Fachrichtung 6.2 - Informatik
Naturwissenschaftlich-Technische Fakultät I
- Mathemathik und Informatik -
Universität des Saarlandes

Polymorphic Lambda Calculus with Dynamic Types

Bachelorarbeit

Angefertigt unter der Leitung von Prof. Dr. Gert Smolka
Betreuung durch Prof. Dr. Gert Smolka und Dipl. Inform. Guido Tack

Matthias Berg

September 2004

Eidesstattliche Erklärung

Hiermit erkläre ich, Matthias Berg, an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Saarbrücken, im September 2004

Matthias Berg

Danksagung

Mein Dank gilt Prof. Dr. Gert Smolka, der mir dieses interessante Thema angeboten und es mir in intensiven Diskussionen nahe gebracht hat. Desweiteren möchte ich mich bei Guido Tack bedanken, dessen Betreuung mir oft hilfreich war. An dieser Stelle danke ich auch den Mitarbeitern des Lehrstuhls für die angenehme Atmosphäre. Hier ist besonders Andreas Rossberg zu erwähnen, von dessen Erfahrung ich oft profitieren konnte. Bedanken möchte ich mich auch bei meinen Freunden und Kommilitonen für die gelegentliche Ablenkung, um den Kopf mal wieder frei zu bekommen. Nicht zuletzt danke ich meinen Eltern, die mich immer wieder unterstützt und ermutigt haben.

Abstract

In an open programming system dynamic type checking is needed to safely link statically typed components. Since type case destroys the parametricity of type abstraction, dynamically generated type names are needed to hide the implementation of abstract types. Rossberg [8] was the first to analyse the problem and to propose a calculus that solves it. We will greatly simplify Rossberg's calculus by developing a calculus, λ_F^N, an extension of System F that provides constructs for type case and a binder for new type names. We will also develop a calculus, λ_s^L, that extends the simply typed λ-calculus with call-by-need evaluation as this can be used to express lazy linking of components [9]. We will prove the safety properties for both calculi.

Contents

1 Introduction

In open programming systems in general we do not have access to all components of a program at compile time. These components are dynamically linked to the program during run time. Because we cannot type check them at compile time, we need a mechanism to do that at run time. This can be achieved by adding a construct to the language that allows for branching dependent on the type of its argument. Unfortunately such a *type case* destroys the parametricity of type abstraction, as it provides a way to exhibit the implementation of abstract types. A solution to this problem is to add a further construct that generates new type names dynamically to hide the implementation of abstract types.

Rossberg [8] was the first to analyse the problem and to propose a calculus that solves it that way. We will develop a calculus that greatly simplifies Rossberg's calculus. This is possible since his coercions are not needed. Instead we will use a global state that stores the dynamically generated type names and, as a further simplification, we will use evaluation contexts to formulate the reduction rules. Smolka [12] has proposed this design in personal communication with the author.

Additionally we will develop an extension of the simply typed λ-calculus which provides call-by-need evaluation and indicate how this can be used to express lazy linking. Lazy linking is a key mechanism in programming systems such as Java and Alice, because it allows for loading a module at the latest possible time. This can mean that a module is never loaded if it is not needed. Again Rossberg [9] formulated lazy linking in a calculus using scope extrusion. We will use a different approach and realize laziness using a global state containing the expressions that evaluate lazily.

For most of our notations we will use the style found in the textbook [7].

1.1 Type Case

To express dynamic type analysis we introduce a type case construct providing branching dependent on the type of a subterm. There have already been different formulations for dynamic type analysis that also use some kind of type case. Examples are dynamics [1], intentional type analysis [4] and extensional polymorphism [3]. We will use the construct Rossberg [8] used in his calculus:

$$\texttt{case}\ t_1 : T_1\ \texttt{of}\ x : T_2 \Rightarrow t_2\ \texttt{else}\ t_3$$

Here t_1, t_2 and t_3 are terms, T_1 and T_2 are types and T_1 must be the type of t_1. The annotation of t_1 with its type T_1 is not necessary, as the type T_1 could be inferred during reduction. But we want to design a calculus with independent typing and reduction rules. The evaluation semantics of this type case is to evaluate $t_2[x := t_1]$ iff $T_1 = T_2$ dynamically or t_3 otherwise. An example explaining how to use a type case is the following polymorphic function that if given a type and a term of this type returns a string representation of the type:

$$\begin{aligned} rep = \lambda X.\lambda x : X.&\texttt{case}\ x : X\ \texttt{of}\ x' : bool \Rightarrow \text{"}bool\text{"} \\ &\texttt{else case}\ x : X\ \texttt{of}\ x' : int \Rightarrow \text{"}int\text{"} \\ &\texttt{else}\ \text{"}unknown\text{"} \end{aligned}$$

1.2 Abstract Types

Information hiding is an important technique for handling larger programs. Hiding implementation details permits dividing the whole programming task into smaller modules by preventing strong dependencies between them. Examples of programming languages providing some kind of information hiding are SML and object oriented languages like Java or C++.

Information hiding can be achieved by using abstract types. An abstract type consists of a type name, a concrete representation type, a signature and some operations to construct, deconstruct or manipulate instances of the abstract type. The signature defines the types of the operations that are visible outside, where the representation type is substituted with the type name. The representation type itself is only visible for the operations and inaccessible outside, which has the effect that the only way of handling instances of the abstract type is by using the operations defined in its interface. Therefore changes of the implementation of an abstract type have no effect on programs using this abstract type (apart from efficiency) as long as the signature and the semantics of the operations remain unchanged.

As an illustration we will declare an abstract type in pseudo language which realises a number that provides three operations: *zero* for creating a new number, *get* for deconstructing a number and inspecting its value and *succ* for computing the successor of a number. The type name introduced is *Number* and has the representation type *int*:

$$
\begin{array}{l}
\mathit{abstype}\ \mathit{Number} = \mathit{int} \\
\mathit{signature}: \\
\quad \mathit{zero} : \mathit{Number} \\
\quad \mathit{get} : \mathit{Number} \to \mathit{int} \\
\quad \mathit{succ} : \mathit{Number} \to \mathit{Number} \\
\mathit{operations}: \\
\quad \mathit{zero} = 0 \\
\quad \mathit{get} = \lambda n : \mathit{int}.n \\
\quad \mathit{succ} = \lambda n : \mathit{int}.n + 1
\end{array}
$$

Instances of this number will be of type *Number* and outside the implementation the type checking forbids expressions like $n+1$, if n is a number. The internal representation of *Number* is *int*, but this is hidden and therefore the only means for manipulating n is by using one of the operations of number.

As we are aiming at a calculus that extends system F with some constructs, we do not need to include constructs to express abstract types in this calculus. System F is powerful enough to encode type abstraction [7]. We will now explain shortly how this encoding works. Let us again consider the example from above in a simplified version that only supports successor calculation:

$$
\begin{array}{l}
\mathit{abstype}\ \mathit{Number} = \mathit{int} \\
\mathit{signature}: \\
\quad \mathit{succ} : \mathit{Number} \to \mathit{Number} \\
\mathit{operations}: \\
\quad \mathit{succ} = \lambda n : \mathit{int}.n + 1
\end{array}
$$

Of course this number is not usable because it is impossible to construct new instances of the type *Number*, but this aspect is no object of consideration here. The type we encode

number with is $\forall X.(\forall Number.(Number \rightarrow Number) \rightarrow X) \rightarrow X$. This is the standard encoding of existential types [7] (which are the standard interpretation of abstract types). All abstract types have this form. They are given a result type X and and some client of the type $\forall X'.T \rightarrow X$. During evaluation the abstract type gives the client the representation type and the implementation of the operations as arguments. An encoding of the number looks like this:

$$\lambda X.\lambda client : (\forall Number.(Number \rightarrow Number) \rightarrow X).\ client\ \ int\ \ (\lambda n : int.n + 1)$$

Now we can see how information hiding works. For the client all instances of the abstract type are of type *Number*. The representation type *int* is inaccessible and therefore the client must use the provided operations to manipulate numbers.

1.3 Dynamically Generated Type Names

When we use type case and abstract types in combination the following problem arises: Information hiding by abstract types is no longer guaranteed. If our previously defined function *rep* gets the type *Number* and something of type *Number* as argument the desired result is "*unknown*". Unfortunately this is not what really happens. The dynamic type of a number is its representation type *int* which results in *rep* returning "*int*". This small example shows that introducing a type case destroys the parametricity of type abstraction because it makes it possible to expose implementation details of abstract types.

Rossberg [8] solved this problem by generating new type names dynamically. He used coercions to convert the type of terms from representation type to type name and vice versa. Our approach will also generate new type names, but we will not use coercions. We will use a global state that stores the type names and their representation instead. This greatly simplifies the reduction and typing rules of our calculus compared to Rossberg's.

The idea is to use a construct that dynamically introduces a type name for some type that can be used instead of that type. We will use the following syntax:

$$\texttt{new}\ \ X = T\ \ \texttt{in}\ \ t$$

Here X is the type name standing for type T and can be used in the term t. To explain how this restores parametricity of type abstraction we rewrite our previous example as follows:

$\texttt{new}\ \ X = int\ \ \texttt{in}$
abstype $Number = X$
signature :
  $zero : Number$
  $get : Number \rightarrow int$
  $succ : Number \rightarrow Number$
operations :
  $zero = 0$
  $get = \lambda n : X.n$
  $succ = \lambda n : X.n + 1$

The dynamic type of a number is still its representation type, but this has changed to X, resulting in our small example above returning "*unknown*". The desired property of

$x \in \mathit{Var}$			Variables
$X \in \mathit{TVar}$			Type Variables
$T \in \mathit{Typ}$	$::=$	$X \mid T \to T \mid \forall X.T$	Types
$v \in \mathit{Val}$	$::=$	$\lambda x : T.t \mid \lambda X.t \mid x$	Values
$t \in \mathit{Ter}$	$::=$	$x \mid \lambda x : T.t \mid t\ t \mid \lambda X.t \mid t\ T$	Terms
		$\mid \texttt{case}\ v : T\ \texttt{of}\ x : T \Rightarrow t\ \texttt{else}\ t \mid \texttt{new}\ X = T\ \texttt{in}\ t$	
$\kappa \in \mathit{Kind}$	$::=$	$*$	Kinds
$\Gamma \in \mathit{Env}$	$::=$	$\emptyset \mid \Gamma, x : T \mid \Gamma, X : \kappa$	Environments
$N \in \mathit{State}$	$::=$	$\phi \mid N, X = T$	States

Figure 1: λ_F^N-Syntax

dynamic opacity is difficult to formalise, so we will rely on our intuitive understanding that this construction solves the problem arising from using type case together with abstract types.

Communication with Rossberg [11] showed that the approach with the global state does not restore parametricity in systems with dependent types. The calculus we will develop will be based on system F, which does not provide dependent types.

2 The λ_F^N-Calculus

We will now develop a calculus based on system F that has extensions for type case and dynamic type name generation. The calculus will be referred to as λ_F^N. It is not necessary to include constructs for handling type abstraction in this calculus because system F is powerful enough to encode abstract types as explained in section 1.2.

2.1 Syntax

Figure 1 shows the syntax of λ_F^N. The constructs for type case and type name generation are the ones introduced in the previous section. As you can see `case` only allows values in its first component. This will simplify our reduction rules, since we will not need to reduce these values. It is no loss of generality as we can rewrite `case` terms as follows:

$$\texttt{case}\ t : T\ \texttt{of}\ x' : T'\ \texttt{in}\ t_1\ \texttt{else}\ t_2 \equiv (\lambda x : T.\texttt{case}\ x : T\ \texttt{of}\ x' : T'\ \texttt{in}\ t_1\ \texttt{else}\ t_2)\, t$$

This is also the reason why we include variables in the definition of values. *State* will be used as global state storing the dynamically generated type names and their respective representation. *Env* is the environment containing the free type variables and the types of free term variables. *State* and *Env* are modelled as lists. It is also possible to model them as partial functions, but the list constructions preserves the order of the entries and will simplify some of our proofs. *Kind* has been included to unify the representation of environments. Sometimes we will use *State* more like a queue; if we write $X = T, N$ this means $X = T$ is the leftmost element and if we write $N, X = T$ it is the rightmost element of the list and N is this list without the entry $X = T$.

In order to avoid variable capturing, we consider terms and types modulo alpha conversion and assume the convention that binders always introduce fresh variable names. This is also known as the Barendregt Convention.

$E ::= \circ \mid E\ t \mid v\ E \mid E\ T$

$$
\begin{array}{lrcll}
(1) & E((\lambda x : T.t)\ v) \mid N & \longrightarrow & E(t[x := v]) \mid N & \\
(2) & E((\lambda X.t)\ T) \mid N & \longrightarrow & E(t[X := T]) \mid N & \\
(3) & E(\texttt{case}\ v : T\ \texttt{of}\ x : T' \Rightarrow t\ \texttt{else}\ t') \mid N & \longrightarrow & E(t[x := v]) \mid N & (\text{if } T = T') \\
(4) & E(\texttt{case}\ v : T\ \texttt{of}\ x : T' \Rightarrow t\ \texttt{else}\ t') \mid N & \longrightarrow & E(t') \mid N & (\text{if } T \neq T') \\
(5) & E(\texttt{new}\ X = T\ \texttt{in}\ t) \mid N & \longrightarrow & E(t) \mid N, X = T & (X \text{ fresh})
\end{array}
$$

Figure 2: λ_F^N-Reduction

2.2 Reduction

As mentioned above we use a global state to store the dynamically generated type names and their respective representation. Therefore we define our reduction rules over pairs of terms and states, written $t|N$. We will call such pairs *configurations*. We define the reduction rules using evaluation contexts. Figure 2 contains the rules realizing a small-step call-by-value strategy. The first two rules are known from system F. Rules 3 and 4 realize the type dependent branching of the type case construct. Rule 5 defines the reduction of the `new` construct. The abstracted type and its name are just appended to the global state. It is easy to verify that these reduction rules are deterministic. Note that the global state N is not touched in rules 1-4 and only extended in rule 5. This indicates that a type that has been abstracted with `new` cannot be inspected during reduction.

2.3 Typing

Figure 3 defines five different relations:

$\vdash \Gamma$	well-formedness of environments
$\vdash N$	well-formedness of states
$\Gamma \vdash T : \kappa$	well-formedness of types
$\Gamma \vdash t : T$	well-typedness of terms
$\Gamma \vdash t\|N : T$	well-typedness of configurations

Environments and states are lists of pairs. The relations $\vdash \Gamma$ and $\vdash N$ contain only environments or states where each variable occurs as left component in a pair once at the most. Therefore we can use well-formed environments and states like partial functions. Using a state as a function means to substitute all type names with their respective representation types:

$$
\begin{array}{rcl}
\phi(t) & = & t \\
(N, X = T)(t) & = & N(t[X := T])
\end{array}
$$

The application of a state to a type is defined analogically. If we have $\Gamma = \emptyset$, we will omit Γ as abbreviation and write $\vdash t : T$ to express that t is a well-typed closed term.

Note that in reduction the state grows to the right whereas the well-formedness rules for states always deal with the leftmost entry. Because states grow to the right all dependencies in a state are from right to left, meaning that for an entry $X = T$ no free variable in T is bound in any entry to the right of its position. Therefore we check states from left to right as

Environments

$$\frac{}{\vdash \emptyset} \qquad \frac{\Gamma \vdash T : * \quad x \notin dom(\Gamma)}{\vdash \Gamma, x : T} \qquad \frac{\vdash \Gamma \quad X \notin dom(\Gamma)}{\vdash \Gamma, X : \kappa}$$

States

$$\frac{\vdash \Gamma}{\Gamma \vdash \phi} \qquad \frac{\Gamma \vdash T : \kappa \quad \Gamma, X : \kappa \vdash N}{\Gamma \vdash X = T, N}$$

Types

$$\frac{\vdash \Gamma \quad X \in dom(\Gamma)}{\Gamma \vdash X : \Gamma X} \qquad \frac{\Gamma \vdash T_1 : * \quad \Gamma \vdash T_2 : *}{\Gamma \vdash T_1 \to T_2 : *} \qquad \frac{\Gamma, X : * \vdash T : *}{\Gamma \vdash \forall X.T : *}$$

Terms

$$\frac{\vdash \Gamma \quad x \in dom(\Gamma)}{\Gamma \vdash x \ : \ \Gamma\ x} \qquad \frac{\Gamma \vdash t : \forall X.T' \quad \Gamma \vdash T : *}{\Gamma \vdash t\ T : T'[X := T]}$$

$$\frac{\Gamma, x : T \vdash t : T'}{\Gamma \vdash \lambda x : T.t \ : \ T \to T'} \qquad \frac{\Gamma \vdash v : T \quad \Gamma, x : T' \vdash t : T'' \quad \Gamma \vdash t' : T''}{\Gamma \vdash \texttt{case}\ v : T\ \texttt{of}\ x : T' \Rightarrow t\ \texttt{else}\ t' \ : \ T''}$$

$$\frac{\Gamma \vdash t : T' \to T \quad \Gamma \vdash t' : T'}{\Gamma \vdash t\ t' : T} \qquad \frac{\Gamma \vdash T : \kappa \quad \Gamma \vdash t[X := T] : T'}{\Gamma \vdash \texttt{new}\ X = T\ \texttt{in}\ t \ : \ T'}$$

$$\frac{\Gamma, X : * \vdash t : T}{\Gamma \vdash \lambda X.t \ : \ \forall X.T}$$

Configurations

$$\frac{\Gamma \vdash N \quad \Gamma \vdash Nt : T}{\Gamma \vdash t|N : T}$$

Figure 3: λ_F^N-Typing

this allows that Γ binds exactly those variables the currently checked entry in the state can depend on.

The typing rules for the two λ-abstractions require the introduced variable not to be bound by Γ, because otherwise appending this variable to Γ would result in a malformed environment. So the typing rules assure that these binders must always introduce new variable names. This is conform with the previously mentioned Barendregt Convention.

It is possible to formulate the typing rules with a different approach that combines the rules for typing terms and configurations. The resulting relation $\Gamma \vdash t|N : T$ would have rules for each syntactic form of terms. Here we use the approach to have a simpler relation $\Gamma \vdash t : T$ that is used by the rule for typing configurations by applying state N to t.

A nice property of our typing rules is that given $\Gamma \vdash t : T$ it follows that all free variables of t and T must be bound by Γ and that Γ is well-formed. This is quite obvious if we look at the rule for term variables. Here x must be bound by Γ and Γ must be well-formed, implying that type Γx is well-formed, which means that all its free variables are bound by Γ.

From this property and the rule for configurations it follows directly that $\Gamma \vdash t|N : T$ implies that the free variables of T contain no type name that is bound by N.

3 Properties of λ_F^N

In this section we will prove the safety properties for our calculus, which are the unique type, the progress, and the preservation property. Additionally we will show that in contrast to system F the calculus λ_F^N is not normalising any more.

3.1 Uniqueness of Types

To be able to use the inference rules in our typing relations not only as implication from premises to conclusion, but also vice versa, we formulate the following inversion lemma:

Lemma 3.1 *(Inversion):*

Environments:
$\vdash \Gamma, x : T \implies \Gamma \vdash T : * \ \wedge \ x \notin dom(\Gamma)$
$\vdash \Gamma, X : \kappa \implies \vdash \Gamma \ \wedge \ X \notin dom(\Gamma)$

States:
$\Gamma \vdash \phi \implies \vdash \Gamma$
$\Gamma \vdash X = T, N \implies \Gamma \vdash T : \kappa \ \wedge \ \Gamma, X : \kappa \vdash N$

Types:
$\Gamma \vdash X : \kappa \implies \vdash \Gamma \ \wedge \ X \in dom(\Gamma) \ \wedge \ \Gamma\, X = \kappa$
$\Gamma \vdash T_1 \to T_2 : * \implies \Gamma \vdash T_1 : * \ \wedge \ \Gamma \vdash T_2 : *$
$\Gamma \vdash \forall X.T : * \implies \Gamma, X : * \vdash T : *$

Terms:
$\Gamma \vdash x : T \implies \vdash \Gamma \ \wedge \ x \in dom(\Gamma) \ \wedge \ \Gamma\, x = T$
$\Gamma \vdash \lambda x : T.t \ : \ T \to T' \implies \Gamma, x : T \vdash t : T'$
$\Gamma \vdash t\ t' : T \implies \exists T' : \ \Gamma \vdash t : T' \to T \ \wedge \ \Gamma \vdash t' : T'$
$\Gamma \vdash \lambda X.t \ : \ \forall X.T \implies \Gamma, X : * \vdash t : T$
$\Gamma \vdash t\ T : T' \implies \exists X, T'' : \ \Gamma \vdash t : \forall X.T'' \ \wedge \ \Gamma \vdash T : * \ \wedge \ T' = T''[X := T]$

$\Gamma \vdash \texttt{case}\ v : T\ \texttt{of}\ x : T' \Rightarrow t\ \texttt{else}\ t'\ :\ T''\ \implies$
$\Gamma \vdash v : T\ \wedge\ \Gamma, x : T' \vdash t : T''\ \wedge\ \Gamma \vdash t' : T''$
$\Gamma \vdash \texttt{new}\ X = T\ \texttt{in}\ t\ :\ T'\ \implies\ \Gamma \vdash T : \kappa\ \wedge\ \Gamma \vdash t[X := T] : T'\ \wedge\ X \notin dom(\Gamma)$
Configurations:
$\Gamma \vdash t|N : T\ \implies\ \Gamma \vdash N\ \wedge\ \Gamma \vdash Nt : T$

Proof : Because we have exactly one rule per syntactic form, the lemma follows immediately from the definition of our relations.

□

Our proofs will often use induction on terms. Unfortunately the typing rule of the `new` construct does not allow structural induction on terms because in its premise we need to type the term $t[X := T]$ and this substitution prohibits a pure structural argumentation. Hence we define a size of terms that does not depend on the types contained in a term:

size(x) = 1
size($\lambda x : T.t$) = size(t)+1
size($t_1\, t_2$) = size(t_1)+size(t_2)+1
size($\lambda X.t$) = size(t)+1
size($t\ T$) = size(t)+1
size($\texttt{case}\ v : T_1\ \texttt{of}\ x : T_2 \Rightarrow t_1\ \texttt{else}\ t_2$) = size($v$)+size($t_1$)+size($t_2$)+1
size($\texttt{new}\ X = T\ \texttt{in}\ t$) = size($t$)+1

Theorem 3.2 *(Uniqueness for terms):*

$\Gamma \vdash t : T\ \wedge\ \Gamma \vdash t : T'\quad \implies\quad T = T'$

Proof : Induction on the size of t
Case size(t) = 1:
$\implies t = x$ for some variable x.
Let $\Gamma \vdash x : T\ \wedge\ \Gamma \vdash x : T'$
It follows that $T = \Gamma\ x = T'$.

Case size(t) > 1:
Case $t = \lambda x : T_1.t_2$:
Let $\Gamma \vdash t : T\ \wedge\ \Gamma \vdash t : T'$
It follows that $T = T_1 \to T_2\ \wedge\ T' = T_1 \to T_2'$ for some types T_2 and T_2'.
Furthermore we have $\Gamma, x : T_1 \vdash t_2 : T_2\ \wedge \Gamma, x : T_1 \vdash t_2 : T_2'$ (Inversion).
The induction hypothesis gives us $T_2 = T_2'$ and therefore $T = T'$.

Case $t = t_1\ t_2$:
Let $\Gamma \vdash t_1\ t_2 : T\ \wedge\ \Gamma \vdash t_1\ t_2 : T'$
It follows that $\Gamma \vdash t_1 : T_1 \to T \wedge \Gamma \vdash t_1 : T_2 \to T'$ for some types T_1 and T_2 (Inversion).
The induction hypothesis gives us $T_1 \to T = T_2 \to T'$ which implies $T = T'$.

Case $t = \lambda X.t_1$:
Let $\Gamma \vdash \lambda X.t_1 : T\ \wedge\ \Gamma \vdash \lambda X.t_1 : T'$
It follows that $T = \forall X.T_1\ \wedge\ T' = \forall X.T_1'$ for some types T_1 and T_1'.

Furthermore we have $\Gamma, X : * \vdash t_1 : T_1 \ \wedge \ \Gamma, X : * \vdash t_1 : T_1'$ (Inversion).
The induction hypothesis gives us $T_1 = T_1'$ and therefore $T = T'$.

Case $t = t_1\, T_1$:
Let $\Gamma \vdash t_1\, T_1 : T \ \wedge \ \Gamma \vdash t_1\, T_1 : T'$
It follows that $\Gamma \vdash t_1 : \forall X.T_2 \ \wedge \ \Gamma \vdash t_1 : \forall X.T_2'$ for some types T_2 and T_2'.
Furthermore we have $T = T_2[X := T_1]$ and $T' = T_2'[X := T_1]$.
The induction hypothesis gives us $\forall X.T_2 = \forall X.T_2'$.
It follows that $T_2 = T_2'$ and therefore $T = T'$.

Case $t = \texttt{case}\ v : T_1\ \texttt{of}\ x : T_2 \Rightarrow t_2\ \texttt{else}\ t_3$:
Let $\Gamma \vdash t : T \ \wedge \ \Gamma \vdash t : T'$
By inversion it follows that $\Gamma \vdash t_3 : T \ \wedge \ \Gamma \vdash t_3 : T'$.
The induction hypothesis gives us $T = T'$.

Case $t = \texttt{new}\ X = T_1\ \texttt{in}\ t_1$:
Let $\Gamma \vdash t : T \ \wedge \ \Gamma \vdash t : T'$
By inversion it follows that $\Gamma \vdash t_1[X := T_1] : T \ \wedge \ \Gamma \vdash t_1[X := T_1] : T'$.
Because $\text{size}(t_1[X := T_1]) < \text{size}(t)$, the induction hypothesis gives us $T = T'$

□

Corollary 3.3 *(Uniqueness):*

$$\Gamma \vdash t|N : T \ \wedge \ \Gamma \vdash t|N : T' \quad \Longrightarrow \quad T = T'$$

Proof : Immediately from theorem 3.2 with use of the inversion lemma.

□

3.2 Progress

The progress property states that a closed, well-typed configuration contains either a value or it can be reduced.

Theorem 3.4 *(Progress):*

$$\vdash t|N : T \quad \Longrightarrow \quad t \in \mathit{Val} \ \vee \ \exists t', N' : t|N \longrightarrow t'|N'$$

Proof : Structural induction on t
Case $t = x$:
trivial. x is not a closed term.

Case $t = \lambda x : T.t'$:
trivial. $\lambda x : T.t'$ is a value.

Case $t = t_1 t_2$:
Let $\vdash t_1 t_2|N : T$. By inversion it follows that $\vdash N \wedge \ \vdash N(t_1 t_2) : T$.
$\Longrightarrow \vdash (Nt_1)(Nt_2) : T$
$\Longrightarrow \vdash Nt_2 : T_2$ and $\vdash Nt_1 : T_2 \to T$ for some type T_2 (inversion).
$\Longrightarrow \vdash t_2|N : T_2$ and $\vdash t_1|N : T_2 \to T$

By induction hypothesis we may assume:
$t_1 \in \mathit{Val} \ \vee \ \exists t_1', N' : t_1|N \longrightarrow t_1'|N'$ and
$t_2 \in \mathit{Val} \ \vee \ \exists t_2', N' : t_2|N \longrightarrow t_2'|N'$
Case $t_1 \notin \mathit{Val}$:
$\Longrightarrow t_1|N \longrightarrow t_1'|N'$ for some term t_1' and some state N'.
So there exists a context E and terms s, s' such that $t_1 = E(s)$ and $E(s)|N \longrightarrow E(s')|N'$.
With context $E' = E\ t_2$ we have $t|N = E'(s)|N \ \wedge \ E'(s)|N \longrightarrow E'(s')|N'$.
Case $t_1 \in \mathit{Val}$ and $t_2 \notin \mathit{Val}$:
$\Longrightarrow t_2|N \longrightarrow t_2'|N'$ for some term t_2' and some state N'.
So there exists a context E and terms s, s' such that $t_2 = E(s)$ and $E(s)|N \longrightarrow E(s')|N'$.
With context $E' = t_1\ E$ we have $t|N = E'(s)|N \ \wedge \ E'(s)|N \longrightarrow E'(s')|N'$.
Case $t_1 \in \mathit{Val}$ and $t_2 \in \mathit{Val}$:
So $t_1 = \lambda x : T_2.s$ for some term s ($t_1 = \lambda X.t'$ or $t_1 = x$ would not type correctly).
With $E = \circ$ we have:
$t|N = E((\lambda x : T_2.s)\ t_2)|N \wedge E((\lambda x : T_2.s)\ t_2)|N \longrightarrow E(s[x := t_2])|N$

Case $t = \lambda X.t_1$:
trivial. $\lambda X.t_1$ is a value.

Case $t = t_1\, T_1$:
Let $\vdash t_1\, T_1 \mid N : T$. By inversion it follows that $\vdash N \wedge \ \vdash N(t_1 T_1) : T$
$\Longrightarrow \vdash (N t_1)(N T_1) : T$
$\Longrightarrow \vdash t_1|N : \forall X.T'$ with $T = T'[X := N T_1]$
By induction hypothesis we may assume:
$t_1 \in \mathit{Val} \ \vee \ \exists t_1', N' : t_1|N \longrightarrow t_1'|N'$
Case $t_1 \in \mathit{Val}$:
$\Longrightarrow t_1 = \lambda X.t_2$ for some term t_2 ($t_1 = \lambda x : T'.t'$ or $t_1 = x$ would not type correctly).
With $E = \circ$ we have $t|N = E((\lambda X.t_2)T_1)|N \ \wedge \ E((\lambda X.t_2)T_1)|N \longrightarrow E(t_2[X := T_1])|N$
Case $t_1 \notin \mathit{Val}$:
$\Longrightarrow t_1|N \longrightarrow t_1'|N'$ for some term t_1' and some state N'
So there exists a context E and terms s, s' such that $t_1 = E(s)$ and $E(s)|N \longrightarrow E(s')|N'$.
With context $E' = E\ T$ we have $t|N = E'(s)|N \ \wedge \ E'(s)|N \longrightarrow E'(s')|N'$.

Case $t = \texttt{case}\ \ v : T_1\ \ \texttt{of}\ \ x : T_2 \Rightarrow t_2\ \ \texttt{else}\ \ t_3$:
Case $T_1 = T_2$: with $E = \circ$ we have $t|N = E(t)|N \ \wedge \ E(t)|N \longrightarrow E(t_2[x := v])|N$
Case $T_1 \neq T_2$: with $E = \circ$ we have $t|N = E(t)|N \ \wedge \ E(t)|N \longrightarrow E(t_3)|N$

Case $t = \texttt{new}\ \ X = T_1\ \ \texttt{in}\ \ t_1$:
With $E = \circ$ we have $t|N = E(t)|N \ \wedge \ E(t)|N \longrightarrow E(t_1)|N, X = T_1$ (X fresh)

□

3.3 Preservation

The goal of this section is to prove that well-typed configurations do not alter their type during further reduction. This is a bit harder to prove than the previous properties, because we need some lemmas first. Lemma 3.5 allows us to state the well-typedness of some term t in some context E:

Lemma 3.5 *:*

$$\Gamma \vdash E(t)|N : T \quad \Longrightarrow \quad \exists T' : \Gamma \vdash Nt : T'$$

Proof : Structural induction on E:
Case $E = \circ$:
Follows trivially by inversion, as $E(t) = t$

Case $E = E'\ t'$:
Let $\Gamma \vdash E(t)|N : T$. This is equivalent to $\Gamma \vdash E'(t)\, t'|N : T$
By inversion it follows that $\Gamma \vdash E'(t)|N : T'' \to T$ for some type T''
The induction hypothesis gives us $\exists T' : \Gamma \vdash Nt : T'$

Case $E = v\ E'$:
Let $\Gamma \vdash E(t)|N : T$. This is equivalent to $\Gamma \vdash v\, E'(t)|N : T$
By inversion it follows that $\Gamma \vdash E'(t)|N : T''$ for some type T''
The induction hypothesis gives us $\exists T' : \Gamma \vdash Nt : T'$

Case $E = E'\ T$:
Let $\Gamma \vdash E(t)|N : T$. This is equivalent to $\Gamma \vdash E'(t)\, T|N : T$
By inversion it follows that $\Gamma \vdash E'(t)|N : \forall X.T''$ for some type T''
The induction hypothesis gives us $\exists T' : \Gamma \vdash Nt : T'$

□

The weakening lemma 3.6 will be needed in the proof of lemma 3.7.

Lemma 3.6 *(Weakening):*

$$\Gamma \vdash t : T \ \wedge \ X \notin dom(\Gamma) \quad \Longrightarrow \quad \Gamma, X : * \vdash t : T$$

Proof : Straightforward induction on t.

□

Lemma 3.7 is the well-known substitution lemma that allows us to argue about type preservation during β-reduction.

Lemma 3.7 *(Substitution):*

$$\Gamma, x : T' \vdash t : T \ \wedge \ \Gamma \vdash t' : T' \quad \Longrightarrow \quad \Gamma \vdash t[x := t'] : T$$

Proof : Induction on the size of t:
Case size$(t) = 1$:
Case $t = x' \ \wedge \ x' = x$:
Let $\Gamma, x : T' \vdash x' : T \ \wedge \ \Gamma \vdash t' : T'$. So it follows that $T' = (\Gamma, x : T')(x') = T$.
As $t[x := t'] = t'$ we have $\Gamma \vdash t[x := t'] : T$.

Case $t = x' \ \wedge \ x' \neq x$:
Let $\Gamma, x : T' \vdash x' : T$. As $x' \neq x$ it follows that $x'[x := t'] = x' \ \wedge \ \Gamma \vdash x' : T$.

Case size$(t) > 1$:

Case $t = \lambda x' : T_1.t_2$:
Let $\Gamma, x : T' \vdash t : T \ \wedge \ \Gamma \vdash t' : T'$.
From our typing rules it follows $x \neq x'$.
Type T must have the form $T_1 \to T_2$ and $\Gamma, x : T', x' : T_1 \vdash t_2 : T_2$ for some type T_2.
We can exchange the order of the environment to $\Gamma, x' : T_1, x : T'$ because
neither x nor x' introduce new type variables: $\Gamma, x' : T_1, x : T' \vdash t_2 : T_2$
By induction hypothesis it follows $\Gamma, x' : T_1 \vdash t_2[x := t'] : T_2$.
That implies $\Gamma \vdash \lambda x' : T_1.(t_2[x := t']) : T_1 \to T_2$
which is equivalent to $\Gamma \vdash (\lambda x' : T_1.t_2)[x := t'] : T_1 \to T_2$.

Case $t = t_1\, t_2$:
Let $\Gamma, x : T' \vdash t_1\, t_2 : T \ \wedge \ \Gamma \vdash t' : T'$.
Inversion leads to $\Gamma, x : T' \vdash t_1 : T_2 \to T \ \wedge \Gamma, x : T' \vdash t_2 : T_2$ for some type T_2.
By induction hypothesis it follows $\Gamma \vdash t_1[x := t'] : T_2 \to T \ \wedge \ \Gamma \vdash t_2[x := t'] : T_2$.
So we obtain $\Gamma \vdash (t_1 t_2)[x := t'] : T$.

Case $t = \lambda X.t_2$:
Let $\Gamma, x : T' \vdash t : T \ \wedge \ \Gamma \vdash t' : T'$.
Type T must have the form $\forall X.T_2$ and $\Gamma, x : T', X : * \vdash t_2 : T_2$ for some type T_2.
We may assume that X is not free in T'.
$\Longrightarrow$ We can exchange the order of the environment to $\Gamma, X : *, x : T' \ (X \notin FV(T'))$:
$\Gamma, X : *, x : T' \vdash t_2 : T_2$
The weakening lemma 3.6 gives us $\Gamma, X : * \vdash t' : T'$.
By induction hypothesis it follows $\Gamma, X : * \vdash t_2[x := t'] : T_2$.
$\Longrightarrow \Gamma \vdash \lambda X.(t_2[x := t']) : \forall X.T_2$
$\Longrightarrow \Gamma \vdash (\lambda X.t_2)[x := t'] : \forall X.T_2$

Case $t = t_1\, T_1$:
Let $\Gamma, x : T' \vdash t_1\, T_1 : T \ \wedge \ \Gamma \vdash t' : T'$.
$\Longrightarrow \Gamma, x : T' \vdash t_1 : \forall X.T_2$ with $T_2[X := T_1] = T$ and $\Gamma \vdash T_1 : *$ (by inversion)
By induction hypothesis it follows $\Gamma \vdash t_1[x := t'] : \forall X.T_2$.
$\Longrightarrow \Gamma \vdash (t_1[x := t'])T_1 : T$
$\Longrightarrow \Gamma \vdash (t_1 T_1)[x := t'] : T$

Case $t = \texttt{case}\ v : T_1\ \texttt{of}\ x' : T_2 \Rightarrow t_2\ \texttt{else}\ t_3$:
Let $\Gamma, x : T' \vdash t : T \ \wedge \ \Gamma \vdash t' : T'$ (From the typing rules follows $x \neq x'$).
$\Longrightarrow \Gamma, x : T' \vdash v : T_1 \ \wedge \ \Gamma, x : T', x' : T_2 \vdash t_2 : T \ \wedge \ \Gamma, x : T' \vdash t_3 : T$ (by inversion)
$\Longrightarrow \Gamma, x : T' \vdash v : T_1 \ \wedge \ \Gamma, x' : T_2, x : T' \vdash t_2 : T \ \wedge \ \Gamma, x : T' \vdash t_3 : T$
By induction hypothesis it follows:
$\Gamma \vdash v[x := t'] : T_1 \ \wedge \ \Gamma, x' : T_2 \vdash t_2[x := t'] : T \ \wedge \ \Gamma \vdash t_3[x := t'] : T$
$\Longrightarrow \Gamma \vdash \texttt{case}\ v[x := t'] : T_1\ \texttt{of}\ x' : T_2 \Rightarrow t_2[x := t']\ \texttt{else}\ t_3[x := t'] : T$
$\Longrightarrow \Gamma \vdash (\texttt{case}\ v : T_1\ \texttt{of}\ x' : T_2 \Rightarrow t_2\ \texttt{else}\ t_3)[x := t'] : T$

Case $t = \texttt{new}\ X = T_1\ \texttt{in}\ t_1$:
Let $\Gamma, x : T' \vdash t : T \ \wedge \ \Gamma \vdash t' : T'$.
$\Longrightarrow \Gamma, x : T' \vdash t_1[X := T_1] : T \ \wedge \ \Gamma \vdash T_1 : \kappa$ (by inversion)
Because $\text{size}(t_1[X := T_1]) < \text{size}(t)$, the induction hypothesis gives us:

$\Gamma \vdash t_1[X := T_1][x := t'] : T$
We may assume that X is not free in t'.
$\Longrightarrow \Gamma \vdash t_1[x := t'][X := T_1] : T$
$\Longrightarrow \Gamma \vdash \texttt{new}\ X = T_1\ \texttt{in}\ t_1[x := t'] : T$
$\Longrightarrow \Gamma \vdash (\texttt{new}\ X = T_1\ \texttt{in}\ t_1)[x := t'] : T$

□

Lemma 3.8 states that in some context E the substitution of some term t with some term s of the same type does not change the type of the whole term.

Lemma 3.8 *:*

$$\Gamma \vdash E(t)|N : T\ \wedge\ \Gamma \vdash Nt : T'\ \wedge\ \Gamma \vdash Ns : T' \quad \Longrightarrow \quad \Gamma \vdash E(s)|N : T$$

Proof : Structural induction on E
Case $E = \circ$
Let $\Gamma \vdash E(t)|N : T\ \wedge\ \Gamma \vdash Nt : T'\ \wedge\ \Gamma \vdash Ns : T'$.
$\Longrightarrow \Gamma \vdash Nt : T\ \wedge\ \Gamma \vdash Nt : T'$ (by inversion)
The uniqueness property gives us $T = T'$. $E(s) = s$ leads to $\Gamma \vdash E(s)|N : T$.

Case $E = E'\,t'$:
Let $\Gamma \vdash E(t)|N : T\ \wedge\ \Gamma \vdash Nt : T'\ \wedge\ \Gamma \vdash Ns : T'$.
It follows that $\Gamma \vdash E'(t)|N : T'' \to T\ \wedge\ \Gamma \vdash Nt' : T''$ for some type T''.
The induction hypothesis leads us to $\Gamma \vdash E'(s)|N : T'' \to T\ \wedge\ \Gamma \vdash Nt' : T''$.
This implies $\Gamma \vdash E(s)|N : T$.

Case $E = v\,E'$:
Let $\Gamma \vdash E(t)|N : T\ \wedge\ \Gamma \vdash Nt : T'\ \wedge\ \Gamma \vdash Ns : T'$.
It follows that $\Gamma \vdash Nv : T'' \to T\ \wedge\ \Gamma \vdash E'(t)|N : T''$ for some type T''.
The induction hypothesis leads us to $\Gamma \vdash Nv : T'' \to T\ \wedge\ \Gamma \vdash E'(s)|N : T''$.
This implies $\Gamma \vdash E(s)|N : T$.

Case $E = E'\,T''$:
Let $\Gamma \vdash E(t)|N : T\ \wedge\ \Gamma \vdash Nt : T'\ \wedge\ \Gamma \vdash Ns : T'$.
It follows that $\Gamma \vdash E'(t)|N : \forall X.T_1\ \wedge\ \Gamma \vdash NT'' : *$ with $T = T_1[X := NT'']$.
The induction hypothesis leads us to $\Gamma \vdash E'(s)|N : \forall X.T_1$.
This implies $\Gamma \vdash E(s)|N : T$.

□

The last two lemmas we need for proving preservation state that we can instantiate type variables with well-formed types. Here we take advantage of our design decision to model environments as lists and not as partial functions, since we can use the order of the entries to argue about dependencies.

Lemma 3.9 *(Type Substitution for Types):*

$$\Gamma, X : *, \Gamma' \vdash T : \kappa\ \wedge\ \Gamma \vdash T' : * \quad \Longrightarrow \quad \Gamma, \Gamma'[X := T'] \vdash T[X := T'] : \kappa$$

Proof : Straightforward structural induction on T

□

Lemma 3.10 *(Type Substitution):*

$$\Gamma, X : *, \Gamma' \vdash t : T \ \wedge \ \Gamma \vdash T' : * \quad \Longrightarrow \quad \Gamma, \Gamma'[X := T'] \vdash t[X := T'] : T[X := T']$$

Proof : Induction on the size of t:
Let $\Gamma, X : *, \Gamma' \vdash t : T \ \wedge \ \Gamma \vdash T' : *$.
From the typing rules it follows that X does not occur in Γ.

Case size$(t) = 1$:
$\Longrightarrow t = x$ for some variable x.
From the typing rules it follows $\vdash \Gamma, \Gamma'[X := T']$.
Case $x \in dom(\Gamma)$:
$\Longrightarrow X$ is not free in T and therefore $\Gamma, \Gamma'[X := T'] \vdash t[X := T'] : T[X := T']$.
Case $x \in dom(\Gamma')$:
$\Longrightarrow \Gamma, \Gamma'[X := T'] \vdash t[X := T'] : T[X := T']$, since $t[X := T'] = x$

Case size$(t) > 1$:
Case $t = \lambda x_1 : T_1.t_2$:
$\Longrightarrow \Gamma, X : *, \Gamma', x_1 : T_1 \vdash t_2 : T_2$ with $T = T_1 \to T_2$ for some type T_2 (inversion)
By induction hypothesis it follows:
$\Gamma, \Gamma'[X := T'], x_1 : T_1[X := T'] \vdash t_2[X := T'] : T_2[X := T']$
$\Longrightarrow \Gamma, \Gamma'[X := T'] \vdash \lambda x_1 : T_1[X := T'].t_2[X := T'] : T_1[X := T'] \to T_2[X := T']$
$\Longrightarrow \Gamma, \Gamma'[X := T'] \vdash (\lambda x_1 : T_1.t_2)[X := T'] : (T_1 \to T_2)[X := T']$

Case $t = t_1\, t_2$:
$\Longrightarrow \Gamma, X : *, \Gamma' \vdash t_1 : T_2 \to T \ \wedge \ \Gamma, X : *, \Gamma' \vdash t_2 : T_2$ for some type T_2.
By induction hypothesis it follows:
$\Gamma, \Gamma'[X := T'] \vdash t_1[X := T'] : (T_2 \to T)[X := T']$ and
$\Gamma, \Gamma'[X := T'] \vdash t_2[X := T'] : T_2[X := T']$
$\Longrightarrow \Gamma, \Gamma'[X := T'] \vdash t_1[X := T']t_2[X := T'] : T[X := T']$
$\Longrightarrow \Gamma \vdash (t_1\, t_2)[X := T'] : T[X := T']$

Case $t = \lambda X_1.t_1$:
$\Longrightarrow \Gamma, X : *, \Gamma', X_1 : * \vdash t_1 : T_1$ with $T = \forall X_1.T_1$ for some type T_1.
From the typing rules it follows $X \neq X_1$.
By induction hypothesis it follows $\Gamma, \Gamma'[X := T'], X_1 : * \vdash t_1[X := T'] : T_1[X := T']$.
$\Longrightarrow \Gamma, \Gamma'[X := T'] \vdash \lambda X_1.t_1[X := T'] : \forall X_1.T_1[X := T']$
$\Longrightarrow \Gamma, \Gamma'[X := T'] \vdash (\lambda X_1.t_1)[X := T'] : (\forall X_1.T_1)[X := T']$, since $X \neq X_1$.

Case $t = t_1\, T_2$:
$\Longrightarrow \Gamma, X : *, \Gamma' \vdash t_1 : \forall X_1.T_1$ with $T = T_1[X_1 := T_2]$ and $\Gamma, X : *, \Gamma' \vdash T_2 : *$.
By induction hypothesis it follows $\Gamma, \Gamma'[X := T'] \vdash t_1[X := T'] : (\forall X_1.T_1)[X := T']$.
From the typing rules and the Barendregt Convention it follows:
$X \neq X_1 \wedge X_1 \notin FV(T') \wedge \Gamma, \Gamma'[X := T'] \vdash T_2[X := T'] : *$.
$\Longrightarrow \Gamma, \Gamma'[X := T'] \vdash t_1[X := T']T_2[X := T'] : T_1[X := T'][X_1 := T_2[X := T']]$

The substitution $[X := T'][X_1 := T_2[X := T']]$ is equivalent to $[X_1 := T_2][X := T']$
$\Longrightarrow \Gamma \vdash t_1[X := T']T_2[X := T'] : T_1[X_1 := T_2][X := T']$
$\Longrightarrow \Gamma \vdash (t_1\, T_2)[X := T'] : T[X := T']$

Case $t = \texttt{case}\ v : T_1\ \texttt{of}\ x : T_2 \Rightarrow t_2\ \texttt{else}\ t_3$:
$\Longrightarrow \Gamma, X : *, \Gamma' \vdash v : T_1\ \wedge\ \Gamma, X : *, \Gamma', x : T_2 \vdash t_2 : T\ \wedge\ \Gamma, X : *, \Gamma' \vdash t_3 : T$ (inversion)
By induction hypothesis it follows:
$\Gamma, \Gamma'[X := T'] \vdash v[X := T'] : T_1[X := T'] \wedge$
$\Gamma, \Gamma'[X := T'], x : T_2[X := T'] \vdash t_2[X := T'] : T[X := T'] \wedge$
$\Gamma, \Gamma'[X := T'] \vdash t_3[X := T'] : T[X := T']$
$\Longrightarrow \Gamma, \Gamma'[X := T'] \vdash \texttt{case}\ v[X := T'] : T_1[X := T']$
$\texttt{of}\ x : T_2[X := T'] \Rightarrow t_2[X := T']$
$\texttt{else}\ t_3[X := T']$
$: T[X := T']$
$\Longrightarrow \Gamma, \Gamma'[X := T'] \vdash (\texttt{case}\ v : T_1\ \texttt{of}\ x : T_2 \Rightarrow t_2\ \texttt{else}\ t_3)[X := T'] : T[X := T']$

Case $t = \texttt{new}\ X_1 = T_1\ \texttt{in}\ t_1$:
$\Longrightarrow \Gamma, X : *, \Gamma' \vdash T_1 : \kappa\ \wedge\ \Gamma, X : *, \Gamma' \vdash t_1[X_1 := T_1] : T$
Without loss of generality we may assume $X_1 \notin FV(T_1)$.
Because size$(t_1[X_1 := T_1]) <$ size(t), the induction hypothesis gives us:
$\Gamma, \Gamma'[X := T'] \vdash t_1[X_1 := T_1][X := T'] : T[X := T']$.
We want to show $\Gamma, \Gamma'[X := T'] \vdash (\texttt{new}\ X_1 = T_1\ \texttt{in}\ t_1)[X := T'] : T[X := T']$
Without loss of generality we may assume $X_1 \notin FV(T')$ and $X \neq X_1$.
It suffices to show $\Gamma, \Gamma'[X := T'] \vdash t_1[X := T'][X_1 := T_1[X := T']] : T[X := T']$
and $\Gamma, \Gamma'[X := T'] \vdash T_1[X := T'] : \kappa$.
$\Gamma, \Gamma'[X := T'] \vdash T_1[X := T'] : \kappa$ follows from lemma 3.9, since $\Gamma, X : *, \Gamma' \vdash T_1 : \kappa$.
The substitution $[X := T'][X_1 := T_1[X := T']]$ is equivalent to $[X_1 := T_1][X := T']$.
$\Gamma, \Gamma'[X := T'] \vdash t_1[X_1 := T_1][X := T'] : T[X := T']$ follows by induction hypothesis.
□

We are now equipped to prove the preservation property:

Theorem 3.11 *(Preservation):*

$$\Gamma \vdash t|N : T\ \wedge\ t|N \longrightarrow t'|N' \quad \Longrightarrow \quad \Gamma \vdash t'|N' : T$$

Proof :
Let $\Gamma \vdash t|N : T\ \wedge\ t|N \longrightarrow t'|N'$. Then there exist a context E and terms t_0, t'_0 such that $t = E(t_0)$ and $t' = E(t'_0)$ and there are five possibilities:
$t_0 = (\lambda x : T_1.t_2)v\ \wedge\ t'_0 = t_2[x := v]\ \wedge\ N' = N$
or $t_0 = (\lambda X.t_2)T_1\ \wedge\ t'_0 = t_2[X := T_1]\ \wedge\ N' = N$
or $t_0 = \texttt{case}\ v : T_1\ \texttt{of}\ x : T_2 \Rightarrow t_2\ \texttt{else}\ t_3\ \wedge\ t'_0 = t_2[x := v]\ \wedge\ N' = N\ \wedge\ T_1 = T_2$
or $t_0 = \texttt{case}\ v : T_1\ \texttt{of}\ x : T_2 \Rightarrow t_2\ \texttt{else}\ t_3\ \wedge\ t'_0 = t_3\ \wedge\ N' = N\ \wedge\ T_1 \neq T_2$
or $t_0 = \texttt{new}\ X = T_1\ \texttt{in}\ t_1\ \wedge\ t'_0 = t_1\ \wedge\ N' = (N, X = T_1)$

Case $t_0 = (\lambda x : T_1.t_2)v\ \wedge\ t'_0 = t_2[x := v]\ \wedge\ N' = N$:
$\Longrightarrow \Gamma \vdash N((\lambda x : T_1.t_2)v) : T_2$ for some type T_2 by lemma 3.5, since $\Gamma \vdash E(t_0)|N : T$.
$\Longrightarrow \Gamma \vdash \lambda x : NT_1.Nt_2 : NT_1 \to T_2\ \wedge\ \Gamma \vdash Nv : NT_1$

$\Longrightarrow \Gamma, x : NT_1 \vdash Nt_2 : T_2 \ \wedge \ \Gamma \vdash Nv : NT_1$ (by inversion)
$\Longrightarrow \Gamma \vdash Nt_2[x := Nv] : T_2$ by lemma 3.7
$\Longrightarrow \Gamma \vdash N(t_2[x := v]) : T_2$.
$\Longrightarrow \Gamma \vdash E(t_2[x := v])|N : T$ by lemma 3.8, since $\Gamma \vdash E(t_0)|N : T$ and $\Gamma \vdash Nt_0 : T_2$.
$\Longrightarrow \Gamma \vdash E(t_0')|N' : T$, since $t_0' = t_2[x := v]$

Case $t_0 = (\lambda X.t_2)T_1 \ \wedge \ t_0' = t_2[X := T_1] \ \wedge \ N' = N$:
$\Longrightarrow \Gamma \vdash N((\lambda X.t_2)T_1) : T_2$ for some type T_2 by lemma 3.5, since $\Gamma \vdash E(t_0)|N : T$.
$\Longrightarrow \Gamma \vdash N(\lambda X.t_2) : \forall X.T_2'$ with $T_2 = T_2'[X := NT_1]$ and $\Gamma \vdash NT_1 : *$ (by inversion)
Without loss of generality we may assume $X \notin dom(N)$ and X does not occur in Γ.
$\Longrightarrow \Gamma, X : * \vdash Nt_2 : T_2'$ (by inversion)
$\Longrightarrow \Gamma \vdash (Nt_2)[X := NT_1] : T_2'[X := NT_1]$ by lemma 3.10
$\Longrightarrow \Gamma \vdash N(t_2[X := T_1]) : T_2$
$\Longrightarrow \Gamma \vdash E(t_2[X := T_1])|N : T$ by lemma 3.8, since $\Gamma \vdash E(t_0)|N : T$ and $\Gamma \vdash Nt_0 : T_2$.
$\Longrightarrow \Gamma \vdash E(t_0')|N' : T$, since $t_0' = t_2[X := T_1]$

Case $t_0 = \texttt{case} \ v : T_1 \ \texttt{of} \ x : T_2 \Rightarrow t_2 \ \texttt{else} \ t_3 \ \wedge \ t_0' = t_2[x := v] \ \wedge \ N' = N \ \wedge \ T_1 = T_2$:
$\Longrightarrow \Gamma \vdash Nt_0 : T_3$ for some type T_3 by lemma 3.5, since $\Gamma \vdash E(t_0)|N : T$.
$\Longrightarrow \Gamma, x : NT_2 \vdash Nt_2 : T_3 \ \wedge \ \Gamma \vdash Nv : NT_1$ (by inversion)
$\Longrightarrow \Gamma \vdash (Nt_2)[x := Nv] : T_3$ by lemma 3.10, since $T_1 = T_2$.
$\Longrightarrow \Gamma \vdash N(t_2[x := v]) : T_3$
$\Longrightarrow \Gamma \vdash E(t_2[x := v])|N : T$ by lemma 3.8, since $\Gamma \vdash E(t_0)|N : T$ and $\Gamma \vdash Nt_0 : T_3$.
$\Longrightarrow \Gamma \vdash E(t_0')|N' : T$, since $t_0' = t_2[x := v]$

Case $t_0 = \texttt{case} \ v : T_1 \ \texttt{of} \ x : T_2 \Rightarrow t_2 \ \texttt{else} \ t_3 \ \wedge \ t_0' = t_3 \ \wedge \ N' = N \ \wedge \ T_1 \neq T_2$:
$\Longrightarrow \Gamma \vdash Nt_0 : T_3$ for some type T_3 by lemma 3.5, since $\Gamma \vdash E(t_0)|N : T$.
$\Longrightarrow \Gamma \vdash Nt_3 : T_3$ (by inversion)
$\Longrightarrow \Gamma \vdash E(t_3)|N : T$ by lemma 3.8, since $\Gamma \vdash E(t_0)|N : T$ and $\Gamma \vdash Nt_0 : T_3$.
$\Longrightarrow \Gamma \vdash E(t_0')|N' : T$, since $t_0' = t_3$

Case $t_0 = \texttt{new} \ X = T_1 \ \texttt{in} \ t_1 \ \wedge \ t_0' = t_1 \ \wedge \ N' = (N, X = T_1)$:
$\Longrightarrow \Gamma \vdash N(\texttt{new} \ X = T_1 \ \texttt{in} \ t_1) : T_2$ for some type T_2 by lemma 3.5, since $\Gamma \vdash E(t_0)|N : T$.
Without loss of generality we may assume $X \notin dom(N)$.
$\Longrightarrow \Gamma \vdash \texttt{new} \ X = NT_1 \ \texttt{in} \ Nt_1 : T_2$
$\Longrightarrow \Gamma \vdash Nt_1[X := NT_1] : T_2 \ \wedge \ \Gamma \vdash NT_1 : *$ (by inversion)
$\Longrightarrow \Gamma \vdash N(t_1[X := T_1]) : T_2$
$\Longrightarrow \Gamma \vdash E(t_1[X := T_1])|N : T$ by lemma 3.8, since $\Gamma \vdash E(t_0)|N : T$ and $\Gamma \vdash Nt_0 : T_2$.
$\Longrightarrow \Gamma \vdash E(t_1)[X := T_1]|N : T \quad (X \text{ fresh})$
$\Longrightarrow \Gamma \vdash E(t_1)|(N, X = T_1) : T$
$\Longrightarrow \Gamma \vdash E(t_0')|N' : T$, since $t_0' = t_1$ and $N' = (N, X = T_1)$.

□

During reduction the state is only extended and never inspected, therefore the reduction rules are independent of the contents of the state. It is possible to remove the states from the reduction rules and to forget the information what type some variable X stands for. Thereby we would lose the ability to check the type of the intermediate configurations, but as we have proven the preservation property we know that reduction cannot go wrong. We know that there always exists some state that allows us to give a configuration the type we started with.

3.4 Normalisation

System F is a normalising calculus. It is interesting that this property does not hold for λ_F^N [10]. We will show this by constructing a diverging term. In the untyped λ-calculus a simple diverging term is $(\lambda x.xx)(\lambda x.xx)$. When this term is reduced by β-reduction the result is the term itself. In the simply typed λ-calculus, a term like $\lambda x : T_1.xx$ is not typable, because the typing rules require T_1 to be an arrow type containing itself as left component ($T_1 = T_1 \to T$). This would result in an infinite type. But in system F we can write a typable version of this term by using type application:

$$\lambda x : \forall X.X \to T.x(\forall X.X \to T)x$$

This term has the type $(\forall X.X \to T) \to T$ and it is still impossible to apply this term to itself. By using our type case we can write a similar term that allows self-application:

$$D = \lambda X.\lambda x : X.\texttt{case}\ x : X\ \texttt{of}\ x' : \forall X.X \to T \Rightarrow x'(\forall X.X \to T)x'\ \texttt{else}\ t_T$$

If t_T is a term of type T (and X is not free in T), then D is of type $\forall X.X \to T$ and therefore the term $\Omega = D(\forall X.X \to T)D$ is well-typed. Ω has the following reduction sequence:

$$\begin{aligned}
&D(\forall X.X \to T)D \\
\longrightarrow\ &(\lambda x : \forall X.X \to T.\texttt{case}\ x : \forall X.X \to T\ \texttt{of}\ x' : \forall X.X \to T \Rightarrow x'(\forall X.X \to T)x'\ \texttt{else}\ t_T)D \\
\longrightarrow\ &\texttt{case}\ D : \forall X.X \to T\ \texttt{of}\ x' : \forall X.X \to T \Rightarrow x'(\forall X.X \to T)x'\ \texttt{else}\ t_T \\
\longrightarrow\ &D(\forall X.X \to T)D \\
\longrightarrow\ &...
\end{aligned}$$

In a similar way it is possible to construct a fixed point operator . This shows that λ_F^N can not only express diverging terms, is it even powerful enough to express recursion.

$$\begin{aligned}
F = \lambda X_1.\lambda X_2.\lambda x_\perp : X_1 \to X_2.\lambda x_f : (X_1 \to X_2) \to X_1 \to X_2.\lambda X.\lambda x : X.\lambda x_1 : X_1. \\
x_f(\texttt{case}\ x : X\ \texttt{of}\ x_g : \forall X.X \to X_1 \to X_2 \Rightarrow x_g(\forall X.X \to X_1 \to X_2)x_g\ \texttt{else}\ x_\perp)x_1
\end{aligned}$$

$$\begin{aligned}
\mathit{fix} = \lambda X_1.\lambda X_2.\lambda x_\perp : X_1 \to X_2.\lambda x_f : (X_1 \to X_2) \to X_1 \to X_2. \\
(F\,X_1\,X_2\,x_\perp\,x_f)(\forall X.X \to X_1 \to X_2)(F\,X_1\,X_2\,x_\perp\,x_f)
\end{aligned}$$

As the `case` construct in F expects in its `else` part something of type $X_1 \to X_2$ we give *fix* the argument $x_\perp$. Therefore this fixed point operator is only usable for inhabited types $X_1 \to X_2$.

4 Laziness

In this section we will develop some constructs that allow us to change the evaluation strategy of our calculus from call-by-value to call-by-need which is also called lazy evaluation. This means evaluating a term at the latest possible time which can even mean that this term is never evaluated. A term has to be evaluated if it occurs as left-hand side of an application. The effect of this evaluation strategy is that we only evaluate terms we really need to evaluate. When working with modules a useful application is lazy linking, meaning that a module is only loaded if it is really used and not only contained in a branch of the program that will not be executed.

$x \in \mathit{Var}$			Variables
$X \in \mathit{TVar}$			Type Variables
$T \in \mathit{Typ}$	$::=$	$X \mid T \to T$	Types
$t \in \mathit{Ter}$	$::=$	$x \mid \lambda x : T.t \mid t\ t \mid \texttt{lazy}\ x\ \texttt{=}\ t\ \texttt{in}\ t$	Terms
$v \in \mathit{Val}$	$::=$	$\lambda x : T.t \mid x$	Values
$S \in \mathit{Stack}$	$::=$	$\phi \mid S, x$	Stacks
$\mu \in \mathit{State}$	$=$	$\mathit{Var} \xrightarrow{\mathit{fin}} \mathit{Ter}$	States
$\Gamma \in \mathit{Env}$	$=$	$\mathit{Var} \xrightarrow{\mathit{fin}} \mathit{Typ}$	Environments

Figure 4: λ_s^L-Syntax

4.1 Simply Typed Laziness

To understand lazy evaluation we will first formalise it for an extension of the simply typed λ-calculus we will call λ_s^L. An example of a concurrent calculus that provides a form of laziness is [6]. Some of its concepts will be used in our calculus, but it will be deterministic. For enabling laziness we will use a construct that corresponds to the lazy expressions found in Alice [2]:

$$\texttt{lazy}\ x\ \texttt{=}\ t\ \texttt{in}\ t'$$

Like in a `let` construct the variable x can be used in t' as a name for the term t. But the semantics consists in evaluating t' and delaying the evaluation of t until we reach a term of the form $x\,v$. Here x occurs as left hand side of an application, which means that we cannot proceed without evaluating t. When the evaluation of t is finished, we substitute all occurrences of x with its result and proceed with evaluating t'. The substitution of all occurrences of x prevents that t is evaluated several times as it would happen, if during further reduction we reached another term of the form $x\,v'$.

During the evaluation of t it may happen that a further lazy term needs to be evaluated and this term can trigger the evaluation of even more lazy terms. Therefore we must organise the evaluations of these terms somehow. The natural way to do this is to use a stack storing the information which terms we need to evaluate. To indicate that some term t needs to be evaluated next we will push its corresponding variable x on top of the stack.

Additionally we also need to memorise the information that a variable x is a lazy name for a term t. For this purpose we introduce a finite partial function $\mu \in \mathit{Var} \xrightarrow{\mathit{fin}} \mathit{Ter}$ with elements of the form $x = t$. We will call such functions *states*. If we write $\mu, x = t$ we mean the state $\mu \cup \{x = t\}$ where $x \notin \mathit{dom}(\mu)$. For simplicity we also model environments as functions mapping variables to their type. The list construction as in the development of λ_F^N is not necessary since we do not need an order of the variables. We will write $\Gamma, x : T$ analogically to the notation for states.

Figure 4 contains the complete syntax of λ_s^L. Since we want to use evaluation contexts like in figure 2 the values need to include variables. This prevents the attempt to evaluate lazy variables if they occur as right-hand side of an application.

The reduction rules in figure 5 are defined over pairs of states μ and stacks S, written $\mu|S$. We call such pairs *configurations*. If we want to evaluate some term t, we start with the configuration $\{x = t\}|\phi, x$ where x is an arbitrary variable. We always reduce the term whose variable is on top of the stack. If we reduce a `lazy` term, a new entry is added to the state

$E ::= \circ \mid E\ t \mid v\ E$

$$
\begin{array}{lrcll}
(1) & \mu, x = E((\lambda x' : T.t)\, v) \mid S, x & \longrightarrow & \mu, x = E(t[x' := v]) \mid S, x & \\
(2) & \mu, x = E(\texttt{lazy}\ x_1\ \texttt{=}\ t_1\ \texttt{in}\ t_2) \mid S, x & \longrightarrow & \mu, x_1 = t_1, x = E(t_2) \mid S, x & (x_1 \text{ fresh}) \\
(3) & \mu, x = E(x_1\, v) \mid S, x & \longrightarrow & \mu, x = E(x_1\, v) \mid S, x, x_1 & (\text{if } x_1 \in dom(\mu)) \\
(4) & \mu, x = v \mid S, x & \longrightarrow & \mu[x := v] \mid S & (\text{if } S \neq \phi)
\end{array}
$$

Figure 5: λ_s^L-Reduction

and if we attempt to reduce an application with a lazy variable as left-hand side, we push this variable onto the stack, which results in the evaluation of its corresponding term.

The last reduction rule treats the situation that the term to be reduced is already a value. If the stack contains more than one variable, we can pop the topmost variable from the stack and substitute all its occurrences with this value. If the stack contains only one variable, this means that the whole evaluation is done.

We will now explore what it means for a configuration to be well-typed. Let us suppose the state of a configuration contains the elements $x_1 = E(x_2)$ and $x_2 = E(x_1)$. Here x_2 is free in the term of x_1 and x_1 is free in the term of x_2. It is obvious that the types of x_1 and x_2 depend on each other, which means that in general we cannot give a unique type to x_1 if we do not know the type of x_2 and vice versa. Therefore we must forbid such situations in our typing rules. Let us regard the inner dependencies of a state. A lazy variable x_1 is called *directly dependent* on a lazy variable x_2 iff x_2 is free in the lazy term of x_1. The following relation defines the direct dependencies in some state μ:

$$dep_\mu = \{(x_1, x_2) \mid \{x_1, x_2\} \subseteq dom(\mu)\ \wedge\ x_2 \in FV(\mu\, x_1)\}$$

We will write $<_\mu$ for the transitive closure of dep_μ. In general $x_1 <_\mu x_2$ implies that we cannot give a type to x_1, if we do not know the type of x_2. We require $<_\mu$ to be acyclic as this implies the existence of variables without dependencies. Such variables are called *maximal*. Analogically a variable is called *minimal* iff there exists no variable that depends on it. In section 4.2.1 we will prove that acyclic states allow for unique typing.

As you can see in the third reduction rule a growing stack respects the order defined by the state, i.e. the pushed variable is greater than the variable on top of the stack. We will include this property in our typing rules as we need it to argue about type preservation in the fourth reduction rule.

Figure 6 contains the typing rules of λ_s^L. The typing rule for configurations requires the existence of an environment that gives every lazy term the type of its respective variable. The type of a configuration is the type this environment maps the minimal stack variable x_0 to. The rule for typing a `lazy` term is like for `let` constructs.

4.2 Properties of λ_s^L

For the proofs we need an inversion lemma like it is formulated in section 3.1. We will not write it down explicitly, since such a lemma is trivial to formulate and to prove.

4.2.1 Uniqueness of Types

Lemma 4.1 *(Uniqueness for terms):*

Terms

$$\frac{x \in dom(\Gamma)}{\Gamma \vdash x \ : \ \Gamma\ x} \qquad \frac{\Gamma \vdash t : T' \to T \quad \Gamma \vdash t' : T'}{\Gamma \vdash t\ t' : T}$$

$$\frac{\Gamma, x : T \vdash t : T'}{\Gamma \vdash \lambda x : T.t \ : \ T \to T'} \qquad \frac{\Gamma \vdash t : T \quad \Gamma, x : T \vdash t' : T'}{\Gamma \vdash \texttt{lazy}\ x\ \texttt{=}\ t\ \texttt{in}\ t' : T'}$$

Stacks

$$\frac{x \in dom(\mu)}{\mu \vdash \phi, x : x} \qquad \frac{x_1 <_\mu x_2 \quad \mu \vdash S, x_1 : x}{\mu \vdash S, x_1, x_2 : x}$$

Configurations

$$\frac{\begin{array}{ccc} dom(\Gamma) \cap dom(\mu) = \emptyset & \mu \vdash S : x_0 & <_\mu \ acyclic \\ \exists \Gamma' \supseteq \Gamma : dom(\Gamma') = dom(\Gamma) \cup dom(\mu) \ \wedge \ \Gamma' x_0 = T \ \wedge \ \forall x = t \in \mu : \Gamma' \vdash t : \Gamma' x \end{array}}{\Gamma \vdash \mu | S : T}$$

Figure 6: λ_s^L-Typing

$$\Gamma \vdash t : T \ \wedge \ \Gamma \vdash t : T' \quad \Longrightarrow \quad T = T'$$

Proof : Structural induction on t
The cases $t = x$, $t = \lambda x : T_1.t_2$ and $t = t_1\ t_2$ are like in the proof of theorem 3.2. The only new case is the one for the `lazy` construct:

Case $t = \texttt{lazy}\ x\ \texttt{=}\ t_1\ \texttt{in}\ t_2$:
Let $\Gamma \vdash t : T \ \wedge \ \Gamma \vdash t : T'$
By inversion it follows that $\Gamma \vdash t_1 : T_1 \ \wedge \ \Gamma, x : T_1 \vdash t_2 : T$ and
$\Gamma \vdash t_1 : T_1' \ \wedge \ \Gamma, x : T_1' \vdash t_2 : T'$ for some types T_1 and T_1'
The induction hypothesis gives us $T_1 = T_1'$ and $T = T'$.

□

To prove the uniqueness for configurations we need to prove that for well-typed configurations the existence of the environment Γ' is unique. This follows immediately from this lemma:

Lemma 4.2 *(Unique environments):*

$$<_\mu \ acyclic \ \wedge \ dom(\Gamma) \cap dom(\mu) = \emptyset \ \wedge \ dom(\Gamma') = dom(\Gamma'') = dom(\Gamma) \cup dom(\mu) \ \wedge$$
$$\Gamma' \supseteq \Gamma \ \wedge \ \Gamma'' \supseteq \Gamma \ \wedge \ \forall x = t \in \mu : \Gamma' \vdash t : \Gamma' x \ \wedge \ \Gamma'' \vdash t : \Gamma'' x$$
$$\Longrightarrow \quad \Gamma' = \Gamma''$$

Proof : Induction on $|\mu|$
Case $|\mu| = 0$:
trivial.

Case $|\mu| > 0$:
As $<_\mu$ is acyclic it must hold: $\exists \mu_1, x_1, t_1 : \mu = (\mu_1, x_1 = t_1) \ \wedge \ x_1$ is maximal.

$\Longrightarrow \forall x \in FV(t_1) : x \in dom(\Gamma)$
$\Longrightarrow \Gamma \vdash t_1 : \Gamma' x_1 \ \wedge \ \Gamma \vdash t_1 : \Gamma'' x_1$
From lemma 4.1 it follows $\Gamma' x_1 = \Gamma'' x_1$
Let $T_1 = \Gamma' x_1$. It follows:
$<_{\mu_1}$ *acyclic* $\wedge$ $dom(\Gamma, x_1 : T_1) \cap dom(\mu_1) = \emptyset \wedge$
$dom(\Gamma') = dom(\Gamma'') = dom(\Gamma, x_1 : T_1) \cup dom(\mu_1) \wedge$
$\Gamma' \supseteq \Gamma, x_1 : T_1 \ \wedge \ \Gamma'' \supseteq \Gamma, x_1 : T_1 \ \wedge \ \forall x = t \in \mu_1 : \Gamma' \vdash t : \Gamma' x \ \wedge \ \Gamma'' \vdash t : \Gamma'' x$
By induction hypothesis it follows $\Gamma' = \Gamma''$

□

Corollary 4.3 *(Uniqueness):*

$$\Gamma \vdash \mu|S : T \ \wedge \ \Gamma \vdash \mu|S : T' \quad \Longrightarrow \quad T = T'$$

Proof : Immediately from lemma 4.2 with use of the inversion lemma.

□

4.2.2 Progress

At first we prove that every term matches a pattern of the reduction rules:

Lemma 4.4 *(Context structure):*

$$\forall t : t \in Val \ \vee \ \exists E, v_1, v_2, x, t', t'' : \ t = E(v_1\, v_2) \ \vee \ t = E(\texttt{lazy } x \texttt{ = } t' \texttt{ in } t'')$$

Proof : Structural induction on t
Case $t = x_1 \ \vee \ t = \lambda x_1.t_1 \ \vee \ t = \texttt{lazy } x_1 \texttt{ = } t_1 \texttt{ in } t_2$:
trivial.

Case $t = t_1\, t_2$:
Case $t_1 \notin Val$:
By induction hypothesis it follows:
There exist E, v_1, v_2, x, t', t'' such that $t_1 = E(v_1\, v_2) \ \vee \ t_1 = E(\texttt{lazy } x \texttt{ = } t' \texttt{ in } t'')$.
Let $E' = E\, t_2$
$\Longrightarrow t = E'(v_1\, v_2) \ \vee \ t = E'(\texttt{lazy } x \texttt{ = } t' \texttt{ in } t'')$

Case $t_1 \in Val \ \wedge \ t_2 \notin Val$:
analogically to case $t_1 \notin Val$.

Case $t_1 \in Val \ \wedge \ t_2 \in Val$:
$\Longrightarrow$ with $E = \circ$ it follows $t = E(t_1\, t_2)$ (trivial).

□

Theorem 4.5 *(Progress):*

$$\vdash \mu|S : T \quad \Longrightarrow \quad (\exists \mu_1, x, v : \mu = (\mu_1, x = v) \ \wedge \ S = \phi, x) \ \vee \ \exists \mu', S' : \mu|S \longrightarrow \mu'|S'$$

Proof :
Let $\vdash \mu|S : T$.
$\Longrightarrow \exists \mu_1, S_1, x, t : \mu = (\mu_1, x = t) \ \wedge \ S = S_1, x$ (by inversion μ and S must not be empty)
From lemma 4.4 follows:
$t \in \mathit{Val} \ \vee \ \exists E, v_1, v_2, x_1, t_1, t_2 : \ t = E(v_1\, v_2) \ \vee \ t = E(\texttt{lazy}\ x_1\ \texttt{=}\ t_1\ \texttt{in}\ t_2)$
Case $t \in \mathit{Val} \ \wedge \ S_1 = \phi$:
trivial.

Case $t \in \mathit{Val} \ \wedge \ S_1 \neq \phi$:
$\Longrightarrow \mu|S \longrightarrow \mu_1[x := t]|S_1$

Case $t = E(v_1\, v_2)$:
Case $v_1 = \lambda x_1 : T_1.t_1$:
$\Longrightarrow \mu|S \longrightarrow \mu_1, x = E(t_1[x_1 := v_2])|S$
Case $v_1 = x_1$:
From $\vdash \mu|S : T$ it follows that $x_1 \in \mathit{dom}(\mu)$.
$\Longrightarrow \mu|S \longrightarrow \mu|S, x_1$

Case $t = E(\texttt{lazy}\ x_1\ \texttt{=}\ t_1\ \texttt{in}\ t_2)$:
$\Longrightarrow \mu|S \longrightarrow \mu_1, x_1 = t_1, x = E(t_2)|S$

□

4.2.3 Preservation

To prove the preservation property we lemmas similar to the ones in section 3.3.

Lemma 4.6 *:*

$$\Gamma \vdash E(t) : T \quad \Longrightarrow \quad \exists T' : \Gamma \vdash t : T'$$

Proof : Structural induction on E.
Cases are similar to the ones for lemma 3.5.

□

Lemma 4.7 *(Weakening):*

$$\Gamma \vdash t : T \ \wedge \ x \notin \mathit{dom}(\Gamma) \quad \Longrightarrow \quad \Gamma, x : T' \vdash t : T$$

Proof : Straightforward induction on t.

□

Lemma 4.8 *(Substitution):*

$$\Gamma, x : T' \vdash t : T \ \wedge \ \Gamma \vdash t' : T' \quad \Longrightarrow \quad \Gamma \vdash t[x := t'] : T$$

Proof : Structural induction on t
The cases $t = x$, $t = \lambda x : T_1.t_2$ and $t = t_1\ t_2$ are like in the proof for lemma 3.7. The only new case is the one for the `lazy` construct:

Case $t = \texttt{lazy}\ x_1\ \texttt{=}\ t_1\ \texttt{in}\ t_2$:

Let $\Gamma, x : T' \vdash t : T \ \wedge \ \Gamma \vdash t' : T'$.
By inversion it follows $\Gamma, x : T' \vdash t_1 : T_1 \ \wedge \ \Gamma, x : T', x_1 : T_1 \vdash t_2 : T$
The weakening lemma 4.7 gives us $\Gamma, x_1 : T_1 \vdash t' : T'$
By induction hypothesis it follows $\Gamma \vdash t_1[x := t'] : T_1 \ \wedge \ \Gamma, x_1 : T_1 \vdash t_2[x := t'] : T$
This implies $\Gamma \vdash (\texttt{lazy}\ x_1\ \texttt{=}\ t_1\ \texttt{in}\ t_2)[x := t'] : T$

□

Lemma 4.9 *:*

$$\Gamma \vdash E(t) : T \ \wedge \ \Gamma \vdash t : T' \ \wedge \ \Gamma \vdash t' : T' \quad \Longrightarrow \quad \Gamma \vdash E(t') : T$$

Proof : Structural induction on E.
Cases are similar to the ones for lemma 3.8.

□

Theorem 4.10 *(Preservation):*

$$\Gamma \vdash \mu|S : T \ \wedge \ \mu|S \longrightarrow \mu'|S' \quad \Longrightarrow \quad \Gamma \vdash \mu'|S' : T$$

Proof :
Let $\Gamma \vdash \mu|S : T \ \wedge \ \mu|S \longrightarrow \mu'|S'$.
By inversion it follows:
$dom(\Gamma) \cap dom(\mu) = \emptyset \ \wedge \ \mu \vdash S : x_0 \ \wedge \ <_\mu$ *acyclic* and there exists some $\Gamma' \supseteq \Gamma$ such that
$dom(\Gamma') = dom(\Gamma) \cup dom(\mu) \ \wedge \ \Gamma' x_0 = T \ \wedge \ \forall x = t \in \mu : \Gamma' \vdash t : \Gamma' x$.
Furthermore there are four possibilities:
Case 1: $\mu|S = (\mu_1, x = E((\lambda x_1 : T_1.t_1)v))|S_1, x \ \wedge \ \mu'|S' = (\mu_1, x = E(t_1[x_1 := v]))|S_1, x$
Case 2: $\mu|S = (\mu_1, x = E(\texttt{lazy}\ x_1\ \texttt{=}\ t_1\ \texttt{in}\ t_2))|S_1, x \wedge \mu'|S' = (\mu_1, x_1 = t_1, x = E(t_2))|S_1, x$
Case 3: $\mu|S = (\mu_1, x = x_1\, v)|S_1, x \ \wedge \ \mu'|S' = (\mu_1, x = x_1\, v)|S_1, x, x_1 \ \wedge \ x_1 \in dom(\mu_1)$
Case 4: $\mu|S = (\mu_1, x = v)|S_1, x \ \wedge \ \mu'|S' = \mu_1[x := v]|S_1 \ \wedge \ S_1 \neq \phi$

Case 1: $\mu|S = (\mu_1, x = E((\lambda x_1 : T_1.t_1)v))|S_1, x \ \wedge \ \mu'|S' = (\mu_1, x = E(t_1[x_1 := v]))|S_1, x$
$\Longrightarrow \Gamma' \vdash E((\lambda x_1 : T_1.t_1)v) : \Gamma' x$ (by inversion)
$\Longrightarrow \Gamma' \vdash (\lambda x_1 : T_1.t_1)v : T_0$ for some type T_0 by lemma 4.6.
$\Longrightarrow \Gamma' \vdash (\lambda x_1 : T_1.t_1) : T_1 \to T_0 \ \wedge \ \Gamma' \vdash v : T_1$ (by inversion)
$\Longrightarrow \Gamma', x_1 : T_1 \vdash t_1 : T_0$ (by inversion)
$\Longrightarrow \Gamma' \vdash t_1[x_1 := v] : T_0$ by lemma 4.8, since $\Gamma' \vdash v : T_1$.
Since $\Gamma' \vdash E((\lambda x_1 : T_1.t_1)v) : \Gamma' x$ and $\Gamma' \vdash (\lambda x_1 : T_1.t_1)v : T_0$ and $\Gamma' \vdash t_1[x_1 := v] : T_0$,
it follows by lemma 4.9:
$\Gamma' \vdash E(t_1[x_1 := v]) : \Gamma' x$
Therefore it holds: $\forall x = t \in \mu' : \Gamma' \vdash t : \Gamma' x$
The state μ' does not contain more dependencies than μ: $dep_{\mu'} \subseteq dep_\mu$
$\Longrightarrow <_{\mu'}$ is acyclic, since $<_\mu$ is acyclic.
As we changed only the term of the topmost stack variable, the stack S' respects μ':
$\mu' \vdash S' : x_0$
Therefore it holds: $\Gamma \vdash \mu'|S' : T$

Case 2: $\mu|S = (\mu_1, x = E(\texttt{lazy}\ x_1\ \texttt{=}\ t_1\ \texttt{in}\ t_2))|S_1, x \wedge \mu'|S' = (\mu_1, x_1 = t_1, x = E(t_2))|S_1, x$
$\Longrightarrow \Gamma' \vdash E(\texttt{lazy}\ x_1\ \texttt{=}\ t_1\ \texttt{in}\ t_2) : \Gamma' x$ (by inversion)

$\Longrightarrow \Gamma' \vdash \texttt{lazy}\ x_1\ \texttt{=}\ t_1\ \texttt{in}\ t_2 : T_0$ for some type T_0 by lemma 4.6.
$\Longrightarrow \Gamma' \vdash t_1 : T_1\ \wedge\ \Gamma', x_1 : T_1 \vdash t_2 : T_0$ for some type T_1 by inversion (x_1 is fresh).
By lemma 4.7 it follows:
$\Gamma', x_1 : T_1 \vdash t_1 : (\Gamma', x_1 : T_1)x_1$ and $\Gamma', x_1 : T_1 \vdash \texttt{lazy}\ x_1\ \texttt{=}\ t_1\ \texttt{in}\ t_2 : T_0$
Furthermore lemma 4.7 gives us: $\forall x = t \in \mu : \Gamma', x_1 : T_1 \vdash t : (\Gamma', x_1 : T_1)x$
$\Longrightarrow \Gamma', x_1 : T_1 \vdash E(\texttt{lazy}\ x_1\ \texttt{=}\ t_1\ \texttt{in}\ t_2) : (\Gamma', x_1 : T_1)x$
Since $\Gamma', x_1 : T_1 \vdash \texttt{lazy}\ x_1\ \texttt{=}\ t_1\ \texttt{in}\ t_2 : T_0$ and $\Gamma', x_1 : T_1 \vdash t_2 : T_0$ it follows by lemma 4.9:
$\Longrightarrow \Gamma', x_1 : T_1 \vdash E(t_2) : (\Gamma', x_1 : T_1)x_1$
Therefore it holds: $\forall x = t \in \mu' : \Gamma', x_1 : T_1 \vdash t : (\Gamma', x_1 : T_1)x$
Furthermore we have: $dom(\Gamma) \cap dom(\mu') = \emptyset$ and $dom(\Gamma', x_1 : T_1) = dom(\Gamma) \cup dom(\mu')$
As the terms responsible for the dependencies of the stack variables are not changed in μ', it holds that $\mu' \vdash S' : x_0$.
The last thing to show is that $<_{\mu'}$ is acyclic. Obviously $\mu' \backslash \{x_1 = t_1\}$ is acyclic.
The only variable in μ' that can depend on x_1 is x, but x_1 cannot depend on x.
$\Longrightarrow <_{\mu'}$ is acyclic.
Therefore it holds: $\Gamma \vdash \mu' | S' : T$

Case 3: $\mu | S = (\mu_1, x = x_1\, v) | S_1, x\ \wedge\ \mu' | S' = (\mu_1, x = x_1\, v) | S_1, x, x_1\ \wedge\ x_1 \in dom(\mu_1)$
As $x = x_1\, v \in \mu$ it follows: $x <_\mu x_1$.
$\Longrightarrow \mu' \vdash S' : x_0$, since $\mu \vdash S : x_0$ and $\mu = \mu'$.
Therefore it holds: $\Gamma \vdash \mu' | S' : T$

Case 4: $\mu | S = (\mu_1, x = v) | S_1, x\ \wedge\ \mu' | S' = \mu_1[x := v] | S_1\ \wedge\ S_1 \neq \phi$
All free lazy variables in v are greater than x. Variables depending on x are less than x.
Therefore the substitution $[x := v]$ cannot introduce new dependencies.
$\Longrightarrow <_{\mu'}$ is acyclic.
The substitution does no affect dependencies between variables which are smaller than x.
Since S_1 contains only variables which are smaller than x, it follows: $\mu' \vdash S' : x_0$
There exist Γ_1, T_1 such that $\Gamma' = \Gamma_1, x : T_1\ \wedge\ \Gamma_1 \vdash v = T_1$. Lemma 4.8 gives us:
$\forall x_1 = t_1 \in \mu_1 : \Gamma_1 \vdash t_1[x := v] : \Gamma' x_1$
$\Longrightarrow \forall x_1 = t_1 \in \mu' : \Gamma_1 \vdash t_1 : \Gamma_1 x_1$
As $dom(\Gamma) \cap dom(\mu') = \emptyset$ and $dom(\Gamma_1) = dom(\Gamma) \cup dom(\mu')$ it holds: $\Gamma \vdash \mu' | S' : T$

□

4.3 Lazy Linking

In open programming systems loading modules may be quite expensive if for example the module is only available over a slow network connection. Therefore it is of interest to load modules only if they are really needed and not only used in some branch of the program that will not be executed. We want a module to be loaded at the latest possible time which includes that the module might never be loaded. Such a mechanism is called lazy linking. As we have seen before dynamic linking needs a form of dynamic type analysis. Therefore we will formalise lazy linking in a calculus which provides the type case we have used before. We will not include a construct for dynamic type name generation, because it is not needed to understand lazy linking. The laziness formalised in the previous section provides some background we will use in the development of this calculus, λ_F^L, an extension of system F.

We will use abstract types (see section 1.2) as modules. Basically all modules we consider consist of some type T and some term t that may provide operations to work with instances of type T. Therefore loading a module introduces two new names which can be used for type T and term t. The `lazy` construct we defined previously introduces only one binder, hence we change its syntax to the following form:

`lazy` $<X, x>$ `=` t `in` t'

Here we require t to be a module. The scope of the type name X and the variable x is the term t'. There are two possible situations which can trigger the evaluation of term t. In the first one x occurs as left hand side of an application. During the reduction of a type case the two contained types need to be compared. If X is free in one of these types, the substitution of X with its representation may change the outcome of this comparison. Therefore we will evaluate a module if its corresponding type name is free in a type case. This strategy is a bit too eager, since it can trigger the evaluation of modules even if the outcome of the comparison is determined in advance, but it simplifies the needed reduction rules.

As we allow not only lazy term variables but also lazy type variables the state we use contains elements of the following form: $<X, x> = t$. Since the variables X and x of an abstract type always occur in combination, it is sufficient to store only term variables in the stack. We want a state to bind every variable once at the most. In the simply typed case this is valid because a state is a function. Since states map pairs of variables, this property does not hold anymore. In the following we only consider states where each variables occurs in the domain once at the most. If we write $BV(\mu)$ for some state μ, this is the set of all variables bound by μ.

To express modules (or abstract types) we will not use the encoding explained in section 1.2. In this encoding the implementation is passed to the clients by applying the abstract type to them. This scheme is not adaptable for our state and stack model as we cannot apply a term to a state or even a configuration. Therefore we will extend the calculus with constructs to express abstract types. We will use the construction developed by Mitchell and Plotkin [5] who gave abstract types an existential type. The notation we will use to express abstract types is:

$<T, t>$ `as` $\exists X.T'$

Here T is the representation type, t contains the operations and T' is the signature using the type name X. We impose a small restriction on type abstraction as abstract types must not provide more than one operation (the one of type T'). This restriction can be compensated for by including things like records in the calculus, which would provide a way to pack more than one operation in the type T'. We will not include records in our calculus to keep it simpler and focus on the more important topics.

The complete syntax of the calculus λ_F^L can be found in figure 7. As you can see, we model environments as lists again. This will simplify the rules for well-formed environments. The reduction rules in figure 8 are similar to the ones of λ_s^L and system F. Rule eight is the one which triggers the evaluation of modules, whose type name is free in a type case. Since such a type case can contain several type names and we want to formulate deterministic reduction rules, we need to clarify which module should be evaluated first. For this purpose we introduce the function first_μ which returns deterministically the variable of such a module. As we must

$x \in \mathit{Var}$			Variables
$X \in \mathit{TVar}$			Type Variables
$T \in \mathit{Typ}$	$::=$	$X \mid T \to T \mid \forall X.T \mid \exists X.T$	Types
$t \in \mathit{Ter}$	$::=$	$x \mid \lambda x:T.t \mid t\ t \mid \lambda X.t \mid t\ T \mid$ `case` $v:T$ `of` $x:T \Rightarrow t$ `else` t $\mid$ `lazy <`X, x`> =` t `in` $t \mid$ `<`T, t`> as` T	Terms
$v \in \mathit{Val}$	$::=$	$\lambda x:T.t \mid \lambda X.t \mid x \mid$ `<`T, v`> as` T	Values
$\kappa \in \mathit{Kind}$	$::=$	$*$	Kinds
$\Gamma \in \mathit{Env}$	$::=$	$\emptyset \mid \Gamma, x:T \mid \Gamma, X:\kappa$	Environments
$\mu \in \mathit{State}$	$=$	$\mathit{TVar} \times \mathit{Var} \xrightarrow{fin} \mathit{Ter}$	States
$S \in \mathit{Stack}$	$::=$	$\phi \mid S, x$	Stacks

Figure 7: λ_F^L-Syntax

evaluate all these modules, it is not essential what strategy this function uses to determine a module. One possibility is to select the module whose type name is in the leftmost position in the types T and T' of the type case.

For the typing relations we have to define, what the dependency relation looks like:

$$\mathit{dep}_\mu = \{(x_1, x_2) \mid \exists X_1, X_2 : \{\texttt{<}X_1, x_1\texttt{>}, \texttt{<}X_2, x_2\texttt{>}\} \subseteq \mathit{dom}(\mu) \ \wedge\ \{X_2, x_2\} \cap FV(\mu\, x_1) \neq \emptyset\}$$

Now the typing rules in figure 9 are straightforward. As usual for abstract types we require the condition that the type of a `lazy` construct must not contain the abstract type.

$E ::= \circ \mid E\ t \mid v\ E \mid E\ T \mid \texttt{<}T, E\texttt{>}\ \texttt{as}\ T$

(1) $\quad \mu, \texttt{<}X, x\texttt{>} = E((\lambda x' : T.t)\, v) \mid S, x$
$\longrightarrow \mu, \texttt{<}X, x\texttt{>} = E(t[x' := v]) \mid S, x$

(2) $\quad \mu, \texttt{<}X, x\texttt{>} = E((\lambda X.t)\, T) \mid S, x$
$\longrightarrow \mu, \texttt{<}X, x\texttt{>} = E(t[X := T]) \mid S, x$

(3) $\quad \mu, \texttt{<}X, x\texttt{>} = E(\texttt{lazy}\ \texttt{<}X_1, x_1\texttt{>}\ \texttt{=}\ t_1\ \texttt{in}\ t_2) \mid S, x$
$\longrightarrow \mu, \texttt{<}X_1, x_1\texttt{>} = t_1, \texttt{<}X, x\texttt{>} = E(t_2) \mid S, x$ $\qquad (X_1 \text{ and } x_1 \text{ fresh})$

(4) $\quad \mu, \texttt{<}X, x\texttt{>} = E(x_1\, v) \mid S, x$
$\longrightarrow \mu, \texttt{<}X, x\texttt{>} = E(x_1\, v) \mid S, x, x_1$ $\qquad (\text{if } x_1 \in BV(\mu))$

(5) $\quad \mu, \texttt{<}X, x\texttt{>} = E(x_1\, T) \mid S, x$
$\longrightarrow \mu, \texttt{<}X, x\texttt{>} = E(x_1\, T) \mid S, x, x_1$ $\qquad (\text{if } x_1 \in BV(\mu))$

(6) $\quad \mu, \texttt{<}X, x\texttt{>} = E(\texttt{case}\ v : T\ \texttt{of}\ x' : T' \Rightarrow t\ \texttt{else}\ t') \mid S, x$
$\longrightarrow \mu, \texttt{<}X, x\texttt{>} = E(t[x' := v]) \mid S, x$ $\qquad \begin{pmatrix} BV(\mu) \cap FV(T, T') = \emptyset \\ T = T' \end{pmatrix}$

(7) $\quad \mu, \texttt{<}X, x\texttt{>} = E(\texttt{case}\ v : T\ \texttt{of}\ x' : T' \Rightarrow t\ \texttt{else}\ t') \mid S, x$
$\longrightarrow \mu, \texttt{<}X, x\texttt{>} = E(t') \mid S, x$ $\qquad \begin{pmatrix} BV(\mu) \cap FV(T, T') = \emptyset \\ T \neq T' \end{pmatrix}$

(8) $\quad \mu, \texttt{<}X, x\texttt{>} = E(\texttt{case}\ v : T\ \texttt{of}\ x' : T' \Rightarrow t\ \texttt{else}\ t') \mid S, x$
$\longrightarrow \mu, \texttt{<}X, x\texttt{>} = E(\texttt{case}\ v : T\ \texttt{of}\ x' : T' \Rightarrow t\ \texttt{else}\ t') \mid S, x, x_1$ $\qquad \begin{pmatrix} BV(\mu) \cap FV(T, T') \neq \emptyset \\ \mathit{first}_\mu(T, T') = x_1 \end{pmatrix}$

(9) $\quad \mu, \texttt{<}X, x\texttt{>} = \texttt{<}T, v\texttt{>}\ \texttt{as}\ T' \mid S, x$
$\longrightarrow \mu[X := T][x := v] \mid S$ $\qquad (\text{if } S \neq \phi)$

Figure 8: λ_F^L-Reduction

Environments

$$\frac{}{\vdash \emptyset} \qquad \frac{\Gamma \vdash T : * \quad x \notin dom(\Gamma)}{\vdash \Gamma, x : T} \qquad \frac{\vdash \Gamma \quad X \notin dom(\Gamma)}{\vdash \Gamma, X : \kappa}$$

Types

$$\frac{\vdash \Gamma \quad X \in dom(\Gamma)}{\Gamma \vdash X : \Gamma X} \qquad \frac{\Gamma \vdash T_1 : * \quad \Gamma \vdash T_2 : *}{\Gamma \vdash T_1 \to T_2 : *} \qquad \frac{\Gamma, X : * \vdash T : *}{\Gamma \vdash \forall X.T : *}$$

Terms

$$\frac{\vdash \Gamma \quad x \in dom(\Gamma)}{\Gamma \vdash x \ : \ \Gamma\ x} \qquad \frac{\Gamma, X : * \vdash t : T}{\Gamma \vdash \lambda X.t \ : \ \forall X.T}$$

$$\frac{\Gamma, x : T \vdash t : T'}{\Gamma \vdash \lambda x : T.t \ : \ T \to T'} \qquad \frac{\Gamma \vdash t : \forall X.T' \quad \Gamma \vdash T : *}{\Gamma \vdash t\ T : T'[X := T]}$$

$$\frac{\Gamma \vdash t : T' \to T \quad \Gamma \vdash t' : T'}{\Gamma \vdash t\ t' : T} \qquad \frac{\Gamma \vdash v : T \quad \Gamma, x : T' \vdash t : T'' \quad \Gamma \vdash t' : T''}{\Gamma \vdash \texttt{case}\ v : T\ \texttt{of}\ x : T' \Rightarrow t\ \texttt{else}\ t' \ : \ T''}$$

$$\frac{\Gamma \vdash t : T'[X := T]}{\Gamma \vdash \texttt{<}T, t\texttt{>}\ \texttt{as}\ \exists X.T' : \exists X.T'} \qquad \frac{\Gamma \vdash t : \exists X.T \quad \Gamma, X : *, x : T \vdash t' : T' \quad X \notin FV(T')}{\Gamma \vdash \texttt{lazy <}X, x\texttt{> = }t\ \texttt{in}\ t' : T'}$$

Stacks

$$\frac{x \in BV(\mu)}{\mu \vdash \phi, x : x} \qquad \frac{x_1 <_\mu x_2 \quad \mu \vdash S, x_1 : x}{\mu \vdash S, x_1, x_2 : x}$$

Configurations

$$\frac{\begin{array}{c} dom(\Gamma) \cap BV(\mu) = \emptyset \qquad \mu \vdash S : x_0 \qquad <_\mu \ acyclic \\ \exists \Gamma' \supseteq \Gamma : dom(\Gamma') = dom(\Gamma) \cup BV(\mu) \ \wedge \ \Gamma' x_0 = T \ \wedge \ \forall x = t \in \mu : \Gamma' \vdash t : \Gamma' x \end{array}}{\Gamma \vdash \mu | S : T}$$

Figure 9: λ_F^L-Typing

References

[1] M. Abadi, L. Cardelli, B. Pierce, and D. Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111-130, Jan. 1995.

[2] Alice Team. *The Alice System.* Programming Systems Lab, Saarland University. `http://www.ps.uni-sb.de/alice/`. 2003

[3] Catherine Dubois, François Rouaix, Pierre Weis. Extensional polymorphism. *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p.118-129, January 23-25, 1995, San Francisco, California, United States.

[4] Robert Harper, Greg Morrisett. Compiling polymorphism using intensional type analysis. *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p.130-141, January 23-25, 1995, San Francisco, California, United States.

[5] John C. Mitchell , Gordon D. Plotkin, Abstract types have existential type, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, v.10 n.3, p.470-502, July 1988

[6] Joachim Niehren, Jan Schwinghammer, Gert Smolka. A Concurrent Lambda Calculus with Futures. Technical Report. Programming Systems Lab, Saarland University. 2004.

[7] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, Feb, 2002.

[8] Andreas Rossberg. Generativity and dynamic opacity for abstract types. *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 241-252, 2003.

[9] Andreas Rossberg. What are Components. Programming Systems Lab, Saarland University. Slides, March 2004.

[10] Andreas Rossberg, Didier Le Botlan. Personal communication. Programming Systems Lab, Saarland University. March 2004.

[11] Andreas Rossberg. Personal communication. Programming Systems Lab, Saarland University. September 2004.

[12] Gert Smolka. Personal communication. Programming Systems Lab, Saarland University. March 2004.

[13] Eijiro Sumii and Benjamin C. Pierce, A bisimulation for dynamic sealing. *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of programming languages.* ACM Press, January 2004.