Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science
Bachelor's Program in Computer Science

Bachelor's Thesis

Effectful Computation in Moggi's Calculus with Records and Subtyping

submitted by
Dieter Brunotte
on October 30, 2006

Supervisor

Prof. Dr. Gert Smolka

Advisor

Dr. Jan Schwinghammer

Reviewers

Prof. Dr. Gert Smolka
Prof. Dr.-Ing. Holger Hermanns

Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Saarbrücken, October 30, 2006

Abstract

Many theoretical results about programming languages are stated for "effect-free" lambda calculus only, or add effects (such as I/O and state) in an ad-hoc manner. With his computational monads, Moggi presented an abstract framework that fits a wide range of computational effects. Some languages, notably object-oriented ones, feature subtyping in addition to effects.

In my Bachelor thesis I present an extension of Moggi's computational monads with subtyping. As an application of the theory we show that Abadi and Cardelli's imperative object calculus can be embedded into our system, in a way that ensures type safety.

Contents

1 Introduction

In "pure" lambda calculus one just has variables, abstractions and applications. For programming languages with effects like I/O-operations, storage, side-effects, exceptions and so on this is not enough. One way to solve this problem is to extend the language in an ad-hoc manner with the required features. There is much literature about ad hoc extensions for pure lambda calculus, for instance [10].

The drawback of adding some extensions in an ad hoc-manner is that the language changes, e.g. by adding some new constructs (more detailed examples follow in this section) and so basic properties of typed lambda calculi must be proven again. The properties we are interested in are type preservation, progress and a unique (minimal) type property that is important for type checking. Note that in general typing requires an environment Γ giving types to the free variables (not bound by a lambda) of the term.

Type Preservation. This means that for an arbitrary term t for that we can give a certain type A and if we can evaluate t to some other term t' then this term has again type A. So e.g. the evaluation of term with type integer yields again some term that has typ interger.

Progress. This is a property only for terms typable in the empty environment (no variable bound to a type). We call them closed terms. We define the values of the calculus, a subset of terms containing all the terms that cannot be reduced anymore. Progress states that all closed and typable terms are either values or can be reduced.

Unique Types. This means we can give any term t only one type. As we will see subtyping removes that property from typing relation, but we can find a unique type with a "strong" typing that stands in a subtype relation to all other types of the term. That is in calculi with subtyping, the unique type property of simply typed lambda calculus, is replaced by a minimal type property.

The proofs for these properties usually are very similar for extended lambda calculi. So in the proofs of these properties one often uses the trick to say that cases for the construct common with the pure lambda calculus are analogues to that calculus. One of the aims of this thesis is to give a lambda calculus that can be extended easily with effect features like storage and allows to prove important properties like preservation and progress in a modular way using the properties for our calculus.

A problem of effects is that they can change a "global" context such as a output string. If we have a term allowing 2 different reductions and so depending on which reduction we evaluate first can change the following evaluation of terms in 2 different values. Normally, we give an evalution strategie synchronizing the effects in semantic of the languege (example in Section 1.2). With computational Monads Moggi had already considered in [6, 9] a way to synchronize effects. For this purpose he introduced a "let"-construct. We give examples how this construct works in Section 1.3.

Most object-oriented programming languages have a feature Moggi's work doesn't consider: Subtyping. Subtyping is a relation between types; one says a

type A is subtype of type B if one can use terms of type A where you expect to get a type B. A simple example are floating point numbers and integers. Of course, an integer number can always be transformed into a floating point. So if you e.g. want to add two floating points it is always possible to use an integer instead of one or two of the floating points when we transform them to floating points.

More typically for subtyping are record types of object languages when we want to define similar classes of objects. An example is to consider points in the plane. The first class which we call point simply contains the two real number fields x, y giving the coordinates of points. In the second class colored points we additionally want to have color for a point. Now suppose we have a function for two objects of class point, e.g. determining the Euclidian distance. For this function we just need to know the fields x and y. Since the class colored point has all fields of point, there is no technical reason to forbid using a colored point instead of a point.

There is already much literature about subtyping for lambda calculi without monads. Hence, the subtype behaviour of the standard constructs such as functional types is already well-known. One knows that a functional type $A_1 \to A_2$ is subtype of $B_1 \to B_2$ if A_2 is subtype of B_2, but we have the so called contravariance for argument types so that B_1 must be subtype of A_1. More details in Section 2.2 where we consider Subtyping again, or in literature such as [10].

Here, I consider a lambda calculus featuring both, Moggi's theory about computational monads and subtyping. In the remainder of this section I give some basics of lambda calculi and an introduction of Moggi's work with computational monads. In Section 2 I introduce a calculus offering the desired features whose soundness I prove in Section 3 along with some other properties. A main characteristic of the calculus is that we can prove the soundness properties in a emphasised way where the proofs for extended calculi have to be redone only for the new cases. In Section 4 we will see an example realizing an imperative object calculus [2].

1.1 Pure Lambda Calculus

We start by looking at the definition of a pure lambda calculus. We define it in Figure 1 analogous to [10], and start with the simply typed lambda calculus (and not the untyped one). In contrast to Pierce, we use the capital letters A, B, C as notation of an arbitrary type, and reserve the letter T as contructor of the new monadic type. As terms we just have variables, abstractions and application to create terms. Hence, for the evaluation of terms we need just one proper reduction, i.e. $(\lambda x : A.t')t \to t'[x := t]$ where $t'[x := t]$ denotes that we replace all free (not bound by a lambda in term t') x in t' by t. Furthermore we can reduce t_1 or t_2 of an application term $t_1 t_2$, but t_2 only if t_1 is a lambda term and we may not reduce under lambda.

So far we have given the intuition of the untyped calculus. As one can see in reduction rules one can interpret lambda as functions. Hence, one can classify them by what kind of term they are taking as input and what kind of term

$$\begin{array}{rcll} A & \in & Typ & = & A \to A \\ t & \in & Ter & = & x \mid \lambda x : T.t \mid tt \end{array}$$

Figure 1: 'Types and terms of "pure" lambda calculus

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \text{(T-VAR)} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : A \to B}\ \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : A \to B \qquad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1 t_2 : B}\ \text{(T-App)}$$

Figure 2: Typing rule for "pure" simply typed lambda calculus

they are returning, and introduce the typing with the type construct $A \to B$ ("taking A, returning B"). In other calculi one still has other type constructs or constants for further term constructs, but here we just need functional types to introduce typing for all terms. To define typing recursively we also need a typing environment Γ to give the variables some type. We give the typing rules for the simply typed lambda calculus in Figure 2. T-Var just reads the type of x from Γ. By T-Abs, a λ-abstraction $\lambda x : A.t'$ has functional type with type A of variable x as argument type and the result type must be the type of t' where we type function argument x to A. T-App states that for an application term $t_1 t_2$ the subterm t_1 must have a functional type that takes something of the type of t_2.

Furthermore, this calculus fulfills the properties stated in the introduction (can be read in much literature about lambda-calculus like [10]).

1.2 Lambda Calculus with Effects in ad-hoc Manner

After having seen the basic simply typed lambda calculus we will now consider some examples for extensions with effects, added in an ad-hoc way. The first example we present is a lambda calculus featuring the effect of output. Then we continue by an extension with storage.

Example 1. Lambda calculus with output strings

As one can see in Figure 3 we now have an output alphabet Σ containing the letters to create output strings. Whenever we perform the new operation print

$$\begin{array}{rcll} \sigma & \in & \Sigma^* & \\ T & \in & Typ & = & \cdots \mid unit \\ t & \in & Ter & = & \cdots \mid print\ \sigma \mid () \end{array}$$

Figure 3: Lambda calculus with output strings

$$\frac{}{\Gamma \vdash \mathit{print}\ \sigma : \mathit{unit}}\ \text{(T-PRINT)} \qquad \frac{}{\Gamma \vdash () : \mathit{unit}}\ \text{(T-UNIT)}$$

Figure 4: New typing rules for lambda calculus with output string

with a string σ, we add σ at the end of the global output $S \in \Sigma$, which is a result of the evaluation of the term. Since we added new form to the calculus, we need some additional typing and evaluation rules for the new term constructs. It also is required a new standard type unit to have a type for output operations.

Additionally, we have to be able to observe the output string everywhere. This we can do by adding an output string to the terms.

$$\begin{array}{lcll} S \mid t & \rightarrow & S \mid t' & \text{if } t \rightarrow t' \text{ in pure calculus} \\ S \mid \mathit{print}\ \sigma & \rightarrow & S.\sigma \mid () & \end{array}$$

Another problem is that in non-deterministic evaluation of lambda-calculus. We can produce different outputs from the same term and there is no construction to synchronise the output. The following term makes the problem clear; we use the sequencing ";"-operater of SML where $t_1; t_2$ means we first evaluate t_1, "throw away" the result and finally evaluate t_2. As usual this can be simulated by term $(\lambda x : T_1.t_2)t_1$ where T_1 is the type of t_1 and x is a fresh variable.

$$\begin{array}{lcl} & & \epsilon \mid (\lambda x : \mathit{unit}.x; x)(\mathit{print}\ \sigma) \\ \ldots & \rightarrow & \sigma \mid (\lambda x : \mathit{unit}.x; x)() \\ & \rightarrow & \sigma \mid (); () \\ & \rightarrow & \sigma \mid () \end{array}$$

$$\begin{array}{lcl} \ldots & \rightarrow & \epsilon \mid \mathit{print}\ \sigma; \mathit{print}\ \sigma \\ & \rightarrow & \sigma \mid (); \mathit{print}\ \sigma \\ & \rightarrow & \sigma \mid \mathit{print}\ \sigma \\ & \rightarrow & \sigma.\sigma \mid () \end{array}$$

As we can see in the second case we print σ two times, in the other case the output string is σ. In the next section we will see a solution with monads. In ad-hoc manner one usually solves this by giving an evaluation strategy such as deterministic call-by-name (cbn) and call-by-value (cbv). In cbv-evaluation one evaluates first the function arguments and then performs the application, as done in the first example with output just σ. In contrast, in cbn the substitution of arguments is done immediately, like in the second example leading to output $\sigma.\sigma$.

Example 2. Side-effects with storage

Here we need a new type constructor $ref\ T$ for memory cells of type T together with some new operations and a new *unit* standard type.

$$\begin{array}{rcl} read_A & : & ref\,A \to A \\ write_A & : & ref\,A \to A \to unit \end{array}$$

Similar to the global output string before we have to add an store S for the evaluation.

$$\begin{array}{lcll} S \mid t & \to & S \mid t' & \text{if } t \to t' \text{ in pure calculus} \\ S \mid read_T\ l & \to & S \mid Sl & \\ S \mid read_T\ l\ t & \to & S[l := t] \mid () & \end{array}$$

Similar problems to output arise in the case of non-deterministic evaluation.

1.3 From Basic Lambda Calculus to Monadic Types

So last subsection we have seen how one can add effects in ad-hoc manner. Now we see Moggi's solution with monads for synchronising the effects. Here we have the additional construct $let\ x \Leftarrow t_1\ in\ t_2$ in the language. The intuition of this construct is that all effects of t_1 happen before the effects of t_2 where x is substitited by the result of t_1. Furthermore to distinguish between computations with and without effects, Moggi introduced a new type constructor T, the so-called monadic type constructor. The term construct $[t]$ creates a monadic value out of an arbitrary term t. So all terms with effectful reduction have the requirement of monadic type and hence the let-construct and the subterms t_1, t_2 must have a monadic type. The monadic term construct $[t]$ also stops effectful evalution of t. The only possibility to get the t under [] is with a new reduction rule for let: if we have done the computations with effects of t_1 one normally gets a term of the form $[t]$; in the term t_2 we then do not substitute the x by $[t]$, but we substitute by t. Here is a formal notation of this rule.

$$let\ x \Leftarrow [t_1]\ in\ t_2 \xrightarrow{\epsilon} t_2[x := t_1]$$

Now we can consider again the example from the last subsection with output strings. Depending on the evaluating strategie (cbv, cbn) we can produce different output strings with the same term. In Table 1 we give translation function for both interpretation into the calculus with the let-construct. As one can see these functions also must be defined for types, but in the example we just need to translate *unit* with itself. We also give a rule for the operator ";" instead of using the simulating term since cbn interpretation differ from the SML interpretation. Note that the result type of *print* changes to $T\ unit$ because evaluation causes an effect. Further examples for this kind of translations and one for types can be found in [6]. About other practical use of monads there is much literature, e.g. by Philip Wadler([11], [13] or [12]).

Now we give the translation for the example term with two different output

t	$cbv(t)$	$cbn(t)$
x	$[x]$	x
$\lambda x : A.t'$	$[\lambda x : cbv(A).cbv(t')]$	$[\lambda x : T\ cbn(A).cbn(t')]$
$t_1 t_2$	$let\ f \Leftarrow cbv(t_1)\ in\ let\ y \Leftarrow cbv(t_2)\ in\ f\ y$	$let\ f \Leftarrow cbn(t_1)\ in\ f\ cbn(t_2)$
$t_1; t_2$	$let\ z \Leftarrow cbv(t_1)\ in\ cbv(t_2)$	$let\ z \Leftarrow cbn(t_1)\ in\ cbn(t_2)$
$print\ \sigma$	$print\ \sigma$	$print\ \sigma$
$()$	$[()]$	$[()]$

Table 1: Cbv and Cbn translation

strings and show that the translations produce the same output strings.

$t = (\lambda x : unit.x; x)(print\ \sigma)$
$cbv(t) = let\ f \Leftarrow [\lambda x : unit.let\ z \Leftarrow [x]\ in\ [x]]\ in\ let\ y \Leftarrow (print\ \sigma)\ in\ f\ y$
$cbn(t) = let\ f \Leftarrow [\lambda x : Tunit.let\ z \Leftarrow x\ in\ x]\ in\ f\ (print\ \sigma)$

$\epsilon \mid cbv(t)$

$$
\begin{aligned}
\ldots &\rightarrow \epsilon \mid let\ y \Leftarrow (print\ \sigma)\ in\ (\lambda x : unit.let\ z \Leftarrow [x]\ in\ [x])\ y \\
&\rightarrow \sigma \mid let\ y \Leftarrow [()]\ in\ (\lambda x : unit.let\ z \Leftarrow [x]\ in\ [x])\ y \\
&\rightarrow \sigma \mid (\lambda x : unit.let\ z \Leftarrow [x]\ in\ [x])() \\
&\rightarrow \sigma \mid let\ z \Leftarrow [()]\ in\ [()] \\
&\rightarrow \sigma \mid [()]
\end{aligned}
$$

$\epsilon \mid cbn(t)$

$$
\begin{aligned}
\ldots &\rightarrow \epsilon \mid (\lambda x : Tunit.let\ z \Leftarrow x\ in\ x)\ (print\ \sigma) \\
&\rightarrow \epsilon \mid let\ z \Leftarrow print\ \sigma\ in\ print\ \sigma \\
&\rightarrow \sigma \mid let\ z \Leftarrow [()]\ in\ print\ \sigma \\
&\rightarrow \sigma \mid print\ \sigma \\
&\rightarrow \sigma.\sigma \mid [()]
\end{aligned}
$$

2 The Calculus

In the last section we have seen the problems of adding extensions with effects to lambda calculus. We have seen that Moggi found a way to address these with computational monads, but did not consider the problem of subtyping. So this chapter we will now introduce a lambda calculus solving the problems.

2.1 Definition of the Calculus

First we give a definition of terms and types of the extended Moggi calculus in Figure 5. As in every lambda calculus terms can have variables, abstractions and applications. To synchronise effects we have let-expression as used by Moggi and the monadic constructor []. We see how synchronisation works when we consider the reduction rules of the calculus (Section 2.4). Since we want to have subtyping in the language it makes sense to introduce records because there is not a problem to use a record which has more fields if you just need a record with fewer fields. Production rules for subtyping follow in Section 2.2. To realize the extension of the language in a principal manner, we decided to have some constants in our language that can be defined on demand. So if you need some new constructs for any extension, you define $Const$ with some emphasized constants. If e.g. you want to be able to do some arithmetic operations on numbers you can add the numbers $0, 1, 2, \ldots$ with type int and operations like $plus$ with type $int \to int \to int$ as new constants. As new proper reduction rule you can define $plus\ i\ j \to i + j$. Since we don't have the type int we can add this type by defining $BasicTypes$, the set of basic types, so that it contains a type int. So adding new types isn't a problem in our language.

Knowing the terms the most type constructs are defined intuitively. There is an arrow type ($\to$) for abstractions as in pure lambda calculus. For monads we have the type constructor T, for records a record type, for subtyping a type Top which is a supertype (inverse direction of subtype; if A is subtype of B, then B is a supertype of A) of every other type, and the possibility to add some other base types or type constructors. We need the type constructor to be able to create types like referenceces (necassary for our example in Section 4) or Lists. Note that it is possible to define all type constructs in TC . Top and basic types would have arity 0, the monadic type constructor T 1, and the arrow with arity 2. Also record types could be simulated for every fixed set of labels $\{l_i \mid i \in [n]\}$ with arity n. (Notation: $[n]$ stands for the set $\{1, 2 \ldots, n\}$ of first n numbers.)

We finish this subsection with a definition of the free variables of the calculus. We will need to know them for some proper reduction rules and some auxiliary lemmas in the next section. The definition is very intuitive, noting that λ and monadic let are the only binding constructs.

$$
\begin{array}{rcll}
b & \in & BasicTypes & \\
F & \in & TC & \textit{constant set of type constuctors with an arity function } a : TC \to \mathbf{N} \\
A, B & \in & Ty & = \; Top \mid b \mid A \to B \mid TA \mid \{l_i : A_i \;^{i=1\ldots n}\} \\
 & & & \mid F\ A_1\ \ldots\ A_{a(F)} \\
c : A & \in & Const & \\
t & \in & Ter & = \; x \mid c \mid \lambda x : A.t \mid tt \mid [t] \mid let\, x \Leftarrow t\, in\, t \mid \\
 & & & t.l \mid \{l_i = t_i \;^{i=1\ldots n}\}
\end{array}
$$

Figure 5: Moggi's Calculus with records

$$
\begin{array}{lcl}
FV(x) & = & \{x\} \\
FV(c) & = & \{\} \\
FV(\lambda x : A.t) & = & FV(t) - \{x\} \\
FV(t_1\, t_2) & = & FV(t_1) \cup FV(t_2) \\
FV([t]) & = & FV(t) \\
FV(let\, x \Leftarrow t_1\, in\, t_2) & = & FV(t_1) \cup (FV(t_2) - \{x\}) \\
FV(t.l) & = & FV(t) \\
FV(\{l_i = t_i \;^{i=1\ldots n}\}) & = & \bigcup_{i=1}^{n} FV(t_i)
\end{array}
$$

2.2 Subtype Relation

Before we consider the typing of the terms we introduce the subtype relation. The rules for subtyping are defined in Figure 6 and are analogous to [10]. We start with the general subtype rules and the rule for functional types. The first rule (S-Top) gives us that Top is a supertype of every type. The other generell rules tell that every type has itself as subtype (S-Refl) and the subtype relation is transitive (S-Trans). Probably the most interesting rule is the rule for functional types (S-Arrow) with the so called contravariance of argument type. That means if $A_1 \to A_2 <: B_1 \to B_2$ then we must have $B_1 <: A_1$ and not as one could expect $A_1 <: B_1$. This has the following reason: suppose you have the term fx where f is a function with argument type B_1 and x has the same type; if we now have a function f' with type A_1 which has subtype of f one should be able to write $f'x$; but to be able to do this f' must be able to handle argument x of type B_1 which is only possible iff $B_1 <: A_1$.

The next subtype rules are the ones for records analogous to [10]. The last rule (T-Mon) is the one for the monadic type saying that if a types is a subtype of another type then their monadic types have the same relation. In other words, the type constructor T is covariant. Establishing this subtype relation for monads is one of the aims of this thesis.

We don't define any subtype rules for type constructors. However, we don't allow it here. An investigation of this is left to future work.

Standard typing rules for subtyping:

$$\frac{}{A <: Top}\ \text{(S-Top)} \qquad \frac{A <: B \qquad B <: C}{A <: C}\ \text{(S-Trans)}$$

$$\frac{}{A <: A}\ \text{(S-Refl)} \qquad \frac{B_1 <: A_1 \qquad A_2 <: B_2}{A_1 \to A_2 <: B_1 \to B_2}\ \text{(S-Arrow)}$$

Additional subtyping rules for records :

$$\frac{\forall i \in [n] : A_i <: B_i}{\{l_i : A_i\ ^{i=1,\ldots,n}\} <: \{l_i : B_i\ ^{i=1,\ldots,n}\}}\ \text{(S-RcdDepth)}$$

$$\frac{}{\{l_i : A_i\ ^{i=1,\ldots,n+k}\} <: \{l_i : A_i\ ^{i=1,\ldots,n}\}}\ \text{(S-RcdWidth)}$$

$$\frac{\{l_i : A_i\ ^{i=1,\ldots,n}\} \quad \text{is a permutation of} \quad \{k_i : B_i\ ^{i=1,\ldots,n}\}}{\{l_i : A_i\ ^{i=1,\ldots,n}\} <: \{k_i : B_i\ ^{i=1,\ldots,n}\}}\ \text{(S-RcdPerm)}$$

Subtyping rule for monads :

$$\frac{A <: B}{TA <: TB}\ \text{(S-Mon)}$$

Figure 6: Subtype rules for the calculus

$$\frac{}{\Gamma, x : A \vdash x : A}\ (\text{T-Var}) \qquad \frac{c : A \in Const}{\Gamma \vdash c : A}\ (\text{T-Const}) \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : A \to B}\ (\text{T-Abs})$$

$$\frac{\Gamma \vdash t_1 : A \to B \qquad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1 t_2 : B}\ (\text{T-App}) \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash [t] : TA}\ (\text{T-Mon})$$

$$\frac{\Gamma \vdash t : TA \qquad \Gamma, x : A \vdash t' : TB}{\Gamma \vdash let\, x \Leftarrow t\, in\, t' : TB}\ (\text{T-Let})$$

$$\frac{\forall i \in [n] : \Gamma \vdash t_i : A_i}{\Gamma \vdash \{l_i = t_i\ ^{i=1\ldots n}\} : \{l_i : A_i\ ^{i=1\ldots n}\}}\ (\text{T-Rcd})$$

$$\frac{\Gamma \vdash t : \{l_i : A_i\ ^{i=1\ldots n}\} \qquad j \in [n]}{\Gamma \vdash t.l_j : A_j}\ (\text{T-Proj})$$

Additional typing rule to use the subtyping:

$$\frac{\Gamma \vdash t : A \qquad A <: B}{\Gamma \vdash t : B}\ (\text{T-Sub})$$

Figure 7: Typing rules for the calculus

2.3 Weak Typing Relation

We see the definitions of typing rules in Figure 7. Most are well-known and analogues of the known lamba type rules presented by Pierce in [10]. A new type rule is T-Const, where the type of a defined constant is just read from the set *Const*. So types of constants must be given to type any term with constants.

The rule T-Let for let-expressions requires monadic types for both subterms. So we have to stay in monadic type.

For subtyping we have the rule T-Sub, which allows to give a term typable with type A a supertype B. So the "weak" type of a term is not unique, e.g. with the S-Top rule we can give every typable term the type *Top*. But in Section 3.3 we will see that there also is a unique "strong" type of a term t which is subtype of all possible types of t.

All other type rules are intuitive and known from other lambda calculi, e.g. T-Mon where we can create a term of monadic type of A by putting a term t of type A in $[t]$. Viewing $T\ A$ as the type of computations yielding values in A, $[t]$ is a trivial computation that immediately returns t.

Effect notation:

$$\begin{array}{l} \alpha \in Eff \quad ::= \quad \emptyset \mid \epsilon \\ \epsilon \in pEff \end{array}$$

Proper Reduction rules:

$$\begin{array}{llcll} (\beta) & (\lambda x : A.t')t & \rightarrow & t'[x := t] & \\ (\eta) & \lambda x : A.tx & \rightarrow & t & x \notin FV(t) \\ (\beta.rec) & \{l_i = t_i \ ^{i=1\ldots n}\}.l_j & \rightarrow & t_j & j \in [n] \\ (\eta.rec) & \{l_i = t.l_i \ ^{i=1,\ldots,n}\} & \rightarrow & t & \\ (\beta.let) & let\ x \Leftarrow [t]\ in\ t' & \stackrel{\emptyset}{\rightarrow} & t'[x := t] & \\ (\eta.let) & let\ x \Leftarrow t\ in\ [x] & \stackrel{\emptyset}{\rightarrow} & t & \\ (assoc) & let\ y \Leftarrow (let\ x \Leftarrow t_1\ in\ t_2) in\ t_3 & \stackrel{\emptyset}{\rightarrow} & let\ x \Leftarrow t_1\ in (let\ y \Leftarrow t_2\ in\ t_3) & x \notin FV(t_3) \end{array}$$

Figure 8: Proper Reduction rules and effects

2.4 Reduction Relation

We equip the calculus with a small-step operational semantics, defined using elementary "proper" reduction rules and "evaluation contexts" in the style of Felleisen and Wright [14]. In Figure 8 we define the proper reduction rules of the calculus. The β-rule was already introduced in Section 1.1. The intuition of η-rule is that we can reproduce any function f by a lambda-abstraction that just applies its argument variable to f. Similar rules we have for records. $\beta.rec$ just states how to get the content of a record field. The idea of $\eta.rec$ is if every field of record takes out the content of the same term t with its own label then t must behave as the original record. Then we have rules with the empty effect for the let-construct. $\beta.let$ was introduced in Section 1.3. We expect a term t of monadic type to finally evaluate to $[t']$. Hence, if t is subterm in a let bound to a variable x then the next step would to take t' out of [] and replace all x by t'. So the lefthand side of $\eta.let$ evalutes to t. As last rule we state $assoc$ which is a standard equivalence for many kinds of let-constructs.

Notice that this list of rules is not complete since our language offers the feature to add new reduction rules to the calculus such that we can use e.g. the new constants in $Const$ (as in the example with integers and the plus operator in Section 2.1). Since we may have different effects in our language we have to define what kinds of effects appear (by defing *pEff* and *Eff*) and when (on which reduction) they are generated. So we have the empty effect $\emptyset$ and we have some proper effects which can be defined on demand, analogous to the sets of constants and basic types. As we can see in Figure 8 not every reduction has an effect observed on its arrow, so we distinguish between effect-free reduction and reductions with effect. We call reductions $t \rightarrow t'$ effect-free reductions $t \stackrel{\epsilon}{\rightarrow} t'$ effectful observing effect ϵ. Since we want to use effects to synchronize we do not allow effectful reductions everywhere, i.e. we only may reduce on top-level (term has left-hand side of a proper reduction rule with effect) or in the first

term of a let-expression. Effect-free reductions are allowed everywhere. We explain more precise when we introduce the reduction contexts below.

An intuitive wish of new constants $c \in Const$ with functional type is that we can reduce well-typed terms of the form $c\ t_1\ \ldots\ t_n$ whenever they have a non-functional type. Therefore we require that new proper reduction rules are added to the language of the form $c\ t_1\ t_2\ \ldots\ t_n \to t'$ for constants defined in $Const$ with functional type, so that we can always reduce terms of the form $c\ t_1\ t_2\ \ldots\ t_n$ without functional type.

If these reductions need the possibility of synchronisation we add a proper effect on its reduction arrow. But since we want to use our new monadic type for synchronization, we want to allow effectful reduction only if we have a monadic type. So if we define an additional reduction rule with effect for any constant c in $Const$ this constant must have a type of the form $A_1 \to \cdots \to A_n \to T\ A$, and the left-hand side of the new proper reduction rule has excactly n arguements. As we will see below in an example n may be 0.

Fortunately, we prove the main properties of the calculus in Section 3 without the need that we can alway reduce terms as discribed in the last paragraphs. But we need some other properties we state when we do these proofs next section.

The empty effect $\emptyset$ is used for the standard reduction of let-expressions as we see them in Figure 8. It is also allowed to define new reduction with let-expression when e.g. a new monadic construct requires that. Here we also define a effectful reduction rule for new constants $c : T\ A \in Const$, so that we always have a reduction for well-typed terms of the form $let\ x \Leftarrow c\ in\ t'$ Mostly, it is enough to define just one rule with effects, since it is allowed to reduce the c in the let-term effectful. A simple example is that we have print constants for arbitrary output strings s, e.g. $print_s : T\{\} \in Const$. Here we can define the reductions $print_s \xrightarrow{s} [\{\}]$.

So far, we don't have any new rules for constants with record type. Intuitively, we have to give reduction rules for all labels of the record type. But knowing all this rules it is easy to find an equivalent record using no constant of record type.

We realize synchronisation by using different reduction contexts for effect-free reduction and reductions with effects. Now we define contexts for effect-free reductions and for reduction with effects. Furthermore, when we talk about contexts we use C for a context for effectfree reduction and E for the ones with effect. Note that every context E is also a context for effect-free reductions.

$$\begin{array}{rcl} C \in Context & = & \circ \mid Ct \mid tC \mid \lambda x : A.C \mid \{l_i = t_i\ ^{i=1,\ldots,j-1}, l_j = C, l_i = t_i\ ^{i=j+1,\ldots,n}\} \mid \\ & & let\ x \Leftarrow C\ in\ t \mid let\ x \Leftarrow t\ in\ C \mid C.l \mid [C] \\ E \in EContext & = & \circ \mid let\ x \Leftarrow E\ in\ t \end{array}$$

We have the following context reduction rules to produce further reductions:

$$\frac{t \to t'}{C[t] \xrightarrow{\emptyset} C[t']} \qquad \frac{t \xrightarrow{s} t'}{E[t] \xrightarrow{s} E[t']}$$

Alternative:

$$\frac{t \to t'}{C[t] \to C[t']} \qquad \frac{t \to t'}{t \xrightarrow{\emptyset} t'} \qquad \frac{t \xrightarrow{\alpha} t'}{E[t] \xrightarrow{\alpha} E[t']}$$

Thus one can perform a reduction with effect only on toplevel or under the left side of a let expression. That means in a term $let\ x \Leftarrow t_1\ in\ t_2$ all effects of t_1 are observed before the effects of t_2, because we are not allowed to perform effectful reductions of t_2. In contrast, in a term like $(\lambda x.t_2)t_1$ in a simply typed lambda calculus with effects as in Section 1.2 the order in which effects had appeared had not been predictable.

Contexts C allow effect-free reduction everywhere. So we can do effect-free reductions that do not need to be synchronised where we want (even below λs), and if we need a synchronisation we use let expressions and effectful reductions to do so.

Furthermore, the context rules allow to label every effect-free reduction with the empty effect $\emptyset$.

3 Soundness Proofs

Last section we have defined a new calculus. Now we want to see that this calculus is safe, i.e. it has certain properties we will prove this section. The properties I'm interested in are Type Preservation and Progress. We will also show that every typable term has an strong type which is a subtype of every weak type.

3.1 Proof of Type Preservation

We start with the Type Preservation theorem. Like Pierce in [10] we need some auxiliary lemmas and theorems for this proof. The first two lemmas we proove are extented versions of Lemmas 15.3.2 and 15.3.3 in [10]. So they hold in the extended Moggi calculus, but we need some additional properties because we have more constructs in our language. After that we continue with preservation under substitution which we need to prove preservation of β-rules, and finally we show the preservation theorem.

3.1.1 Inversion

Lemma 3.1 (Inversion of the Subtype Relation)**.**

1. *If $B <: A_1 \to A_2$, then B has the form $B_1 \to B_2$, with $A_1 <: B_1$ and $B_2 <: A_2$.*
2. *If $B <: \{l_i : A_i{}^{i=1,\ldots,n}\}$, then B has the form $\{k_j : B_j{}^{j=1\ldots m}\}$, with at least the labels $\{l_i{}^{i=1\ldots n}\}$ i.e., $\{l_i{}^{i=1\ldots n}\} \subseteq \{k_j{}^{j=1\ldots m}\}$ and with $B_j <: A_i$ for each common label $l_i = k_j$.*
3. *If $B <: TA$ then B has the form TB', and $B' <: A$.*

Proof. This proof is by induction on the derivation of the subtype relation. It is not necessary to consider all cases in the lemma, e.g. we will not consider S-Top. This is because every part of the lemma requires a special form on the right-hand side of $<:$. Similarly we need not consider S-Refl because it already implies the required properties.

1. $B <: A_1 \to A_2$

   - S-TRANS :
     We have $B <: C$ and $C <: A_1 \to A_2$.
     By induction, C has form $C_1 \to C_2$ and $A_1 <: C_1$, $C_2 <: A_2$.
     Hence by induction, also follows B has form $B_1 \to B_2$, and $C_1 <: B_1$ and $B_2 <: C_2$.
     By S-Trans rule, $A_1 <: B_1$ and $B_2 <: A_2$.

   For S-Arrow the Lemma holds immediately. All other rules do not have the right form on the right-hand side.

2. $B <: \{l_i : A_i\ ^{i=1,\ldots,n}\}$

   - S-TRANS:
     We have $B <: C$ and $C <: \{l_i : A_i\ ^{i=1,\ldots,n}\}$.
     By induction, C has form $\{m_k : C_k\ ^{k=1\ldots l}\}$ and $\{l_i\ ^{i=1\ldots n}\} \subseteq \{m_k\ ^{k=1\ldots l}\}$ and $C_k <: A_i$ for each common label $l_i = m_k$.
     Again by induction, B has the form $\{k_j : B_j\ ^{j=1\ldots m}\}$ and $\{m_k\ ^{k=1\ldots l}\} \subseteq \{k_j\ ^{j=1\ldots m}\}$ and $B_j <: C_k$ for each common label $m_k = k_j$.
     Hence, $\{l_i\ ^{i=1\ldots n}\} \subseteq \{k_j\ ^{j=1\ldots m}\}$ and by S-Trans $B_j <: A_i$ for each common label $l_i = m_k = k_j$.
   - S-RCDDEPTH/S-RCDPERM :
     The required properties follow directly from the rule.
   - S-RCDWIDTH :
     The form and label subset property follow directly. Because each label $k_i = l_i$ has the same types the property $B_i <: A_i$ follows with S-Refl.

   Other rules are not applicable.

3. $B <: TA$

   - S-TRANS :
     We have $B <: C$ and $C <: TA$.
     By induction, C has form TC' and $C' <: A$.
     Hence again by induction, B has form TB' and $B' <: C'$.
     By S-Trans, $B' <: A$.
   - S-MON :
     The required properties follow directly.

   For the other rules there is nothing to show.

□

Lemma 3.2 (Inversion)**.**

1. *If* $\Gamma \vdash \lambda x : B_1.t' : A_1 \to A_2$*, then* $A_1 <: B_1$ *and* $\Gamma, x : B_1 \vdash t' : A_2$*.*
2. *If* $\Gamma \vdash \{k_a = t_a\ ^{a=1\ldots m}\} : \{l_i : A_i\ ^{i=1\ldots n}\}$*, then* $\{l_i\ ^{i=1\ldots n}\} \subseteq \{k_a\ ^{a=1\ldots m}\}$ *and* $\Gamma \vdash t_a : A_i$ *for each common label* $k_a = l_i$*.*
3. *If* $\Gamma \vdash t'.l_j : A$ *then* $\Gamma \vdash t' : \{k_i : A_i\ ^{i=1,\ldots n}\}$ *and* $\{k_i\ ^{i=1,\ldots n}\}$ *contains the label* l_j *and* $A_i <: A$ *for* $k_i = l_j$*.*
4. *If* $\Gamma \vdash t_1 t_2 : A$ *then* $\exists A' : \Gamma \vdash t_1 : A' \to A$*,* $\Gamma \vdash t_2 : A'$*.*
5. *If* $\Gamma \vdash x : B$*, then for some* A *we have* $x : A \in \Gamma$ *and* $A <: B$*.*

6. *If* $\Gamma \vdash [t'] : TA$ *, then* $\Gamma \vdash t' : A$.

7. *If* $\Gamma \vdash let\ x \Leftarrow t'\ in\ t : TA$ *, then* $\Gamma \vdash t' : TA'$ *and* $\Gamma, x : A' \vdash t : TA$ *for some* A'.

Proof. We prove this by a straightforward induction on type derivations. For every part of this lemma we only need to consider T-SUB and the type rule for the corresponding term construct; the others are not applicable because of the form.

1. • case T-ABS :
   The hypothesis of the rule are
   $\Gamma, x : B_1 \vdash t' : A_2 \qquad B_1 = A_1 \qquad B_2 = A_2$
   Hence, $A_1 <: B_1$ holds by S-REFL.
   • case T-SUB :
   $\Gamma \vdash \lambda x : B_1.t' : C \qquad C <: A_1 \to A_2$
   By Inversion of subtype relation, C has the form $C_1 \to C_2$ and $A_1 <: C_1$ and $C_2 <: A_2$.
   By induction, $C_1 <: B_1$ and $\Gamma, x : B_1 \vdash t' : C_2$.
   By S-TRANS, $A_1 <: B_1$ and by T-SUB $\Gamma, x : B_1 \vdash t' : A_2$.

2. • case T-RCD :
   $\forall a \in [n] : \Gamma \vdash t_a : A_a$ and $k_a = l_a$ and hence, $\{l_i\ ^{i=1\ldots n}\} = \{k_a\ ^{a=1\ldots m}\}$.
   • case T-SUB :
   $\Gamma \vdash \{k_a = t_a\ ^{a=1\ldots m}\} : C \qquad C <: \{l_i : A_i\ ^{i=1\ldots n}\}$
   By inversion of subtype relation, C has the form $\{m_j : C_j\ ^{j=1\ldots l}\}$ and $\{l_i\ ^{i=1\ldots n}\} \subseteq \{m_j\ ^{j=1\ldots l}\}$ and $C_j <: A_i$, whenever $m_j = l_i$.
   By induction, $\{m_j\ ^{j=1\ldots l}\} \subseteq \{k_a\ ^{a=1\ldots m}\}$ and $\Gamma \vdash t_a : C_j$ whenever $k_a = m_j$.
   Hence by T-SUB, $\Gamma \vdash t_a : A_i$, whenever $k_a = m_j = l_i$.

3. • case T-PROJ :
   The required properties are given by the type rule.
   • case T-SUB :
   $\Gamma \vdash t'.l_j : A' \qquad A' <: A$
   By induction, $\Gamma \vdash t' : \{k_i : A_i\ ^{i=1,\ldots n}\}$ and $\{k_i\ ^{i=1,\ldots n}\}$ contains the label l_j and $A_i <: A'$ for $k_i = l_j$. And hence by S-TRANS, $A_i <: A$.

4. • case T-APP :
   The required properties are given by the typing rule.
   • case T-SUB :
   $\Gamma \vdash t_1 t_2 : A'' \qquad A'' <: A$
   By induction, there is A' such that $\Gamma \vdash t_1 : A' \to A''$ and $\Gamma \vdash t_2 : A'$.
   By subtype rules, $A' \to A'' <: A' \to A$. Hence with T-SUB, $\Gamma \vdash t_1 : A' \to A$.

5. • case T-VAR
   By typing rule $x : B \in \Gamma$. Here, $A = B$, hence, $A <: B$ by S-Refl.
   • case T-SUB
   $\Gamma \vdash x : A' \qquad A' <: B$
   By induction, for some A we have $x : A \in \Gamma$ and $A <: A'$, so by S-Trans $A <: B$.

6. • case T-MON :
   $\Gamma \vdash t' : A$ given by type rule.
   • case T-SUB :
   $\Gamma \vdash [t'] : B \qquad B <: TA$
   By inversion of subtype relation, B has the form TB' and $B' <: A$.
   By induction, $\Gamma \vdash t' : B'$. With T-Sub, $\Gamma \vdash t' : A$.

7. • case T-LET :
   Required properties follow directly from type rule.
   • case T-SUB :
   $\Gamma \vdash let\ x \Leftarrow t'\ in\ t : B \qquad B <: TA$
   By inversion of subtype relation, B has the form TB' and $B' <: A$.
   By induction, $\Gamma \vdash t' : TA'$ and $\Gamma, x : A' \vdash t : TB'$ for some A'. By T-Sub, $\Gamma, x : A' \vdash t : TA$.

□

3.1.2 Preservation of Types under Substitution

Before we prove the preservation under substitution we state a short lemma which is easy to proof that we need in the following proof. We don't proof it, but we should know it when it is used in proof.

Lemma 3.3 (Strengthening)**.** *If* $\Gamma, x : B \vdash t : A$ *and* $x \notin FV(t)$*, then* $\Gamma \vdash t : A$.

Proof. The proof is by straight-forward induction on type derivation.

□

Theorem 3.4 (Preservation of types under substitution)**.** *If* $\Gamma, x : B \vdash t : A$ *and* $\Gamma \vdash t' : B$ *then* $\Gamma \vdash t[x := t'] : A$

Proof. We prove the theorem by structural induction on type derivations. The proof is a big case analysis.
Base cases:(T-VAR/T-CONST)

- case $t = x$:
  We have $t[x := t'] = t'$. Hence, $\Gamma \vdash t[x := t'] : B$. By inversion of typing rules, we get $B <: A$. So we have again $\Gamma \vdash t[x := t'] : A$.

- case $t = y \neq x (t = c)$:

  Here $t[x := t'] = t = y (= c)$. Because $x \notin FV(t) = \{y\} (= \{\})$, by strengthening we don't need the x in the type environment Γ to type t. Hence, $\Gamma \vdash y : A$ ($\Gamma \vdash c : A$).

Induction step:

- case T-ABS :

  $t = \lambda x' : A'.t''$ (where $x' \notin FV(t'), x' \neq x$)

  $t[x := t'] = \lambda x' : A'.t''[x := t']$

  $A = A' \to A''$ $\quad$ $\Gamma, x : B, x' : A' \vdash t'' : A''$ $\quad$ for some A''

  By induction hypothesis, $\Gamma, x' : A' \vdash t''[x := t'] : A''$.

  By typing we get $\Gamma \vdash \lambda x' : A'.t''[x := t'] : A$.

  In case $x' = x$ we have $t[x := t'] = t$. Because x does not occur free in t we don't need it to type t. Hence, $\Gamma \vdash t : A$.

- case T-APP :

  $t = t_1 t_2$

  $t[x := t'] = t_1[x := t']t_2[x := t']$

  $\Gamma, x : B \vdash t_1 : A' \to A$ $\quad$ $\Gamma, x : B \vdash t_2 : A'$ $\quad$ for some A'

  By induction, $\Gamma \vdash t_1[x := t'] : A' \to A$ and $\Gamma \vdash t_2[x := t'] : A'$.

  By typing, $\Gamma \vdash t_1[x := t']t_2[x := t'] : A$.

- case T-MON :

  $t = [t'']$

  $t[x := t'] = [t''[x := t']]$

  $\Gamma, x' : B \vdash t'' : A'$ $\quad$ $A = TA'$

  By induction, $\Gamma \vdash t''[x := t'] : A'$.

  By typing, $\Gamma \vdash [t''[x := t']] : TA'$.

- case T-RCD :

  $t = \{l_i = t_i \, ^{i=1...n}\}$

  $t[x := t'] = \{l_i = t_i[x := t'] \, ^{i=1...n}\}$

  $\forall i \in [n] : \Gamma, x : B \vdash t_i : A_i$ $\quad$ $A = \{l_i : A_i \, ^{i=1...n}\}$

  By induction, $\forall i \in [n] : \Gamma \vdash t_i[x := t'] : A_i$.

  By typing, $\Gamma \vdash \{l_i = t_i[x := t'] \, ^{i=1...n}\} : \{l_i : A_i \, ^{i=1...n}\}$.

- case T-PROJ :

  $t = t''.l_j$

  $t[x := t'] = t''[x := t'].l_j$

  $\Gamma, x : B \vdash t'' : \{l_i : A_i \ ^{i=1...n}\} \qquad A = A_j \qquad j \in [n]$

  By induction, $\Gamma \vdash t''[x := t'] : \{l_i : A_i \ ^{i=1...n}\}$.

  By typing, $\Gamma \vdash t''[x := t'].l_j : A$.

- case T-LET (1):

  $t = let\ x' \Leftarrow t_1\ in\ t_2$ (where $x \neq x'$)

  $t[x := t'] = let\ x' \Leftarrow t_1[x := t']\ in\ t_2[x := t']$

  $\Gamma, x : B \vdash t_1 : TA' \qquad \Gamma, x : B, x' : A' \vdash t_2 : TA'' \qquad A = TA''$ for some A''

  By induction, $\Gamma \vdash t_1[x := t'] : TA'$ and $\Gamma, x' : A' \vdash t_2[x := t'] : TA''$.

  By typing, $\Gamma \vdash t[x := t'] : TA''$.

- case T-LET (2):

  $t = let\ x \Leftarrow t_1\ in\ t_2$

  $t[x := t'] = let\ x \Leftarrow t_1[x := t']\ in\ t_2$

  $\Gamma, x : B \vdash t_1 : TA'' \qquad \Gamma, x : B, x : A'' \vdash t_2 : TA' \qquad A = TA''$

  By induction, $\Gamma \vdash t_1[x := t'] : TA''$. Because in the second writing the type B of x is overwritten by type A'' we don't need $x : B$ to type t_2. So $\Gamma, x : A'' \vdash t_2 : TA'$.

  By typing, $\Gamma \vdash t[x := t'] : A$.

- case T-SUB :

  $\Gamma, x : B \vdash t : A'$ and $A' <: A$

  By induction, $\Gamma \vdash t[x := t'] : A'$.

  By typing, we get $\Gamma \vdash t[x := t'] : A$.

□

3.1.3 Type Preservation Theorem for Moggi's Calculus with Records and Subtyping

We prove preservation for the case where there are no additional effectful reductions. However, effect-free reductions are permitted. As already stated in Section 2 we must assume a type preservation for the new proper reduction rules without effects, e.g. we must assume for rule $p_1 \rightarrow p_2$ that whenever $\Gamma \vdash p_1 : A$ we also must have $\Gamma \vdash p_2 : A$. This property usually must be shown for any new proper reduction rule.

We are not able to prove this lemma also for effectful reduction since as we will see in our example with references in Section 4 the reduction could depent on a "global" context such as the store. Depending on the state of the store, an operation *deref* l to read the content of a certain cell l in store can reduce to different terms if we perform them for different stores. Hence, for the new reductions with effect we must reprove preservation in the contexts E. See Section 4 for an example of this. For all other rules we can apply the following theorem.

Theorem 3.5 (Preservation)**.** *If* $\Gamma \vdash t : A$, $t \to t'$ *(or* $t \xrightarrow{\alpha} t'$ *), then* $\Gamma \vdash t' : A$.

Proof. Again we prove the theorem by induction on type derivations. We perform a case analysis.

- cases T-VAR :

  Vacoously true, because there does not exist any t' such that $t \to t'$.

- case T-CONST :

  $t = c \qquad c : A \in Const$

  Here we may have $c \to t'$ as additional proper reduction rule without effect. A restriction of these rules is that they must preserve types.

- case T-ABS :

  $t = \lambda x : A_1.t'' \qquad \Gamma, x : A_1 \vdash t'' : A_2 \qquad A = A_1 \to A_2$

  - subcase η-reduction :

    $t'' = t'x \qquad x \notin FV(t')$

    Applying inversion to $t'x$ we get that for some A', $\Gamma, x : A_1 \vdash t' : A' \to A_2$ and $\Gamma, x : A_1 \vdash x : A'$, and because $x : A_1$ is in the type environment we have $A_1 <: A'$ by inversion. So we have $A' \to A_2 <: A_1 \to A_2$, and so $\Gamma, x : A_1 \vdash t' : A_1 \to A_2$. Because of the side condition that $x \notin FV(t')$ we can apply strengthening and get $\Gamma \vdash t' : A_1 \to A_2$.

  - subcase $C = \lambda x : A.C'$:

    $t'' \to t''' \qquad t' = \lambda x : A_1.t'''$

    By induction, $\Gamma, x : A_1 \vdash t''' : A_2$, hence $\Gamma \vdash \lambda x : A_1.t''' : A_1 \to A_2$.

- case T-APP :

  $t = t_1 t_2 \qquad \Gamma, x : B \vdash t_1 : A' \to A \qquad \Gamma, x : B \vdash t_2 : A'$

  - subcase $C = C't$:

    $t_1 \to t_1' \qquad t' = t_1' t_2$

    By induction, $\Gamma \vdash t_1' : A' \to A$.

    By typing, $\Gamma \vdash t' : A$.

- subcase $C = tC'$:
  $t_2 \to t_2'$ $\quad$ $t' = t_1 t_2'$
  By induction, $\Gamma \vdash t_2' : A'$.
  By typing, $\Gamma \vdash t' : A$.
- subcase β-reduction :
  $t_1 = \lambda x : A''.t_1'$ $\quad$ $t' = t_1'[x := t_2]$
  By inversion, $A' <: A''$ and $\Gamma, x : A' \vdash t_1' : A$
  By applying the type preservation under substitution we get $\Gamma \vdash t' : A$.
- subcase additional reduction rule for constant :
  Here t has the form $c\ t_1'\ \ldots\ t_n' t_2$. Since we have restricted such additional rules to fulfill the type preservation property there is nothing to show.

- case T-MON :
  $t = [t'']$ $\quad$ $\Gamma \vdash t'' : A'$ $\quad$ $A = TA'$
  Reduction in context $C = [C']$, we have $t'' \to t'''$, $t' = [t''']$.
  By induction, $\Gamma \vdash t''' : A'$.
  By typing, $\Gamma \vdash [t'''] : TA'$.

- case T-LET :
  $t = let\ x \Leftarrow t_1\ in\ t_2$ $\quad$ $\Gamma \vdash t_1 : TA'$ $\quad$ $\Gamma, x : A' \vdash t_2 : TB$
  $A = TB$
  - subcase $C = let\ x \Leftarrow C'\ in\ t_2$ / $E = let\ x \Leftarrow E'\ in\ t_2$:
    $t_1 \to t_1'$ $t_1 \xrightarrow{\alpha} t_1'$ $\quad$ $t' = let\ x \Leftarrow t_1'\ in\ t_2$
    By induction, $\Gamma \vdash t_1' : TA'$.
    By typing, $\Gamma \vdash t' : A$.
  - subcase $C = let\ x \Leftarrow t_1\ in\ C'$:
    $t_2 \to t_2'$ $\quad$ $t' = let\ x \Leftarrow t_1\ in\ t_2'$
    By induction, $\Gamma, x : A' \vdash t_2' : TB$.
    By typing, $\Gamma \vdash t' : A$.
  - subcase β.let reduction :
    $t_1 = [t_1']$ $\quad$ $t' = t_2[x := t_1']$
    By inversion, $\Gamma \vdash t_1' : A'$.
    By type preservation under substitution, we get $\Gamma \vdash t_2[x := t_1'] : A$.
  - subcase η.let reduction :
    $t_2 = [x]$ $\quad$ $t' = t_1$
    We can easily see that $TA' <: TB$, (by inversion $\Gamma, x : A' \vdash x : B$ and $A' <: B$, then use S-Mon) hence we can type $\Gamma \vdash t_1 : TB$.

– subcase assoc-reduction :

$t_1 = let\ y \Leftarrow t_3\ in\ t_4 \qquad y \notin FV(t_2)$
$t' = let\ y \Leftarrow t_3\ in(let\ x \Leftarrow t_4\ in\ t_2)$
By applying inversion to the inner let t_1 we get $\Gamma \vdash t_3 : TA''$ and $\Gamma, y : A'' \vdash t_4 : TA'$ for some A''.

Now we can type t' with the following inference. (*) on the lines means that this typing is given by any of the assumptions. (STRENGTH) denotes an application of the strengthening lemma.

$$\dfrac{\dfrac{}{\Gamma \vdash t_3 : TA''}\ (*) \quad \dfrac{\dfrac{}{\Gamma, y : A'' \vdash t_4 : TA'}\ (*) \quad \dfrac{\dfrac{}{\Gamma, x : A' \vdash t_2 : TB}\ (*)}{\Gamma, y : A'', x : A' \vdash t_2 : TB}\ \text{(STRENGTH)}}{\Gamma, y : A'' \vdash let\ x \Leftarrow t_4\ in\ t_2 : TB}\ \text{(T-LET)}}{\Gamma \vdash let\ y \Leftarrow t_3\ in(let\ x \Leftarrow t_4\ in\ t_2) : TB}\ \text{(T-LET)}$$

- case T-RCD :

$t = \{l_i : t_i\ ^{i=1,\dots,n}\} \qquad A = \{l_i : A_i\ ^{i=1,\dots,n}\} \qquad \forall i \in [n] : \Gamma \vdash t_i : A_i$

  – subcase $C = \{l_i = t_i\ ^{i=1,\dots,j-1}, l_j = C, l_i = t_i\ ^{i=j+1,\dots,n}\}$:

  $t_j \to t'_j \qquad t' = \{l_i = t_i\ ^{i=1,\dots,j-1}, l_j = t'_j, l_i = t_i\ ^{i=j+1,\dots,n}\}$
  By induction, $\Gamma \vdash t'_j : A_j$.
  By typing, $\Gamma \vdash \{l_i = t_i\ ^{i=1,\dots,j-1}, l_j = t'_j, l_i = t_i\ ^{i=j+1,\dots,n}\} : A$.

  – subcase η.rec-reduction :

  $\forall i \in [n] : t_i = t'.l_i$
  By inversion $\Gamma \vdash t' : B$ and B has the form $\{l_i : B_i\ ^{i=1,\dots,m}\}$ containing at least the labels $\{l_i\ ^{i=1,\dots,m}\}$ and for every label l_i $B_i <: A_i$. Hence, $B <: A$ and so by T-SUB $\Gamma \vdash t' : A$.

- case T-PROJ :

$t = t''.l_j \qquad \Gamma \vdash t'' : \{l_i : A_i\ ^{i=1,\dots,n}\} \qquad A = A_j \qquad j \in [n]$

  – subcase $C = C'.l_j$:

  $t'' \to t''' \qquad t' = t'''.l_j$
  By induction, $\Gamma \vdash t''' : \{l_i : A_i\ ^{i=1,\dots,n}\}$.
  By typing, $\Gamma \vdash t'''.l_j : A$.

  – subcase β.rec reduction :

  $t'' = \{k_a = t_a\ ^{a=1,\dots,m}\} \qquad t' = t_h \qquad$ for $k_h = l_j$.
  By inversion, $\{l_i\ ^{i=1,\dots,n}\} \subseteq \{k_a\ ^{a=1,\dots,m}\}$ and $\Gamma \vdash t_i : A_j$ whenever $l_i = k_a$. Let $k_h = l_j$. Then $t' = t_h$ and $\Gamma \vdash t_h : A_j$.

□

Remark: This property is only true in this direction. It is possible that we can give t' a type that t does not have. Let's consider the following term $t = (\lambda x : Top.x)t'$. Independent of the type of t', we can give t only the type Top. But t' may have any arbitrary subtype of Top different from the type Top.

3.2 Progress of the Calculus

In this section we show the progress property, i.e. we define a set of values which cannot be reduced anymore and show that all other terms typable in empty type environment are still reducible.

3.2.1 Normalform of the Values

We start by defining what the values of our calculus are. Normally, values are closed terms (terms typable in the empty type envirenment) that have no reduction rule to apply. It turns out that this is complicated for our calculus. One reason is that we may reduce effect-free under lambdas or on record fields or we only could reduce there observing effect, so that we simply cannot place there arbitrary term or value again. These terms under lambda now can have a variable and hence they aren't typable anymore in empty type envirenment. But to prove Progress we must also give some properties about such term typable in arbitrary type envirenment.

Another problem is that the set of values can depend on the additional reduction rules we added to the language. So if we add a rule for $c\ t_1\ \ldots\ t_n$ this term should not be a value anymore. For this reason we give a modular progress proof in the sence that we define values without using new reductions with effects. If then we want to show progress for a specific calculus with all reductions, we define the new value set Val' and show that it equals the set $Val - \{t \mid t \xrightarrow{\alpha}\}$, i.e. every term in $Val - Val'$ has a reduction. So we give now the definition of values using to auxilary set of values with nested recursion and then explain why they are necessary and how they work.

$$
\begin{array}{rcll}
a \in Val_{APP} & = & x \mid am \mid a.l & \\
m \in Val_{MON} & = & a \mid let\ x \Leftarrow m\ in\ m \mid v & \\
v \in Val & = & \lambda x : A.m & \text{where } m \neq tx \text{ or } x \in FV(t) \\
 & & \mid \{l_i = m_i\ ^{i=1\ldots n}\} & \text{where } m_i \neq v'.l_i \text{ for some } i \in [n] \\
 & & \mid [m] & \\
 & & \mid c\ m_1 \ldots m_n & \text{if } c\ m_1 \ldots m_n \not\rightarrow \\
 & & \mid v.l & \text{where } v \neq \{l_i = m_i\ ^{i=1,\ldots,n}\} \\
 & & \mid let\ x \Leftarrow v\ in\ m & \text{where } v \neq [m] \text{ and } v \neq let \ldots \\
 & & \mid v\ m & \text{where } v \neq \lambda x : A.m \text{ and} \\
 & & & v \neq c\ m_1 \ldots m_n
\end{array}
$$

The set Val is the "real" set of values we want to define. The class Val_{MON}

contains terms that maybe still can be reduced, but only by observing an effect. This class also contains all terms of the other defined classes. These terms we put in the definition of Val where we only may reduce without observing effects, e.g. under λ or on record field. This way we exclude terms in Val that still have possible effect-free reductions where only effectfree reduction is allowed, and keep the possibility to place there terms that only have reductions observing effects. Note that all terms in Val_{MON} that aren't in Val or Val_{APP} have a monadic type if they are typable in an arbitray environment.

Since we can have variables in some of the subterms we use an additional class Val_{APP} producing such subterms. Not embedding these remaining terms in Val_{MON} has the advantage that we can forbid terms of monadic type as functions or as records and it ensures all terms $t \in Val_{MON} - Val - Val_{APP}$ have a monadic type. And it turns out that we only need to take out one further construct of Val_{MON} to get only the terms having a reduction with effect so it gets easier to state an auxiliary Progress Theorem in this section. Also oberve that all terms of Val_{APP} are not typable in the empty environment because they contain free variables.

In Val we still have the constructs $v.l$, $v\ m$ and $let\ x \Leftarrow v\ in\ m$, one normally wants to exclude from the set of values. We can do this if we give reduction rules for all terms of the form $c\ m_1\ \ldots\ m_n$ without functional type. For $v.l$, $v\ m$ this is also the case if terms $c\ m_1\ \ldots\ m_n$ with monadic type are not reducible by some additional proper reduction rule. Lemma 3.8 helps to prove this, because it shows that we can express such values as $C[c\ m_1\ \ldots\ m_n]$. Here we still need them because some of these reductions from constants have effects and the theorem we prove here does not take additional effectful reductions in account.

We also state two other useful lemmata: Lemma 3.6 states that we have defined the values so that we can't reduce anymore, or only with effect in Val_{MON}. Lemma 3.7 is an auxiliary lemma we need in the progress proof, stating that values of a certain type have a certain form.

Lemma 3.6. *Let t be a term with $\Gamma \vdash t : A$.*

1. *If $t \in Val_{MON}$, then $\nexists t' : t \to t'$.*
2. *If $t \in Val$ or Val_{APP}, then $\nexists t', \alpha : t \to t' \vee t \xrightarrow{\alpha} t'$.*

Proof. Proof by induction on type derivations.

□

Lemma 3.7 (Normalform of values)**.** *Let v be an value and $\Gamma \vdash v : A$.*

1. *If $\Gamma \vdash v : A_1 \to A_2$, then it has one of the following forms:*
   (a) $\lambda x : A'.m$
   (b) $c\ m_1 \ldots m_n$ *where* $c\ m_1 \ldots m_n \not\to$
   (c) $v.l$ *where* $v \neq \{l_i = m_i\ ^{i=1,\ldots,n}\}$
   (d) $v\ m$ *where* $v \neq \lambda x : A.m$ *and* $v \neq c\ m_1 \ldots m_n$

2. *If* $\Gamma \vdash v : T\ A'$, *then it has one of the following forms:*

   (a) $[m]$
   (b) $let\ x \Leftarrow v\ in\ m$ *where* $v \neq [m]$ *and* $v \neq let \ldots$
   (c) $c\ m_1 \ldots m_n$ *where* $c\ m_1 \ldots m_n \not\rightarrow$
   (d) $v.l$ *where* $v \neq \{l_i = m_i\ ^{i=1,\ldots,n}\}$
   (e) $v\ m$ *where* $v \neq \lambda x : A.m$ *and* $v \neq c\ m_1 \ldots m_n$

3. *If* $\Gamma \vdash v : \{k_j : A_i\ ^{j=1\ldots m}\}$, *then it has one of the following form*

   (a) $\{l_i = m_i\ ^{i=1,\ldots,n}\}$
   (b) $c\ m_1 \ldots m_n$ *where* $c\ m_1 \ldots m_n \not\rightarrow$
   (c) $v.l$ *where* $v \neq \{l_i = m_i\ ^{i=1,\ldots,n}\}$
   (d) $v\ m$ *where* $v \neq \lambda x : A.m$ *and* $v \neq c\ m_1 \ldots m_n$

Proof. Proof by induction on type derivations.

□

Lemma 3.8. *Let* v *be a value of the form* $c\ m_1 \ldots m_n$, $v'.l$, $v'\ m$ *or let* $x \Leftarrow v'$ *in* m. *If* $\Gamma \vdash v : A$, *then there is a context* C *produced without using the context production rules* tC *and let* $\ldots$ *in* C *for effect-free contexts and a term* $t = c\ m_1\ \ldots\ m_n$ *with* $\Gamma \vdash t : B$ *such that* $v = C[t]$.

Furthermore, if B *has the form* $T\ B'$ *then for some type* A', $\Gamma \vdash v : TA'$ *and* C *is either* $\circ$ *or let* $x \Leftarrow \circ$ *in* m *which is also a context* E *for effectful reductions.*

If B *is a functional type, then* $C = \circ$.

As a direct consequence of this lemma for all forms except $c\ m_1\ \ldots\ m_n$ the subterm t has a nonfunctional type in Σ. So if we can reduce all terms with nonfunctional type of the form $c\ m_1\ \ldots\ m_n$, we can reduce all terms of the forms $v'.l$, $v'\ m$ or $let\ x \Leftarrow v'\ in\ m$.

Proof. We prove this by induction on the derivation of the typing relation.

- <u>case T-APP:</u>
  - subcase $v = c\ m_1 \ldots m_n$
    Here, we have $C = \circ$, $t = v$ and hence, $A = B$.
  - subcase $v = v'\ m$
    $\Gamma \vdash v' : A' \rightarrow A \qquad \Gamma \vdash m : A'$
    By induction there is a context C' and a term $t' = c\ m_1 \ldots m_n = t$ with $\Gamma \vdash t : B$ and $v' = C'[t']$. Hence, for $C = C'\ m$ we have $v = C[t]$. B cannot be a monadic type since then, by induction v' had a monadic type, a contradiction. If B would be a funtional type then $C' = \circ$ and hence, $v' = t'$ and so v' has a form excluded by definition of values. Because we excluded production rule tC for effect-free contexts we are not allowed to search for a t in m.

- case T-PROJ:

  $v = v'.l_j \qquad \Gamma \vdash v' : \{l_i : A_i \,^{i=1,\ldots,n}\} \qquad j \in [n]$

  By induction there is a context C' and a term $t' = c\ m_1 \ldots m_n = t$ with $\Gamma \vdash t : B$ and $v' = C'[t']$. Hence, for $C = C'.l_j$ we have $v = C[t]$. B can't be a monadic (functional) type since then, by induction v' had a monadic (functional) type, a contradiction.

- case T-LET:

  $v = let\ x \Leftarrow v' in\ m \qquad \Gamma \vdash v' : TA'$

  By induction there is a context C' and a term $t' = c\ m_1 \ldots m_n = t$ with $\Gamma \vdash t : B$ and $v' = C'[t']$. Hence, for $C = let\ x \Leftarrow C' in\ m$ we have $v = C[t]$. Furthermore, if $B = T\ B'$, then we know that C is also a context E' and hence C is also a context E for effectful reductions. Since let-expressions for v' are excluded, it must hold that $E' = \circ$. We are not allowed to search for t in m because lemma forbids use of rule *let ... in C*. Again we can exclude a functional type B, since then v' had a functional type, a contradiction.

- case T-SUB:

  $\Gamma \vdash v : A' \qquad A' <: A$

  By induction v has the required properties.

The other typing rules are not applicable because they need a special form for value v that the lemma does not cover.

□

3.2.2 Progress Proof

Here we prove the Progress Theorem using a modified auxiliary Progress Theorem which implies the correctness of the original version directly. One modification of this auxiliary theorem is that we use an arbitrary type environment Γ since we have also to state properties for the terms under lambdas and after the "in" of let-expression, which can have free variables. And we have a second modification that we distinguish between reduction with and without effect. As already stated we prove the theorem not using additional reductions with effects. The standard Progress Theorem would be as follows.

Theorem 3.9 (Progress)**.** *Let $t \in Ter$ and $\emptyset \vdash t : A$. Then either $t \in Val$, or $\exists t' \in Ter, \alpha \in pEff : t \to t' \vee t \xrightarrow{\alpha} t'$.*

Proof. The correctness of the Progress Theorem follows directly from the auxiliary Progress below. All we need to show is that all terms in Val_{APP} and terms of the form $let\ x \Leftarrow a\ in\ m$ with $a \in Val_{APP}$, and $m \in Val_{MON}$ are not typable in the empty environment. For Val_{APP} this is easy to prove by induction with base case of a variable. For the second form we just have to see that we only can type our term in empty environment if we can do this with subterm $a \in Val_{APP}$. □

Now we can state and prove the auxilary Progress Theorem.

Theorem 3.10 (Auxiliary Progress)**.** *For every term t and every type environment Γ, if $\Gamma \vdash t : A$, then one of the following statements holds:*

1. *$t \in Val$ or Val_{APP}, or t has the form let $x \Leftarrow a$ in m with $a \in Val_{APP}$, and $m \in Val_{MON}$*
2. *$t \in Val_{MON}$ and $\exists t', \alpha : t \xrightarrow{\alpha} t'$*
3. *$\exists t' : t \to t'$*

Proof. Again we prove the theorem by structural induction on type derivations and perform a big case analysis. Note that all cases of statement 1 imply that $t \in Val_{MON}$.

- case T-VAR :
  $t = x \qquad x : A \in \Gamma$
  Every variable x is in Val_{APP}.
- case T-ABS :
  $t = \lambda x : A_1.t'' \qquad \Gamma, x : A_1 \vdash t' : A_2 \qquad A = A_1 \to A_2$
  1. subcase: $t'' = t'x, x \notin FV(t')$
     Hence, one can reduce t to t' by η-reduction. So statement 3 holds.
  2. subcase: $t'' \in Val_{MON}$, $t'' \neq t'x$ or $x \in FV(t')$
     So $t \in Val$ by definition and hence statement 1 holds.
  3. subcase: $t'' \notin Val_{MON}$
     By induction, there is t''' such that $t'' \to t'''$. Hence $\lambda x : A_1.t'' \to \lambda x : A_1.t'''$.
- case T-APP :
  $t = t_1 t_2 \qquad \Gamma \vdash t_1 : A' \to A \qquad \Gamma \vdash t_2 : A' \qquad$ for some type A'
  We don't need to consider the case that t_1 is an irreducible let-expression because they cannot have functional type and so we had an ill-typed term.
  1. subcase: $t_1 \in Val_{APP}, t_2 \in Val_{MON}$
     So $t \in Val_{APP}$ by definition.
  2. subcase: $t_1 \in Val, t_2 \in Val_{MON}$
     By Lemma 3.7 value t_1 has one of the following forms:
     (a) $\lambda x : A'.m$
     Here, we can apply β-reduction and reduce $t \to m[x := t_2]$.
     (b) $c\ m_1 \dots m_n$ where $c\ m_1 \dots m_n \not\to$
     Either we can reduce now or we have $c\ m_1 \dots m_n\ t_2 \not\to$ and so $t \in Val$.

(c) $v.l$ where $v \neq \{l_i = m_i \ ^{i=1,\ldots,n}\}$
Here $v.l\ t_2 \in Val$ because $v.l$ is not fobidden in the side-condition.

(d) $v\ m$ where $v \neq \lambda x : A.m$ and $v \neq c\ m_1 \ldots m_n$
Again, we have $t \in Val$.

3. subcase: $t_1 \notin Val_{MON}$
Because t_1 is typeable in the type environment Γ one of the three statements must hold by induction. Since $t_1 \notin Val_{MON}$ there exists a term t_1' such that $t_1 \to t_1'$.
Now we can do the reduction $t_1 t_2 \to t_1' t_2$, so statement 3 holds.

4. subcase: $t_1 \in Val_{MON}, t_2 \notin Val_{MON}$
analogous to 2. subcase, reduce to $t_1 t_2'$.

5. subcase $t_1 \in Val_{MON}, t_2 \in Val_{MON}$
Here we have nothing to show because either $t_1 \in Val, Val_{APP}$ (already covered by other cases) or t_1 is a let-expression which has monadic type or Top. But application requires a functional type for t_1.

- <u>case T-RCD :</u>
$t = \{l_i = t_i \ ^{i=1\ldots n}\} \qquad \forall i \in [n] : \Gamma \vdash t_i : A_i \qquad A = \{l_i : A_i \ ^{i=1,\ldots,n}\}$

1. subcase: $\forall i \in [n] : t_i = t'.l_i$
here we can reduce $t \to t'$ by η.rec-reduction.

2. subcase: $\forall i \in [n] : t_i \in Val_{MON}$ and $\exists i \in [n] : t_i \neq t'.l_i$
By definiton t is already a value and hence statement 1 holds.

3. subcase: $\exists j \in [n] : t_j \notin Val_{MON}$
Since $\Gamma \vdash t_j : A_j$, by induction and the fact that t_j is not in Val_{MON} there exists t_j' such that $t_j \to t_j'$.
Now we can do the reduction $\{l_i = t_i \ ^{i=1\ldots n}\} \to \{l_i = t_i \ ^{i=1,\ldots,j-1}, l_j = t_j', l_i = t_i \ ^{i=j+1\ldots n}\}$, so statement 3 holds.

- <u>case T-PROJ :</u>
$t = t''.l_j \qquad \Gamma \vdash t'' : \{l_i : A_i \ ^{i=1,\ldots,n}\} \qquad j \in [n]$

1. subcase $t'' \in Val_{APP}$
So $t \in Val_{APP}$ by definition.

2. subcase: $t'' \in Val$
By Lemma 3.7 t'' has one of the following form:

(a) $\{l_i = m_i \ ^{i=1,\ldots,n}\}$
Here, t can be reduced with a $\beta.rec$-reduction to m_a with $k_a = l_j$.
So statement 3 holds.

(b) $c\ m_1 \ldots m_n$ where $c\ m_1 \ldots m_n \not\to$
$t \in Val$

(c) $v.l$ where $v \neq \{l_i = m_i\ ^{i=1,\ldots,n}\}$
$t \in Val$

(d) $v\ m$ where $v \neq \lambda x : A.m$ and $v \neq c\ m_1 \ldots m_n$
$t \in Val$

3. subcase: $t'' \notin Val_{MON}$
By induction there exists a t''' such that $t'' \rightarrow t'''$. So we can perform the reduction $t''.l_j \rightarrow t'''.l_j$. So, here statement 3 holds.

4. subcase $t'' \in Val_{MON}$
Same argument as for subcase 5 of T-APP. Here we need a record type instead of a functional type, which also creates a contradiction.

- <u>case T-MON :</u>
$t = [t'']$ $\quad \Gamma \vdash t'' : A'$ $\quad A = TA'$

1. subcase: $t'' \in Val_{MON}$
By definition t is already a value.

2. subcase: $t'' \notin Val_{MON}$
By induction there is t''' such that $t'' \rightarrow t'''$ hence we can reduce $[t''] \rightarrow [t''']$.

- <u>case T-LET :</u>
$t = let\ x \Leftarrow t_1\ in\ t_2$ $\quad \Gamma \vdash t_1 : TA'$ $\quad \Gamma, x : A' \vdash t_2 : TA''$
$A = TA''$ $\quad$ for some A'

1. subcase: $t_1 \in Val_{APP}, t_2 \in Val_{MON}$
So $t \in Val_{MON}$ by definition.

2. subcase: $t_1 \in Val, t_2 \in Val_{MON}$
By Lemma 3.7 t_1 has one of the following forms:

(a) $[m]$
Here, we can apply the $\beta.let$-reduction to get $let\ x \Leftarrow [m]\ in\ N \xrightarrow{\emptyset} N[x := m]$. So statement 2 holds.

(b) $let\ y \Leftarrow v\ in\ m$ where $v \neq [m]$ and $v \neq let \ldots$
Here, we can perform an assoc reduction and get the term $let\ y \Leftarrow v\ in\ let\ x \Leftarrow m\ in\ t_2$. So statement 2 holds.

(c) $c\ m_1 \ldots m_n$ where $c\ m_1 \ldots m_n \not\rightarrow$
$t \in Val$.

(d) $v.l$ where $v \neq \{l_i = m_i\ ^{i=1,\ldots,n}\}$
$t \in Val$.

(e) $v\ m$ where $v \neq \lambda x : A.m$ and $v \neq c\ m_1 \ldots m_n$
$t \in Val$.

3. subcase: t_1 has the form $let\ y \Leftarrow a\ in\ m$ with $a \in Val_{APP}, m \in Val_{MON}$, $t_2 \in Val_{MON}$

   $t \in Val_{MON}$ is clear. We can do an *assoc*-reduction.

   $let\ x \Leftarrow (let\ y \Leftarrow a\ in\ m)\ in\ t_2 \xrightarrow{\emptyset} let\ y \Leftarrow a\ in\ let\ x \Leftarrow m\ in\ t_2$. So here also holds statement 2.

4. subcase: $t_1 \in Val_{MON}$ and $\exists t_1', \alpha : t_1 \xrightarrow{\alpha} t_1'$,$t_2 \in Val_{MON}$

   In this case $t \in Val_{MON}$ and we have the reduction $let\ x \Leftarrow t_1\ in\ t_2 \xrightarrow{\alpha} let\ x \Leftarrow t_1'\ in\ t_2$. So statement 2 holds.

5. subcase: $t_1 \notin Val_{MON}$

   By induction, there is t_1' with $t_1 \to t_1'$ So statement 3 holds with the reduction $let\ x \Leftarrow t_1\ in\ t_2 \to let\ x \Leftarrow t_1'\ in\ t_2$.

6. subcase: $t_2 \notin Val_{MON}$

   By induction there is t_2' such that $t_2 \to t_2'$. Hence we have the reduction $let\ x \Leftarrow t_1\ in\ t_2 \to let\ x \Leftarrow t_1\ in\ t_2'$ and statement 3 holds.

- case T-SUB :

  $\Gamma \vdash t : A' \qquad A' <: A$

  By induction one of the statements holds for t.

□

3.3 Strong Typing

One problem of the weak typing relation is that terms do not have unique types which makes it harder to determine the type of a term algorithmically, as even just check that a given judgement $\Gamma \vdash t : A$ is derivable. In this subsection we will see that there is a unique strong type A for every term t which is a subtype of all possible weak types, and conversely all supertypes of A are weak types of t. So if we want to know if a certain type is a weak type of t we can determine the strong types and test for subtype.

3.3.1 Strong Typing Relation

In Figure 9 we can see the strong typing rules, which derive judgements of the form $\Gamma \vdash_{str} t : A$ most rules analogues to the weak typing relation. We don't have a rule T-Sub anymore creating different types for the same terms. Modified is only the rule for applications. This is because we have the possibility to give the function an argument which has a subtype of the required argument type and we cannot type the argument to a supertype anymore by a T-Sub rule. So we solve the problem by simply adding a subtype test to the rule.

$$\frac{}{\Gamma, x : A \vdash_{str} x : A}\ \text{(T-VAR)} \qquad \frac{c : A \in Const}{\Gamma \vdash_{str} c : A}\ \text{(T-Const)} \qquad \frac{\Gamma, x : A \vdash_{str} t : B}{\Gamma \vdash_{str} \lambda x : A.t : A \to B}\ \text{(T-Abs)}$$

$$\frac{\Gamma \vdash_{str} t_1 : A \to B \qquad \Gamma \vdash_{str} t_2 : A' \qquad A' <: A}{\Gamma \vdash_{str} t_1 t_2 : B}\ \text{(T-App)}$$

$$\frac{\Gamma \vdash_{str} t : A}{\Gamma \vdash_{str} [t] : TA}\ \text{(T-Mon)} \qquad \frac{\Gamma \vdash_{str} t_1 : TA \qquad \Gamma, x : A \vdash_{str} t_2 : TB}{\Gamma \vdash_{str} let\ x \Leftarrow t_1\ in\ t_2 : TB}\ \text{(T-Let)}$$

$$\frac{\forall i \in [n] : \Gamma \vdash_{str} t_i : A_i}{\Gamma \vdash_{str} \{l_i = t_i{}^{i=1\ldots n}\} : \{l_i : A_i{}^{i=1\ldots n}\}}\ \text{(T-Rcd)}$$

$$\frac{\Gamma \vdash_{str} t : \{l_i : A_i{}^{i=1\ldots n}\} \qquad j \in [n]}{\Gamma \vdash_{str} t.l_j : A_j}\ \text{(T-Proj)}$$

Figure 9: Strong subtype rules

3.3.2 Properties of the Strong Typing Relation

Definition: $\Gamma' \leq \Gamma \iff Dom(\Gamma') = Dom(\Gamma) \wedge \forall x \in Dom(\Gamma) : \Gamma' x <: \Gamma x$

Lemma 3.11 (Subsumption property)**.** *If* $\Gamma \vdash t : A$ *and* $\Gamma' \leq \Gamma$ *and* $A <: B$*, then* $\Gamma' \vdash t : B$ *.*

Proof. By a straightforward induction on typing derivations we can show $\Gamma' \vdash t : A$. Then by T-Sub follows $\Gamma' \vdash t : B$.

□

Lemma 3.12. $\Gamma \vdash t : A \iff \exists B : \Gamma \vdash_{str} t : B$ *and* $B <: A$*.*

Proof.

" $\Leftarrow$ " :

We first show that $\Gamma \vdash_{str} t : B \Rightarrow \Gamma \vdash t : B$ by induction on strong typing derivation. The required property $\Gamma \vdash t : A$ with $B <: A$ follows directly by T-Sub rule for weak typing.

In the induction proof we only consider the rule for applications because the other rules have the same form in both typings. These rules can be proven by applying the induction hypothesis to the premises of the rule and retyping the same way with the weak typing rule

- <u>case T-APP:</u>
  $\Gamma \vdash_{str} t_1 : A' \to B \qquad \Gamma \vdash_{str} t_2 : A'' \qquad A'' <: A'$
  By induction, $\Gamma \vdash t_1 : A' \to B$ and $\Gamma \vdash t_2 : A''$
  By weak T-Sub, $\Gamma \vdash t_2 : A'$, and by weak T-App, $\Gamma \vdash t_1 t_2' : B$.

" $\Rightarrow$ " :
The proof is by straight forward induction on type derivations.

- case T-VAR :

  $t = x \qquad \Gamma \vdash x : A \qquad x : A \in \Gamma$

  So we can type $\Gamma \vdash_{str} x : A$ and $A <: A$ by S-Refl.

- case T-CONST :

  $t = c \qquad \Gamma \vdash c : A \qquad c : A \in Const$

  So we can type $\Gamma \vdash_{str} c : A$ and $A <: A$ by S-Refl.

- case T-ABS :

  $t = \lambda x : A_1.t' \qquad \Gamma, x : A_1 \vdash t' : A_2 \qquad A = A_1 \to A_2$

  By induction, there is B_2 such that $\Gamma, x : A_1 \vdash_{str} t' : B_2$ and $B_2 <: A_2$. Hence, $\Gamma \vdash_{str} t : A_1 \to B_2$ and by S-Arrow, $A_1 \to B_2 <: A_1 \to A_2$.

- case T-APP :

  $t = t_1 t_2 \qquad \Gamma \vdash t_1 : A' \to A \qquad \Gamma \vdash t_2 : A' \qquad$ for some A'

  By induction, there are types B, C such that $\Gamma \vdash_{str} t_1 : B$ and $B <: A' \to A$ and $\Gamma \vdash_{str} t_2 : C$ and $C <: A'$. By inversion of subtype relation, B has the form $B_1 \to B_2$ with $A' <: B_1$ and $B_2 <: A$. By S-Trans, $C <: B_1$, hence $\Gamma \vdash_{str} t : B_2$ with $B_2 <: A$.

- case T-MON :

  $t = [t'] \qquad \Gamma \vdash t' : A' \qquad A = TA' \qquad$ for some A'

  By induction, there is a type B such that $\Gamma \vdash_{str} t' : B$ and $B <: A'$. So $\Gamma \vdash_{str} [t'] : TB$ and by S-MON, $TB <: TA'$.

- case T-LET :

  $t = let\ x \Leftarrow t_1\ in\ t_2 \qquad \Gamma \vdash t_1 : TB' \qquad \Gamma, x : A' \vdash t_2 : TA'$
  $A = TA'$

  By induction, there is a type C such that $\Gamma \vdash_{str} t_1 : C$ and $C <: TB'$. By inversion of the subtype relation, C has the form TC' with $C' <: B'$. Hence, we have $\Gamma, x : C' \leq \Gamma, x : B'$ and so by the subsumption property, $\Gamma, x : C' \vdash t_2 : TA'$.

  By induction, there is a type B such that $\Gamma \vdash_{str} t_2 : B$ and $B <: TA'$. Hence, $\Gamma \vdash_{str} t : B$ and $B <: A$.

- case T-RCD :

  $t = \{l_i = t_i\ ^{i=1,\ldots,n}\} \qquad \forall i \in [n] : \Gamma \vdash t_i : A_i$
  $A = \{l_i : A_i\ ^{i=1,\ldots,n}\}$

  By induction, there are types $B_1, B_2, \ldots, B_n$ such that $\forall i \in [n] : \Gamma \vdash_{str} t_i : B_i \wedge B_i <: A_i$. By S-RcdDepth $\{l_i : B_i\ ^{i=1,\ldots,n}\} <: \{l_i : A_i\ ^{i=1,\ldots,n}\}$, and by typing, $\Gamma \vdash_{str} t : \{l_i : B_i\ ^{i=1,\ldots,n}\}$.

- case T-PROJ :

  $t = t'.l_j \qquad \Gamma \vdash t' : \{l_i : A_i \ ^{i=1,\ldots,n}\} \qquad j \in [n] \qquad A = A_j$

  By induction, $\Gamma \vdash_{str} t' : B$ and $B <: \{l_i : A_i \ ^{i=1,\ldots,n}\}$. By inversion of the subtype relation, B has the form $\{k_a : B_a \ ^{a=1,\ldots,m}\}$ with at least the labels $\{l_i \ ^{1=1,\ldots,n}\}$ and $B_a <: A_i$ whenever $k_a = l_i$.

  Let $k_a = l_j$, then $\Gamma \vdash_{str} t : B_a$ and $B_a <: A_j$.

- case T-SUB :

  $\Gamma \vdash t : C$ and $C <: A$

  By induction, there is B such that $\Gamma \vdash_{str} t : B$ with $B <: C$ and by S-Trans $B <: A$.

□

Lemma 3.13 (Uniqueness of strong Types)**.** *If* $\Gamma \vdash_{str} t : A$ *and* $\Gamma \vdash_{str} t : B$*, then* $A = B$*.*

Proof. The proof is a straight-forward induction on type derivations of the strong typing relation.

□

4 Application: Object Calculus

In this chapter we show an application, considering an imperative object calculus. First we introduce new constants needed for simulating the object calculus, and prove in a modular way that the standard soundness properties also hold in our extended monadic calculus. We introduce a variant of the imperative object calculus of Abadi & Cardelli in [2, 1, 4, 3], very similar to the one Abadi & Leino presented in [5] (without the boolean constructs). Then we show how to translate this calculus into the monadic calculus with subtyping and the extension we now introduce. We show that this encoding respects typing.

4.1 Extensions in the Monadic Calculus

To simulate an object calculus in the monadic language we need to define some constants to realize the implicit references and a new type contructor ref to create references of type A. This type constructor ref must be contained in TC, the set of type constructors with arity 1. In [10] we can read that for $ref\ A <: ref\ B$ we need that $A <: B$ and $B <: A$. But this is only the case if B is a permutation on the fields of the record type nested in A. Here we can also do without additional subtype rules for a type $ref\ A$. So we do not get in conflict with the requirement that no new subtype rules are defined.

Another feature of the imperative object calculus is the implicit recursion when we call a method field of an object. Hence we need a recursion operator in the monadic calculus. So it is clear what new constances we need to define in our constant set $Const$: We need a fixpoint operator for recursion and the standard operations to use references. Since constants in $Const$ need a particular type and these operation are usually type polymorphically depending on one type we need the constants for every possible type. For simplicity, we define them in $Const$ one time using an arbitrary type A.

$$\begin{aligned} Const \quad = \quad & \{new_A : A \to T(refA), update_A : refA \to A \to T\{\}, \\ & deref_A : refA \to TA, fix_A : (TA \to TA) \to TA\} \end{aligned}$$

Here we indexed every new constant with a type which means that by defining one constant we have defined one constant for every possible type.

Note that in a calculus with references we can simulate a the fixpoint fix_A for types that have a closed term t_A ($\emptyset \vdash t_A : A$). For the following translation of the object calculus we can always construct such a term t_A for every translation of an object type, but for technical reasons it is easier to use a constant. The term simulating fix_A would be as follows:

$$\begin{aligned} fix_A = \quad & \lambda f : TA \to TA.let\ l \Leftarrow new_{TA}\ [t_A]\ in\ let\_ \Leftarrow update_{TA}\ l \\ & (let\ x \Leftarrow deref_{TA}\ l\ in\ f\ x) in\ let\ y \Leftarrow deref_{TA}\ l\ in\ y \end{aligned}$$

Additionally we have to perform storage updates while evaluating. Therefore we do not only need a store S, but also some proper effects stating which update operation we perform. Hence, we define the set of proper effects $pEff$ as follows:

$$
\begin{array}{lll}
fix_A\ t & \xrightarrow{fix} & t(fix_A\ t) \\
new_A\ t & \xrightarrow{l=t:A} & [l] \\
update_A\ lt & \xrightarrow{!l:=t} & [\{\}] \\
deref_A\ l & \xrightarrow{?l=t} & [t]
\end{array}
$$

Figure 10: Additional proper reduction rules

$$pEff = \{fix, l = t : A, !l := t, ?l = t\}$$

Every proper effect except for fix indicating that we execute a fixpoint operation stands for a special storage operation. The first one is $l = t : A$. It adds a new cell of type A with initial content t to store, accessible over the label l. We need to give a type because the type of t could depend on the type of some free variables bound in a type environment or could be a subtype of the required cell type. The effect $!l := t$ changes the content of cell l to t, $?l = t$ reads the content t of cell l. Knowing the intuition of labels we define in Figure 10 additional reduction rules where all rules have some new proper effects. Furthermore, it is easy to see that all left-hand sides of the new rules have monadic types.

Furthermore, now terms can have reference labels for that we must be able to give a type. So we also introduce an environment Σ which gives the reference labels a type of the form $ref\ A$. If we do not differ between variables and labels we can use a normal variable for a reference label if we make sure that no label variable is used by the term. This we can do if the l in the effect $l = v : A$ for creating a new cell does not occur in the evaluated term.

So for typing we must use this Σ instead of the empty environment to type any term. But typing only works if we have a consistent store. That means that all labels l in Σ are typed with a type $refA$ and have a value of type A in the corresponding reference cell in a concrete store. Now we give a formal definition of top level evaluation and storage consistency in Figure 11. Note that for reductions without effects or the effect $\emptyset$ and fix we may reduce on top-level without changing S, Σ. Furthermore we use the notation $\Sigma \mid \Gamma \vdash t : A$ as an alternative to $\Sigma, \Gamma \vdash t : A$ to state that Σ is the environment for labels and contains only typings of the form $x : refA$ where x is a label variable.

We still have to prove that the soundness properties Preservation and Progress hold. We start with Preservation and continue with Progress. For the new theorems we need to take care of our storage. Fortunately we can use an analogous formulation to the one Pierce used for his storage extension in Chapter 13 of [10].

Theorem 4.1 (Preservation of Calculus with Storage and Fixpoint)**.** *If* $\Sigma \mid \Gamma \vdash t : A$*, and* $\Sigma \mid \Gamma \vdash S$*, and* $S \mid t \to S' \mid t'$ *then* $\exists \Sigma' : \Sigma \subseteq \Sigma' \wedge \Sigma' \mid \Gamma \vdash t' : A \wedge \Sigma' \mid \Gamma \vdash S'$.

Proof. Finally, here is an example for a modular proof. In the cases that we have an effect-free reduction or a reduction with the empty effect we can argue

$$\frac{t \xrightarrow{l=t'':A} t'}{S \mid t \to S \cup \{(l, t'')\} \mid t'} \; l \notin Dom(S) \text{ AND } l \notin Var(t)$$

$$\frac{t \xrightarrow{!l:=t''} t'}{S \mid t \to S[l := t''] \mid t'} \; l \in Dom(S) \qquad \frac{t \xrightarrow{?l=t''} t'}{S \mid t \to S \mid t'} \; v{=}S(l)$$

Consistency of storage S in Σ, Γ written as $\Sigma \mid \Gamma \vdash S$:
$dom(S) = dom(\Sigma)$
$\forall l : refA \in \Sigma : \Sigma \mid \Gamma \vdash S(l) : A$

Figure 11: Top-level evaluation of storage operations and Consistency of storage

the correctness with the "old" preservation theorem 3.5. Then we prove the preservation for the new reduction rules, and that this holds in the effectfull reduction context E.

We know that if we don't have an effectful reduction with one of the new proper effects we don't have storage operations. So we can choose $\Sigma' = \Sigma$ and $S' = S$ and have $\Sigma' \mid \Gamma \vdash S'$. The old Preservation theorem gives us that $\Sigma, \Gamma \vdash t' : A$ and so we have $\Sigma' \mid \Gamma' \vdash t' : A$.

For the new reduction rules with proper effects we reprove by induction on strong type derivation. The strong type may differ from our weak type A, but we know that it is a subtype of A. So if we can show that we can weakly type t' to our new type we directly know that we can weakly type t' to A. So W.l.o.g. assume that A is the strong type of t.

- case T-APP:
  $t = t_1 t_2$ $\quad \Sigma \mid \Gamma \vdash_{str} t_1 : A' \to A$ $\quad \Sigma \mid \Gamma \vdash_{str} t_2 : B$
  $B <: A'$ $\quad$ for some types A', B.

  - case $\alpha = fix$:
    $t_1 = fix_C$ $\quad \Sigma \mid \Gamma \vdash_{str} fix_C : (TC \to TC) \to TC$ for some type C
    So $A' = TC \to TC$, $A = TC$ and $B <: TC \to TC$. By inversion of subtype relation we know that B has the form $B_1 \to B_2$ with $TC <: B_1$ and $B_2 <: TC$.
    $fix\ t \xrightarrow{fix} t(fix\ t)$ $\quad S \mid fix\ t \to S \mid t(fix\ t)$
    S, Σ and Γ remains unchanged, so $\Sigma' \mid \Gamma \vdash S'$ holds. With the strong subtype rules we can strongly type $t(fix\ t)$ to B_2, and so weakly type it to TC.
  - case α is $l = t_2 : C$:

$t_1 = new_C$ $\quad \Sigma \mid \Gamma \vdash_{str} new_C : C \to T(ref\ C)$

$new_C\ t_2 \xrightarrow{l=t_2:C} t' = [l]$ $\quad S \mid new_C\ t_2 \to S \cup \{(l, t_2)\} \mid t' = [l]$
for some fresh label variable $l \notin Dom(S)$

So $A = T(ref\ C)$,$B <: C$ and $\Sigma' = \Sigma, l : ref\ C$. We can easily see that in Σ', Γ we can type l to $ref\ C$ and so also $[l]$ to $T(ref\ C)$. Now it remains to show that $\Sigma' \mid \Gamma \vdash S'$. Since $\Sigma \mid \Gamma \vdash S$ we know that for all $k : ref\ A \in \Sigma : \Sigma \mid \Gamma \vdash S(k) : A$. By strengthening and l is an fresh variable not in $FV(S(k))$ we know that $\Sigma' \mid \Gamma \vdash S(k) : A$ with $S(k) = S'(k)$. For the new label l we know that $S'(l) = t_2$ was typable in $\Sigma \mid \Gamma$ to type $B <: C$, so by strengthening we can $\Sigma' \mid \Gamma \vdash t_2 : C$ with $t_2 = S'(l)$. Hence $\Sigma' \mid \Gamma \vdash S'$.

- case α *is* $!l := t_2$:

  $t_1 = update_C\ l$ $\quad \Sigma \mid \Gamma \vdash_{str} update_C : ref\ C \to C \to T\{\}$
  $\Sigma \mid \Gamma \vdash_{str} l : ref\ C$ $\quad$ for some type C

  $update_C\ l\ t_2 \xrightarrow{!l:=t_2} [\{\}]$ $\quad S \mid update_C\ l\ t_2 \to S[l := t_2] \mid [\{\}]$

  So $A = T\{\}$, $A' = C$, $\Sigma' = \Sigma$ and $S' = S[x := t_2]$. Obvious, $\Sigma' \mid \Gamma \vdash [\{\}] : T\{\}$. So it remains to show that $\Sigma' \mid \Gamma \vdash S'$ holds. For all labels $k \neq l$ with $k : ref\ D \in \Sigma$ we have $S'(k) = S(k)$ with $\Sigma \vdash \Gamma \vdash S(k) : D$, so we can type $\Sigma' \mid \Gamma \vdash S'(k) : D$. We know that l has type $ref\ C$ in Σ, so for $t_2 = S'(l)$ we have the typing $\Sigma' \mid \Gamma \vdash t_2 : C$.

- case α *is* $?l = v$:

  $t_1 = deref_A$ $\quad t_2 = l$, $B = ref\ A$ $\quad \Sigma \mid \Gamma \vdash_{str} deref_A : ref\ A \to A$

  $deref_A\ l \xrightarrow{?l=v} v$ $\quad S \mid deref_A\ l \to S \mid v$

  Here Σ and S remains unchanged, so $\Sigma' \mid \Gamma \vdash S'$ holds. Since l has type $ref\ A$ in Σ we have $\Sigma' \mid \Gamma \vdash S(l) = t' : A$ because of storage consistency.

- case T-LET :

  $t = let\ x \Leftarrow t_1\ in\ t_2$ $\quad \Sigma \mid \Gamma \vdash_{str} t_1 : TC$ $\quad \Sigma \mid \Gamma, x : C \vdash_{str}: A$
  $A = TA'$ $\quad$ for some C, A'

  Because we consider only effectful reduction there is only one case.

  $t_1 \xrightarrow{s} t_1'$ $\quad S \mid t_1 \to S' \mid t_1'$

  $S \mid let\ x \Leftarrow t_1\ in\ t_2 \to S' \mid let\ x \Leftarrow t_1'\ in\ t_2$

  By induction there is $\Sigma \subseteq \Sigma'$ such that $\Sigma' \mid \Gamma \vdash S'$ and $\Sigma' \mid \Gamma \vdash t_1' : TC$. Hence, by typing $\Sigma' \mid \Gamma \vdash let\ x \Leftarrow t_1'\ in\ t_2 : A$.

□

Now we continue showing Progress. Our idea for a modular proof was to define a new value set Val' for a specific calculus depending on the additional

proper reduction rules without effects and show that $Val' = Val - \{t \mid t \xrightarrow{\alpha}\}$ assuming that all terms $t \notin Val$ with $\emptyset \vdash t : A$ have a reduction also in our new store context.

Unfortunately here this way does not work directly. The reason is that now typing of a term depends on a label environment Σ that gives our labels used in store a reference type. So we have to state preservation theorem for terms typeable in environment Σ for labels, but in empty environment for normal variables.

Another problem is that even with that Σ we have more terms in the value set. Fortunately we just have one further form of values, namely a variable l used as reference label with a type of the form $ref\ A$ contained in the set Val_{APP}. So we first can argue with the auxiliary Progress theorem that we have Progress for terms t on the old value set extended with label variables without our additional reduction rules whenever $\Sigma \mid \emptyset \vdash t : A$. When we have defined the new value set Val', we show that $Val' = (Val \cup \{l \mid l \text{ a variable}\}) - \{t \mid \ S \mid t \xrightarrow{\alpha}\}$. Now we can define our new value set Val'.

$$Val' \quad = \quad x \mid \lambda x : A.m \mid [m] \mid \{l_i = m_i\ ^{i=1,\ldots,n}\} \mid c \mid update\ m$$

Now we state our Progress theorem analogous to [10].

Theorem 4.2 (Progress of Calculus with Storage and Fixpoint)**.** *Let t be a term with $\Sigma \mid \emptyset \vdash t : A$ for any label environment Σ. Either $t \in Val'$ or else for any store S with $\Sigma \mid \emptyset \vdash S$, there is some term t' and a store S' such that $S \mid t \to S' \mid t'$ or $S \mid t \xrightarrow{\alpha} S' \mid t'$.*

Proof. First we show that t is either in $Val \cup \{l\}$ or there is a term t' such that $t \to t'$ or $t \xrightarrow{\alpha} t'$.

Since $\Sigma \mid \emptyset \vdash t : A$, we have $\Sigma \vdash t : A$ and so one of the cases of the auxiliary Progress theorem must hold. The cases $t \in Val$, $t \in Val_{MON} \vee \exists t' : t \xrightarrow{\alpha} t'$ and $\exists t' : t \to t'$ fulfill the property. For the case $t \in Val_{APP}$ we have that t is only typable in Σ in case of $t = l$ where $l : ref\ A \in \Sigma$. This we can easily prove by induction and the fact that terms of the form $a\ m$,$a.l$ aren't typable if a can only have a reference type. The last case $t = let\ x \Leftarrow a\ in\ m$ also isn't typable since therefore we must be able to type a term of Val_{APP} to a monadic type.

Because here we didn't consider the new reduction rules with proper effect the only effect we can observe is $\emptyset$. Hence we can do all considered reductions also on top-level.

To complete the proof we must show that we can reduce all terms in $(Val \cup \{l\}) - Val'$. These terms we want to exclude have either the form of an additional proper reduction rule, or the form $v.l$, $v\ m$ or $let\ x \Leftarrow v\ in\ m$.

Now we show by applying Lemma 3.8 that all these terms t contain the left-hand side p_1 of some new proper reduction rule $p_1 \xrightarrow{\alpha} p_2$ as a subterm and there is a context E such that $t = E[p_1]$. Then we show depending on the proper effect that for any store S with $\Sigma \mid \emptyset \vdash S$ that there is store S' and we have the reduction $S \mid t \to S' \mid E[p_2]$.

Since $\Sigma \mid \emptyset \vdash t : A$, we also have $\Sigma \vdash t : A$. So by Lemma 3.8 we know that for all values excluded from $Val \cup \{l\}$ there is context C and a term $t'' = c\, m_1\ \ldots m_n$ with $\Sigma \vdash t'' : B$ such that $t = C[t'']$. Furthermore for the constructs $v'.l$, $v'\ m$ and $let\ x \Leftarrow v'\ in\ m$ of t, type B is nonfunctional. If we consider again our constants we easily see that t'' either must have functional or monadic type. Hence B is monadic, and so C is a context E for effectful reductions.

Since we have defined additional proper reduction rules with effects for all constants with full arguments and hence, we always have a reduction $t'' \xrightarrow{\alpha} t'''$, and hence also $t = E[t''] \xrightarrow{\alpha} E[t'''] = t'$ Let S be an arbitary store with $\Sigma \mid \emptyset \vdash S$. Now we show in a case distinction on the effects that there is a reduction for $S \mid t$.

- α is fix

  $S \mid t \to S \mid t'$ is always possible.

- α is $l = t_1 : A$

  Clearly $t'' = new_A\ t_2$. Here it is alway possible to choose a label l so that $l \notin Dom(\Sigma)$ and $l \notin FV(t)$, since we know the term t and and Σ for typing. Since by consistency of storage $Dom(S) = Dom(\Sigma)$, we can reduce $S \mid t \to S \cup \{(l, v)\} \mid t'$.

- α is $!l := t_1$

  Clearly $t'' = update\ l\ t_1$. Since t'' is typable in Σ applying inversion (parts 4,5) we get that l is typable in Σ and hence that $l \in Dom(\Sigma)$. By consistency of storage l is also in $Dom(S)$. Hence $S \mid t \to S[l := t_1 \mid t'$.

- α is $?l = t_1$

  Clearly $t'' = deref\ l$. With the same argument as last case we have $l \in Dom(S)$. We can alway choose $t''' = [S(l)]$, $S(l) = t_1$ and so we have the reduction $S \mid t \to S \mid t'$.

□

4.2 Object Calculus

Now we define the object calculus we want to translate in the monadic calculus. We start with the definitions of types, terms and values.

$$\begin{array}{rcl} A, B \in Ty & ::= & [f_i : A_i\,^{i=1\ldots n}, m_j : B_j\,^{j=1\ldots m}] \\ b, o \in Obj & ::= & x \mid x.f \mid x.f := y \mid x.m \mid [f_i = x_i\,^{i\in 1\ldots n}, m_j = \varsigma(y_j : A) b_j\,^{j=1\ldots m}] \\ & & \mid let\ x = o\ in\ b \end{array}$$

In the object calculus we distinguish like in many object-oriented programming languages between fields and methods. Methods receive the object itself as argument again, so we have a language with recursion. As one can see we have for many constructs the restriction that we must use variables instead of

This object calculus provides also subtyping with the following subtype rule:

$$\frac{\forall j \in [m] : B_j <: B'_j}{[f_i : A_i \,^{i=1\ldots n+l}, m_j : B_j \,^{j=1\ldots m+k}] <: [f_i : A_i \,^{i=1\ldots n}, m_j : B'_j \,^{j=1\ldots m}]}$$

Typing rules:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \text{ (O-VAR)} \qquad \frac{\Gamma \vdash x : [f_i : A_i \,^{i\in 1\ldots n}, m_j : B_j \,^{j\in 1\ldots m}]}{\Gamma \vdash x.f_k : A_k} \text{ (O-F-PROJ)}$$

$$\frac{\Gamma \vdash x : [f_i : A_i \,^{i\in 1\ldots n}, m_j : B_j \,^{j\in 1\ldots m}] \qquad \Gamma \vdash y : A_k}{\Gamma \vdash x.f_k := y : [f_i : A_i \,^{i\in 1\ldots n}, m_j : B_j \,^{j\in 1\ldots m}]} \text{ (O-F-UPD)}$$

$$\frac{\Gamma \vdash x : [f_i : A_i \,^{i\in 1\ldots n}, m_j : B_j \,^{j\in 1\ldots m}]}{\Gamma \vdash x.m_k : B_k} \text{ (O-M-PROJ)}$$

$$\frac{\begin{array}{c}\forall i \in [n] : \Gamma \vdash x_i : A_i \\ \forall j \in [m] : \Gamma, y_i : A \vdash b_j : B_j \qquad A = [f_i : A_i \,^{i\in 1\ldots n}, m_j : B_j \,^{j\in 1\ldots m}]\end{array}}{\Gamma \vdash [f_i = x_i \,^{i\in 1\ldots n}, m_j = \varsigma(y_i : A) b_j \,^{j\in 1\ldots m}] : A} \text{ (O-OBJ)}$$

$$\frac{\Gamma \vdash o : A' \qquad \Gamma, x : A' \vdash b : A}{\Gamma \vdash \mathit{let}\ x = o\ \mathit{in}\ b : A} \text{ (O-LET)}$$

$$\frac{\Gamma \vdash o : A \qquad A <: B}{\Gamma \vdash o : B} \text{ (O-SUB)}$$

Figure 12: Subtyping rule and typing rules for the imperative object calculus

arbitrary objects. When we talk about the operational semantics we will see that this is just a convinience since one could simulate the version with objects with let-terms, e.g. we can simulate $o.f$ with $\mathit{let}\ x = o\ \mathit{in}\ x.f$.

First we have a look at the subtyping and typing rules of the object calculus in Figure 12. As one can see in the subtype rule we allow subtyping in depth only for methods. This has the reason that we must see the fields as references that have an update ($x.f := y$) and a dereferencing ($x.f$) operation we explain when we talk about the operational semantics of the object calculus. As it is well-known, we can't allow subtyping for references without becoming unsound. The type rules are also intuitive knowing the semantics of object calculus.

Now we still have to talk about the operational semantics of this imperative object calculus which is analogous to [5]. One principle is that objects are handled as references to the real object in a store. So for any term of the form

$[f_i = x_i\,^{i\in 1\ldots n}, m_j = \varsigma(y_i : A)b_j\,^{j=1\ldots m}]$ we create an object with the fields f_i, $i \in [n]$ and the fields m_j, $j \in [m]$ in the store with corresponding initial values and return a reference to this object. Furthermore, we have operations for the fields. The operation $x.f$ just returns the value of field f of the object the variable x is bound to. The update operation $x.f := y$ replaces the field f of the object x with the object bound on y, returning again x. Methods can only be called and not be updated so that we can allow there the subtyping. When a method field m_j with value $\varsigma(y_j)b_j$ is called on object x then we evaluate the body b_j binding the variably y_i on the value x. As last construct we have a let-construct $let\ x = o\ in\ b$. Here we first evaluate o bind the value of o to the variable x and then we evaluate the body b.

4.3 Translation into the Monadic Calculus

We can give a translation of the object calculus into the monadic calculus. We start with the rule for types using the notation A^* for the translation of type A.

$$[f_i : A_i\,^{i=1\ldots n}, m_j : B_j\,^{j=1\ldots m}]^* = \{f_i : ref\ A_i^*\,^{i=1\ldots n}, m_j : T\ B_j^*\,^{j=1\ldots m}\}$$

Observe that here the base case is that of an empty object type $[]^* = \{\}$. Also notice that the translation of any object type is a record type. Furthermore we are now able to extend the translation for type envirenments:

$$\begin{array}{lcl} \emptyset^* & = & \emptyset \\ (\Gamma, x : A)^* & = & \Gamma^*, x^* : A^* \end{array}$$

Now we translate objects from the inperative object calculus into the monadic calculs. Note that all variable $z, self$ must be fresh.

$$\begin{array}{lcl} x^* & = & [x] \\ x.f^* & = & deref\ x.f \\ x.f := y^* & = & let\_ \Leftarrow update\ x.f\ y\ in\ [x] \\ x.m^* & = & x.m \\ [f_i = x_i\,^{i\in 1\ldots n}, m_j & = & let\ z_1 \Leftarrow new\ x_1\ in \\ = s(y_i : A)b_j\,^{j\in 1\ldots m}]^* & & \qquad \ldots \\ & & let\ z_n \Leftarrow new\ x_n\ in \\ & & fix(\lambda self : T\ A^*.[\{f_i = z_i\,^{i\in 1\ldots n}, \\ & & m_j = let\ y_i \Leftarrow self\ in\ b_j^*)\,^{j\in 1\ldots m}\}] \\ let\ x = o\ in\ b^* & = & let\ x \Leftarrow o^*\ in\ b^* \end{array}$$

To convince us of the correctness of the translation we must interpret a record as the reference on the object. Since method fields are constant and we cannot update them per side-effect it doesn't matter if we write them directly in the field or in a reference cell. For the other fields we assign a reference cell that

won't be changed anymore, so knowing the references for the fields we can construct the "reference" for any object directly (assigning in the translation for $[f_i = x_i\,^{i\in 1\ldots n}, m_j = \varsigma(y_i : A)b_j\,^{j=1\ldots m}]$, other constructs do not create objects). Since we don't have the possibility to hide the argument for self application of methods in typing we translate the body and give them a variable self as argument. To realize self application we use *fix* and put the record in the body of a function with self as argument. To avoid evaluation of method bodies before the method is called, we use a let construct instead of an abstraction with argument self. So method fields have a monadic type.

Since we work much with reference operation which have monadic type we must use let constructs to be able to evaluate and hence we must give the translation a monadic type. This is also the reason why we put variables in monadic constructor []. The remaining operations are intuitive.

The idea of this translation comes from [8], where a similar translation of imperative objects into a lambda calculus with state was considered, in the context of a "denotational" semantics. The use of a monadically typed target language appears to be new, but we conjecture that correctness of the translation can be established in our case, too. Here, we show only correctness with respect to typing. To prove the correctness we need also to show that translation of types keeps the subtype relation and an inversion lemma for variables (without proof) and the weakening.

Lemma 4.3 (Preservation of Subtype relation)**.** *If $A <: B$ then $A^* <: B^*$.*

Proof. Proof by induction on the derivation of subtype relation.

By subtype rule, we have $A = [f_i : A_i\,^{i=1\ldots n+l}, m_j : B'_i\,^{j=1\ldots m+k}]$ and $\forall j \in [m] : B'_j <: B_j$. Hence by induction, we have $\forall j \in [m] : B'^*_j <: B^*_j$. Since by S-Refl $\forall i \in [n] : ref\ A^*_i <: ref\ A^*_i$, we have by S-RcdDepth that $\{f_i : ref\ A^*_i\,^{i=1\ldots n}, m_j : B^*_i\,^{j=1\ldots m}\} <: \{f_i : ref\ A^*_i\,^{i=1\ldots n}, m_j : B^*_i\,^{j=1\ldots m}\}$. By S-RcdWidth, we get $\{f_i : ref\ A^*_i\,^{i=1\ldots n+l}, m_j : B'^*_i\,^{j=1\ldots m+k}\} <: \{f_i : ref\ A^*_i\,^{i=1\ldots n}, m_j : B'^*_i\,^{j=1\ldots m}\}$. With S-Trans, we get $A^* = \{f_i : ref\ A^*_i\,^{i=1\ldots n+l}, m_j : B^*_i\,^{j=1\ldots m+k}\} <: \{f_i : ref\ A^*_i\,^{i=1\ldots n}, m_j : B^*_i\,^{j=1\ldots m}\} = B^*$.

□

Lemma 4.4. *If $\Gamma \vdash x : A$, then there is a type A' with $A' <: A$ and $x : A' \in \Gamma$. Hence, we have $\Gamma^* \vdash x : A^*$.*

Lemma 4.5 (Weakening)**.** *If $x \notin FV(t), Dom(\Gamma)$ and $\Gamma \vdash t : A$, then $\Gamma, x : B \vdash t : A$.*

Proof. The proof by straightforward induction on type derivation.

□

Lemma 4.6 (Type correctness of translation)**.** *If $\Gamma \vdash o : A$ in the object calculus, then $\Gamma^* \vdash o^* : T\ A^*$ in the monadic language.*

Proof. We prove this by induction of type derivation for objects.

- case O-VAR:

  $o = x \qquad x : A \in \Gamma$

  $o^* = [x]$

  By definition of translation of Γ we can easily see that $x : A^* \in \Gamma^*$. Hence, $\Gamma^* \vdash x : A^*$ and so by T-Mon $\Gamma^* \vdash [x] : TA^*$.

- case O-F-PROJ:

  $o = x.f_k \qquad \Gamma \vdash x : [f_i : A_i\,^{i\in 1\ldots n}, m_j : B_j\,^{j\in 1\ldots m}] \qquad A = A_k$

  $o^* = deref\ x.f$

  By Lemma 4.4, $\Gamma^* \vdash x : \{f_i : ref\ A_i^*\,^{i\in 1\ldots n}, m_j : B_j^*\,^{j\in 1\ldots m}\}$. Hence $\Gamma^* \vdash x.f_k : ref\ A_k^*$ and so $\Gamma^* \vdash o^* : TA_k^*$.

- case O-F-UPD:

  $o = x.f_k := y \qquad \Gamma \vdash x : [f_i : A_i\,^{i\in 1\ldots n}, m_j : B_j\,^{j\in 1\ldots m}](= A)$
  $\Gamma \vdash y : A_k$

  $o^* = let\ z \Leftarrow update_{A_k^*}\ x.f_k\ y\ in\ [x] \qquad z$ fresh

  By Lemma 4.4, $\Gamma^* \vdash x : A^*$ and $\Gamma^* \vdash y : A_k^*$. So we can type $\Gamma^* \vdash update_{A_k^*}\ x.f_k\ y : T\{\}$. By weakening and $z \notin FV(x)$, $\Gamma^*, z : \{\} \vdash x : A^*$ and by T-Mon, $\Gamma^*, z : \{\} \vdash [x] : TA^*$. By T-Let, $\Gamma^* \vdash o^* : TA^*$.

- case O-M-PROJ:

  $o = x.m_k \qquad \Gamma \vdash x : [f_i : A_i\,^{i\in 1\ldots n}, m_j : B_j\,^{j\in 1\ldots m}] \qquad A = B_k$

  $o^* = x.m_k$

  By Lemma 4.4, $\Gamma^* \vdash x : \{f_i : ref\ A_i^*\,^{i\in 1\ldots n}, m_j : B_j^*\,^{j\in 1\ldots m}\}$. Hence $\Gamma^* \vdash x.m_k : ref\ A_k^*$ and so $\Gamma^* \vdash o^* : TA_k^*$.

- case O-OBJ:

  $o = [f_i = x_i\,^{i\in 1\ldots n}, m_j = s(y_i : A)b_j\,^{j\in 1\ldots m}] \qquad \forall i \in [n] : \Gamma \vdash x_i : A_i$
  $\forall j \in [m] : \Gamma, y_i : A \vdash b_j : B_j \qquad A = [f_i : A_i\,^{i\in 1\ldots n}, m_j : B_j\,^{j\in 1\ldots m}]$

  $o^* = let\ z_1 \Leftarrow new_{A_1^*}\ x_1\ in \ldots let\ z_n \Leftarrow new_{A_n^*}\ x_n\ in\ fix_{A^*}(\lambda self : T\ A^*.[\{f_i = z_i\,^{i\in 1\ldots n}, m_j = let\ y_i \Leftarrow self\ in\ b_j^*)$

  By Lemma 4.4, $\forall i \in [n] : \Gamma^* \vdash x_i : A_i^*$ and so $\Gamma^* \vdash new_{A_i^*}\ x_i : T(ref\ A_i^*)$. By weakening and all z_i are fresh, we have $\forall i \in [n] : \Gamma^*, z_1 : ref\ A_1^*, \ldots z_{i-1} : A_{i-1}^* \vdash new_{A_i^*}\ x_i : T(ref\ A_i^*)$.

  By nested application of T-Let, one gets $\Gamma^* \vdash o^* : TA^*$ if $\Gamma^*, z_1 : ref\ A_1^*, \ldots z_n : A_n^* \vdash fix_{A^*}(\lambda self : T\ A^*.[\{f_i = z_i\,^{i\in 1\ldots n}, m_j = let\ y_i \Leftarrow self\ in\ b_j^*) : A$. After applications of T-App, T-Abs and T-Mon it remains to show $\Gamma^*, z_1 : ref\ A_1^*, \ldots z_n : ref\ A_n^*, self : TA^* \vdash \{f_i = z_i\,^{i\in 1\ldots n}, m_j = let\ y_i \Leftarrow self\ in\ b_j^*) : A^*$.

  By applying T-Rcd, it remains to prove that $\forall j \in [m] : \Gamma^*, z_1 : ref\ A_1^*, \ldots z_n : ref\ A_n^*, self : TA^* \vdash let\ y_i \Leftarrow self\ in\ b_j^* : TB_j^*$. And hence, one can see that this is the case if $\Gamma^*, z_1 : ref\ A_1^*, \ldots z_n : ref\ A_n^*, self : TA^*, y_i : A^* \vdash$

$b_j^* : TB_j^*$. By induction, we get $\Gamma^*, y_i : A^* \vdash b_j^* : TB_j^*$, the previous form we get by applying weakening and z_i fresh.

- case O-LET:

  $o = let\ x = o'\ in\ b \qquad \Gamma \vdash o' : B \qquad \Gamma, x : B \vdash b : A$

  $o^* = let\ x \Leftarrow o'^*\ in\ b^*$

  By induction, $\Gamma^* \vdash o'^* : B^*$ and $\Gamma^*, x : B^* \vdash b^* : A^*$. By T-Let, $\Gamma^* \vdash o^* : TA^*$.

- case O-SUB:

  $\Gamma \vdash o : B \qquad B <: A$

  By induction, $\Gamma^* \vdash o^* : TB^*$. By Lemma 4.3 $B^* <: A^*$ and hence by S-Mon, $TB^* <: TA^*$ and by T-Sub, $\Gamma^* \vdash o^* : TA^*$.

□

5 Future Work

So far, we have seen a monadic calculus with subtyping that can easily be extended. For this calculus we have proven the properties Type Preservation, Progress and the minimal ("strong") type property. In the example of embedding a object calculus into the monadic calculus in Section 4, we have seen that for some extensions we can reprove the properties in a modular way using the correctness of these properties for the unextended calculus. But the work of combing computational monads and subtyping isn't complete yet.

One task left is to investigate the calculus without the restriction not to define new subtype rules, e.g. it is not yet possible to establish a subtype relation between integers and floating points as discribed in the Introduction. For other type constructors as ref from the example in Section 4 it is often reasonable to establish a subtype rule as T-Mon for the monadic type constructor T. A simple example for a type constructor with a covariant subtype rule is $List$. As one cannot change a list element per side effect in the standard semantics we don't need contravariance as for references.

The example of embedding an object calculus into the monadic calculus is not complete since we only proved the type correctness of the translation. Therefore we have to define a particular operational semantic for the object calculus. Since both semantic, the one for the object calculus and the one for the extended monadic calculus with references, depend on a global store, we also need to extend the translation for the state of the store. Then, we must show that whenever we can evaluate a particular object o in the object calculus we can do an equivalent evaluation for the translation of o in the extended monadic calculus.

References and fixpoint operator aren't the only possible extension for the monadic calculus. Many of them had already been investigated for monadic calculi without subtyping. There is much literature about such extensions as exceptions ([7]) or non-determinism, for instance, by Moggie and Wadler in [6] and [13]. These examples must be considered again for the new monadic calculus.

As a last task left, there is extension of the calculus with type polymorhism. This can't be embedded by defining the sets in the definition of calculus. Some theory about type polymorphism for non-monadic calculi can be found in Part 5 of [10]. So here we must extend the language of the calculus and reprove the basic properties.

References

[1] M. Abadi and L. Cardelli. An imperative object calculus. *Theory and Practice of Object Systems*, 1(3):151–166, 1995.

[2] M. Abadi and L. Cardelli. *A theory of objects*. Springer-Verlag, New York, 1996.

[3] M. Abadi and L. Cardelli. A theory of primitive objects. In *International Conference on Functional Programming*, pages 63–74, Baltimore, Sept. 1998. ACM.

[4] M. Abadi, L. Cardelli, and R. Viswanathan. An interpretation of objects and object types. In *POPL*, pages 396–409, 1996.

[5] M. Abadi and K. R. M. Leino. A logic of object-oriented programs. Technical Report 161, Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, 1998. Order from src-report@src.dec.com.

[6] N. Benton, J. Hughes, and E. Moggi. Monads and effects. In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *APPSEM*, volume 2395 of *Lecture Notes in Computer Science*, pages 42–122. Springer, 2000.

[7] N. Benton and A. Kennedy. Exceptional syntax. *J. Funct. Program*, 11(4):395–410, 2001.

[8] S. N. Kamin and U. S. Reddy. Two semantic models of object-oriented languages. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 464–495. MIT Press, 1994.

[9] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[10] B. C. Pierce. *Types and Programming languages*. MIT Press, 2002.

[11] P. Wadler. The essence of functional programming. In *POPL*, pages 1–14, 1992.

[12] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer-Verlag, 1995.

[13] P. Wadler. The marriage of effects and monads. In *International Conference on Functional Programming*, pages 63–74, Baltimore, Sept. 1998. ACM.

[14] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput*, 115(1):38–94, Nov. 1994.