Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 256 (2009) 67–85
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Selective Memoization with Box Types

Favio Ezequiel Miranda-Perea[1]
Lourdes Del Carmen González-Huesca[2]

Departamento de Matemáticas
Facultad de Ciencias UNAM
Circuito Exterior s/n, Cd. Universitaria,
México D.F. 04510, México

Abstract

Memoization is a useful technique to eliminate computational redundancy. A memo function remembers all the arguments to which it has been applied, together with their corresponding results, by storing them in a table. This table is consulted before each functional call to determine if the particular argument is in it. If so, the call is skipped and the stored result is returned; otherwise the call is performed and its result added to the table. Acar, Belloch and Harper present a framework to apply memoization selectively, that is, enabling the programmer to determine precisely the dependences between the input and the result of a function. This framework is efficient and yields programs whose performance can be analyzed using standard techniques. The language, implemented as an SML library, is based on a modal type system which allows the programmer to reveal the true data input/output dependences in a program. However, the modality seems to be an ad-hoc choice for the implementation. In this paper we develop selective memoization, using instead box types, corresponding to the necessitation modality $\Box$. We also include non-memoized functions, and provide full proofs of type safeness and soundness of the dynamic semantics with respect to an effect-free system which is later translated into the very well-known language PCF .

Keywords: selective memoization, modal types, box types, adaptive computation, type safeness, functional programming.

1 Introduction

Memoization is a well known technique to avoid repeated computation which has been around a half century [6]. It refers to the tabulation of the results of a computation to elude their repeated calculation and has been extensively used in several areas such as dynamic programming [3], incremental computation [12] and others [10]. To be adequate for implementation a memoization framework must provide control over equality tests, space management as well as a precise identification of dependences between the input and the output of memoized code. The *selective*

[1] Email: favio@matematicas.unam.mx

[2] Email: l_gonzalez@uxmcc2.iimas.unam.mx

doi:10.1016/j.entcs.2009.11.006

memoization framework presented in [2] provides control over equality and identification of dependences, and some control over space management. When detecting dependencies, it is essential not to omit any, or the function caching will be unsound. However, it is also important not to introduce too broad dependencies, or the memoization will be ineffective. For the technique to be most effective, each function call's dependencies must be recorded as precisely as possible. As an example consider the following simple function: `fun f(x,y,z) = if x > 0 then y else z`. The choice point (`x>0`) causes the arguments on which `f` depends, to change in a dynamic way. For instance in the call to `f(1,2,3)` the result depends only on `x` and `y`; the value of `z` being irrelevant. Moreover observe that the evaluation of `f(x,y,z)` does not depend on the exact value of the first argument x, since it suffices to know the sign of x; therefore, a later invocation to say `f(3,2,35)` in which the result is identical to the one in the previous call, for the argument y is identical, should yield a table lookup using as key the former input `(1,2,3)`. To handle this kind of function calls efficiently, instead of using the input arguments to index the memo table, a list of choice points, called events, is used. Thus, this list, called branch of events, records the control flow of information from the input to the result of a function. This technique improves the memoization process, as showed in [2], although it could still yield recalculations as in the case of `f(-1,5,2)`, since the argument `-1` corresponds to a different event, namely `not` ($\mathtt{x} > 0$).

An incremental exploration process guided by a modal type reveals the needed dependences to build a branch. However, the modality ! employed in [2] for this purpose, seems to be an ad-hoc choice for the implementation. To build the type $!\mathsf{T}$, called a bang type, the underlying type T is required to be indexable, which means that T must admit an injective function, that maps each value of T to a unique integer. Therefore, the type $!\mathsf{T}$ makes explicit the indexable feature of the type T[3], which is needed for the implementation with hash tables, and although box types are mentioned as an important extension and even used in the implementation they were not formalized. In this paper we propose a system for selective memoization with essentially the same evaluation semantics but using a static semantics which dismisses bang types in favor of box types as defined in [11]. This type, corresponding to the necessitation modality in logic, has been used for different applications such as mobility and locality in distributed computation [9], secure information flow [8] or extensions with persistent code [13]. To our purposes $\Box\mathsf{T}$ can be thought of as a type which encapsulates certain ordinary values by means of immutable references, that is, there is a way to allocate values (box) and to deallocate them ($\mathsf{let\,box}$). However, we cannot assign a new value to the same box. Furthermore, the encapsulation of a value v by a box signals that v should be explored to reveal its control and data dependences. Particularly, in practice, memoized functions involve a box type in their domains.

As we will rely on the language MFL of [2] we give here a brief comparison between this system and our proposal: In this paper we are mainly interested in the mechanism of identification of dependences and their formal definition and behavior,

[3] $!\mathsf{T}$ could even be syntax sugar for the product type $\langle \mathsf{T}, \mathsf{T} \to \mathsf{Int}\rangle$ as implemented in [2].

by means of a type system and an evaluation semantics. Although we consider that other aspects of that framework, such as performance and equality tests needed for the implementation, are of great importance, we have decided not to treat them here. However, as our evaluation semantics is very similar to that of MFL, we expect that those aspects behave in a similar way, but of course further research is needed. For the time being, we believe that the advantage of our system over MFL is a theoretical one since, apart from the use of box types, which have a strong logical foundation, we prove here in detail its type safeness, as well as its soundness with respect to a non-memoized semantics, as opposed to the work in [1,2].

Our paper is organized as follows: after this introduction we present a system SM of selective memoization with box types in section 2. The auxiliary system S, which keeps the selectivity mechanism but without effects (storages), is developed in section 3, where we also prove that it is type safe. In section 4 the type safeness of SM is proved by means of a faithful translation to S. A translation of S to PCF is provided in section 5 proving indirectly the soundness of the original effectful evaluation semantics of SM with respect to the purely functional semantics of PCF. Finally in section 6 we provide some closing remarks and future work.

2 Selective Memoization

In this section we present a system for selective memoization based on the original system given in [2] but with the following differences: we eliminate the use of indexable and bang types in favor of box types and include non-memoized functions, case analysis and projections among the terms. With respect to static semantics we propose type judgments with an additional context for locations keeping the dynamic semantics essentially equal to the original one.

The syntax is structured in terms and expressions, in the same sense as in [11]. A term corresponds to a program which is evaluated in an ordinary way, whereas expressions are meant to be evaluated with respect to a given memo table. The type structure engages function, product and sum types as well as the necessitation modality $\Box$, deeply studied in [11] from the type-theoretical point of view.

- *Types.* Types are built from a set of basic types B including the unit type Unit and the integers Int

$$\mathsf{T} ::= \mathsf{B} \mid \mathsf{T} \to \mathsf{T} \mid \mathsf{T} + \mathsf{T} \mid \mathsf{T} \times \mathsf{T} \mid \Box\mathsf{T}$$

- *Terms.* Built from an infinite set of term variables x, an infinite set of resource variables a and primitive operators o, including numbers n

$$t, r, s ::= x \mid a \mid o(t, \ldots, t) \mid \star \mid n \mid \lambda x : \mathsf{T}.r \mid \mathsf{mfun}_\ell(f.a.e) \mid rs \mid \mathsf{inl}_\mathsf{T}\, r \mid \mathsf{inr}_\mathsf{T}\, s \mid$$
$$\mathsf{case}(r, x.s, y.t) \mid \langle r, s\rangle \mid \mathsf{fst}\, r \mid \mathsf{snd}\, r \mid \mathsf{box}\, t$$

where $\mathsf{box}\, t$ is the constructor of box typed terms; and $\mathsf{mfun}_\ell(f.a.e)$ defines a memoized, usually recursive, function with name f, in this case the metavariable ℓ belongs to a set $\mathcal{L}$ of label locations disjoint from ordinary variables and resources. The dot notation on abstractions, case analysis and function declarations denotes binding: in every expression of the form $x.t$ the occurrences of the term variable x in t are considered bound. This binding mechanism using the dot avoids the use of parentheses, the dot signals an opening parentheses which closes as far to the right as syntactically possible. This same convention as well as the usual α-equivalence also apply to expressions below.

- *Expressions.* Built from terms

$$e ::= \mathsf{return}\ \ t \,|\, \mathsf{let\,box}\,(t,\, x.e) \,|\, \mathsf{letprod}\,(t,\, a_1.a_2.e) \,|\, \mathsf{mcase}\,(t,\, a_1.e_1,\, a_2.e_2)$$

  Expressions provide a binding mechanism for either data dependences (ordinary variables of modal type), or control dependences (resource variables). The idea is that an expression, as opposed to a term, will be evaluated with respect to a memo table. This will be made clear later when defining the semantics. Observe that every term can be considered as an expression due to the return constructor.

- *Contexts.* There are three kinds of contexts, which are finite sets of pairs. Variable contexts Γ; resource contexts Δ and location contexts Σ defined by:

$$\Gamma \ ::= \ \cdot \,|\, \Gamma, x : \mathsf{T} \qquad \Delta \ ::= \ \cdot \,|\, \Delta, a :: \mathsf{T} \qquad \Sigma \ ::= \ \cdot \,|\, \Sigma, \ell : \mathsf{T}$$

  where $\cdot$ denotes the empty set.

  Variable contexts correspond to the validity context of [11] whereas resource context correspond to truth contexts. We use a third context for labels to keep exact track of the labels occurring in a term or expression. With aid of this context we will statically ensure that location labels of different functions in a same program will be different.

2.1 Type System

The static semantics is given by a type system which derives judgments of the form $\Gamma \,|\, \Delta \,|\, \Sigma \vdash r : \mathsf{T}$ denoting that the term or expression r is well-typed in contexts Γ, Δ and Σ. The rules for deriving these judgments are defined as follows:

$$\frac{}{\Gamma, x : \mathsf{T} \,|\, \Delta \,|\, \Sigma \vdash x : \mathsf{T}}\ (\mathsf{Tvar}) \qquad \frac{}{\Gamma \,|\, \Delta, a :: \mathsf{T} \,|\, \Sigma \vdash a : \mathsf{T}}\ (\mathsf{Tresource})$$

$$\frac{\Gamma \,|\, \Delta \,|\, \Sigma \vdash t_i : \mathsf{T}_i \ \ (1 \le i \le n) \qquad \vdash o \ : \mathsf{T}_1 \times \cdots \times \mathsf{T}_n \to \mathsf{T}}{\Gamma \,|\, \Delta \,|\, \Sigma \vdash o\,(t_1, \ldots, t_n) : \mathsf{T}}\ (\mathsf{Tbasicop})$$

$$\frac{}{\Gamma \,|\, \Delta \,|\, \Sigma \vdash \star : \mathsf{Unit}}\ (\mathsf{Tunit}) \qquad \frac{}{\Gamma \,|\, \Delta \,|\, \Sigma \vdash n : \mathsf{Int}}\ (\mathsf{Tnum})$$

$$\frac{\Gamma,\, x : \mathsf{T}_1 \,|\, \Delta \,|\, \Sigma \vdash t : \mathsf{T}_2}{\Gamma \,|\, \Delta \,|\, \Sigma \vdash \lambda x : \mathsf{T}_1 .t : \mathsf{T}_1 \to \mathsf{T}_2}\ (\mathsf{Tlam}) \qquad \frac{\Gamma,\, f : \mathsf{T}_1 \to \mathsf{T}_2 \,|\, \Delta,\, a :: \mathsf{T}_1 \,|\, \Sigma \vdash e : \mathsf{T}_2}{\Gamma \,|\, \Delta \,|\, \Sigma, \ell : \mathsf{T}_2 \vdash \mathsf{mfun}_\ell\,(f.a.e) : \mathsf{T}_1 \to \mathsf{T}_2}\ (\mathsf{Tmfun})$$

$$\frac{\Gamma \,|\, \Delta \,|\, \Sigma \vdash t_1 \ : \mathsf{T}_1 \to \mathsf{T}_2 \qquad \Gamma \,|\, \Delta \,|\, \Sigma \vdash t_2 \ : \mathsf{T}_1}{\Gamma \,|\, \Delta \,|\, \Sigma \vdash t_1\, t_2 \ : \mathsf{T}_2}\ (\mathsf{Tapply})$$

$$\frac{\Gamma \,|\, \Delta \,|\, \Sigma \vdash t \ : \mathsf{T}_1}{\Gamma \,|\, \Delta \,|\, \Sigma \vdash \mathsf{inl}_{\mathsf{T}_2}\, t \ : \mathsf{T}_1 + \mathsf{T}_2}\ (\mathsf{Tinl}) \qquad \frac{\Gamma \,|\, \Delta \,|\, \Sigma \vdash t \ : \mathsf{T}_2}{\Gamma \,|\, \Delta \,|\, \Sigma \vdash \mathsf{inr}_{\mathsf{T}_1}\, t \ : \mathsf{T}_1 + \mathsf{T}_2}\ (\mathsf{Tinr})$$

$$\frac{\begin{array}{c}\Gamma|\,\Delta|\,\Sigma \vdash t : \mathsf{T}_1 + \mathsf{T}_2 \\ \Gamma,\, x_1 : \mathsf{T}_1|\,\Delta|\,\Sigma \vdash t_1 : \mathsf{T} \\ \Gamma,\, x_2 : \mathsf{T}_2|\,\Delta|\,\Sigma \vdash t_2 : \mathsf{T}\end{array}}{\Gamma|\,\Delta|\,\Sigma \vdash \mathsf{case}\,(t,\, x_1.t_1,\, x_2.t_2) : \mathsf{T}}\ (\mathsf{Tcase})$$

$$\frac{\Gamma\,|\,\Delta\,|\,\Sigma \vdash t_1\ : \mathsf{T}_1 \qquad \Gamma\,|\,\Delta\,|\,\Sigma \vdash t_2\ : \mathsf{T}_2}{\Gamma\,|\,\Delta\,|\,\Sigma \vdash \langle t_1, t_2\rangle\ : \mathsf{T}_1 \times \mathsf{T}_2}\ (\mathsf{Tpair})$$

$$\frac{\Gamma\,|\,\Delta\,|\,\Sigma \vdash t\ : \mathsf{T}_1 \times \mathsf{T}_2}{\Gamma\,|\,\Delta\,|\,\Sigma \vdash \mathsf{fst}\,t\ : \mathsf{T}_1}\ (\mathsf{Tfst}) \qquad \frac{\Gamma\,|\,\Delta\,|\,\Sigma \vdash t\ : \mathsf{T}_1 \times \mathsf{T}_2}{\Gamma\,|\,\Delta\,|\,\Sigma \vdash \mathsf{snd}\,t\ : \mathsf{T}_2}\ (\mathsf{Tsnd})$$

$$\frac{\Gamma\,|\,\cdot\,|\,\Sigma \vdash t : \mathsf{T}}{\Gamma\,|\,\Delta\,|\,\Sigma \vdash \mathsf{box}\,t : \Box\,\mathsf{T}}\ (\mathsf{Tbox})$$

Observe that in rule (Tbasicop) the typing of the primitive operator o is supposed to be given beforehand. Furthermore, in the rule (Tmfun) we expand the label context, guaranteeing thus, that every new memoized function term is associated to a unique label. In the case of rule (Tbox) the requirement of an empty resource context in the premise, ensures that the term t does not have free resources before its encapsulation in box t this is necessary for an adequate operational behavior of boxed expressions. A logical motivation for this rule can be found in [11].

Expressions are typed as follows:

$$\frac{\Gamma|\,\cdot\,|\,\Sigma \vdash t : \mathsf{T}}{\Gamma|\,\Delta|\,\Sigma \vdash \mathsf{return}\ t\ : \mathsf{T}}\ (\mathsf{Treturn}) \qquad \frac{\Gamma|\,\Delta|\,\Sigma \vdash t : \Box\,\mathsf{T}_1 \qquad \Gamma, x : \mathsf{T}_1|\,\Delta|\,\Sigma \vdash e : \mathsf{T}_2}{\Gamma|\,\Delta|\,\Sigma \vdash \mathsf{let\,box}(t,\, x.e) : \mathsf{T}_2}\ (\mathsf{Tletbox})$$

$$\frac{\Gamma|\,\Delta|\,\Sigma \vdash t : \mathsf{T}_1 \times \mathsf{T}_2 \qquad \Gamma|\,\Delta, a_1 :: \mathsf{T}_1,\ a_2 :: \mathsf{T}_2|\,\Sigma \vdash e : \mathsf{T}}{\Gamma|\,\Delta|\,\Sigma \vdash \mathsf{letprod}\,(t,\, a_1.a_2.e) : \mathsf{T}}\ (\mathsf{Tletprod})$$

$$\frac{\begin{array}{c}\Gamma|\,\Delta|\,\Sigma \vdash t : \mathsf{T}_1 + \mathsf{T}_2 \\ \Gamma|\,\Delta,\, a_1 :: \mathsf{T}_1|\,\Sigma \vdash e_1 : \mathsf{T} \\ \Gamma|\,\Delta,\, a_2 :: \mathsf{T}_2|\,\Sigma \vdash e_2 : \mathsf{T}\end{array}}{\Gamma|\,\Delta|\,\Sigma \vdash \mathsf{mcase}\,(t,\, a_1.e_1,\, a_2.e_2) : \mathsf{T}}\ (\mathsf{Tmcase})$$

The rule (Treturn) provides an inclusion of terms in expressions, it is a special purpose rule for memoization with no analogue in modal logic. Its premise requires not to have free-resources, this will ensure that any dependence on a memoized function argument is made explicit in program code before introducing a return statement. On the other hand, the rule (Tletbox) is the elimination rule for box types. The expression constructor let box binds a boxed value to an ordinary variable, which may be used without restriction. Again a logical motivation for this rule can be found on [11]. To bind further resources with other ordinary terms we use the constructors letprod and mcase. As an example, consider the following definition of the Fibonacci function $\mathsf{mfib} : \Box\,\mathsf{Int} -> \mathsf{Int}$

```
mfun_ℓ mfib (a::□Int) = letbox x = a in
                          return (if x<2 then x
                                      else mfib (box (x-1)) +
                                           mfib (box (x-2))).
```

The let box constructor in this definition causes the underlying value of parameter a, which is a resource, to be exposed before performing the two recursive calls. This, instead of generating an immediate evaluation, will produce a lookup in a memo table, as formalized in the following subsection. Moreover, the domain is given a box type to signal the fact that an actual argument to the function should be explored, since the final result depends completely on this value.

2.2 Dynamic Semantics

The evaluation of a term or expression is given by two mutually defined big-step semantics, which involve stores of memo tables indexed by branches that trace choice points in the evaluation process. For terms, the semantics $\Downarrow^t$ is quite standard. This semantics interacts with the one used for expressions in the case of an application of a memoized function. The semantics for expressions $\Downarrow^e_{\beta @ \ell}$ performs an evaluation according to a given branch β which will be used to query the memo table stored at a given location ℓ. We define now these concepts in detail.

The set of values is a subset of terms defined as follows:

$$v ::= \star \mid n \mid \lambda x : \mathsf{T}.t \mid \mathsf{mfun}_\ell(f.a.e) \mid \mathsf{inl}\, v \mid \mathsf{inr}\, v \mid \langle v, v\rangle \mid \mathsf{box}\, v$$

We have carefully defined a set of values which consists of terms including neither variables nor resources; as opposed to the set of values defined in [2].

Definition 2.1 A branch β is a list of events ε. An event signals a choice point in the evaluation of an expression. Such a point arises either by a case analysis or by a boxed value. The definitions are:

$$\beta ::= \bullet \mid \varepsilon \cdot \beta \qquad\qquad \varepsilon ::= !v \mid \mathsf{inl} \mid \mathsf{inr}$$

Definition 2.2 A memo table θ is a partial function $\theta : \mathcal{B} \to \mathcal{V}_\mathsf{T}$ mapping branches to values of a given type T. We write $\theta[\beta \mapsto v]$ for the extension of θ that binds β with v, always assuming that $\beta \notin dom(\theta)$.

Definition 2.3 A store μ is a partial function with finite domain $\mu : \mathcal{L} \to \mathcal{T}$ mapping location labels ℓ to memo tables θ. A store is *initial* if and only if it contains only empty memo tables, that is, $\forall \ell \in dom(\mu)(\mu(\ell) = \varnothing)$.
We write $\mu[\ell \mapsto \theta]$ for the extension of μ that binds ℓ with θ, always assuming that $\ell \notin dom(\mu)$. Moreover, when $\ell \in dom(\mu)$, we write $\mu[\ell \leftarrow \theta]$ for the update of store μ which binds ℓ to θ.

The evaluation semantics for terms is given by a relation $\mu, t \Downarrow^t v, \mu'$ modeling the evaluation process of term t with respect to store μ resulting in a final value v and store μ', defined by the following inference rules: [4]

[4] For the sake of a direct comparison observe that this semantics corresponds to $\sigma, t \Downarrow^t v, \sigma'$ in [2].

$$\frac{\mu,\ t_1 \Downarrow^t v_1,\ \mu_1 \quad \ldots \quad \mu_{n-1},\ t_n \Downarrow^t v_n,\ \mu_n \quad \mu_n,\ o(v_1,\ldots,v_n) \Downarrow^t v, \mu'}{\mu,\ o\ (t_1,\ldots,t_n) \Downarrow^t v,\ \mu'}\ (\mathsf{Ebasicop})$$

$$\frac{}{\mu,\ \star \Downarrow^t\ \star,\ \mu}\ (\mathsf{Eunit}) \qquad \frac{}{\mu,\ n \Downarrow^t\ n,\ \mu}\ (\mathsf{Enum})$$

$$\frac{}{\mu,\ \lambda x : \mathsf{T}.t \Downarrow^t\ \lambda x : \mathsf{T}.t,\ \mu}\ (\mathsf{Elam})$$

$$\frac{\ell \notin dom(\mu)}{\mu,\ \mathsf{mfun}_\ell\,(f.a.e) \Downarrow^t\ \mathsf{mfun}_\ell\,(f.a.e),\ \mu[\ell \mapsto \varnothing]}\ (\mathsf{Emfun}) \qquad \frac{\ell \in dom(\mu)}{\mu,\ \mathsf{mfun}_\ell\,(f.a.e) \Downarrow^t\ \mathsf{mfun}_\ell(f.a.e),\ \mu}\ (\mathsf{Emfun-in})$$

$$\frac{\begin{array}{c}\mu,\ t_1 \Downarrow^t \lambda x : \mathsf{T}.t,\ \mu_1 \\ \mu_1,\ t_2 \Downarrow^t v',\ \mu_2 \\ \mu_2,\ t[x := v'] \Downarrow^t v,\ \mu'\end{array}}{\mu,\ t_1 t_2 \Downarrow^t\ v,\ \mu'}\ (\mathsf{Eapply}) \qquad \frac{\begin{array}{c}\mu,\ t_1 \Downarrow^t \quad v_1 = \mathsf{mfun}_\ell\,(f.a.e),\ \mu_1 \\ \mu_1,\ t_2 \Downarrow^t \quad v_2,\ \mu_2 \\ \mu_2,\ e[f,a := v_1,v_2] \Downarrow^e_{\bullet @ \ell} v,\ \mu'\end{array}}{\mu,\ t_1 t_2 \Downarrow^t\ v,\ \mu'}\ (\mathsf{Emapply})$$

$$\frac{\mu,\ t \Downarrow^t\ v,\ \mu'}{\mu,\ \mathsf{inl}\,t \Downarrow^t\ \mathsf{inl}\,v,\ \mu'}\ (\mathsf{Einl}) \qquad \frac{\mu,\ t \Downarrow^t\ v,\ \mu'}{\mu,\ \mathsf{inr}\,t \Downarrow^t\ \mathsf{inr}\,v,\ \mu'}\ (\mathsf{Einr})$$

$$\frac{\begin{array}{c}\mu,\ t \Downarrow^t \mathsf{inl}\,v,\ \mu_1 \\ \mu_1, t_1[x_1 := v] \Downarrow^t v_1,\ \mu'\end{array}}{\mu,\ \mathsf{case}\,(t,\ x_1.t_1,\ x_2.t_2) \Downarrow^t\ v_1,\ \mu'}\ (\mathsf{Ecase-l}) \qquad \frac{\begin{array}{c}\mu,\ t \Downarrow^t \mathsf{inr}\,v,\ \mu_1 \\ \mu_1, t_2[x_2 := v] \Downarrow^t v_2,\ \mu'\end{array}}{\mu,\ \mathsf{case}\,(t,\ x_1.t_1,\ x_2.t_2) \Downarrow^t\ v_2,\ \mu'}\ (\mathsf{Ecase-r})$$

$$\frac{\begin{array}{c}\mu,\ t_1 \Downarrow^t v_1,\ \mu_1 \\ \mu_1,\ t_2 \Downarrow^t v_2,\ \mu'\end{array}}{\mu,\ \langle t_1,t_2\rangle \Downarrow^t\ \langle v_1,v_2\rangle,\ \mu'}\ (\mathsf{Epair})$$

$$\frac{\mu,\ t \Downarrow^t\ \langle v_1,\, v_2\rangle,\ \mu'}{\mu,\ \mathsf{fst}\,t \Downarrow^t\ v_1,\ \mu'}\ (\mathsf{Efst}) \qquad \frac{\mu,\ t \Downarrow^t\ \langle v_1,\, v_2\rangle,\ \mu'}{\mu,\ \mathsf{snd}\,t \Downarrow^t\ v_2,\ \mu'}\ (\mathsf{Esnd})$$

$$\frac{\mu,\ t \Downarrow^t\ v,\ \mu'}{\mu,\ \mathsf{box}\,t \Downarrow^t\ \mathsf{box}\,v,\ \mu'}\ (\mathsf{Ebox})$$

The substitution operations $r[x := s]$ and $r[a := s]$ where r, s can be terms or expressions –appearing in some rules above– are defined in a standard fashion, including an implicit renaming of bound variables. Observe that a memoized function is a value, but its evaluation can yield the creation of a new empty memo table, in case a memo table for this function does not already exists. Moreover, the application of such a function launches an expression evaluation starting from the empty branch, which means that an exploration of dependences begins.

The corresponding semantics for expressions is given by a relation $\mu, e \Downarrow^e_{\beta @ \ell} v, \mu'$, meaning that the evaluation of expression e with respect to store μ results in a value v and store μ'; all according to the query of branch β at the memo table stored at location ℓ in μ.

The deriving rules for this semantics are [5] :

[5] Again, for a direct comparison our relation $\mu, e \Downarrow^e_{\beta @ \ell} v, \mu'$ corresponds to $\sigma, l : \beta, e \Downarrow^e v, \sigma'$ in [2].

$$\frac{\mu(\ell)(\beta) = v}{\mu,\ \mathsf{return}\ t \Downarrow^e_{\beta @ \ell}\ v,\ \mu}\ (\mathsf{Efound}) \qquad \frac{\mu(\ell) = \theta \quad \beta \notin dom(\theta) \quad \mu,\ t \Downarrow^t\ v,\ \mu' \quad \mu'(\ell) = \theta'}{\mu,\ \mathsf{return}\ t \Downarrow^e_{\beta @ \ell}\ v,\ \mu'[\ell \leftarrow \theta'[\beta \mapsto v]]}\ (\mathsf{Enotfound})$$

$$\frac{\mu,\ t \Downarrow^t\ \mathsf{box}\, v,\ \mu_1 \quad \mu_1,\ e[x := v] \Downarrow^e_{!v\cdot\beta @ \ell}\ v',\ \mu'}{\mu, \mathsf{let\,box}\,(t,\ x.e) \Downarrow^e_{\beta @ \ell}\ v',\ \mu'}\ (\mathsf{Eletbox}) \qquad \frac{\mu,\ t \Downarrow^t \langle v_1,\ v_2\rangle,\ \mu_1 \quad \mu_1,\ e[a_1, a_2 := v_1,\ v_2] \Downarrow^e_{\beta @ \ell} v,\ \mu'}{\mu,\ \mathsf{letprod}\,(t,\ a_1.a_2.e) \Downarrow^e_{\beta @ \ell}\ v,\ \mu'}\ (\mathsf{Eletprod})$$

$$\frac{\mu,\ t \Downarrow^t\ \mathsf{inl}\, v,\ \mu_1 \quad \mu_1, e_1[a_1 := v] \Downarrow^e_{\mathsf{inl}\cdot\beta @ \ell} v_1,\ \mu'}{\mu,\ \mathsf{mcase}\,(t,\ a_1.e_1,\ a_2.e_2) \Downarrow^e_{\beta @ \ell}, v_1,\ \mu'}\ (\mathsf{Emcase-l}) \qquad \frac{\mu,\ t \Downarrow^t\ \mathsf{inr}\, v,\ \mu_1 \quad \mu_1, e_2[a_2 := v] \Downarrow^e_{\mathsf{inr}\cdot\beta @ \ell} v_2,\ \mu'}{\mu,\ \mathsf{mcase}\,(t,\ a_1.e_1,\ a_2.e_2) \Downarrow^e_{\beta @ \ell}, v_2,\ \mu'}\ (\mathsf{Emcase-r})$$

There is a subtlety in the definition of rule (Enotfound). Observe that for $\theta'[\beta \mapsto v]$ to be defined we must have $\beta \notin dom(\theta')$, but the rule only requires $\beta \notin dom(\theta)$. We will justify the soundness of this rule definition later in lemma 4.5.

The main goal of this paper is to prove the type safeness of the above system, stated next.

Theorem 2.4

(i) *If* $\Gamma|\cdot|\Sigma \vdash t : \mathsf{T}$ *and* $\mu, t \Downarrow^t v, \mu'$ *with* μ *initial, then there exists* Σ' *such that* $\Gamma|\cdot|\Sigma' \vdash v : \mathsf{T}$.

(ii) *If* $\Gamma|\cdot|\Sigma \vdash e : \mathsf{T}$ *and the evaluation* $\mu, e \Downarrow^e_{\beta @ \ell} v, \mu'$ *originates in an application*[6] $\mu^\star, t_1 t_2 \Downarrow^t v^\star, \mu^{\star\prime}$ *with* $\mu^\star$ *initial, then there exists* Σ' *such that* $\Gamma|\cdot|\Sigma' \vdash v : \mathsf{T}$.

It is worth noting that part (ii) of this theorem does not have a counterpart in [1], which is an important omision since it is needed for the proof of part (i).

To prove this theorem we will translate SM to a system S defined in the following section. We will prove that S is type safe and that the translation is faithful getting as a corollary a proof of theorem 2.4. For part (i) this idea is sketched without details in [1]. Here we provide a full proof.

3 A selective system without effects

To prove the safeness for the system SM we define an auxiliary system S which keeps the selective feature but avoids memoization. This system is quite similar to the original one, with some differences mainly in the dynamic semantics for expressions. With this, we provide a detailed alternative to the missing definitions and properties

[6] This means that $\mu, e \Downarrow^e_{\beta @ \ell} v\, \mu'$ arises as part of a derivation of the premise involving $\Downarrow^e_{\bullet @ \ell'}$ in an instance of the rule (Emapply).

of the relations $\Downarrow^{\mathsf{t}}_{\mathsf{p}}, \Downarrow^{\mathsf{e}}_{\mathsf{p}}$ of [1,2], which are critical to the proof of soundness for their system MFL.

- *Types.* The same types used for SM
- *Terms.* The same terms as for SM except for the memoized functions declaration where now there is no label associated with a function. That is, we use terms of the form $\mathsf{mfun}(f.a.e)$ instead of $\mathsf{mfun}_\ell(f.a.e)$. This class of term is called named function.
- *Expressions.* The same as for SM
- *Contexts.* We keep the contexts for ordinary variables Γ and for resource variables Δ.
- *Type system.* The same rules as for SM, adapted accordingly to the terms of S, eliminating all the label contexts Σ. For instance the typing rule for named functions is:

$$\frac{\Gamma, f : \mathsf{T}_1 \to \mathsf{T}_2 \,|\, \Delta, a :: \mathsf{T}_1 \vdash e : \mathsf{T}_2}{\Gamma \,|\, \Delta \vdash \mathsf{mfun}(f.a.e) : \mathsf{T}_1 \to \mathsf{T}_2}$$

This inference system satisfies the usual structural properties of monotonicity, exchange and contraction. Moreover, since the inferences are syntax-directed it also satisfies typing inversion. These properties can be shown routinely by structural induction.

3.1 Dynamic semantics

The evaluation relation for terms is completely analogous to the one for SM, we simply eliminate the stores and modify the class of values accordingly. For expressions, the evaluation relation denoted $e \Downarrow^{e}_{\beta;v_f} v$ depends now on a branch β and a functional value v_f of the form $\mathsf{mfun}(f.a.e)$. For the cases of $\mathsf{let\,box}$, $\mathsf{letprod}$ or mcase statements the evaluation rules are obtained from the ones for $\Downarrow^{e}_{\beta @\ell}$ replacing $\beta@\ell$ for $\beta; v_f$ accordingly. For the case of a return statement we need a partial access function $v_f@\beta$, defined below, which will compute a return statement obtained by exploring the given branch β.

Definition 3.1 Given a term or expression r and a branch β we define the partial access function $r@\beta$ as follows:

$$\begin{array}{rclcrcl}
\mathsf{mfun}(f.a.e) \;@\; \beta & = & e \;@\; \beta & \quad & \mathsf{letprod}(t, a_1.a_2.e) \;@\; \beta & = & e \;@\; \beta \\
\mathsf{return}\, t \;@\; \bullet & = & \mathsf{return}\, t & & \mathsf{mcase}(t, a_1.e_1, a_2.e_2) \;@\; \beta\,\widehat{}\,\mathsf{inl} & = & e_1 \;@\; \beta \\
\mathsf{let\,box}(t, x.e) \;@\; \beta\,\widehat{}\,!v & = & e[x := v] \;@\; \beta & & \mathsf{mcase}(t, a_1.e_1, a_2.e_2) \;@\; \beta\,\widehat{}\,\mathsf{inr} & = & e_2 \;@\; \beta
\end{array}$$

where the notation $\beta\widehat{}\varepsilon$ makes explicit the last event ε of a branch whose previous elements are those of β.

The evaluation of a return statement can now be defined as

$$\frac{v_f@\beta = \mathsf{return}\ t \qquad t \Downarrow_p^t v}{\mathsf{return}\ t \Downarrow_{\beta;v_f}^e\ v}\ (\mathsf{Ereturn})$$

This rule causes the evaluation of the argument t of the return statement to always take place. The side condition $v_f@\beta = \mathsf{return}\, t$ makes sure that the branch β really corresponds to the exploration of the functional value v_f which we are using to evaluate.

The dynamic semantics for terms $t \Downarrow_p^t v$ is analogous to the one for SM, the main difference arises in the rule for application of named functions which is:

$$\frac{t_1 \ \Downarrow_p^t \ v_1 = \mathsf{mfun}(f.a.e) \qquad t_2 \ \Downarrow_p^t \ v_2 \qquad e[f, a := v_1,\, v_2] \Downarrow_{\bullet\,;v_1}^e\ v}{t_1\, t_2 \Downarrow_p^t v}\ (\mathsf{Emapply})$$

3.2 *Type safeness for* S

It is conventional to prove the type safeness for S, since this is an effect-free system. We state next a standard substitution lemma needed in the proof.

Lemma 3.2 (Substitution lemma) *The static semantics of system* S *satisfies:*

(i) *If* $\Gamma, x : \mathsf{R}\,|\Delta \vdash t : \mathsf{T}$ *and* $\Gamma|\cdot \vdash r : \mathsf{R}$ *then* $\Gamma|\Delta \vdash t[x := r] : \mathsf{T}$.

(ii) *If* $\Gamma, x : \mathsf{R}\,|\Delta \vdash e : \mathsf{T}$ *and* $\Gamma|\cdot \vdash r : \mathsf{R}$ *then* $\Gamma|\Delta \vdash e[x := r] : \mathsf{T}$.

(iii) *If* $\Gamma|\Delta, a :: \mathsf{R} \vdash t : \mathsf{T}$ *and* $\Gamma|\Delta \vdash r : \mathsf{R}$ *then* $\Gamma|\Delta \vdash t[a := r] : \mathsf{T}$.

(iv) *If* $\Gamma|\Delta, a :: \mathsf{R} \vdash e : \mathsf{T}$ *and* $\Gamma|\Delta \vdash r : \mathsf{R}$ *then* $\Gamma|\Delta \vdash e[a := r] : \mathsf{T}$.

Proof Straightforward induction on the first given derivation in each case. □

Now we can prove the safeness theorem.

Theorem 3.3 *System* S *is type safe, that is:*

(i) *If* $\Gamma|\cdot \vdash t : \mathsf{T}$ *and* $t \Downarrow_p^t v$ *then* $\Gamma|\cdot \vdash v : \mathsf{T}$.

(ii) *If* $\Gamma|\cdot \vdash e : \mathsf{T}$ *and* $e \Downarrow_{\beta;v_f}^e v$, *for any* β *and* v_f, *then* $\Gamma|\cdot \vdash v : \mathsf{T}$.

Proof We prove both parts simultaneously by performing induction on both evaluation relations $\Downarrow_p^t$ and $\Downarrow_{\beta;v_f}^e$.

Let us show the cases for applications of named functions and return statements.

- Assume that $t = t_1 t_2$ with $t_1 t_2 \Downarrow_p^t v$ derived from $t_1 \Downarrow_p^t v_1 = \mathsf{mfun}(f.a.e)$, $t_2 \Downarrow_p^t v_2$ and $e[f, a := v_1, v_2] \Downarrow_{\bullet\,;v_1}^e v$. As $\Gamma|\cdot \vdash t_1 t_2 : \mathsf{T}$, inversion of typing implies that $\Gamma|\cdot \vdash t_1 : \mathsf{R} \to \mathsf{T}$ and $\Gamma|\cdot \vdash t_2 : \mathsf{R}$. By I.H. of part (i) we get $\Gamma|\cdot \vdash v_1 : \mathsf{R} \to \mathsf{T}$ and $\Gamma|\cdot \vdash v_2 : \mathsf{T}$. Next observe that the typing of v_1 implies that $\Gamma, f : \mathsf{R} \to \mathsf{T}\,|a :: \mathsf{R} \vdash e : \mathsf{T}$. Therefore by the substitution lemma parts (ii) and (iv) we obtain $\Gamma|\cdot \vdash e[f, a := v_1, v_2] : \mathsf{T}$. Finally, as $e[f, a := v_1, v_2] \Downarrow_{\bullet\,;v_1}^e v$, the I.H. for part (ii) of this theorem yields $\Gamma|\cdot \vdash v : \mathsf{T}$.

- Assume $e = \mathsf{return}\, t$ with $\mathsf{return}\, t \Downarrow^{e}_{\beta;v_f} v$ derived from $v_f@\beta = \mathsf{return}\, t$ and $t \Downarrow^{t}_{p} v$. $\Gamma|\cdot \vdash \mathsf{return}\, t : \mathsf{T}$ yields, from inversion of typing, $\Gamma|\cdot \vdash t : \mathsf{T}$ which by I.H. for part (i), together with $\mathsf{return}\, t \Downarrow^{e}_{\beta;v_f} v$ leads us to $\Gamma|\cdot \vdash v : \mathsf{T}$.

□

This system will later be translated into the well-known system PCF. In this way we indirectly reduce the system of selective memoization to a pure functional system proving soundness of the original memoized semantics with respect to the pure semantics of PCF.

4 A translation from SM to S

In this section we develop a faithful translation from SM to S and use it to prove the type safeness of SM.

Definition 4.1 The translation $(\cdot)^-$ from the terms and types of SM to the terms and types of S is defined as follows:

- The translation on types is the identity function.
- The translation on terms is the forgetful map on labels. In particular we have $\mathsf{mfun}_\ell(f.a.e)^- = \mathsf{mfun}(f.a.e^-)$ and $(\mathsf{return}\, t)^- = \mathsf{return}\, t^-$

Later on, we will also apply this translation to branches β, obtaining a branch β^- by replacing each event of the form $!v$ occurring in β, by $!(v^-)$.

Important features of our translation are its compatibility with substitution and its compliance with typing derivations.

Lemma 4.2 *If r is a term or expression then $r[x := t]^- = r^-[x := t^-]$ and $r[a := t]^- = r^-[a := t^-]$.*

Proof Straightforward simultaneous induction on terms and expressions. □

Proposition 4.3 *The translation $t \mapsto t^-$ respects types.*

(i) *If $\Gamma|\Delta|\Sigma \vdash t : \mathsf{T}$ then $\Gamma|\Delta \vdash t^- : \mathsf{T}$.*

(ii) *If $\Gamma|\Delta|\Sigma \vdash e : \mathsf{T}$ then $\Gamma|\Delta \vdash e^- : \mathsf{T}$.*

Proof Straightforward simultaneous induction on typing derivations. □

To prove that our translation is faithful we need to simulate a store μ in an adequate way. To this purpose we use function tables which associate a label location with a named function value.

Definition 4.4 A function table τ is a partial function $\tau : \mathcal{L} \to \mathcal{F}$ with finite domain, mapping location labels to functional values of the form $\mathsf{mfun}(f.a.e)$.

Using function tables and access functions we can justify the subtlety in the definition of the evaluation rule (Enotfound) by means of the following

Lemma 4.5 *If $\mu, t \Downarrow^t v, \mu'$, $\mu(\ell)@\beta = \mathsf{return}\, t$ and $\mu(\ell)(\beta)$ is undefined then $\mu'(\ell)(\beta)$ is also undefined.*

Proof See [1]. □

For the evaluation simulation to succeed, we need to associate an adequate function table τ with a given store μ. The needed intrinsic relationship between a store μ in SM and a table τ in S is that every memoized function f, for which there is a memo table stored in μ at location ℓ, should have a corresponding named function f^- as image of the same location ℓ in τ. To achieve this we will use the following

Definition 4.6 Let μ be a store, τ a function table, r a term or expression and e an expression.

- τ is consistent with r if and only if for each subterm of r of the form $\mathsf{mfun}_\ell(f.a.e)$, we have $\tau(\ell) = \mathsf{mfun}(f.a.e^-)$
- τ is consistent with μ if and only if for all $\ell \in dom(\mu)$ and $\beta \in dom(\mu(\ell))$, if $\mu(\ell)(\beta) = v$ then τ is consistent with v.
- τ is compatible with μ if and only if:
  - · $dom(\mu) = dom(\tau)$
  - · τ is consistent with μ.
  - · For all $\ell \in dom(\mu)$, $\beta \in dom(\mu(\ell))$ and t term, if $\mu(\ell)(\beta) = v$ and $\tau(\ell)@\beta^- = \mathsf{return}\, t^-$ then $t^- \Downarrow_p^t v^-$. Let us call this condition ($\diamond$).

To prove that the translation simulates the evaluation relation we will need the following concepts.

Definition 4.7 An augmented branch γ is a list of augmented events ϵ. An augmented event records choice points and the bindings of resource variables.

$$\gamma ::= \bullet \mid \epsilon \cdot \gamma$$
$$\epsilon ::= (v) \mid !v \mid \langle v_1, v_2\rangle \mid \mathsf{inl}\, v \mid \mathsf{inr}\, v$$

Partial access functions for augmented branches are defined as follows:

$$\mathsf{mfun}_\ell(f.a.e) \;@\; \gamma\char`^(v) = e[f, a := \mathsf{mfun}_\ell(f.a.e), v] \;@\; \gamma$$
$$e \;@\; \bullet = e$$
$$\mathsf{let\,box}(t, x.e) \;@\; \gamma\char`^!v = e[x := v] \;@\; \gamma$$
$$\mathsf{letprod}(t, a_1.a_2.e) \;@\; \gamma\char`^\langle v_1, v_2\rangle = e[a_1, a_2 := v_1, v_2] \;@\; \gamma$$
$$\mathsf{mcase}(t, a_1.e_1, a_2.e_2) \;@\; \gamma\char`^\mathsf{inl}\, v = e_1[a_1 := v] \;@\; \gamma$$
$$\mathsf{mcase}(t, a_1.e_1, a_2.e_2) \;@\; \gamma\char`^\mathsf{inr}\, v = e_2[a_1 := v] \;@\; \gamma$$

Augmented branches are a device needed in the proof of proposition 4.9 below. Given an augmented branch γ we can easily obtain a simple branch denoted γ°, by forgetting events of the form (v) and $\langle v_1, v_2\rangle$, as well as the event v in the injections $\mathsf{inl}\, v$, $\mathsf{inr}\, v$. Moreover the translation γ^- of an augmented branch γ is defined by replacing each value v, occuring in γ, by v^-.

Operationally, the augmented branches give the same results as simple branches as ensured by the following

Lemma 4.8 *Augmented branches do not modify return statements. That is, if* $\tau(\ell)@\gamma = \mathsf{return}\, t$ *then* $\tau(\ell)@\gamma^\circ = \mathsf{return}\, t$

Proof See [1]. □

The following proposition states the preservation of the selective memoized semantics of SM with respect to the selective semantics without memoization of S, and allows us to conclude the faithfulness of the translation.

Proposition 4.9 *The translation* $t \mapsto t^-$ *simulates evaluation under the following conditions:*

(i) *Let* τ *be consistent with* t *and compatible with store* μ. *If* $\mu, t \Downarrow^t v, \mu'$ *then* $t^- \Downarrow^t_p v^-$ *and there exists* $\tau' \supseteq \tau$ *such that* τ' *is consistent with* v *and compatible with* μ'.

(ii) *Let* τ *be consistent with expression* e *and compatible with store* μ, β *be a simple branch and* γ *be an augmented branch. If* $\mu, e \Downarrow^e_{\beta@\ell} v, \mu'$, $\gamma^\circ = \beta$ *and* $\tau(\ell)@\gamma^- = e^-$ *then there exists* $\tau' \supseteq \tau$ *such that* τ' *is consistent with* v *and compatible with* μ' *and* $e^- \Downarrow^e_{\beta\,;\,\tau'(\ell)} v^-$

Proof The two parts are proved simultaneously by induction on both evaluation relations. We show the most important cases, application of a named function and return statements.

- $t = t_1 t_2$ and $\mu, t_1 t_2 \Downarrow^t v, \mu'$ derived from $\mu, t_1 \Downarrow^t v_1 = \mathsf{mfun}_\ell(f.a.e), \mu_1$, $\mu_1, t_2 \Downarrow^t v', \mu_2$ and $\mu_2, e[f, a := v_1, v_2] \Downarrow^e_{\bullet@\ell} v, \mu'$. Let τ be consistent with t and compatible with μ. In particular τ is consistent with t_1 which by I.H. yields $t_1^- \Downarrow^t_p v_1^-$ and $\tau_1 \supseteq \tau$ consistent with v_1 and compatible with μ_1. Applying now the I.H. for t_2 with τ_1 which is possible for τ_1 extends τ and therefore τ_1 is also consistent with t_2 we obtain $t_2^- \Downarrow^t_p v_2^-$ and $\tau_2 \supseteq \tau_1$ consistent with v_2 and compatible with μ_2. We observe now that τ_2 is also consistent with $e[f, a := v_1, v_2]$ for τ_2 is consistent with v_1, v_2 and also with e for e is a subexpression of v_1. It suffices now to show a γ such that $\gamma^\circ = \bullet$ and $\tau_2(\ell)@\gamma^- = (e[f, a := v_1, v_2])^-$. For then the I.H. for $e[f, a := v_1, v_2]$ will get us a τ such that $e[f, a := v_1, v_2]^- \Downarrow^e_{\bullet\,;\,\tau'(\ell)} v^-$ which will yield $t^- \Downarrow^t_p v^-$ as desired. Define $\gamma = (v_2) \cdot \bullet$, then

$$\begin{aligned}
\tau_2(\ell)@\gamma^- &= v_1^-@\gamma- \\
&= e^-[f, a := v_1^-, v_2^-] \\
&= (e[f, a := v_1, v_2])^- \quad \text{(by lemma 4.2)}
\end{aligned}$$

- $e = \mathsf{return}\, t$ and $\mu, e \Downarrow^e_{\beta@\ell} v, \mu'$ derived with the rule found, therefore $\mu' = \mu$ and $\mu(\ell)(\beta) = v$. Assume τ consistent with $\mathsf{return}\, t$ and compatible with μ, $\gamma^\circ = \beta$ and $\tau(\ell)@\gamma^- = (\mathsf{return}\, t)^- = \mathsf{return}\, t^-$. Observe that, as $\mu(\ell)(\beta) = v$ the compatibility of τ with μ implies that τ is consistent with v. Moreover, as $\tau(\ell)@\gamma^- = \mathsf{return}\, t^-$, lemma 4.8 implies $\tau(\ell)@(\gamma^-)^\circ = \mathsf{return}\, t^-$. But $(\gamma^-)^\circ = (\gamma^\circ)^- = \beta^-$. Therefore $\tau(\ell)@\beta^- = \mathsf{return}\, t^-$ and again by compatibility we get $t^- \Downarrow_p^t v^-$ which finally, using the rule for return, yields $\mathsf{return}\, t^- \Downarrow_{\beta\,;\,\tau(\ell)}^e v^-$. Therefore, in this case it suffices to take $\tau' = \tau$.

- $e = \mathsf{return}\, t$ and $\mu, e \Downarrow_{\beta@\ell}^e v, \mu'$ derived with the rule $\mathsf{notfound}$. Assume τ consistent with $\mathsf{return}\, t$ and compatible with μ, $\gamma^\circ = \beta$ and $\tau(\ell)@\gamma^- = (\mathsf{return}\, t)^- = \mathsf{return}\, t^-$. By I.H. there exists $\tau_1 \supseteq \tau$ consistent with v and compatible with μ' and $t^- \Downarrow_p^t v^-$. Take $\tau' = \tau_1$. It remains to be proved that τ' is compatible with $\mu'' = \mu'[\ell \leftarrow \theta'[\beta \mapsto v]]$. We show this now:
  · $dom(\mu'') = dom(\tau')$ because μ'' is only an update of μ'.
  · τ' is consistent with μ''. Observe that, as τ' is consistent with μ' it suffices to show consistency for the update $\mu''(\ell)(\beta) = v$. In this case we need to show that τ' is consistent with v, but this is true by definition of τ'.
  · To show that condition $(\diamond)$ holds, it suffices again to show it for ℓ, β and an arbitrary term t' such that $\mu''(\ell)(\beta) = v$ and $\tau(\ell)@\beta^- = \mathsf{return}\, t'^-$. But observe that, $\tau(\ell)@\beta^- = \tau(\ell)@\gamma^- = \mathsf{return}\, t^-$ which implies that $t = t'$, for $\_@\_$ is a function. Hence we have to show $t^- \Downarrow_p^t v^-$ but this was consequence of the I.H.

$\square$

It is easy to see that, if $t = f\,v$ is an application of a memoized function with no free variables f to a value v, the part (i) of proposition 4.9 entails that the translation of this memoized function computes indeed an application $t^- = f^- v^-$, which corresponds to a non-memoized function with the same outcome as f. In particular, the memoized function given in page 5 will compute the actual Fibonacci function.

Next we give some conditions which guarantee the existence of function tables as required by proposition 4.9.

Lemma 4.10 *If $(e\,@\,\gamma)$ is defined, then for every augmented event ϵ, $(e\,@\,\gamma)\,@\,\epsilon = e\,@\,\epsilon \cdot \gamma$*

Proof Straightforward induction on the augmented branch γ.

$\square$

Lemma 4.11 *Let e be an expression such that μ, $e \Downarrow_{\beta@\ell}^e v$, μ' originates in an application $\mu^\star, t_1 t_2 \Downarrow_p^t v^\star, \mu^{\star\prime}$ with $\mu^\star$ initial, then there exists a table function τ_e and an augmented branch γ such that τ_e is consistent with e and compatible with μ, $\gamma^\circ = \beta$ and $\tau_e(\ell)@\gamma^- = e^-$.*

Proof According to the statement of the lemma we have the following situation:

$$\dfrac{\begin{array}{ccc} & & \dfrac{\mu, e \Downarrow^e_{\beta @\ell} v, \mu'}{\mu'',\ e' \Downarrow^e_{\beta_1 @\ell} v,\ \mu'} \\ \vdots & \vdots & \vdots \\ \mu^\star,\ t_1 \Downarrow^t v_1 = \mathsf{mfun}_\ell(f.a.e_1), \mu_1 & \mu_1,\ t_2 \Downarrow^t v_2, \mu_2 & \mu_2, e_1[f, a := v_1, v_2] \Downarrow^e_{\bullet @\ell} v^\star,\ \mu^{\star\prime} \end{array}}{\mu^\star,\ t_1 t_2 \Downarrow^t v^\star,\ \mu^{\star\prime}}$$

The proof is by induction on the number n of inference rules from the evaluation of $e_1[f, a := v_1, v_2]$ up to the evaluation of e.

- Induction basis: $n = 0$ which means there is no inference rule, that is, $e = e_1[f, a := v_1, v_2]$ and $v_1 = \mathsf{mfun}_\ell(f.a.e_1)$. As $\mu^\star$ is initial we can build a τ consistent with t_1t_2 and compatible with $\mu^\star$. From this τ, repeated applications of proposition 4.9 yield the desired τ_e. In particular we have $\tau_e(\ell) = \mathsf{mfun}(f.a.e_1^-)$. Define now $\gamma = (v_2) \cdot \bullet$, and observe that $\tau_e(\ell)@\gamma^- = e_1^-[f, a := v_1^-, v_2^-] \cdot \bullet = e_1^-[f, a := v_1^-, v_2^-]$. But by lemma 4.2 $e_1^-[f, a := v_1^-, v_2^-] = e_1[f, a := v_1, v_2]^- = e^-$, which completes the proof.
- Inductive step: assume that there are $n+1$ inference rules from $e_1[f, a := v_1, v_2]$ up to e. The proof proceeds by a case analysis on the top rule and is illustrated for the case of rule ($\mathsf{Eletbox}$). In this case we have $e' = \mathsf{let\,box}(t, x.e_1)$, $e = e_1[x := v_1], \beta = !v_1 \cdot \beta_1$ and $\mu'', t \Downarrow^t_p \mathsf{box}\, v_1, \mu$. By I.H. there are $\tau_{e'}$ and γ_1 such that $\tau_{e'}$ is consistent with e' and compatible with $\mu'', \tau_{e'}(\ell)@\gamma_1^- = e'^-$ and $\gamma_1^\circ = \beta_1$. In particular $\tau_{e'}$ is consistent with t and therefore, by proposition 4.9, there is a τ'' such that $\tau'' \supseteq \tau_{e'}$ is consistent with $\mathsf{box}\, v_1$ and compatible with μ. Moreover, τ'' is also consistent with $e = e_1[x := v_1]$, and as it was compatible with μ we can define $\tau_e = \tau''$. Define now $\gamma = !v_1 \cdot \gamma_1$ and observe that $\gamma^\circ = !v_1 \cdot \gamma_1^\circ = !v_1 \cdot \beta_1 = \beta$. This suffices to show that $\tau_e(\ell)@\gamma^- = e^-$.

$$\begin{aligned} \tau_e(\ell)@\gamma^- &= \tau_{e'}(\ell)@\gamma^- \\ &= \tau_{e'}(\ell)@(!v_1^- \cdot \gamma_1^-) \\ &= (\tau_{e'}(\ell)@\gamma_1^-)@!v_1^-) \quad \text{(by lemma 4.10)} \\ &= \mathsf{let\,box}(t^-, x.e_1^-)@!v_1^- \\ &= e_1^-[x := v_1^-]@\bullet \\ &= e_1^-[x := v_1^-] = e^- \quad \text{(by lemma 4.2)} \end{aligned}$$

The cases for $\mathsf{letprod}$ and mcase are analogous.

□

After a final lemma, we will be able to prove now the type safeness for system SM .

Lemma 4.12 *Let r be a term or expression of* SM. *If $\Gamma|\Delta \vdash r^- : \mathsf{T}$ then there exists a label context Σ such that $\Gamma|\Delta|\Sigma \vdash r : \mathsf{T}$*

Proof Straightforward induction on the typing derivation of r^-. □

4.1 Type safeness for SM

We can now develop the proof of type safeness for SM stated in theorem 2.4.

Proof of theorem 2.4

- Part (i). Assume $\Gamma|\cdot|\Sigma \vdash t : \mathsf{T}$ and $\mu, t \Downarrow^t v, \mu'$ with μ an initial store. Let $\mathcal{L}_t$ be the set of labels occurring in t. Without loss of generality we can assume $dom(\mu) = \mathcal{L}_t = dom(\Sigma)$. Observe that the typing ensures that for every $\ell \in dom(\tau)$ there is a unique expression e such that $\mathsf{mfun}_\ell(f.a.e)$ occurs in t. Therefore we can define a function table τ with $dom(\tau) = dom(\mu)$ by defining $\tau(\ell) = \mathsf{mfun}(f.a.e^-)$ for every $\ell \in dom(\tau)$. In this way τ is consistent with t by construction and it is compatible with μ due to the initiality of μ. Therefore we can apply part (i) of proposition 4.9 to get a $\tau' \supseteq \tau$ such that τ' is consistent with v, compatible with μ' and $t^- \Downarrow^t_p v^-$. On the other hand, proposition 4.3 yields the typing $\Gamma|\cdot \vdash t^- : \mathsf{T}$ which together with $t^- \Downarrow^t_p v^-$ implies, by type safeness of system S (prop. 3.3), that $\Gamma|\cdot \vdash v^- : \mathsf{T}$. Finally lemma 4.12 yields a Σ' such that $\Gamma|\cdot|\Sigma' \vdash v : \mathsf{T}$.
- Part (ii). Assume $\Gamma|\cdot|\Sigma \vdash e : \mathsf{T}$ and $\mu, e \Downarrow^e_{\beta @ \ell} v, \mu'$ originating in an application $\mu^\star, t_1 t_2 \Downarrow^t_p v^\star, \mu^{\star\prime}$ with $\mu^\star$ initial. By lemma 4.11 there is a τ_e consistent with e and compatible with μ, and a γ such that $\tau_e(\ell)@\gamma^- = e^-$ and $\gamma^\circ = \beta$. The proof proceeds analogous to part (i).

$\square$

We have now fulfilled our main goal. However, although system S does not have explicit effects, from a strict point of view, this system is still not pure, for it keeps the distinction between types and expressions as well as the selectivity feature. To solve this problem we provide a translation of S to the purely functional language PCF.

5 A translation from S to PCF

In section 4 we have developed a faithful translation from SM to S, a system which can be considered to be effect-free. However we are not comfortable claiming that this translation shows the soundness of the memoized semantics with respect to a pure functional semantics. Our main reason to this remark is that the operational semantics of S is not pure strictly speaking, for it refers to a function table and upholds the selective mechanism of the original system SM. Moreover, the return instruction, which has a strong imperative flavor, is maintained. To avoid this problem we give a translation $(\cdot)^*$ from S to the very well-known pure functional language PCF.

The translation on types is the identity function except in the case of a box type where the definition is $(\Box\,\mathsf{T})^* = \mathsf{T}^*$. Furthermore, terms and expressions of S collapse into terms of PCF according to the following table, where every resource variable a is mapped to a unique variable x_a of PCF.

r	r^*
x	x
a	x_a
$\mathsf{box}\, t$	t^*
$\mathsf{mfun}_\ell(f.a.e)$	$\mathsf{fun}(f.x_a.e^*)$
$\mathsf{return}\, t$	t^*
$\mathsf{let\,box}(t,\, x.e)$	$\mathsf{let}(t^*,\, x.e^*)$
$\mathsf{letprod}(t,\, a_1.a_2.e)$	$\mathsf{let}(\mathsf{snd}\, t^*,\, x_{a_2}.(\mathsf{let}(\mathsf{fst}\, t^*,\, x_{a_1}.e^*)))$
$\mathsf{mcase}(t,\, a_1.e_1,\, a_2.e_2)$	$\mathsf{case}(t^*,\, x_{a_1}.e_1^*,\, x_{a_2}.e_2^*)$

For ease of presentation we have assumed named functions fun and let expressions as primitives in PCF, though they are indeed syntax sugar. All syntactic forms missed in the table are defined in a homomorphic way.

The following results prove that the translation is faithful. Their proofs are completely conventional due to the fact that S does not have actual effects.

Proposition 5.1 *Let r be a term or expression. If $\Gamma|\Delta \vdash r : \mathsf{T}$ then $\Gamma^*, \Delta^* \vdash r^* : \mathsf{T}^\star$, where $\Gamma^* = \{x : \mathsf{T}^\star \mid x : \mathsf{T} \in \Gamma\}$ and $\Delta^* = \{x_a : \mathsf{T}^* | a :: \mathsf{T} \in \Delta\}$.*

Proposition 5.2 *The translation $(\cdot)^*$ satisfies:*

(i) *If $t \Downarrow_p^t v$ then $t^* \Downarrow_{\mathsf{PCF}} v^*$.*

(ii) *If $e \Downarrow_{\beta;v_f}^e v$ then $e^* \Downarrow_{\mathsf{PCF}} v^*$.*

Finally an adequate combination of propositions 4.9 and 5.2 allows us to claim the soundness of the original memoization semantics with respect to a pure functional semantics.

6 Conclusions and Final Remarks

The framework MFL of selective memoization makes explicit the performance effects of memoization by capturing control and data dependences between the input and the result of a memoized function. Moreover it yields programs whose running times can be analyzed by standard techniques and has been implemented as an SML library. An essential feature of the system is the use of a modal type to reveal dependences. In this paper we have presented a framework SM, similar to the one developed in [2], with the following improvements from the theoretical point of view: instead of the bang modality, which in our opinion, was an ad-hoc choice for the implementation of the original system, we use the box type corresponding to the necessitation modality deeply discussed in [11] from the logical point of view. We provide our system with a static semantics which keeps exact track of the location labels occurring in a term or expression. Hence, we statically ensure the uniqueness of the label assignment for memoized functions. Type safeness for SM is proven by its translation to an effect-free system which is later translated to the purely functional language PCF. This way we have shown that selective memoization does not affect the outcome of evaluation as compared to a purely functional non-memoizing semantics.

Concerning current and future work, above all we do not discard a reformulation of the dynamic semantics of our systems in the framework of operational structural semantics, which would allow to pursuit an analysis of evaluation progress. We are working towards an implementation of the system in HASKELL, based on the framework of polytypic memoization of [5]. From the logical point of view neither MFL nor SM have a clear sight under the Curry-Howard correspondence. Hence an interesting question is to formulate a system which directly corresponds to the necessitation fragment of the judgmental reconstruction of modal logic given in [11]. In particular the typing relations for terms and expressions should be different, but mutually defined as in the case of evaluation relations; and the return instruction should be dismissed. Regarding further extensions of our system, a natural one is to add the type $\bigcirc\,\mathsf{T}$ representing monadic memoization and other computations of type T.
Memoization becomes more complicated when the language includes composite value types like records, in that case the data dependencies should be as fine-grained as possible, as discussed in [4]. Therefore, it is interesting to pursuit the modeling of such dependencies in our framework. Last, but not least, and regarding the inclusion of recursive types, we recall that memoization is most effective when applied to the kind of programs used in dynamic programming, which usually are neither simple iterative nor primitive-recursive, but course-of-value recursive. This fact leads us to investigate an extension with inductive types modeling the latter principle, motivated by our previous work in [7].

Acknowledgement

We are thankful to the anonymous referees for the helpful comments and suggestions regarding the contents and presentation of this paper. We also gratefully acknowledge the LSFA 2009 travel grants committee for providing partial funds to attend the meeting and Martha Buschbeck Alvarado for improving the English manuscript.

References

[1] Acar U. A., G. E. Blelloch and R. Harper, *Selective memoization*, Technical Report CMU-CS-02-194, Carnegie Mellon University, Computer Science Department, 2002.

[2] Acar U. A., G. E. Blelloch and R. Harper, *Selective memoization*, In Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (2003), 14-25.

[3] Cormen T. H., C. Leiserson and R. Rivest, “Introduction to algorithms”, MIT Press, 1990.

[4] Heydon A., R. Levin and Y. Yu, *Caching function calls using precise dependencies*, In Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, 311-320.

[5] Hinze R., *Memo functions, polytypically!*, Proceedings of the 2nd Workshop on Generic Programming (2000), 17-32.

[6] Michie D., *"Memo" Functions and Machine Learning*, Nature (1968), 218:19-22.

[7] Miranda-Perea F. E., *Some Remarks on Type Systems for Course-of-value Recursion* To appear in Electronic Notes in Theoretical Computer Science. Elsevier Science Holland 2009.

[8] Miyamoto K. and A. Igarashi, *A modal foundation for secure information flow*, In Proceedings of Workshop on Foundations of Computer Security (2004), 187-203.

[9] Moody J., *Logical Mobility and Locality Types*, In Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (2004), 69-84.

[10] Norvig P., *Techniques for Automatic Memoization with Applications to Context-Free Parsing*, Computational Linguistics **17** (1991), 91-98.

[11] Pfenning F. and R. Davies, *A judgmental reconstruction of modal logic*, Mathematical Structures in Computer Science **11** (2001), 511-540.

[12] Pugh W. and T. Teitelbaum, *Incremental Computation via Function Caching*, In Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (1989), 315-328.

[13] Yuse Y. and A. Igarashi, *A modal type system for multi-level generating extensions with persistent code*, In Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming (2006), 201-212.