exists d \in D)(\cup [[\phi]]_{g_n [d/x]} = 0) \Longleftrightarrow$ by the structure of Booleans,

  $(\exists d \in D)(\exists n \in w)([[\phi]]_{g_n [d/x]} = 0) \Longleftrightarrow$ by logical laws,

  $(\exists n \in w)(\exists d \in D)([[\phi]]_{g_n [d/x]} = 0) \Longleftrightarrow$ by definition,

$(\exists n \in w)([[\forall x\phi]]_{g_n} = 0) \iff$ by the structure of Booleans,
$\cup[[\forall x\phi]]_{g_n} = 0$

- Assume $[[\forall x\phi]]_{\cup g_n} = 1 \iff$ by definition,
  $(\forall d \in D)([[\phi]]_{\cup g_n[d/x]} = 1) \iff$ by induction,
  $(\forall d \in D)(\cup[[\phi]]_{g_n[d/x]} = 1) \iff$ by the structure of Booleans,
  $(\forall d \in D)(\exists n \in w)([[\phi]]_{g_n[d/x]} = 1) \iff u \subseteq d$ and monotonicity,
  $(\exists n \in w)([[\phi]]_{g_n[u/x]} = 1) \iff$ monotonicity,
  $(\exists n \in w)(\forall d \in D)([[\phi]]_{g_n[d/x]} = 1) \iff$ by definition,
  $(\exists n \in w)([[\forall x\phi]]_{g_n} = 1) \iff$ by the structure of Booleans,
  $\cup[[\forall x\phi]]_{g_n} = 1$

Therefore $[[\forall x\phi]]$ is continuous.

By adopting this definition, we have: $[[\forall x\phi]]_g = 1$ iff $(\forall d \in D)([[\phi]]_{g[d/x]} = 1)$.

As $[[\phi]]$ is continuous, therefore monotonic and as $u \subseteq d$ (where, as noted above, u is the undefined) for each d in D then we get: $(\forall d \in D)([[\phi]]_{g[d/x]} = 1)$ iff $[[\phi]]_{g[u/x]} = 1$.

This clause has serious consequences. I shall illustrate this by taking in the formal language an element u' which names u (I.e. $[[u']]_g = u$ always). Now see what happens if we take ϕ to be: $x = u'$. Applying the above clause we get:

$[[x = u']]_{g[u/x]} = 1$ iff $(\forall d \in D)([[x = u']]_{g[d/x]} = 1)$ which implies:

$u = u$ iff $(\forall d \in D)(d = u)$.

That is absurd.

Hence, even from the model theoretical point of view we have a problem of combining type freeness and logic. Of course, models of the type free λ-calculus with logic exist and we mention two of them ([Aczel 80] and [Scott 75]).

In summary, theories and models for the type free λ-caculus with logic are needed. Such theories and models have been offered by various people and in various ways. Of the contributions to the model problem solution, we mention the work of Scott in his combinators and classes, Feferman in his recursive models of T_0 and Aczel in his Frege structures. There is also the famous method of constructing models using the stabilisation ordinal theorems of Gupta-Herzberger. Solutions to the theory were proposed by Aczel, Feferman, Scott, Flagg and Myhill, Fitch, Girard, Gilmore, Turner, Skolem, Ackerman and infinitely more. Those solutions restricted one or more of the three concepts which lead to Curry's paradox. That is, the solutions restricted either β-conversion, or MP or DT. From the programming paradigms point of view, very few attempts have been made at combining expressiveness with logic. The need, however, for the combination of expressive languages and strong logics is unquestionable (see [Feferman 84]). In fact, there is no doubt that we need full expressiveness in computing science and that we need to express self referential terms. It is well known for example, how important it is to discuss the semantics of recursion using the presence of the fixed point operators. Logic moreover, is at the heart of programming language semantics and of theorem proving. How can we hence push away logic only because we need expressivity and because expressivity and logic lead to paradoxes?

Therefore, this paper aims at providing a very clear system which extends ML in exactly those two areas of expressiveness and logic and which is consistent. The solution should of course be to keep as much as possible of expressivity and logic without facing the paradoxes.

Of course we will face the question that there are other systems which are expressive and have logic in them. Paulson' HOL is such a system. Our reply is that, yes HOL is expressive and have logic in it but its expressivity in terms of self referential terms is similar to that of Milner's ML. In fact the originality of HOL is that it combines logic to a system as expressive as ML. Our system on the other hand combines logic to a system more expressive than ML. So for us not only we have logic, but we have also self referential terms that could not exist in ML, such as $\lambda x.xx$ and $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$.

1.2 Type freeness and Polymorphism

Before we dive into this section, let us attempt to explain what we mean by type freeness and polymorphism. We understand by a type free theory, a theory where terms are well structured but all information about types is unimportant. In such a theory, any two terms can be combined together to result in a term. This is something not accepted in some type theories where two terms can only be combined together if their types match.

Example 1.1 *$\lambda x.x$ is a type free term and the term $(\lambda x.x)(\lambda x.x)$ is a legal one. In some type theories however, we have to say what is the type of x in $\lambda x.x$. For example, $\lambda x : e.x$ where e is the type of objects, is of type $e \rightarrow e$. In such a theory, $\lambda x : e.x$ cannot be applied to itself, but only to things of type e.*

The notion of polymorphism however is quite different from that of type freeness. We say that a theory is polymorphic if functions are not statically typed and the concept of function is defined by what the function does independently of the specific domains on which it operates.

Example 1.2 *A theory where the identity function $\lambda x.x$ has for type $\alpha \rightarrow \alpha$ where α is a variable type, is polymorphic in that α can be instantiated to any type, such as integers, booleans and so on. In a statically typed language however, the identity function has to be given its type at the start and so we speak of the identity function over the integers, the identity function over the booleans and so on.*

Of course there are levels of polymorphism. A theory may allow some functions to be polymorphic and not others. A type free theory on the other hand may result in different notions of polymorphism depending on the concept of type built on the top of it.

Example 1.3 *The language ML of Milner is based on Curry's language $\lambda_{\rightarrow}$Curry (see Section 2.1). This language has for syntax of expressions that of the type free λ-calculus, yet this language is not polymorphic enough to allow terms such as $\lambda x.xx$ to be typechecked. This is due to the non rich notion of type built on it.*

So far we have only talked about the concepts of type freeness and polymorphism without talking about their relation to programming languages. Programming languages however, whether functional, logic or object oriented languages, are facing the problem that their underlying formalism is not polymorphic, or type free enough. In fact, imperative languages such as Pascal are based on the idea that functions, procedures, and hence their operands have a unique type (such languages are said to be monomorphic). Such a problem of strict typing is faced by many programming languages and attempts have been made in order to avoid the problem. In fact now, one finds functional languages (such as Milner's ML) which are polymorphic.

Example 1.4 *The function* len *which finds the length of a list can be defined in ML as follows:*

rec
len [] = 0
$||$ *len* $(a.x) =$ *len* $x + 1$
end

where the only important fact about the function len *is that its argument is a list which could be a list of integers, characters, Boolean, or a list of lists. That is:*

len*: (list *a)* $\rightarrow$ *integer, where *a refers to type variables.*

*If a user wanted to find the length of a list of integers then *a would be specialised or instantiated to* integer *and the function* len *would now possess the type [integer]* $\rightarrow$ *integer.*

Object oriented languages too are beginning to accommodate polymorphism. This is because in object oriented languages, the notion of data type is very important and in these data types there are definite sets of operations which need to be instantiated with different instances. Therefore these sets of operations will need to be defined polymorphically. Moreover, the notion of inheritence in these languages is also very important and an object inherits the properties of other objects above it in the graph. In this inheritence process properties too will have to be instantiated; this instantiation is nothing more than a specialisation of a polymorphic object.

The polymorphism used so far however in programming languages, is still not strong enough to allow self-referencial terms such as the fixed point operator Y which is defined as $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$. Such a Y cannot be given a type in languages such as ML and hence cannot be used as expressions in those languages. The terms $\omega = (\lambda x.xx)(\lambda x.xx)$ and $R = (\lambda x.\neg xx)(\lambda x.\neg xx)$ face the same problem as Y. However, we think it important that terms like $\lambda x.xx$ and Y exist in any formulation of programming languages if only because of self reference and self application that exist in such languages.

It might be argued that $\lambda x.xx$ and Y are not needed, by saying that instead of $(\lambda x.xx)f$ one can use $(\lambda x.\lambda y.xy)ff$, and Y can be defined by its characteristic equation $YE = E(YE)$. We disagree with this opinion because in languages like ML, even though f in $(\lambda x.\lambda y.xy)f\,f$, gets applied to f, the first f is of a different type than the second one. In fact the two functions f are different functions (for they have different types) even though they do the same thing. Hence in languages like ML, we do not have real self application. Furthermore, in such languages, it is impossible to typecheck $\lambda x.xx$ or $(\lambda x.xx)f$. On the other hand, assume we work with a language which actually does have real self application, and where $(\lambda x.\lambda y.xy)ff$ actually applies f to itself. This means that f (even though polymorphic) takes an element of its whole type as an argument. This approach we agree with, and even though $\lambda x.\lambda y.xy$ and $\lambda x.xx$ work in a similar way for f, they are still different functions. In other words here, typechecking usually assumes the following two principles:

1. All occurrences of a variable which are bound by a given λ must be assigned the same type.
2. Distinct occurrences of a given free variable are allowed to be assigned different types.

Example 1.5 *In* $(\lambda x.xx)f$*, both occurrences of* x *in* xx *have the same type, whereas in* $(\lambda x.\lambda y.xy)ff$*, the two occurrences of* f *have distinct types.*

According to ML's approach which assumes those two principles, $\lambda x.xx$ cannot be typechecked because types don't usually contain their arrow types. Hence if x is of type $\alpha \to \beta$, how do we know that this x accepts an object of type $\alpha \to \beta$ as an argument? Our approach on the other hand, assumes these two principles too, but there is the extra condition that always $(\alpha \to \beta) \leq \alpha$. Hence if x is of type $\alpha \to \beta$, it is also of type α. So x has two types α and $\alpha \to \beta$. Moreover, xx is well defined and of type β. Also, in $(\lambda x.\lambda y.xy)ff$, according to our approach, the two occurrences of f have the same type $\alpha \to \beta$, but also this type $\leq \alpha$. Hence f has two types, α and $\alpha \to \beta$.

In the system offered in this paper, we start from the type free lambda calculus. Hence everything starts without a type, and all combinations of terms are allowed. In fact, anything can be applied to anything else and the result is a term. If we come to typecheck any term which does not contain free variables, then its type is given if it exists. For example, $\lambda x.xx$ is type checked to $(\alpha_0 \to \alpha_1) \to \alpha_1$. However, if we ask to typecheck x in an environment where the type of x is undefined, then an "error message" will result. We should typecheck x in an environment in which x is declared to be of a particular type. Now if we typecheck $(\lambda x : p.x)y$ in an environment where y is an object (we write $y : e$) and where p is the type of propositions then an "error-message" will result informing us that p and e mismatch as types. This is of course the case because e is not subsumed by p, and the system deduces that $(\lambda x : p.x)$ which is of type $p \to p$ cannot apply to arguments of type e, but can only apply to terms whose type is subsumed by p (i.e. who are contained in p). If however we typecheck $(\lambda x : \alpha_0.x)y$ where $y : \alpha_1$ and α_0, α_1 are type variables, then the system will deduce that the type of $(\lambda x : \alpha_0.x)$ is $\alpha_0 \to \alpha_0$ and it will try to check and see if $\alpha_0 \leq \alpha_1$ but as α_1 is a variable, the system makes α_1 become α_0 and returns α_0 as the result. Of course in this section we have mixed the mathematical activity of attributing a type to a term and the mechanical activity of typechecking a term. These two activities are unquestionably different things but our paper is concerned with both.

1.3 The paradise of the type free lambda calculus

Let us start by asking a few questions and attempting to answer them. These questions concern the notions of "types", "typed" and "type free" theories. "Type" is this construct that we associate to a term in a typed theory so that we can make sense of some term combination. In a type free theory on the other hand, any combination is allowed.

Question 1. Are types or levels necessary in the avoidance of the paradoxes?.

Answer Not necessarily. For example, ZF was another solution to the paradox where we don't need to classify sets iteratively ([Boolos 71]), yet the Foundation Axiom FA was included in ZF despite the fact that it was shown that antifoundation axioms are consistent with ZF (see [Aczel 84] for such a discussion). The Foundation Axiom FA is $(\exists x)(x \in a) \to (\exists x \in a)(\forall y \in x)\neg(y \in a)$. As a corollary of it, we do not get solutions to $x = \{x\}$, or $x = \{\{x\}\}$. Moreover, the inclusion of FA was unnecessary and it was not the responsible axiom for avoiding the paradox.

Question 2. Are types needed?

Answer Yes of course. The fact that we ask for the full expressive power of the type free λ-calculus does not mean that types are not needed. In fact when we ask for a type free set theory, or a set theory where the definition of a set may be impredicative, we don't go and forget completely about sets. In type free theories, one asks for the furthest expressive power, where we can live with self reference and impredicativity but without paradoxes. The better

such an expressive system is, the more we are moving towards type freeness. Just it is enough to remember that up to the construction of the paradoxes, the ideal system was of course type free. Due to the paradoxes, helas this type free paradise had to be abandoned. Types too found an attractive place in the history of foundation and in most areas of applications of logic. For after all types help in the classification of programs, in the mixing of terms and so on. And moreover they play an important role in explaining the paradoxes (if such an explanation is actually possible). For example, Girard's system F ([Girard 86]) is no less type free than Feferman's theory T_0 yet types play a valuable role in that system with respect to impredicativity. The difference between F and T_0 might be in the explicitness or implicitness of the typing scheme. Now even though one works in a type free system such as that of Feferman, one needs to introduce types such as recursive types, dependent types and the like. After all many of our proofs are for a particular collection of objects and not for all possible objects. Exactly as in set theory, intersection, union and so on are absolute necessity. Note also that a fully type free language cannot accommodate an unrestricted logic together with an unrestricted β-conversion.

Question 3. So if types are needed why talk about type free theories? Why not ignore type freeness?

Answer. The reason is that we may not want to be inflexible from the start if we could afford to be flexible. Type free theories are very elegant and simple, so we can have a clear picture of how much we have and how the paradox is avoided. Then the detail of constructing types if followed will produce all the polymorphic higher order types that are needed. So a lot of unnecessary details (like constructing types) are left till later which will make it easier to prove results about the strength of the system, the expressive power, completeness and so on. Also from the point of view of computation, type free theories could be regarded as first order theories and hence are computationally more tractable than typed theories. Completeness also holds for first order logics but has to be forced for higher order ones. Hence what I am arguing for is the use of type freeness followed by the construction of flexible polymorphic types. It is also the case that the self referentiality of language requires type freeness. So we can talk about a property having itself as a property. For example, the property of those things equal to themselves has itself as a property.

That programming language theory needs a type free background to capture polymorphism and self reference, and that programming languages are implicitly typed, makes it desirable to have a type checking algorithm. Type checking ensures that the application of a function to its arguments is done properly. The purpose of type-checking is to avoid nonsensical operations like adding a character to a truth value. More precisely a type error occurs if a function F, of type $T \to T'$, is applied to an argument which is not of type T. In this paper, self reference is allowed and paradoxes are avoided in our theory which starts type free but where the type checker finds those types that are legitimate. In fact, we do not work with and construct types inside the λ-calculus as a theory of functions only, but aim for the most expressive part which contains logic yet remains consistent. This is done in the system where everything starts by being a term of the type free λ-calculus. Hence everything starts without a type, and all combinations of terms are allowed. In fact, anything can be applied to anything else and the result is a term. However only the typeable terms can be typechecked and the result of the typechecking is their type. For example, the self-application function $\lambda x.xx$, which takes a function and applies it to itself is typable and is type checked to $(\alpha_0 \to \alpha_1) \to \alpha_1$, according to our typing system, where α_0, α_1 are variable types. Our way of avoiding the paradox is by disallowing special kind of types, the *circular types*. Those circular

types have the form $(\alpha_0 \to \alpha_1) \to \alpha_1$ where $\alpha_1 \leq p$, where p is the type of propositions. Hence above even though we said that $\lambda x.xx$ is typable, the type of its abstracted variable x, cannot be $(\alpha_0 \to \alpha_1)$ where $\alpha_1 \leq p$. This system is the type free system where all types except circular ones can be constructed.

2 Type Theory and polymorphism

In Type Theory, various formulations of the typing systems have been provided, some of which can type check $\lambda x.xx$ and/or Y and some cannot (see [Barendregt, Hemerik 90]). All these type systems, use the following as their underlying syntax of types $s ::= x|c|s \to s$ which says that a type is either a variable or a constant or an arrow. Type systems such as λ_2, λ_μ and $\lambda_\cap$ (see [Barendregt, Hemerik 90]), add other types to this set of types in order to typecheck more terms such as Y and $\lambda x.xx$; those which use only the above syntax of types, even though they can be polymorphic, they cannot typecheck Y or $\lambda x.xx$. Milner's ML is such an example; it is based on the system $\lambda_\to$ of Curry (see [Barendregt, Hemerik 90]) which uses the simple syntax of types $T ::= \alpha|c|T \to T$, and it is unable of typing Y or any term which involves self application except in an ad-hoc way.

Let us here overview $\lambda_\to, \lambda_2, \lambda_\mu$ and $\lambda_\cap$ and see what they can do for $\lambda x.xx$ and Y. In these systems we understand by an environment Γ to be a partial function from term variables to the set of types. This is given by the following definition:

Definition 2.1 *An environment is a set of type assignments $(V : T)$ which assigns the type T to the variable V, such that a variable is not assigned two different types. We let Γ range over environments.*

Notation 2.2 *When $(V : T) \in \Gamma$, we say that the type of V in the environment Γ is T. Moreover, we define the free variables of a term T, $FV(T)$, in the usual way and say that $V \in FV(\Gamma)$ iff $V \in FV(T')$ for some $(V', T') \in \Gamma$. Moreover, the notation $\Gamma \vdash E : T$ means that from the environment Γ, we can deduce that the expression E has type T.*

2.1 The system $\lambda_\to$Curry

Definition 2.3 *(Expressions and Types of $\lambda_\to$Curry)*
Expressions are $E ::= V|E_1E_2|\lambda V.E$
Types are $T ::= \alpha|c|T \to T$

Definition 2.4 *(Rules of $\lambda_\to$Curry)*
Rules of $\lambda_\to$Curry are defined as follows:

$$\frac{(V : T) \in \Gamma}{\Gamma \vdash V : T} \tag{1}$$

$$\frac{\Gamma \vdash E_1 : T \to T" \qquad \Gamma \vdash E_2 : T}{\Gamma \vdash E_1E_2 : T"} \tag{2}$$

$$\frac{(V : T) \cup \Gamma \vdash E : T'}{\Gamma \vdash \lambda V.E : T \to T'} \tag{3}$$

Example 2.5 *In* $\lambda_{\rightarrow}$*Curry,* $\lambda x.\lambda y.xy$ *can be seen to be of type* $(T \rightarrow T') \rightarrow T \rightarrow T'$ *as follows:*

(i) $x : T \rightarrow T'$ *hyp*
(ii) $y : T$ *hyp*
(iii) $x : T \rightarrow T'$ *(i), reit*
(iv) $xy : T'$ *(ii), (iii), (2)*
(v) $\lambda y.xy : T \rightarrow T'$ *(i) ... (iv), (3)*
(vi)$\lambda x.\lambda y.xy : (T \rightarrow T') \rightarrow T \rightarrow T'$ *(i) ... (v), (3)*

In $\lambda_{\rightarrow}$, we cannot typecheck $\lambda x.xx$ nor Y.

2.2 The system λ_2

Definition 2.6 *(Expressions and Types of* λ_2*)*
Expressions are $E ::= V|E1E2|\lambda V.E$
Types are $T ::= \alpha|c|T \rightarrow T|\forall\alpha.T$

Definition 2.7 *(Rules of* λ_2*)*
Rules of λ_2 *are (1) + (2) + (3) + (4) + (5) where:*

$$\frac{\Gamma \vdash E_1 : \forall\alpha.T}{\Gamma \vdash E_1 : T[\alpha := T']} \tag{4}$$

$$\frac{\Gamma \vdash E : T \qquad \alpha \notin FV(\Gamma)}{\Gamma \vdash E : \forall\alpha.T} \tag{5}$$

Example 2.8 *In* λ_2*, that* $\lambda x.xx$ *is of type* $\forall\alpha.(\forall\beta.\beta) \rightarrow \alpha$ *can be seen as follows:*

(i) $x : \forall\beta.\beta$ *hyp*
(ii) $x : \beta \rightarrow \alpha$ *(i), (4)*
(iii) $x : \beta$ *(i) , (4)*
(iv) $xx : \alpha$ *(ii), (iii), (2)*
(v) $\lambda x.xx : (\forall\beta.\beta) \rightarrow \alpha$ *(i) ... (iv), (3)*
(vi) $\lambda x.xx : \forall\alpha.(\forall\beta.\beta) \rightarrow \alpha$ *(v), (5)*

However, The fixed point term Y is not typable in λ_2 nor is $(\lambda x.xx)(\lambda x.xx)$.

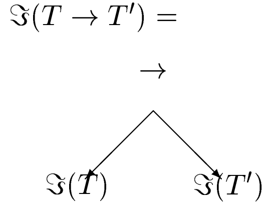
2.3 The system λ_μ

Definition 2.9 *(Expressions and Types of* λ_μ*)*
Expressions are $E ::= V|E_1E_2|\lambda V.E$
Types are $T ::= \alpha|c|T \rightarrow T|\mu\alpha.T$

Moreover, in λ_μ we need the following concept:

Definition 2.10 *(Approximation of Types)*
We say that $T \approx T'$ *iff* $\Im(T) = \Im(T')$ *where* $\Im(T)$ *is a tree defined as follows:*
$\Im(\alpha) = \alpha$
$\Im(c) = c$

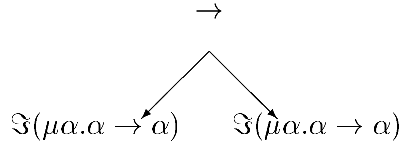
$\Im(T \rightarrow T') =$



$\Im(\mu\alpha.T) = \Im(T[\alpha := \mu\alpha.T])$ *if "defined", else* $\bullet$.

Example 2.11 *Here are two terms and their corresponding images by* $\Im$.

1. $\Im(\mu\alpha.\alpha) = \bullet$
2. $\Im(\mu\alpha.\alpha \rightarrow \alpha) =$



Definition 2.12 *(Rules of* λ_μ*)*
Rules are (1) + (2) + (3) + (6) where

$$\frac{\Gamma \vdash E_1 : T \qquad T \approx T'}{\Gamma \vdash E_1 : T'} \tag{6}$$

Example 2.13 *Let* $T' = \mu\alpha.(\alpha \rightarrow T)$, *then* $T' \approx T' \rightarrow T$. *Now,* $\lambda x.xx$ *gets the type* $T' \rightarrow T$ *as follows:*

(i)	$x : T'$	*hyp*
(ii)	$x : T' \rightarrow T$	*(i), (6)*
(iii)	$xx : T$	*(i), (ii), (2)*
(iv)	$\lambda x.xx : T' \rightarrow T$	*(i) ... (iii) (3)*

Example 2.14 *That* Y *is of type* $(T \rightarrow T) \rightarrow T$ *can be seen as follows:*

(i)	$f : T \rightarrow T$	*hyp*
(ii)	$x : T'$	*hyp*
(iii)	$x : T' \rightarrow T$	*(ii), (6)*
(iv)	$xx : T$	*(ii), (iii), (2)*
(v)	$f(xx) : T$	*(i), (iv), (2)*
(vi)	$\lambda x.f(xx) : T' \rightarrow T$	*(i) ... (v), (3)*
(vii)	$\lambda x.f(xx) : T'$	*(vi), (6)*
(viii)	$(\lambda x.f(xx))\lambda x.f(xx) : T$	*(vi), (vii), (2)*
(ix)	$\lambda f.(\lambda x.f(xx))\lambda x.f(xx) : (T \rightarrow T) \rightarrow T$	*(i) ... (viii), (3)*

$(\lambda x.xx)(\lambda x.xx)$ can be typechecked to T. Moreover if we started with $T' = \mu\alpha.\alpha \rightarrow \alpha$ then we could typecheck $Y(\lambda x.xx)$ to T'.

2.4 The system $\lambda_\cap$

Definition 2.15 *(Expressions and Types of $\lambda_\cap$)*
Expressions are $E ::= V|E_1E_2|\lambda V.E$
Types are $T ::= \alpha|c|T \to T|T \cap T$ *with* ω *a constant type.*
Types are ordered by $\leq$ *and where* $T \leq \omega$ *for every type* T. *Moreover,* $\leq$ *is symmetric, transitive, closed under intersection and satisfies amongst other things that* $T \cap T' \leq T'$.

Definition 2.16 *(Rules of $\lambda_\cap$)*
Rules are (1) + (2) + (3) + (6) + (7) + (8) + (9) + (10) where

$$\frac{\Gamma \vdash E_1 : T \cap T'}{\Gamma \vdash E_1 : T \qquad \Gamma \vdash E_1 : T'} \tag{7}$$

$$\frac{\Gamma \vdash E_1 : T \qquad \Gamma \vdash E_1 : T'}{\Gamma \vdash E_1 : T \cap T'} \tag{8}$$

$$\frac{\Gamma \vdash E_1 : T \qquad T \leq T'}{\Gamma \vdash E_1 : T'} \tag{9}$$

$$\frac{}{\Gamma \vdash E_1 : \omega} \tag{10}$$

Example 2.17 *That* $\lambda x.xx$ *has type* $(T \cap (T \to T')) \to T'$ *can be seen as follows:*

(i)	$x : T \cap (T \to T')$	*hyp*
(ii)	$x : T \to T'$	$(i),(7)$
(iii)	$x : T$	$(i),(7)$
(iv)	$xx : T'$	$(ii),(iii),(2)$
(v)	$\lambda x.xx : (T \cap (T \to T')) \to T'$	$(i)\ldots(iv),(3)$

In $\lambda_\cap$ however, $(\lambda x.xx)(\lambda x.xx)$ gets the type ω due to the failure of the system in finding the more specific type for it. Moreover, Y is not typable in $\lambda_\cap$.

Our aim in this paper is not to extend the syntax of types by allowing forall, recursive or intersection types as in λ_2, λ_μ and $\lambda_\cap$, but to provide a typing system similar to ML, except that the matching between types takes a different form than that in ML. The reason why ML cannot typecheck $\lambda x.xx$ and Y is that even though ML is based on the type free λ-calculus, its typing principles leave $*a \to *b$ and $*a$ (where $*a$ and $*b$ are any types) incomparable. On the other hand, the structure of the models of the type free λ-calculus demands that $(*a \to *b) \leq *a$, and this ordering is the basis of applying functions to themselves. Take for example, $\lambda x.xx$, the operator occurrence of x requires that x be of type $*a \to *b$, and for this occurrence to apply to x, x must also be of type $*a$.

Like ML we will construct a polymorphic type system based on the type free λ-calculus. Unlike ML however, the relation between types will include that every arrow type is included in its domain space. This system will allow typing the self referential term $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$, the self application function $\lambda x.xx$ and all the possible mixtures of Y and $\lambda x.xx$.

3 The system L_λ

3.1 Expressions

Let our term variables be $x, x', y, y', z, z' \ldots$, let $V, V', V", \ldots$ range over these variables, let $\alpha_0, \alpha_1, \ldots$ be our type variables and let $\beta, \beta_0, \beta_1 \ldots$, range over these variables. We let $E, E', E", \ldots E_1, E_2, \ldots, \Phi, \Psi, \ldots$, range over expressions and $T, T', T_1, T_2, \ldots$ range over type expressions.

Definition 3.1 *(Types)*
We will construct types inside this language as follows:
$T ::= \beta \mid Basic \mid (T_1 \rightarrow T_2)$
$Basic ::= p \mid t \mid e$

Here p is the type of propositions, t is the type of truths (that is of all the true propositions) and e is the type of objects. In fact e contains everything, variable types, basic types and arrow types. This is the case due to the subsumption relation $\leq$ on the types defined in Definition 3.8.

Definition 3.2 *(Expressions)*
We assume the following syntax of terms:
$E = V|(E_1E_2)|(\lambda V.E_1)|\Omega E|(\lambda V : T.E_1)|(E_1 \wedge E_2)|(E_1 \rightarrow E_2)|(\neg E_1)|(\forall V.E_1)|(\forall V : T.E_1)$

Hence as seen from the syntax, we work inside the type free λ-calculus with logic but we also allow types. All the above terms should be obvious except for ΩE. This is to be understood as saying that E is a proposition. It is needed to make the construction of logic inside the type free λ-calculus non paradoxical (see [Kamareddine 89], [Aczel 80], [Beeson 84]). More precisely, even though $(\lambda x.\neg xx)(\lambda x.\neg xx) = \neg(\lambda x.\neg xx)(\lambda x.\neg xx)$, the paradox does not arise because there is no way to prove that $\Omega(\lambda x.\neg xx)$.[2] Finally, we assume the usual conventions for the dropping of parentheses when no confusion occurs and say that $E \equiv E$' iff E and E' are exactly the same.

Definition 3.3 *(Substitution)*[3]
We define $E[E'/V]$ the result of substituting E' for each free occurrence of V in E as follows:

[2]Our syntax of terms (excluding those that involve logic) is similar to that of Milner except that we do not include the *if, let* and *fix* constructs; these can however be built out of other ones.
[3]These rules are used in the implementation in Section 7.

(S1)	$V[E'/V] \equiv E'$
(S2)	$V'[E'/V] \equiv V'$ *if not* $(V' \equiv V)$
(S3)	$(E_1E_2)[E'/V] \equiv (E_1[E'/V])(E_2[E'/V])$
	$(E_1 \wedge E_2)[E'/V] \equiv (E_1[E'/V]) \wedge (E_2[E'/V])$
	$(E_1 \rightarrow E_2)[E'/V] \equiv (E_1[E'/V]) \rightarrow (E_2[E'/V])$
	$(\neg E_1)[E'/V] \equiv \neg(E_1[E'/V])$
(S4)	*If* M *is* V *or* M *is* $V : T$ *then*
	$(\lambda M.E_1)[E'/V] \equiv \lambda M.E_1$
	$(\forall M.E_1)[E'/V] \equiv \forall M.E_1$
(S5)	*If* M' *is* V' *or* M' *is* $V' : T$ *then*
	$(\lambda M'.E_1)[E'/V] \equiv \lambda M'.E_1[E'/V]$ *if not* $(V' \equiv V)$ *and* $V' \notin free(E')$ *or* $V \notin free(E_1)$
	$(\forall M'.E_1)[E'/V] \equiv \forall M'.E_1[E'/V]$ *if not* $(V' \equiv V)$ *and* $V' \notin free(E')$ *or* $V \notin free(E_1)$
(S6)	*If* (M' *is* V' *or* M' *is* $V' : T$) *and* (M" *is* V" *or* V" $: T$) *then*
	$(\lambda M'.E_1)[E'/V] \equiv \lambda M".E_1[V"/V'][E'/V]$
	if not $(V' \equiv V)$ *and* $V' \in free(E')$, *and* $V \in free(E_1)$ *and* $V" \notin free(E'E_1)$
	$(\forall M'.E_1)[E'/V] \equiv \forall M".E_1[V"/V'][E'/V]$
	if not $(V' \equiv V)$ *and* $V' \in free(E')$, *and* $V \in free(E_1)$ *and* $V" \notin free(E'E_1)$

As we said before, the typed terms are built out of the type free ones. Hence, we will restrict attention to the untyped fragment. We assume the well known three axioms of the type free λ-calculus (there are of course other axioms and rules which will be gradually introduced below):

Definition 3.4 *(Axioms of the type free λ-calculus)*
The following three axioms are assumed in our system:
(α) $\lambda V.E \rightarrow_\alpha \lambda V'.E[V'/V] \qquad if V' \notin free(E)$
(β) $(\lambda V.E)E' \rightarrow_\beta E[E'/V]$
(η) $\lambda V.EV \rightarrow_\eta E \qquad if V \notin free(E)$.

We write $E \rightarrow_\beta E'$ (respectively $E \rightarrow_\alpha E'$ and $E \rightarrow_\eta E'$) iff E' is obtained from E by reducing any subterm of E using (β) (respectively (α) and (η)).

If $E \rightarrow_\beta E'$ (respectively $E \rightarrow_\alpha E'$ and $E \rightarrow_\eta E'$) then we say E β-reduces (respectively α-reduces and η-reduces) to E'.

If an expression may be reduced by (β) or (η), we say that it *contains a β-redex* or an η-redex. An expression of the form $(\lambda V.E)E'$ is called a *β-redex* and the corresponding term $E[E'/V]$ is called its *contractum*. An expression of the form $\lambda x.Ex$ where $x \notin free(E)$ is called an *η-redex*. Its *contractum* is E.

Definition 3.5 *(Reduction)*
We define $\prec$ to be the rfelexive and transitive closure of $\rightarrow$ where $E \rightarrow E' \Leftrightarrow E \rightarrow_\beta E'$ or $E \rightarrow_\eta E'$ or $E \rightarrow_\alpha E'$. When $E \prec E'$, we say that E reduces to E'.

Definition 3.6 *(Equality)*
We define equality to be the smallest equivalence relation containing $\prec$. If $E = E'$, we say that E equals to E'.

Definition 3.7 *(Normal Form)*
An expression is in normal form if it does not contain an η-redex or a β-redex, an expression E has a normal form if $E = E'$ for some E' in normal form.

3.2 Types and their semantic justification

As explained at the end of Section 2, the reason why ML cannot typecheck $\lambda x.xx$ and Y is that even though Milner's ML is based on the type free λ-calculus, its typing principles leave $*a \to *b$ and $*a$ (where $*a$ and $*b$ are any types) incomparable. On the other hand, the structure of the models of the type free λ-calculus demands that $(*a \to *b) \leq *a$, and this ordering is the basis of applying functions to themselves. Based on this observation, the relation between types will include that every arrow type is included in its domain space. This relation $\leq$ is defined as follows:

Definition 3.8 *(Subsumption Relation)*
The ordering/subsumption relation on types is given by the following rules:
i) $T \leq e$
ii) $t \leq p$
iii) $(T \to T') \leq T$
iv) $T \leq T$
v) if $T \leq T'$ *and* $T' \leq T$ *then* $T = T'$
vi) if $T \leq T'$ *and* $T' \leq T"$ *then* $T \leq T"$
vii) if $T \leq T'$ *then* $(T_1 \to T) \leq (T_1 \to T')$

In other words, everything is an object, true propositions are propositions, $\leq$ is a partial order and $(T \to)$ is monotonic. moreover, it is mainly clause iii) which enables us to have self application in the system.

We say that by $(T \leq T')$, T *subsumes* T'; intuitively it means that any expression which is of type T is also of type T'.

Due to the presence of logic and self application, we will use the notion of circular types defined in Definition 4.14, to avoid the paradoxes. When an expression E has type T we write $E : T$. In particular we write $\Phi : p$ for Φ a proposition and $\Phi : t$ for Φ true. We write $T \equiv T'$ if the types T and T' are syntactically the same. Our syntax of types is very similar to that of Milner ([Milner 78]) except that we restrict attention to the domain e which is a model of the type free λ-calculus. We follow Milner in defining *monotypes* to be types which contain no type variables and use μ, ν, Π, to range over monotypes. As Milner we use the word *polytype* to describe that a type may contain type variables.

3.3 The typing rules with respect to the new ordering and the typing of Y and self application.

We carry over here the definition of an environment and the notation $\Gamma \vdash E : T$ as given in definition 2.1 and Notation 2.2. The following rules associate types to the expressions of the type free part. Those expressions involving logic will be type checked later.

Definition 3.9 *(Typing λ-expressions) The following typing rules accommodate in the usual typing rules, the notion of ordering:*

$$\frac{(V : T) \in \Gamma}{\Gamma \vdash V : T} \tag{11}$$

$$\frac{\Gamma \vdash E : T \qquad T \leq T'}{\Gamma \vdash E : T'} \tag{12}$$

$$\frac{\Gamma \vdash E_1 : T \to T' \qquad \Gamma \vdash E_2 : T}{\Gamma \vdash E_1 E_2 : T'} \tag{13}$$

$$\frac{\{(V : T)\} \cup \Gamma \vdash E : T'}{\Gamma \vdash \lambda V.E : T \to T'} \tag{14}$$

From the above, it is obvious that some expressions have many types. For example, $\lambda x.x$ is of type $\alpha \to \alpha$ for any type variable α.

Now let us illustrate with typing $\lambda x.xx$ and Y.

Example 3.10 *$\lambda x.xx$ has type $(\alpha_0 \to \alpha_1) \to \alpha_1$:*

(i) $x : \alpha_0 \to \alpha_1$ *Assumption*
(ii) $\alpha_0 \to \alpha_1 \leq \alpha_0$ *clause iii) of* $\leq$
(iii) $x : \alpha_0$ *(i), (ii), (12)*
(iv) $xx : \alpha_1$ *(i), (iii), (13)*
(v) $\lambda x.xx : (\alpha_0 \to \alpha_1) \to \alpha_1$ *(i) … (iv), (14)*

Example 3.11 *$\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ has type $(\alpha_2 \to \alpha_2) \to \alpha_2$:*

(i) $f : \alpha_2 \to \alpha_2$ *assumption*
(ii) $x : (\alpha_1 \to \alpha_2) \to \alpha_2$ *assumption*
(iii) $(\alpha_1 \to \alpha_2) \to \alpha_2 \leq \alpha_1 \to \alpha_2$ *clause iii) of* $\leq$
(iv) $x : \alpha_1 \to \alpha_2$ *(ii), (iii), (12)*
(v) $xx : \alpha_2$ *(ii), (iv), (13)*
(vi) $f(xx) : \alpha_2$ *(i), (v), (13)*
(vii) $\lambda x.f(xx) : ((\alpha_1 \to \alpha_2) \to \alpha_2) \to \alpha_2$ *(ii) … (vi), (14)*
(viii) $((\alpha_1 \to \alpha_2) \to \alpha_2) \to \alpha_2 \leq (\alpha_1 \to \alpha_2) \to \alpha_2$ *clause iii) of* $\leq$
(ix) $\lambda x.f(xx) : (\alpha_1 \to \alpha_2) \to \alpha_2$ *(vii), (viii), (12)*
(x) $(\lambda x.f(xx))(\lambda x.f(xx)) : \alpha_2$ *(iii), (ix), (13)*
(xi) $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) : (\alpha_2 \to \alpha_2) \to \alpha_2$ *(i) … (x), (14)*

Example 3.12 *As another example, $(\lambda x : \alpha_0.x)y$ where $y : \alpha_1$ and α_0, α_1 are type variables, is also typable and the system will deduce that the type of $(\lambda x : \alpha_0.x)$ is $\alpha_0 \to \alpha_0$ and it will try to check and see if $\alpha_0 \leq \alpha_1$ but as α_1 is a variable, the system makes α_1 become α_0 and returns α_0 as the result.*

4 Type checking

The type checker is straightforward yet it allows for better polymorphism than other systems because of the subsumption relation that is used. The algorithm for type checking is implemented using *checkexpr* where *checkexpr* is a function with the following functionality: environments $\times$ heap-variables $\times$ terms $\to$ (substitutions $\times$ types $\times$ heap-variables) + error.

Before we explain the type checker we need to describe how we implement the various data types and the various relation on them.

4.1 On the λ-reducer

The implementation of the terms, types and their properties is straightforward except when we come to the reducer. This is because we are using the type free λ-calculus as our basis and hence many reductions will not end in normal forms. The following example illustrates the point:

Example 4.1 $(\lambda x : e \to p.xx)(\lambda x : e \to p.xx)$
$= (\lambda x : e \to p.xx)(\lambda x : e \to p.xx)$ *as* $(e \to p \leq e)$
$= (\lambda x : e \to p.xx)(\lambda x : e \to p.xx)$ *as* $(e \to p \leq e)$
$= \ldots = (\lambda x : e \to p.xx)(\lambda x : e \to p.xx)$ *as* $(e \to p \leq e) = \ldots$ *and so on.*

To be able to implement the reducer of the expressions, we have to be able to deal with such a problem. Because normal order reduction is safe, that is if a term has a normal form then it finds it, we are going to use normal order reduction which works on the leftmost outermost reductions of the terms. Of course normal order reduction will not deal with the above problem of $(\lambda x : e \to p.xx)(\lambda x : e \to p.xx)$. For this we will need an ad-hoc mechanism because of the undecidability of reduction. In fact there are much better lambda reducers than our own and better mechanisms such as head and weak normal forms. For this paper, we take the approach of checking if when reducing E we get an expression which contains E. If so, we stop and return the new expression. Not only reduction is undecidable but equality between terms is undecidable too. In this paper, the equality relation is implemented in terms of reduction and equivalence, so $E = E'$ iff $(reduce E) = (reduce E')$.

There are expressions that the reducer or equality checker don't deal with. The following is an example of such an expression:

Example 4.2 *If we take* Y *to be* $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$*, i.e.* Y *is a fixed point operator, then reduce* $(Y(\lambda x.x))$ *would lead to* $(\lambda x.x)((\lambda x.(\lambda x.x)(xx))(\lambda x.(\lambda x.x)(xx)))$ *whereas we would have liked to get:*$(\lambda x.xx)(\lambda x.xx)$*. The system will deduce that* $Y(\lambda x.x) = (\lambda x.x)(Y(\lambda x.x))$ *and this is trivial because* $(\lambda x.x)E = E$ *for any* E*. However the system will not be able to deduce that* $Y(\lambda x.xx) = (\lambda.xx)(Y(\lambda x.xx))$*. In fact it deduces that they are not equal because when it checks reduce*$(Y(\lambda x.xx))$ *and reduce* $(\lambda x.xx)(Y(\lambda x.xx))$ *it finds two different expressions.*

This of course should not be seen as a deficiency of the system, in fact this is the norm of lambda reducers.

4.2 Subsumption and unification of types

Like Milner's $\leq$, our subsumption relation $\leq$ is transitive and reflexive. Unlike Milner, our $\leq$ gives us that $(\alpha \to \alpha) \to (\alpha \to \alpha) \leq (\alpha \to \alpha) \to \alpha \leq (\alpha \to \alpha) \to e \leq e$ and there is no way to unify the type variable α with another type variable α'.

To replace α's by α''s as in Milner's system, we would need to unify the α's and α''s. For this we need unification on types which saves the binding of types, so we can say that if α and α' can be unified, we have

$(\alpha \to \alpha) \to (\alpha \to \alpha) \leq (\alpha \to \alpha) \to (\alpha' \to \alpha') \leq \alpha \to \alpha' \leq \alpha' \to \alpha \leq \alpha \to \alpha \leq e.$

All the clauses for the subsumption relation given in Definition 3.8 are straightforward to implement except if the types involved contain variable types then unification will come in and some variable types will be instantiated to other types. For example, $\beta \leq T$ will result in

a substitution of types where β is bound to T. We will change $\leq$ to deal with substitutions so that when we write $T \leq T'$, we don't only get a truth value, but a form of unification takes place. This sort of unification will be saved in a substitution function. Due to recursion needs, we start from a type substitution s when we ask the question $T \leq T'$ and we obtain a (possibly) new type substitution s'. This is written as $T \leq_s T' = s'$. Hence, $T \leq_s T' = s'$ will move from substitution s to substitution s' which takes into account some type unification during the process of comparing T and T'.

Before we define $\leq_s$, we need a few auxilliary definitions:

Definition 4.3 *(Type Substitution)*

We define a type substitution *to be a function from types to types which assigns types to type variables. We let SUB be the set of substitutions and let s range over it. Hence each s is a set of elements of the form: (β, T), where no two different elements have β as their first component. For a type T and a substitution s, we let sT be the type obtained by replacing all the type variables in T which appear as first projections in s, by their values in s.*

Example 4.4 *For example, if $s = \{(\beta, e)\}$, and T is $\beta \to \beta$ then sT is $e \to e$.*

Notation 4.5 *In the implementation, in section 7, we take* ob, pr *and* tr *to represent the types* e, p *and* t *respectively.*

Definition 4.6 *(Subsumeset)*

subsumeset *takes a type and finds those types that subsume it. The implementation of such a function is item 1 in 7.4. It is very straightforward and will not be explained further.*

Example 4.7 *subsumeset p = [e; p] and subsumeset* $p \to t = [p \to e; p \to p; p \to t]$.

Now we come to the subsumption relation itself, it is implemented by the function subsume given as item 4 in 7.4. Note the use of the option type (item 7 in 7.1):

type option *a *b = N*a + Y*b

This is so that in case the subsumption fails, we get an error message to the effect. If the subsumption succeeds, we get a substitution. In fact many of our functions will give us results in the type *option.* if the result of a function f is NI, then f fails and I contains a message explaining why the failure occured. If the result of f is YI then f succeeds and I is the desired result of f.

occurs, isarrow, domain, range, scomp, addrem, id-subst and *sub-type* appear in the implementation of subsumption (item 4, 7.4). They are to be understood as follows:

occurs T returns true if there are type variables in T, else it returns false. *Isarrow* tests whether a type is an arrow type (such as $\alpha \to \beta$). *Domain* T and *range* T find the domain and range of an arrow type T. *Scomp* is the composition function which composes two substitutions, *id-subst* is the identity substitution and *addrem* $gxy = g$ everywhere except for x where it gives the value y. We use *sub-type* to apply a substitution to a type. Of course here we will not repeat the implementation of *subsume* from item 4 of 7.4, but note that this function can be roughly translated by the following definition:

Definition 4.8 *(An algorithm for subsumes)*

i) $\mu \leq_s \nu =_{df} s$ *if* $\mu \in$ *(subsumeset* ν*)*

ii) $(T \to T_1) \leq_s T =_{df} s$

iii) $T \leq_s T =_{df} s$
iv) $\beta \leq_s T =_{df} s[T/\beta]$
v) $T \leq_s T_1 =_{df} s$ *if* $(sT = \mu)$ *and* $(sT_1 = \nu)$ *and* $((\mu \leq_s \nu) = s)$
vi) $T \leq_s T_1$ *fails if* $(sT = \mu)$ *and* $(sT_1 = \nu)$ *and* $\mu \leq_s \nu$ *fails*
vii) $T \leq_s T_1 =_{df} (T_1 \leq_s \mu)$ *if* $sT = \mu$
viii) $((T_1 \to T_2) \leq_s (T_1 \to T_2')) =_{df} (T_2 \leq_s T_2')$
ix) $(T \leq_s T') =_{df}$ *find* T'' *such that* $T \leq_s T''$ *and* $T'' \leq_s T'$

T'' is found by the call *subsumed_by* T where *subsumed_by* is defined as item 5 in 7.4. Note here that we have used the concept *subsumed_by* to accommodate the transitivity clause of Definition 3.8. In fact, *subsumed_by* accommodates transitivity through clause ix) of Definition 4.8.

Example 4.9 *subsume takes three arguments, the type substitution, and the two types to be compared. For instance,*

1. *subsume id_subst e p* $= N(''No'')$ *from line 4 of the implementation of subsume.*

2. *If* $(\alpha : p) \in$ *phi then subsume phi* α *e* $= Y(phi)$*, from line 3 of the implementation of subsume.*

Definition 4.10 *We say that a polytype* T *which contains type variables is* cyclic *according to a type substitution* s *iff*
1) $sT \not\equiv T$
2) $sT \leq_s T$
This notion of a cyclic type *is implemented as item 14 in 7.4.*

Example 4.11 β *is cyclic according to* $(\beta, \beta \to \beta_1)$.

Now we define unification of types as follows:

Definition 4.12 *(Unification)*
i) $\mu_1 \approx_s \mu_2 = s$ *if* $\mu_1 \leq_s \mu_2$
ii) $\beta \approx_s T = s[T/\beta]$ *if* $s\beta = \beta$ *and (cyclic* sT*)*
iii) $\beta \approx_s T = s[sT/\beta]$ *if* $s\beta = \beta$ *and* $sT \equiv T$
iv) $\beta \approx_s T = s\beta \leq_s sT$
v) $T \approx_s \beta = sT \leq_s s\beta$
vi) $((T_1 \to T_2) \approx_s (T_3 \to T_4)) = (T_2 \approx_{s1} T_4)$ *where* $s_1 = T_1 \approx_s T_3$

The ML function for this unification is to be found as item 11 in 7.7.

Example 4.13 $(unify\, id_subst\, (\beta, \beta \to \beta'))$ *returns* $Y(id_subst[\beta \to \beta'/\beta])$*, from clause ii) of Definition 4.12. In other words when you unify* β *with* $\beta \to \beta'$ *in the identity substitution, you succeed (you obtain the* Y *part of the type option) and you obtain a substitution which is exactly the same as id_subst except that for* β *it gives* $\beta \to \beta'$*.*

4.3 Type checking the expressions

An important concept for typechecking the expressions of the type free λ-calculus with logic is that of *circular type.* This is implemented as item 15 of 7.4, and it can be formally defined as follows:

Definition 4.14 *(Circular Type)*
We say that a type $(T \to T') \to T''$ is circular iff:

1. *T' and T'' are both monotypes.*
2. *$T' \leq p$ and $T'' \leq p$.*

Example 4.15 *$(\beta \to p) \to t$ and $(e \to p) \to (p \to p)$ are circular types.*

We are ready now to describe our type checking algorithm which will be implemented in 7.9. This algorithm will start from the rules given in Definition 3.9, but takes also into account logic, subsumption and unification of types and our concept of circular types which avoids the paradoxes. The notation $\Gamma \vdash E : T$ means that from the environment Γ, we can deduce that the expression E has type T. The following rules associate types to expressions, however they are supposed to be understood in a procedural way, that is (16) is tried first then (17) and so on. Also when we invoke $\Gamma \vdash a_1, \Gamma \vdash a_2$, then it is to be understood that $\Gamma \vdash a_1$ is executed first and if it succeeds then $\Gamma \vdash a_2$ is invoked but where Γ has been changed as a result of $\Gamma \vdash a_1$. All rules have the form

$$\frac{hypothesis\ h_1, h_2, \ldots, h_n}{conclusion\ C} \tag{15}$$

and if we are at rule R_i testing its hypothesis, $h_1, h_2, \ldots$, hn and one of the h_i fails, we abandon R_i and go to R_{i+1} but all changes to the environment which happened during execution of $h_1, h_2, \ldots, h_n$ are now undone. Now equations (16), …, (29) explain how the typechecker as implemented in *checkexpr* (item 1 of 7.9) has been derived. Basically we start from equations (11), …, (14) and accommodate *logic*, *subsumption* and *unification of types* and *reduction of terms.* Also we must use our notation of circular type to avoid the paradoxes. Note that *checkexpr* takes 3 arguments, the environment in which the expression must be checked, the first free variable from the heap and the expression to be type checked. Now we go to equations (11), …, (14), and expand them in an algorithm upon which the implementation of the type checker will be based. Equations (16), …, (23) will be the replacement of equations (11), …, (14). I.e. equations which accommodate circular types, subsumption and unification in the usual typing schemes. Equations (24), …, (29) accommodate the logical types. Here are these equations, their relation to equations (11), …, (14) and to their implementation in *checkexpr.*

$$\frac{(V : T) \in \Gamma}{\Gamma \vdash V : T} \tag{16}$$

As we see, equation (11) remains unchanged and this is implemented as clause 2 of *checkexpr.* Clause 1 of *checkexpr* implements that the type of *bot* (the bottom element $\bot$) is p.

$$\frac{\begin{array}{c}\Gamma \vdash \lambda V.E_1 : T \to T'', \Gamma \vdash E_2 : T', \Gamma \vdash ct(T'),\\ \Gamma \vdash ct(T \to T''), \Gamma \vdash T' \leq T, \Gamma \vdash reduce((\lambda V.E_1)E_2) : T'''\\ If((\lambda V.E_1)E_2) \text{ is not a subexpression of } reduce((\lambda V.E_1)E_2)\end{array}}{\Gamma \vdash ((\lambda V.E_1)E_2) : T'''} \tag{17}$$

$$\frac{\Gamma \vdash \lambda V.E_1 : T \rightarrow T'', \Gamma \vdash E_2 : T', \Gamma \vdash unify\, T'\, T}{\Gamma \vdash ((\lambda V.E_1)E_2) : T''} \tag{18}$$

The above two equations typecheck terms of the form $((\lambda V.E_1)E_2)$. The first equation deals with the case where both types of $\lambda V.E_1$ and E_2 are constant types, and where the result of $((\lambda V.E_1)E_2)$ has a more specific type than that of the range of $\lambda V.E_1$. The resulting type is the more specific one rather than the general one. The second equation is used in case it is difficult to calculate the more specific type, then the most general one, (the range of $\lambda V.E_1$) is given. These two equations are implemented as clause 3 of *checkexpr*. Note here that the 2 equations might not sound so compatible with one unique clause. All the other details however, such as *ct, reduce* and *subexpression* are tested inside the calls *check_list, list_types* and so on.

$$\frac{\begin{array}{c}\Gamma \vdash \lambda V : T1.E_1 : T \rightarrow T'', \Gamma \vdash E_2 : T', \Gamma \vdash ct(T'), \Gamma \vdash ct(T \rightarrow T''), \\ \Gamma \vdash T' \leq T, \Gamma \vdash reduce((\lambda V.E_1)E_2) : T''' \\ If((\lambda V.E_1)E_2) \text{ is not a subexpression of } reduce((\lambda V.E_1)E_2)\end{array}}{\Gamma \vdash ((\lambda V : T1.E_1)E_2) : T'''} \tag{19}$$

$$\frac{\Gamma \vdash \lambda V : T1.E_1 : T \rightarrow T'', \Gamma \vdash E_2 : T', \Gamma \vdash unify\, T'T}{\Gamma \vdash ((\lambda V : T1.E_1)E_2) : T''} \tag{20}$$

These two equations are similar to (17) and (18) but where the abstracted variable is typed. They are implemented as clause 4 of *checkexpr*.

$$\frac{\Gamma \vdash E_1 : T, \Gamma \vdash E_2 : T', \Gamma \vdash unify\, T\, (T' \rightarrow \beta))}{\Gamma \vdash E_1 E_2 : \beta} \tag{21}$$

This equation deals with the case where the first term E_1 does not have the form of a λ term. For example, in xx, the first x is not a λ term, yet we would like to apply it to the second x. In this case, the first term, is given an arrow type and everything is made to fit. This equation is implemented as clause 5 of *checkexpr*.

Note that we take 5 equations, (17), ..., (21) to accommodate equation (13).

$$\frac{(V : \beta) \cup \Gamma \vdash E : T'\, if\, (\beta \rightarrow T') \text{ non-circular in } \Gamma}{\Gamma \vdash \lambda V.E : \beta \rightarrow T'} \tag{22}$$

$$\frac{(V : T) \cup \Gamma \vdash E : T'\, if\, (T \rightarrow T') \text{ is non-circular in } \Gamma}{\Gamma \vdash \lambda V : T.E : T \rightarrow T'} \tag{23}$$

These two equations replace equation (14). Equation (22) deals with the case where the abstracted variable is untyped and equation (23) deals with the case where the abstracted variable is typed. Those two equations are implemented as clauses 6 and 7 of *checkexpr*.

$$\frac{\Gamma \vdash \lambda V.E : T, \Gamma \vdash unify\, T\, p}{\Gamma \vdash \forall V.E : p} \tag{24}$$

$$\frac{\Gamma \vdash \lambda V : T.E : T', \Gamma \vdash unify\, T'\, p}{\Gamma \vdash \forall V : T.E : p} \tag{25}$$

The above two equations typecheck forall terms, by first typechecking a λ term which corresponds to it and unifying the type of the λ term with p. They are implemented as clauses 9 and 8 of *checkexpr* respectively.

$$\frac{\Gamma \vdash E : T, \Gamma \vdash unify\; T\; p}{\Gamma \vdash \Omega E : p} \tag{26}$$

$$\frac{\Gamma \vdash E : T, \Gamma \vdash unify\; T\; p}{\Gamma \vdash \neg E : p} \tag{27}$$

$$\frac{\Gamma \vdash E_1 : T_1, \Gamma \vdash unify\; T_1\; p, \Gamma \vdash E_2 : T_2, \Gamma \vdash unify\; T_2\; p}{\Gamma \vdash E_1 \wedge E_2 : p} \tag{28}$$

$$\frac{\Gamma \vdash E_1 : T_1, \Gamma \vdash unify\; T_1\; p, \Gamma \vdash E_2 : T_2, \Gamma \vdash unify\; T_2\; p}{\Gamma \vdash E_1 \rightarrow E_2 : p} \tag{29}$$

The above four equations are now obvious. They are implemented as clauses 10, ..., 13 respectively.

Example 4.16 *Now let us see how $\lambda x.xx$ is type checked by the system. In summary the method is as follows:*

(i) $[x : \alpha_0]$ *hyp*
(ii) $\alpha_0 \approx \alpha_0 \rightarrow \alpha_1$ *From unification*
(iii) $xx : \alpha_1$ *From (i), (ii), (21)*
(iv) $\lambda x.xx : (\alpha_0 \rightarrow \alpha_1) \rightarrow \alpha_0$ *From (22)*

The system however, when asked to typecheck $\lambda x.xx$ (by calling typecheck $[\lambda x.xx]$), will follow the steps below (note that check-list $[("x", \alpha_0)][x; x]\alpha_1 =$ Y(id-subst, $[\alpha_0; \alpha_0], \alpha_1)$ and that unify id-subst $(\alpha_0, \alpha_0 \rightarrow \alpha_1) =$ Y(id-subst $[\alpha_0 \rightarrow \alpha_1/\alpha_0]$))

1. checks $[\lambda x.xx]$ [] α_0 1

1.1 checkexpr [] α_0 $\lambda x.xx$

1.1.1 typecheckbodyabs α_0 (checkexpr $[("x", \alpha_0)]$ α_1 xx)

To checkexpr $[("x", \alpha_0)]$ α_1 xx, one has to typecheckapp (check-list $[("x", \alpha_0)]$ $[x; x]$ α_1). I.e. typecheckapp (Y(id-subst, $[\alpha_0; \alpha_0], \alpha_1$)) which is

typecheckapp1 α_1 (unify id-subst $(\alpha_0, \alpha_0 \rightarrow \alpha_1)$). This is

typecheckapp1 α_1 Y(id-subst$[\alpha_0 \rightarrow \alpha_1/\alpha_0]$) which is Y(id-subst$[\alpha_0 \rightarrow \alpha_1/\alpha_0], \alpha_1, \alpha_2$).

Now, typecheckbodyabs α_0 (checkexpr $[("x", \alpha_0)]\alpha_1 xx$) returns $(\alpha_0 \rightarrow \alpha_1) \rightarrow \alpha_0$, the type of $\lambda x.xx$.

Example 4.17 *Y is type checked by the system as follows*

(i)$[f : \alpha_0]$ *hyp*
(ii) $[x : \alpha_1]$ *hyp*
(iii) $\alpha_1 \approx \alpha_1 \rightarrow \alpha_2$ *From unification*
(iv) $xx : \alpha_2$ *From (ii), (iii), (17)*
(v) $\alpha_0 \approx \alpha_2 \rightarrow \alpha_3$ *From unification*
(vi) $f(xx) : \alpha_3$ *From (i), (iv), (v), (21)*
(vii) $\lambda x.f(xx) : (\alpha_1 \rightarrow \alpha_2) \rightarrow \alpha_3$ *From (ii) ... (vi), (22), unification*
(viii) $[x : \alpha_4]$ *hyp*
(ix) $\alpha_4 \approx \alpha_4 \rightarrow \alpha_5$ *From unification*
(x) $xx : \alpha_5$ *From (viii), (ix), (21)*

(xi) $\alpha_5 \approx \alpha_2$ *From unification*
(xii) $xx : \alpha_2$ *From (x), (xi), unification*
(xiii) $f(xx) : \alpha_3$ *From (i), (xii), (v), (21)*
(xiv) $\lambda x.f(xx) : (\alpha_4 \to \alpha_2) \to \alpha_3$ *From (viii) ... (xiii), (22), unification*
(xv) $\alpha_3 \approx \alpha_2$ *From unification*
(xvi) $(\alpha_4 \to \alpha_2) \to \alpha_3 \approx \alpha_1 \to \alpha_2$ *From unification*
(xvii) $(\lambda x.f(xx))(\lambda x.f(xx)) : \alpha_2$ *From (xv), (xvi), (21)*
(xviii) $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) : (\alpha_2 \to \alpha_2) \to \alpha_2$ *From (i) ... (xvii),*

Example 4.18 *The following will give a feel of how the system works. They are examples of what expressions we give the system and what messages or types we get back.*

Expressions	**Types**
1 $\lambda x.x$	$\alpha_0 \to \alpha_0$
2 $\lambda x : e.x$	$e \to e$
3 $\lambda x.xx$	$(\alpha_0 \to \alpha_1) \to \alpha_1$
4 $(\lambda x.xx)(\lambda x.xx)$	α_1
5 $\lambda x : p.xx$	$p \to \alpha_0$
6 $\lambda x : e \to p.xx$	*error:* $(e \to p) \to p$ *is circular*
7 $\forall x : (\alpha_0 \to \alpha_1).xy$	p
8 $\forall x : e.x$	*error, not a proposition*
9 $\forall x : (e \to \alpha_1).xy$	p
10 $\forall x.xx$	p
11 $\lambda x : (\alpha_0 \to \alpha_1).xy$	$(\alpha_0 \to \alpha_1) \to \alpha_1$
12 $\lambda f.(\lambda s : e \to pf(ss))(\lambda s : e \to pf(ss))$	*error:* $(p \to p) \to p$ *is circular*
13 $\lambda f : e \to p.(\lambda s : e \to pf(ss))(\lambda s : e \to pf(ss))$	*error:* $(e \to p) \to p$ *is circular*
14 $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$	$(\alpha_2 \to \alpha_2) \to \alpha_2$
15 $(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))(\lambda x : p.xx)$	p
16 $(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))$	α_2
17 $(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))(\lambda x.xx)$	α_2
18 $(\lambda x.xx)(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))$	α_1
19 $\lambda x.\neg xx$	*error, circular type*
20 $\lambda x : (\alpha_0 \to t).\neg xx$	*error, circular type*
21 $\lambda x : (\alpha_0 \to p).\neg xx$	*error, circular type*
22 $\lambda x.xx \to \bot$	*error, circular type*

5 Theorem proving in the system

Now let us see how the paradoxical sentences do not lead us to problems. Take the following paradoxical sentences:

Let $Russell = \lambda x.\neg xx$
And $AnotherRussell = \lambda x : (a_0 \to t).\neg xx$
And $TypedRussell = \lambda x : (a_0 \to p).\neg xx$
And $Curry = \lambda x.xx \to \bot$

typecheck x where x is any of the above terms returns: an error message informing us that the term has a circular type. So the system does not allow the typing of the paradoxical sentences. However as we have seen in the section on polymorphism above, the system allows and typechecks all self referential terms which are safe. I.e. whereas the system typecheks

$\lambda x.xx$, it does not allow $\lambda x.\neg xx$. This is because it knows that for $\neg$ to make sense, it should apply to a proposition but it cannot make xx be a proposition.

It might be thought that this theory would fall foul of Russell's paradox, due to the fact that xx is a well-formed formula for x of any type $T_1 \to T_2$; and hence by abstracting over $\neg xx$, we could obtain $aa = \neg aa$ where a is $\lambda x.\neg xx$. In particular, if one took x to be of type $e \to p$, then $a = \lambda x.\neg xx$ would be of type $(e \to p) \to p$ and hence aa would be of type p, leading to a contradiction from the above equality. The careful reader however would realise that one of our above steps was wrong. That is, even if x is of type $e \to p$, and even though $\neg xx$ is a proposition, $\lambda x.\neg xx$ is not well-formed. More specifically, its type, $(e \to p) \to p$, is circular. In fact we have a more general result: the paradox does not arise for x of any type $T \to p$. This follows from the following lemma:

Lemma 5.1 *If x is of type $T \to p$, then $\lambda x.\neg xx$ of type $(T \to p) \to p$ is not well-formed.*

Proof:

(i) $x : T \to p$ *hyp*

(ii) $T \to p \leq p$ *from* $\leq$

(iii) $xx : p$ *from (21)*

(iv) $\neg xx : p$ *from (27)*

But as $(T \to p) \to p$ is circular, we cannot apply (23) to get that $\lambda x.\neg xx$ has type $(T \to p) \to p$. In fact we cannot type $\lambda x.\neg xx$. The system comes back and tells us that the type is circular (see term 19 of Example 4.18). □

This might still be unpersuasive however, for the paradox can arise in other ways. For example, take x of type $T \to T'$, where $T' \leq p$. Then xx is of type $T' \leq p$, hence $\neg xx$ is of type p. Now, if $\lambda x.\neg xx$ is a well-formed expression (call it a) then aa is of type p and is equal to aa. Contradiction. In view of this, we have to prove something stronger than Lemma 5.1. This we do via the following lemma:

Lemma 5.2 *If x is of type $T \to T'$, where $T' \leq p$, then $\lambda x.\neg xx$ of type $(T \to T') \to p$ is not well-formed.*

Proof:

(i) $x : T \to T'$ *hyp*

(ii) $T \to T' \leq T$ *from* $\leq$

(iii) $xx : T'$ *from (21)*

(iv) $\neg xx : p$ *from (27), as* $T' \leq p$

But as $(T \to T') \to p$ is circular, we cannot apply (23) to get that $\lambda x.\neg xx$ has type $(T \to T') \to p$. In fact we cannot type $\lambda x.\neg xx$. The system comes back and tells us that the type is circular (see terms 20 and 21 of Example 4.18). □

Up to here, we have only used the type p to express logic, and t has been ignored. We shall show here how the type t is used and demonstrate the idea by showing that we do not face Curry's paradox.

Our version of the Deduction Theorem (DT) has the following form:

(DT) $\Gamma \cup \Phi : t \vdash \Psi : t$ implies $\Gamma \cup \Phi : p \vdash (\Phi \to \Psi) : t$

Modus Ponens (MP) has also the following form:

(MP) $\Gamma \vdash (\Phi \to \Psi) : t$ and $\Gamma \vdash \Phi : t$ implies $\Gamma \vdash \Psi : t$,

If we take a to be the formula

$\lambda x.(xx \to \bot)$,

then by β-conversion,

(1) $aa = aa \to \bot$.

Now, it holds trivially that

(2) $aa : t \vdash aa : t$,

By (1) we derive

(3) $aa : t \vdash aa \to \bot : t$.

and, by Modus Ponens applied to (2) and (3) we get

$aa : t \vdash \bot : t$.

By (DT) we can now derive $aa : p \vdash (aa \to \bot) : t$.

Then also $aa : p \vdash aa : t$.

Given the last two steps, we can again apply Modus Ponens to get

$aa : p \vdash \bot : t$.

However, we cannot show that $aa : p$. In fact $\lambda x.(xx \to \bot)$ is not well formed due to lemma 5.1 above as its type is $(T' \to p) \to p$. This is because if x is of some type T, since xx has to be of type p, we can infer that T must be of the form $T' \to p$. From this it follows that a is of type $(T' \to p) \to p$, which is circular. Hence we do not face Curry's paradox.

This is all the proof theory that we mention about this system in this paper, for more results and properties about the logical properties and the proof theory of the system refer to [Kamareddine 92A]. Also [Kamareddine 92B] and [Kamareddine 92C] present a model of the system together with other systems of the type free λ-calculus with logic.

6 Conclusion

The system provided in this paper has powerful properties. First it is type free. That is, anything structured is an expression and anything non problematic will have a type. These types are polymorphic in the sense that expressions can have many variable types and these variable types may be instantiated to anything. For example, the identity function has type $\alpha_0 \to \alpha_0$, and the identity function applied to objects of type e will result in elements of type e. The polymorphic power of the system comes from the ability to typecheck all polymorphic functions even those which are problematic in other systems. For example the fixed point operator, $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ is typechecked to $(\alpha_2 \to \alpha_2) \to \alpha_2$ and even can apply to itself. Even YY is typechecked to α_2. $f = \lambda x.xx$ is also typechecked to $(\alpha_1 \to \alpha_1) \to \alpha_1$ and f applied to itself is typechecked to α_1. As said earlier, these types can be instantiated so that gg where g is the identity function over e (i.e. $g = \lambda x : e.x$), is typechecked to e naturally. We believe this system is one of the first which can typecheck all the above while remaining a very expressive and simple one. Other polymorphic systems like ML, do not have this polymorphic power. In fact, Y cannot be typechecked in ML. Instead, the fixed point operator is defined trivially by the equation: letrec $YE = E(YE)$, and then this Y is typechecked to $(\alpha_2 \to \alpha_2) \to \alpha_2$. But this is not good enough as one cannot define Y by its λ-expression. Another nice characteristic of the system is its ability to combine logic and the type free λ-calculus while remaining consistent. So even though the Russell sentence $(\lambda x.\neg(xx))$ is a well formed sentence of the system, its type cannot be found. In fact, the system returns an error message explaining that this sentence has a circular type.

The same thing applies to the Curry's sentence ($\lambda x.xx \to \bot$). Of course here, one may wonder if the paradox is really avoided, and may give as an example $F \equiv \lambda f.(\lambda x.f(xx))$ which is typechecked to $(\alpha_2 \to \alpha_2) \to ((\alpha_2 \to \alpha_2) \to \alpha_2)$, and then instantiate it to $F\neg$ which would be of type $(p \to p) \to p$. This does not hold however because $(p \to p) \to p$ is circular and the system does not accept such instantiation. Finally, the system also has error messages which convey the reasons of failure in typechecking and where the failure occured.

7 Program listing

module infixr ”~~~”;
export typevar, tterm, ob, pr, tr, show_type, show_tlist, subsume, ctsubsume, circulartype, change, mysub_type, equaltype, subsumeset, makearrows, subsumed_by, isbasictype, isarrow, domain, range, istlambda, term, free, out, substitute, isin, newvar, rename, show_term, len, betaconverge, etaconverge, alphaconverge, reduceoutermost, reduce, occur, subexpression, equiv, islambda, isinnf, betaconverts, etaconverts, isapp,hasnf, subexp, nodupappend, operator, operand, propconj, propneg, propimpl, propbot, anothereq, cyclictype, fvars, occurs, mem, zip, option, next, ~~~, getsub, gettype, gettvn, print, printerror, iserror, istvar,lookupYN, sub_type, scomp, id_subst, delta, extend, unify, unify_list,addrem, makeprop, composesubs, app_sub_env, typecheckapp, seeprop, occurtype, typecheckbodyabs, checkexpr, check_list, listtypes, typecheckprop, getphi, typechecklapp, checkexpr, checks, typecheck, typecheckapp1;

7.1 Terms, Types and Options Declarations

1. rec type typevar = alpha Int
2. and type tterm = tvar typevar + top (List Char) (List tterm)
3. and ob = top ”OB” []
4. and pr = top ”PR” []
5. and tr = top ”TR” []
6. and type term = bot + var (List Char) + app term term + lambda (List Char) term + tlambda (List Char) tterm term + prop term + conj term term + neg term + impl term term + forall (List Char) term + tforall (List Char) tterm term
7. and type option *a *b = N*a +Y*b

7.2 Printing

1. and show_type (tvar (alpha x)) = itos x
   || show_type (top s l) = if s = ”arrow” then show_tlist l else s
2. and show_tlist ([t1;t2]) = ”(”@show_type t1@” ”@”-”@” ≻ ”@” ”@show_type t2 @”)”@” ”
3. and show_term bot = ”bot”
   || show_term (var v) = ”(var ”@v@”)”
   || show_term (app E E’) = ”(app ”@show_term E@” ”@show_term E’@”)”
   || show_term (lambda v E) = ”(lambda ”@v@” ”@show_term E@”)”
   || show_term (tlambda v t E) = ”(tlambda ”@v@” ”@ show_type t @” ”@show_term E@”)”
   || show_term (conj E E’) = ”(conj ”@show_term E@” ”@show_term E’@”)”
   || show_term (neg E) = ”(neg ”@show_term E@”)”
   || show_term (impl E E’) = ”(imply ”@show_term E@” ”@show_term E’@”)”

```
|| show_term (prop E) = "(prop "@show_term E@")"
|| show_term (forall v E) = "(forall "@v@" "@show_term E@")"
|| show_term (tforall v t E) = "(tforall "@v@" "@ show_type t @" "@show_term E@")"
```

7.3 Properties of terms

1. and len bot =1
   || len (var v) = 1
   || len (app E E') = (len E) + (len E')
   || len (lambda v E) = 1+ (len E)
   || len (tlambda v t E) = 1+ (len E)
   || len (prop E) = (len E)
   || len (conj E E') = (len E) + (len E')
   || len (impl E E') = (len E) + (len E')
   || len (neg E) = (len E)
   || len (forall v E) = 1+ (len E)
   || len (tforall v t E) = 1 + (len E)
2. and occur E E' & (equiv E E') = 1
   ||occur E (app E1 E2) = (occur E E1) + (occur E E2)
   || occur (var v') (lambda v E1) & (v = v') = 1+ (occur (var v') E1)
   || occur E (lambda v E1) = (occur E E1)
   || occur (var v') (tlambda v t E1) & (v = v') = 1+ (occur (var v') E1)
   || occur E (tlambda v t E1) = (occur E E1)
   || occur E (prop E') = (occur E E')
   || occur E (conj E1 E2) = (occur E E1) + (occur E E2)
   || occur E (impl E1 E2) = (occur E E1) + (occur E E2)
   || occur E (neg E') = (occur E E')
   || occur (var v') (forall v E1) & (v = v') = 1+ (occur (var v') E1)
   || occur E (forall v E1) = (occur E E1)
   || occur (var v') (tforall v t E1) & (v = v') = 1+ (occur (var v') E1)
   || occur E (tforall v t E1) = (occur E E1)
   || occur E E' = 0
3. and free bot = []
   || free (var v) = [v]
   || free (app E E') = (free E) @ (free E')
   || free (lambda v E) = out v (free E)
   || free (tlambda v t E) = out v (free E)
   || free (conj E E') = (free E) @ (free E')
   || free (impl E E') = (free E) @ (free E')
   || free (prop E) = (free E)
   || free (neg E) = (free E)
   || free (forall v E) = out v (free E)
   ||free (tforall v t E) = out v (free E)

4. and subexpression E E' & (equiv E E') = true
   ||subexpression E (app E1 E2) = (subexpression E E1) | (subexpression E E2)
   ||subexpression E (conj E1 E2) = (subexpression E E1) | (subexpression E E2)
   ||subexpression E (impl E1 E2) = (subexpression E E1) | (subexpression E E2)
   ||subexpression E (prop E1) = (subexpression E E1)
   ||subexpression E (neg E1) = (subexpression E E1)
   || subexpression (var v') (lambda v E1) = (v' = v) | (subexpression (var v') E1)
   || subexpression (var v') (tlambda v t E1) = (v' = v) | (subexpression (var v') E1)
   ||subexpression E (lambda v E1) = (subexpression E E1)
   ||subexpression E (tlambda v t E1) = (subexpression E E1)
   ||subexpression E (forall v E1) = (subexpression E E1)
   || subexpression E (tforall v t E1) = (subexpression E E1)
   || subexpression E E' = false
5. and equiv bot bot = true
   || equiv (var v) (var v') = (v = v')
   ||equiv (app E1 E2) (app E'1 E'2) = ((equiv E1 E'1) & (equiv E2 E'2))
   ||equiv (conj E1 E2) (conj E'1 E'2) = ((equiv E1 E'1) & (equiv E2 E'2))
   ||equiv (impl E1 E2) (impl E'1 E'2) = ((equiv E1 E'1) & (equiv E2 E'2))
   ||equiv (prop E1) (prop E'1) = (equiv E1 E'1)
   ||equiv (neg E1) (neg E'1) = (equiv E1 E'1)
   ||equiv (lambda v E) (lambda v' E') = ((v = v') & (equiv E E'))
   ||equiv (tlambda v t E) (tlambda v' t' E') = ((v = v') &(t =t') & (equiv E E'))
   ||equiv (forall v E) (forall v' E') = ((v = v') & (equiv E E'))
   ||equiv (tforall v t E) (tforall v' t' E') = ((v = v') & (equiv E E'))
   ||equiv E E' = false
6. and subexp bot = [bot]
   || subexp (var v) = [(var v)]
   || subexp (app E1 E2) = (app E1 E2).(nodupappend (subexp E1) (subexp E2))
   || subexp (lambda v E1) = (nodupappend [(var v)] ((lambda v E1).(subexp E1)))
   || subexp (tlambda v t E1) = (nodupappend [(var v)] ((tlambda v t E1).(subexp E1)))
   || subexp (conj E1 E2) = (conj E1 E2).(nodupappend (subexp E1) (subexp E2))
   || subexp (impl E1 E2) = (impl E1 E2).(nodupappend (subexp E1) (subexp E2))
   || subexp (prop E) = (prop E). (subexp E)
   || subexp (neg E) = (neg E). (subexp E)
   || subexp (forall v E1) = (nodupappend [(var v)] ((forall v E1).(subexp E1)))
   || subexp (tforall v t E1) = (nodupappend [(var v)] ((tforall v t E1).(subexp E1)))
7. and islambda (lambda v E) = true
   || islambda other = false
8. and istlambda (tlambda v t E) = true
   || istlambda other = false
9. and isapp (app E E') = true
   || isapp other = false
10. and operator (app E E') = E
11. and operand (app E E') = E'

7.4 Properties of Types

1. and subsumeset x & (x = ob) = [ob]
   || subsumeset x & (x = pr) = [ob; pr]
   || subsumeset x & (x= tr) = [ob; pr; tr]
   || subsumeset x & (istvar x) = [x]
   || subsumeset x & (isarrow x) = makearrows (domain x) (subsumeset (range x))
2. and makearrows x [] = []
   || makearrows x (y.ys) = (top ”arrow” [x;y]).(makearrows x ys)
3. and ctsubsume x y & (~(occurs x) & ~(occurs y)) = isin x (subsumeset y)
   || ctsubsume x y = false
4. and subsume phi x y & ((ctsubsume x y) | ((isarrow x) & ((domain x) = y))|(x = y)) = Y(phi)
   ||subsume phi (tvar x) y = Y(scomp (addrem id_subst x y) phi)
   ||subsume phi x y & (ctsubsume (sub_type phi x) (sub_type phi y)) = Y(phi)
   ||subsume phi x y & ((~(occurs (sub_type phi x)))&(~(occurs (sub_type phi y)))) = N (”no”)
   ||subsume phi x y & (~(occurs (sub_type phi x))) = subsume phi y x
   ||subsume phi x y = if ((isarrow x) & (isarrow y) & ((domain x) = (domain y)) &
   (~(iserror (subsume phi (range x) (range y))))) then (subsume phi (range x) (range y)) else
   if(~(iserror (subsumed_by (sub_type phi x)))) then subsume phi (getsub (subsumed_by x)) y
   else
   N (”Cannot unify : ”@”types mismatch”@”n”@show_type x @ ”n”@show_type y@”n”)
5. and subsumed_by x & (x = tr) = Y(pr)
   || subsumed_by x & (x = pr) = Y(ob)
   || subsumed_by (top ”arrow” l) = Y(hd l)
   || subsumed_by x = N ”error”
6. and isbasictype x = x = ”OB” | x = ”PR” | x = ”TR”
7. and isarrow (top ”arrow” l) = true
   || isarrow other = false
8. and istvar (tvar x) = true
   ||istvar other = false
9. and domain (top ”arrow” (x.xs)) = x
10. and range (top ”arrow” [x;y]) = y
11. and occurs (tvar x) = true
    ||occurs (top s tlist) = exists (occurs) tlist
12. and fvars (tvar x) = [x]
    || fvars (top s l) = concmap fvars l
13. and equaltype x y =
    x = y | ~(iserror (subsume id_subst x y)) &~(iserror (subsume id_subst y x))
14. and cyclictype phi t =
    (occurs t) & ~((sub_type phi t) = t) & ~(iserror(subsume phi (sub_type phi t) t))
15. and circulartype (top ”arrow” [(top ”arrow” [t’; t1]);t2]) & (~(occurs t1) & ~(occurs t2)) =
    (~(iserror(subsume id_subst t1 pr)) & ~(iserror(subsume id_subst t2 pr)))
    || circulartype other = false

7.5 Substitution of Terms

1. and substitute bot E' v = bot
   || substitute (var v') E' v & (v = v') = E'
   || substitute (var v') E' v = var v'
   || substitute (app E1 E2) E' v = app (substitute E1 E' v) (substitute E2 E' v)
   || substitute (lambda v' E1) E' v & (v = v') = lambda v' E1
   || substitute (tlambda v' t E1) E' v & (v = v') = tlambda v' t E1
   || substitute (tforall v' t E1) E' v & (v = v') = tforall v' t E1
   || substitute (lambda v' E1) E' v & (~(isin v' (free E')) | ~(isin v (free E1))) =
   (lambda v' (substitute E1 E' v))
   || substitute (tlambda v' t E1) E' v &(~(isin v' (free E')) | ~(isin v (free E1))) =
   (tlambda v' t (substitute E1 E' v))
   || substitute (tforall v' t E1) E' v &(~(isin v' (free E')) | ~(isin v (free E1))) =
   (tforall v' t (substitute E1 E' v))
   || substitute (lambda v' E1) E' v & ((isin v' (free E')) & (isin v (free E1))) =
   let new_var = newvar v' (free (app E' E1)) in
   (lambda new_var (substitute (substitute E1 (var new_var) v') E' v))
   || substitute (tlambda v' t E1) E' v & ((isin v' (free E')) & (isin v (free E1)))=
   let new_var = newvar v' (free (app E' E1)) in
   (tlambda new_var t (substitute (substitute E1 (var new_var) v') E' v))
   || substitute (tforall v' t E1) E' v & ((isin v' (free E')) & (isin v (free E1))) =
   let new_var = newvar v' (free (app E' E1)) in
   (tforall new_var t (substitute (substitute E1 (var new_var) v') E' v))
   || substitute (prop E) E' v = prop (substitute E E' v)
   || substitute (conj E1 E2) E' v = conj (substitute E1 E' v) (substitute E2 E' v)
   || substitute (impl E1 E2) E' v = impl (substitute E1 E' v) (substitute E2 E' v)
   || substitute (neg E) E' v = neg (substitute E E' v)
   || substitute (forall v' E1) E' v & (v = v') = forall v' E1
   || substitute (forall v' E1) E' v & (~(isin v' (free E')) | ~(isin v (free E1))) =
   (forall v' (substitute E1 E' v))
   || substitute (forall v' E1) E' v &((isin v' (free E')) & (isin v (free E1))) =
   let new_var = newvar v' (free (app E' E1)) in
   (forall new_var (substitute (substitute E1 (var new_var) v') E' v))

7.6 REDUCTION OF TERMS

1. and betaconverge (app (lambda v E) E') = (true, substitute E E' v)
   || betaconverge (app (tlambda v t E) E') = (true, substitute E E' v)
   || betaconverge other = (false, other)
2. and etaconverge (lambda v (app E E'))& ((E' = (var v)) & ~(isin v (free E))) = (true, E)
   || etaconverge (tlambda v t (app E E'))& ((E' = (var v)) & ~(isin v (free E))) = (true, E)
   || etaconverge other = (false, other)

3. and alphaconverge (lambda v E) = (true, let new_var = newvar (rename v) (free E) in
(lambda new_var (substitute E (var new_var) v)))
|| alphaconverge (tlambda v t E) = (true, let new_var = newvar (rename v) (free E) in
(tlambda new_var t (substitute E (var new_var) v)))
|| alphaconverge other = (false, other)
4. and reduce E = reduceoutermost (snd (etaconverge E))
5. and reduceoutermost bot = bot
|| reduceoutermost (var v) = (var v)
|| reduceoutermost (app E1 E2) & (islambda E1) = let E = (app E1 E2) in
let E' = (snd (betaconverge E)) in if (subexpression E E') then E' else (reduce E')
|| reduceoutermost (app E1 E2) & (istlambda E1) = let E = (app E1 E2) in
let E' = (snd (betaconverge E)) in if (subexpression E E') then E' else (reduce E')
|| reduceoutermost (app E1 E2) = let E' = (reduce E1) in
if (subexpression E1 E') then (app E1 (reduce E2)) else (reduce (app E' E2))
|| reduceoutermost (lambda v E) = (snd (etaconverge (lambda v (reduce E))))
|| reduceoutermost (tlambda v t E) = (snd (etaconverge (tlambda v t (reduce E))))
|| reduceoutermost (prop E) = prop (reduce (snd (etaconverge E)))
|| reduceoutermost (neg E) = neg (reduce (snd (etaconverge E)))
|| reduceoutermost (conj E E') =
conj (reduce (snd (etaconverge E))) (reduce (snd (etaconverge E')))
|| reduceoutermost (impl E E') =
impl (reduce (snd (etaconverge E))) (reduce (snd (etaconverge E')))
|| reduceoutermost (forall v E) = forall v (reduce (snd (etaconverge E)))
|| reduceoutermost (tforall v t E) = tforall v t (reduce (snd (etaconverge E)))
6. and anothereq E E' = (equiv (reduce E) (reduce E'))
7. and isinnf E = let E' = [E1;; E1 ← (subexp E)] in ((null (filter betaconverts E')) & (null (filter etaconverts E')))
8. and hasnf E = isinnf (reduce E)
9. and betaconverts E = fst (betaconverge E)
10. and etaconverts E = fst (etaconverge E)

7.7 Substitution and Unification of Types

1. and sub_type phi t = mysub_type phi t []
2. and mysub_type phi t l & (isin t l) = t
|| mysub_type phi (tvar tvn) l =
let a = phi tvn in if ((a = (tvar tvn)) | (a = ob) | (a = pr) | (a = tr)) then a else
if (istvar a) then (sub_type phi a) else
(top "arrow" [mysub_type phi (domain a) ((tvar tvn).l); mysub_type phi (range a) ((tvar tvn).l)])
|| mysub_type phi (top tcn l) l'= top tcn (map (\u.mysub_type phi u l') l)
3. and scomp sub2 sub1 tvn = sub_type sub2 (sub1 tvn)
4. and id_subst tvn = tvar tvn

5. and delta tvn t tvn1 = if tvn = tvn1 then t else tvar tvn1
6. and composesubs sub t (N w) = N w
   ||composesubs sub t (Y (sub',t',tvn)) = Y (scomp sub' sub,(sub_type sub' t).t',tvn)
7. and app_sub_env phi env = map ((x,y).(x,sub_type phi y)) env
8. and addrem phi tvn t tvn1 = if tvn = tvn1 then t else phi tvn1
9. and change phi tvn t tvn1 = sub_type (addrem phi tvn t) (tvar tvn1)
10. and extend phi tvn t = if t = tvar tvn then Y phi else Y (scomp (delta tvn t) phi)
11. and unify phi ((tvar tvn), t) = if cyclictype phi t then Y(addrem phi tvn t) else let rec phitvn = phi tvn and phit = sub_type phi t in if phitvn = tvar tvn then extend phi tvn phit else unify phi (phitvn, phit)

    ||unify phi ((top tcn ts), (tvar tvn)) = subsume phi (sub_type phi(top tcn ts))(sub_type phi (tvar tvn))

    || unify phi ((top s1 tlist1), (top s2 tlist2)) =

    if (~(occurs (top s1 tlist1))) & (~(occurs (top s2 tlist2))) then

    if ~(iserror(subsume phi (top s1 tlist1)(top s2 tlist2))) then Y phi else N ("Cannot unify : "@"types mismatch"@"n"@show_type (top s2 tlist2) @ "n"@show_type (top s1 tlist1)@"n")

    else unify_list phi (zip tlist1 tlist2)
12. and unify_list phi [] = Y phi

    || unify_list phi ((s,t).sts) = unify phi (s,t) ~~~ (\u.unify_list u sts)

7.8 Logic

1. and propconj (prop E1) (prop E2) = prop (conj E1 E2)
2. and propimpl (prop E1) (prop E2) = prop (impl E1 E2)
3. and propneg (prop E) = prop (neg E)
4. and propbot bot = prop bot

7.9 Type Checking

1. and checkexpr env tvn bot = Y(id_subst,pr,tvn)

   || checkexpr env tvn (var x) = let a = (lookupYN env x) in if (iserror a)

   then N (x@":"@(printerror a)) else Y (id_subst,getsub a,tvn)

   ||checkexpr env tvn (app (lambda x e) e1) =

   typechecklapp env (lambda x e) e1 (check_list env [(lambda x e); e1] tvn)

   ||checkexpr env tvn (app (tlambda x t e) e1) =

   typechecklapp env (tlambda x t e) e1 (check_list env [(tlambda x t e); e1] tvn)

   ||checkexpr env tvn (app e1 e2) = typecheckapp (check_list env [e1; e2] tvn)

   ||checkexpr env tvn (lambda x e) =

   let a =typecheckbodyabs (tvar tvn) (checkexpr ((x,tvar tvn).env) (next tvn) e) in

   if iserror a then a else if (circulartype (gettype a)) then N "circular type" else a

   ||checkexpr env tvn (tlambda x t e) = let a = typecheckbodyabs t (checkexpr ((x,t).env) tvn e) in if iserror a then a else if (circulartype (gettype a)) then N "circular type" else a

   ||checkexpr env tvn (tforall x t e) = typecheckprop(checkexpr ((x,t).env) tvn e)

||checkexpr env tvn (forall x e) = typecheckprop(checkexpr ((x,tvar tvn).env) (next tvn) e)
||checkexpr env tvn (prop e) = typecheckprop(checkexpr env tvn e)
||checkexpr env tvn (neg e) = typecheckprop(checkexpr env tvn e)
||checkexpr env tvn (conj e1 e2) =
let rec a = typecheckprop(checkexpr env tvn e1) in if (~(iserror a))
then let rec b = typecheckprop(checkexpr (app_sub_env (getphi a) env) (gettvn a) e2)
in if (~(iserror b)) then Y(scomp (getphi a)(getphi b), gettype b, gettvn b) else b else a
||checkexpr env tvn (impl e1 e2) =
let rec a = typecheckprop(checkexpr env tvn e1) in if (~(iserror a)) then
let rec b = typecheckprop(checkexpr (app_sub_env (getphi a) env) (gettvn a) e2) in
if (~(iserror b)) then Y(scomp (getphi a)(getphi b), gettype b, gettvn b) else b else a

2. and typecheckprop a = if (~(iserror a)) then
   let b = (gettype a) in if occurs b then makeprop (seeprop (getphi a) b) (gettvn a) b else
   if (iserror(subsume id_subst b pr)) then N ”not a proposition” else a else a

3. and typechecklapp env e1 e2 (N w) = N w
   ||typechecklapp env e1 e2 (Y (phi,[t1;t2],tvn)) =
   if ~(occurs t2) & ~(occurs (domain t1)) & ~(iserror(subsume id_subst t2 (domain t1)))
   & ~(subexpression (app e1 e2) (reduce (app e1 e2))) then
   checkexpr env tvn (reduce (app e1 e2)) else
   if (occurs t2 | occurs (domain t1)| (subexpression (app e1 e2) (reduce (app e1 e2)))) then
   let rec a = (unify phi (t2, domain t1)) in Y(getsub a, sub_type (getsub a) (range t1), tvn)
   else N ”can’t do it”

4. and check_list env [] tvn = Y (id_subst,[],tvn)
   ||check_list env (e.es) tvn = listtypes env e es (checkexpr env tvn e)

5. and listtypes env e es (N w) = N (w@”at ”@” ”@ show_term e@”n”)
   ||listtypes env e es (Y (sub,t,tvn)) =
   composesubs sub t (check_list (app_sub_env sub env) es tvn)

6. and typecheckapp (N w) = N w
   ||typecheckapp (Y (phi,[t1;t2],tvn)) =
   if (isarrow t1) & ~(iserror (subsume id_subst t2 (domain t1))) then
   let a = (getsub (subsume id_subst t2 (domain t1))) in
   Y(scomp a phi, sub_type (scomp a phi) (range t1), tvn) else
   if (isarrow t1) & ~(occurs t1) & ~(occurs t2) then N(”Cannot unify : ”@”types mismatch”@”n”@show_type (domain t1) @ ”n”@show_type t2 @”n”) else
   typecheckapp1 tvn (unify phi (t1,top ”arrow” [t2; (tvar tvn)]))

7. and typecheckapp1 tvn (N w) = N w
   || typecheckapp1 tvn (Y phi) = Y(phi, phi tvn, next tvn)

8. and typecheckbodyabs e (N w)= N w
   ||typecheckbodyabs (tvar tvn)(Y (phi,t,tvn’)) = Y (phi, top ”arrow” [(phi tvn) ;t],tvn’)
   ||typecheckbodyabs e (Y (phi,t,tvn’)) = Y (phi, top ”arrow” [(sub_type phi e); t], tvn’)

9. and typecheck exp = checks exp [] (alpha 0) 1

10. and checks [] env tvn n = []
    ||checks (x.xs) env tvn n = let rec a = checkexpr env tvn x in if (~(iserror a)) then
    (itos n@". "@print a@"n").checks xs env (gettvn a) (n+1) else
    [printerror a@"n"@"in"@" "@show_term x@ "n"@"at"@" "@"line"@" "@itos n@"n"]

7.10 Needed Functions

1. and out x [] = []
   || out x (y.xs) & (x =y) = xs
   ||out x (y.xs) = y.(out x xs)
2. and nodupappend [] x = x
   || nodupappend (x.xs) y & (isin x y) = nodupappend xs y
   || nodupappend (x.xs) y = x.(nodupappend xs y)
3. and isin x [] = false
   || isin x (y.xs) = (x = y) | (isin x xs)
4. and newvar x l = if (isin x l) then newvar (rename x) l else x
5. and rename x = x@"'"
6. and (N w) ~~~ f = N w
   || (Y x) ~~~ f = f x
7. and next (alpha n) = alpha (n+1)
8. and zip [] xs = []
   ||zip (x.xs) [] = []
   || zip (x.xs) (y.ys) = (x,y).(zip xs ys)
9. and mem x [] = false
   || mem x (y.ys) = x =y | mem x ys
10. and getsub (Y x) = x
11. and print (Y (a,b,c)) = show_type b
12. and printerror (N w) = w
13. and iserror (N w) = true
    ||iserror other = false
14. and gettvn (Y (a,b,c)) = c
15. and gettype (Y (a,b,c)) = b
16. and getphi (Y (a,b,c)) = a
17. and lookupYN [] a = N "variable not found"
    || lookupYN ((k,v).env) a = if a = k then Y v else lookupYN env a
18. and makeprop (N w) tvn t = (N w)
    ||makeprop (Y phi) tvn t = Y(phi, sub_type phi t, tvn)
19. and seeprop phi (tvar x) = if phi x = tvar x then Y(change phi x pr) else subsume phi (phi x) pr
    || seeprop phi x & (x = pr | x = tr) = Y(id_subst)
    || seeprop phi (top "arrow" [x; y]) = seeprop phi x ~~~ (\u.seeprop u y)

```
20. and occurtype x & (x = y) = true
    || occurtype x (tvar y) = false
    || occurtype x (top tcn l) = exists (occurtype x) l end
```

7.11 Index of the various functions

This index is in alphabetical order where we go first through the 3 items on one line and then go to the next line.

functions, where	**functions, where**	**functions, where**
”~~~”, 6, 7.10	addrem, 8, 7.7	alphaconverge, 3, 7.6
anothereq, 6, 7.6	app_sub_env, 7, 7.7	betaconverge, 1, 7.6
betaconverts, 9, 7.6	change, 9, 7.7	checkexpr, 1, 7.9
check_list, 4, 7.9	checks, 10, 7.9	circulartype, 15, 7.4
composesubs, 6, 7.7	ctsubsume, 3 7.4	cyclictype, 14, 7.4
delta, 5, 7.7	domain, 9, 7.4	equaltype, 13, 7.4
equiv, 5, 7.3	etaconverge, 2, 7.6	etaconverts, 10, 7.6
extend, 10, 7.7	free, 3, 7.3	fvars, 12, 7.4
getphi, 16, 7.10	getsub, 10, 7.10	gettvn, 14, 7.10
gettype,15, 7.10	hasnf, 8, 7.6	id_subst, 4, 7.7
isapp, 9, 7.3	isarrow, 7, 7.4	isbasictype, 6, 7.4
iserror, 13, 7.10	isin, 3, 7.10	isinnf 7, 7.6
islambda, 7, 7.3	istlambda, 8, 7.3	istvar, 8, 7.4
len, 1, 7.3	listtypes, 5, 7.9	lookupYN, 17, 7.10
makeprop, 18, 7.10	makearrows, 2, 7.4	mem, 9, 7.10
mysub_type, 2, 7.7	newvar, 4, 7.10	next, 7, 7.10
nodupappend, 2, 7.10	ob, 3, 7.1,	occur, 2, 7.3
occurs, 11, 7.4	occurtype, 20, 7.10	operand, 11, 7.3
operator, 10, 7.3	option, 7, 7.1	out, 1, 7.10
pr, 4, 7.1,	print, 11, 7.10	printerror, 12, 7.10
propconj, 1, 7.8	propbot, 4, 7.8	propimpl, 2, 7.8
propneg, 3, 7.8	range, 10, 7.4	reduce, 4, 7.6
reduceoutermost, 5, 7.6	rename, 5, 7.10	scomp, 3, 7.7
seeprop, 19, 7.10	show_term, 3, 7.2	show_tlist, 2, 7.2,
show_type, 1, 7.2,	subexp, 6, 7.3	subexpression, 4, 7.3
substitute, 1, 7.5	subsume, 4, 7.4,	subsumed_by, 5, 7.4
subsumeset, 1, 7.4	sub_type, 1, 7.7	term, 6, 7.1
tr, 5, 7.1,	tterm, 2, 7.1,	typecheck, 9, 7.9
typecheckapp, 6, 7.9	typecheckapp1, 7, 7.9	typecheckbodyabs, 8, 7.9
typechecklapp, 3, 7.9	typecheckprop, 2, 7.9	typevar, 1, 7.1,
unify, 11, 7.7	unify_list, 12, 7.7	zip, 8, 7.10

8 Acknowledgements

I would like to thank Huub ten Eikelder, Rob Hoogerwoord and the anonymous referees for their constructive comments on improving the style and presentation of the paper.

References

[Aczel 80] Aczel, P., Frege structures and the notions of truth and proposition, *Kleene Symposium*, 1980.

[Aczel 84] Aczel, P., *Non-well founded sets*, CSLI Lecture notes No 14, 1984.

[Barendregt, Hemerik 90] Barendregt, H., and Hemerik, C., Types in Lambda calculi and programming languages, *Proceedings of the ESOP conference*, Copenhagen 1990.

[Beeson 84] Beeson, M., *Foundations of constructive Mathematics*, Springer Verlag, Berlin, 1984.

[Boolos 71] Boolos, G., The iterative conception of sets, *Journal of Philosophy LXVIII*, pp 215-231, 1971.

[Feferman 79] Feferman, S., Constructive theories of functions and classes, *Logic Colloquium '78*, M. Boffa et al (eds), pp 159-224, North Holland, 1979.

[Feferman 84] Feferman, S., Towards useful type free theories I, *Journal of Symbolic Logic 49*, pp 75-111, 1984.

[Girard 86] Girard, J.Y., The system F of variable types, fifteen years later, *Theoretical Computer Science 45*, pp 159-192, North-Holland, 1986.

[Kamareddine 89] Kamareddine, F., *Semantics in a Frege Structure*, PhD thesis, University of Edinburgh, 1989.

[Kamareddine 92A] Kamareddine, F., λ-terms, logic, determiners and quantifiers, *Journal of Logic, Language and Information*, Volume 1, No 1, pp 79-103, 1992.

[Kamareddine 92B] Kamareddine, F., Set Theory and Nominalisation, Part I, *Journal of Logic and Computation*, Volume 2, No 5, 1992.

[Kamareddine 92C] Kamareddine, F., Set Theory and Nominalisation, Part II, *Journal of Logic and Computation*, Volume 2, No 6, 1992.

[Martin-Löf 73] Martin-Löf, P., An intuitionistic theory of types: predicative part, *logic colloquium '73* , Rose and Shepherdson (eds), North Holland, 1973.

[Milner 78] Milner, R., A theory of type polymorphism in programming, *Journal of Computer and System Sciences*, Volume 17, No 3, 1978.

[Scott 75] Scott, D., Combinators and classes, in *Lambda Calculus and Computer Science*, Lecture Notes in Computer Science 37, Böhm (ed), Springer, Berlin, pp 1-26, 1975.

[Turner 84] Turner, R., Three Theories of Nominalized Predicates, *Studia Logica XLIV2*, 1984, pp. 165-186.