Quantitative Global Memory[⋆]

Sandra Alves[1], Delia Kesner[2,3], and Miguel Ramos[4,⋆⋆]

[1] CRACS/INESC-TEC, DCC, Faculdade de Ciências, Universidade do Porto
Rua do Campo Alegre s/n, 4169–007 Porto, Portugal
[2] Université Paris Cité, CNRS, IRIF
[3] Institut Universitaire de France
[4] LIACC, DCC, Faculdade de Ciências, Universidade do Porto
Rua do Campo Alegre s/n, 4169–007 Porto, Portugal

Abstract. We show that recent approaches to static analysis based on quantitative typing systems can be extended to programming languages with global state. More precisely, we define a call-by-value language equipped with operations to access a global memory, together with a semantic model based on a (tight) multi-type system that captures exact measures of time and space related to evaluation of programs. We show that the type system is quantitatively sound and complete with respect to the operational semantics of the language.

1 Introduction

The aim of this paper is to extend *quantitative* techniques of *static analysis* based on *multi-types* to programs with *effects*.

Effectful Programs. Programming languages produce different kinds of *effects* (observable interactions with the environment), such as handling exceptions, read/write from a global memory outside its own scope, using a database or a file, performing non-deterministic choices, or sampling from probabilistic distributions. The degree to which these side effects are used depends on each programming paradigm [24] (imperative programming makes use of them while declarative programming does not). In general, avoiding the use of side effects facilitates the formal verification of programs, thus allowing to (statically) ensure their correctness. For example, the functional language Haskell eliminates side effects by replacing them with *monadic* actions, a clean approach that continues to attract growing attention. Indeed, rather than writing a function that returns a raw type, an effectful function returns a raw type inside a useful wrapper – and that wrapper is a monad [34]. This approach allows programming languages to

⋆Supported by: National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia -, within project LA/P/0063/2020, and the project and individual research grant 2021.04731.BD; Base Funding UIDB/00027/2020 of the Artificial Intelligence and Computer Science Laboratory – LIACC - funded by national funds through the FCT/MCTES (PIDDAC); and Cost Action CA20111 EuroProofNet.

⋆⋆Corresponding author.

combine the qualities of both the imperative and declarative worlds: programs produce effects, but these are encoded in such a way that formal verification can be performed very conveniently.

Quantitative Properties. We address quantitative properties of programs with effects using *multi-types*, which originate in the theory of *intersection* type systems. They extend simple types with a new constructor $\cap$ in such a way that a program t is typable with $\sigma \cap \tau$ if t is typable with both types σ and τ independently. Intersection types were first introduced as *models* capturing computational properties of functional programming in a broader sense [14]. For example, termination of different evaluation strategies can be characterized by typability in some appropriate intersection type system: a program t is terminating if and only if t is typable. Originally, intersection enjoys associativity, commutativity, and in particular idempotency (*i.e.* $\sigma \cap \sigma = \sigma$). By switching to a *non-idempotent* intersection constructor, one naturally comes to represent types by multisets, which is why they are called multi-types. Just like their idempotent precursors, multi-types still allow for a characterization of several operational properties of programs, but they also grant a substantial improvement: they provide quantitative measures about these properties. For example, it is still possible to prove that a program is terminating if and only if it is typable, but now an *upper bound* or *exact measure* for the time needed for its evaluation length can be derived from the typing derivation of the program. This shift of perspective, from idempotent to non-idempotent types, goes beyond lowering the logical complexity of the proof: the quantitative information provided by typing derivations in the non-idempotent setting unveils crucial quantitative relations between typing (static) and reduction (dynamic) of programs.

Upper Bounds and Exact Split Measures. Multi-types are extensively used to reason about programming languages from a quantitative point of view, as pioneered by de Carvalho [12,13]. For example, they are able to provide *upper bounds*, in the sense that the evaluation length of a program t *plus* the size of its result (called *normal form*) can be bounded by the size of the type derivation of t. A major drawback of this approach, however, is that the size of normal forms can be exponentially bigger than the length of the evaluation reaching those normal forms. This means that bounding the sum of these two natural numbers at the same time is too rough, and not very relevant from a quantitative point of view. Fortunately, it is possible to extract better measures from a multi-type system. A crucial point to obtain *exact measures*, instead of upper bounds, is to consider minimal type derivations, called *tight derivations*. Moreover, using appropriate refined tight systems it is also possible to obtain *independent* measures (called exact *split* measures) for *length* and for *size*. More precisely, the quantitative typing systems[1] are now equipped with constants and counters, together with an appropriate notion of tightness, which encodes minimality of type derivations. For any tight type derivation Φ of a program t with counters b and d, it is now possible to show that t evaluates to a normal form of size d in exactly b steps.

[1] In this paper, by quantitative types we mean non-idempotent intersection types. Another meaning can be found in [6].

Therefore, the type system is not only *sound*, *i.e.* it is able to *guess* the number of steps to normal form as well as the size of this normal form, but the opposite direction providing *completeness* of the approach also holds.

Contribution. The focus of this paper is on effectful computations, such as reading and writing on a global memory able to hold values in cells. Taking inspiration from the monadic approach adopted in [16], we design a tight quantitative type system that provides exact split measures. More precisely, our system is not only capable of discriminating between length of evaluation to normal form and size of the normal form, but the measure corresponding to the length of the evaluation is split into two different natural numbers: the first one corresponds to the length of standard computation (β-reduction) and the second one to the number of memory accesses. We show that the system is sound *i.e.* for any tight type derivation Φ of t ending with counters (b, m, d), the term t is normalisable by performing b evaluation steps and m memory accesses, yielding a normal form having size d. The opposite direction, giving completeness of the model, is also proved.

In order to gradually present the material, we first develop the technique for a weak (open) call-by-value (CBV) calculus, which can be seen as a contribution per se, and then we encapsulate these preliminary ideas in the general framework of the language with global state.

Summary. Sec. 2 illustrates the technique on a weak (open) CBV calculus. We then lift the technique to the λ-calculus with global state in Sec. 3 by following the same methodology. More precisely, Sec. 3.1 introduces the λ_{gs}-calculus, Sec. 3.2 defines a quantitative type system $\mathcal{P}$. Soundness and completeness of $\mathcal{P}$ w.r.t. λ_{gs} are proved in Sec. 3.3. We conclude and discuss related work in Sec. 4. Due to space limitations we do not include proofs, but they are available in [5].

Preliminary General Notations. We start with some general notations. Given a (one-step) reduction relation $\rightarrow_{\mathcal{R}}$, $\twoheadrightarrow_{\mathcal{R}}$ denotes the reflexive-transitive closure of $\rightarrow_{\mathcal{R}}$. We write $t \twoheadrightarrow^{b} u$ for a reduction sequence from t to u of length b. A term t is said to be (1) in **$\mathcal{R}$-normal form** (written $t \not\rightarrow_{\mathcal{R}}$) iff there is no u such that $t \rightarrow_{\mathcal{R}} u$, and (2) **$\mathcal{R}$-normalizing** iff there is some $\mathcal{R}$-normal form u such that $t \twoheadrightarrow_{\mathcal{R}} u$. The reduction relation $\mathcal{R}$ is normalizing iff every term is $\mathcal{R}$-normalizing.

2 Weak Open CBV

In this section we first introduce the technique of tight typing on a simple language without effects, the weak open CBV. Sec. 2.1 defines the syntax and operational semantics of the language, Sec. 2.2 presents the tight typing system $\mathcal{O}$ and discusses soundness and completeness of $\mathcal{O}$ w.r.t. the CBV language.

2.1 Syntax and Operational Semantics

Weak open CBV is based on two principles: reduction is *weak* (not performed inside abstractions), and terms are *open* (may contain free variables). **Value**,

terms and **weak contexts** are given by the following grammars, respectively:

$$v, w ::= x \mid \lambda x.t \qquad t, u, p ::= v \mid tu \qquad \mathcal{W} ::= \Box \mid \mathcal{W}t \mid t\mathcal{W}$$

We write $\mathtt{Val}$ for the set of all values. Symbol $\mathtt{I}$ is used to denote the identity function $\lambda z.z$.

The sets of **free** and **bound** variables of terms and the notion of α-conversion are defined as usual. A term t is said to be **closed** if t does not contain any free variable, and **open** otherwise. The **size of a term** t, denoted $|t|$, is given by: $|x| = |\lambda x.t| = 0$; and $|tu| = 1 + |t| + |u|$. Since our reduction relation is weak, *i.e.*, reduction does not occur in the body of abstractions, we assign size zero to abstractions.

We now introduce the operational semantics of our language, which models the core behavior of programming languages such as OCaml, where CBV evaluation is *weak*. The **deterministic reduction relation** (written $\rightarrow$), is given by the following rules:

$$\frac{}{(\lambda x.t)v \rightarrow t\{x\backslash v\}}\ (\beta_{\mathtt{v}}) \qquad \frac{t \rightarrow t'}{tu \rightarrow t'u}\ (\mathtt{appL}) \qquad \frac{t \not\rightarrow \quad u \rightarrow u'}{tu \rightarrow tu'}\ (\mathtt{appR})$$

Terms in $\rightarrow$-normal form can be characterized by the following grammars: $\mathtt{no} ::= \mathtt{Val} \mid \mathtt{ne}$ and $\mathtt{ne} ::= x\ \mathtt{no} \mid \mathtt{no}\ \mathtt{ne} \mid \mathtt{ne}\ \mathtt{no}$.

Proposition 1. *Let t be a term. Then $t \in \mathtt{no}$ iff $t \not\rightarrow$.*

In closed CBV [31] (only reducing closed terms), abstractions are the only normal forms, but in open CBV, the following terms turn out to be also acceptable normal forms: xy, $x(\lambda y.y(\lambda z.z))$ and $(\lambda x.x)(y(\lambda z.z))$.

2.2 A Quantitative Type System for the Weak Open CBV

The *untyped* λ-calculus can be interpreted as a *typed* calculus with a single type D, where $D = D \Rightarrow D$ [33]. Applying Girard's [22] *"boring"* CBV translation of intuitionistic logic into linear logic, we get $D = !D \multimap !D$ [1]. Type system $\mathcal{O}$ is built having this equation in mind.

The **set of types** is given by the following grammar:

(Tight Constants)	$\mathtt{tt} ::= \mathtt{v} \mid \mathtt{a} \mid \mathtt{n}$
(Value Types)	$\sigma ::= \mathtt{v} \mid \mathtt{a} \mid \mathcal{M} \mid \mathcal{M} \Rightarrow \tau$
(Multi-Types)	$\mathcal{M} ::= [\sigma_i]_{i \in I}$ where I is a finite set
(Types)	$\tau ::= \mathtt{n} \mid \sigma$

Tight constants are minimal types assigned to terms reducing to normal forms ($\mathtt{v}$ for persistent variables, $\mathtt{a}$ for abstractions or variables that are going to be replaced by abstractions, and $\mathtt{n}$ for neutral terms). Given an arbitrary tight constant $\mathtt{tt}_0$, we write $\overline{\mathtt{tt}_0}$ to denote all the other tight constants in $\mathtt{tt}$ different from $\mathtt{tt}_0$. Multi-types are multisets of value types. A **(typing) environment**, written Γ, Δ, is a function from variables to multi-types, assigning the empty multi-type $[\,]$ to all but a finite set of variables. The domain of Γ is $\mathtt{dom}(\Gamma) := \{x \mid \Gamma(x) \neq [\,]\}$. The **union** of environments, written $\Gamma + \Delta$, is defined by $(\Gamma + \Delta)(x) = \Gamma(x) \sqcup \Delta(x)$, where $\sqcup$ denotes **multiset union**. An example is $(x : [\sigma_1], y : [\sigma_2]) + (x : [\sigma_1], z : [\sigma_2]) = (x : [\sigma_1, \sigma_1], y : [\sigma_2], z : [\sigma_2])$. This notion is extended to a finite union of environments, written $+_{i \in I} \Gamma_i$ (the empty environment is obtained when $I = \emptyset$). We write $\Gamma \setminus\!\!\setminus x$ for the environment $(\Gamma \setminus\!\!\setminus x)(x) = [\,]$ and $(\Gamma \setminus\!\!\setminus x)(y) = \Gamma(y)$ if $y \neq x$ and we write $\Gamma; x : \mathcal{M}$ for $\Gamma + (x : \mathcal{M})$, when $x \notin \mathtt{dom}(\Gamma)$. Notice that Γ and $\Gamma; x : [\,]$ are the same environment.

A **judgement** has the form $\Gamma \vdash^{(b,s)} t : \tau$, where b and s are two natural numbers, representing, respectively, the number of β-steps needed to normalize t, and the size of the normal form of t. The **typing system $\mathcal{O}$** is defined by the rules in Fig. 1. We write $\triangleright \Gamma \vdash^{(b,s)} t : \tau$ if there is a (tree) **type derivation** of the judgement $\Gamma \vdash^{(b,s)} t : \tau$ using the rules of system $\mathcal{O}$. The term t is **$\mathcal{O}$-typable** (we may omit the name $\mathcal{O}$) iff there is an environment Γ, a type τ and counters (b, s) such that $\triangleright \Gamma \vdash^{(b,s)} t : \tau$. We use letters $\Phi, \Psi, \ldots$ to name type derivations, by writing for example $\Phi \triangleright \Gamma \vdash^{(b,s)} t : \tau$.

$$\frac{}{x : [\sigma] \vdash^{(0,0)} x : \sigma}\ (\mathtt{ax}) \qquad \frac{\Gamma \vdash^{(b,s)} t : \tau}{\Gamma \setminus\!\!\setminus x \vdash^{(b,s)} \lambda x.t : \Gamma(x) \Rightarrow \tau}\ (\lambda)$$

$$\frac{\Gamma \vdash^{(b,s)} t : \mathcal{M} \Rightarrow \tau \qquad \Delta \vdash^{(b',s')} u : \mathcal{M}}{\Gamma + \Delta \vdash^{(1+b+b',s+s')} tu : \tau}\ (\mathtt{@}) \qquad \frac{(\Gamma_i \vdash^{(b_i,s_i)} v : \sigma_i)_{i \in I}}{+_{i \in I} \Gamma_i \vdash^{(+_{i \in I} b_i, +_{i \in I} s_i)} v : [\sigma_i]_{i \in I}}\ (\mathtt{m})$$

$$\frac{}{\vdash^{(0,0)} \lambda x.t : \mathtt{a}}\ (\lambda_{\mathtt{p}})$$

$$\frac{\Gamma \vdash^{(b,s)} t : \overline{\mathtt{a}} \qquad \Delta \vdash^{(b',s')} u : \mathtt{tt}}{\Gamma + \Delta \vdash^{(b+b',1+s+s')} tu : \mathtt{n}}\ (\mathtt{@}_{\mathtt{p1}}) \qquad \frac{\Gamma \vdash^{(b,s)} t : \mathtt{tt} \qquad \Delta \vdash^{(b',s')} u : \mathtt{n}}{\Gamma + \Delta \vdash^{(b+b',1+s+s')} tu : \mathtt{n}}\ (\mathtt{@}_{\mathtt{p2}})$$

Fig. 1. Typing Rules of System $\mathcal{O}$

Notice that in rule $(\mathtt{ax})$ of Fig. 1 variables can only be assigned value types σ (in particular no type $\mathtt{n}$): this is because they can only be substituted by values. Due to this fact, multi-types only contain value types. Regarding typing rules $(\mathtt{ax})$, (λ), $(\mathtt{@})$, and $(\mathtt{m})$, they are the usual rules for non-idempotent intersection types [10]. Rules $(\lambda_{\mathtt{p}})$, $(\mathtt{@}_{\mathtt{p1}})$, and $(\mathtt{@}_{\mathtt{p2}})$ are used to type *persistent* symbols, *i.e.* symbols that are not going to be *consumed* during evaluation. More specifically, rule $(\lambda_{\mathtt{p}})$ types abstractions (with type $\mathtt{a}$) that are normal regardless of the typability of its body. Rule $(\mathtt{@}_{\mathtt{p1}})$ types applications that will never reduce to an abstraction on the left (thus of any tight constant that is not $\mathtt{a}$, *i.e.* $\overline{\mathtt{a}}$), while any term reducing to a normal form is allowed on the right (of tight constant $\mathtt{tt}$). Rule $(\mathtt{@}_{\mathtt{p2}})$ also types applications, but when values will never be obtained on the right (only neutral terms of type $\mathtt{n}$). Rule $(\mathtt{ax})$ is also used to type persistent variables, in particular when $\sigma \in \{\mathtt{v}, \mathtt{a}\}$.

A **type** τ is **tight** if $\tau \in \mathtt{tt}$. We write $\mathtt{tight}(\mathcal{M})$, if every $\sigma \in \mathcal{M}$ is tight. A **type environment** Γ is **tight** if it assigns tight multi-types to all variables. A **type derivation** $\Phi \triangleright \Gamma \vdash^{(b,s)} t : \tau$ is **tight** if Γ and τ are both tight.

Example 1. Let $t = (\lambda x.(xx)(yy))(\lambda z.z)$. Let Φ be the following typing derivation:

$$\dfrac{\dfrac{}{x : [[\mathtt{a}] \Rightarrow \mathtt{a}] \vdash^{(0,0)} x : [\mathtt{a}] \Rightarrow \mathtt{a}}\,(\mathtt{ax}) \qquad \dfrac{\dfrac{}{x : [\mathtt{a}] \vdash^{(0,0)} x : \mathtt{a}}\,(\mathtt{ax})}{x : [\mathtt{a}] \vdash^{(0,0)} x : [\mathtt{a}]}\,(\mathtt{m})}{x : [[\mathtt{a}] \Rightarrow \mathtt{a}, \mathtt{a}] \vdash^{(1,0)} xx : \mathtt{a}}\,(\mathtt{@})$$

And Ψ be the following typing derivation:

$$\dfrac{\dfrac{\Phi \qquad \dfrac{\dfrac{}{y : [\mathtt{v}] \vdash^{(0,0)} y : \mathtt{v}}\,(\mathtt{ax}) \qquad \dfrac{}{y : [\mathtt{v}] \vdash^{(0,0)} y : \mathtt{v}}\,(\mathtt{ax})}{y : [\mathtt{v}, \mathtt{v}] \vdash^{(0,1)} yy : \mathtt{n}}\,(\mathtt{@}_{\mathtt{p1}})}{x : [[\mathtt{a}] \Rightarrow \mathtt{a}, \mathtt{a}], y : [\mathtt{v}, \mathtt{v}] \vdash^{(1,1)} (xx)(yy) : \mathtt{n}}\,(\mathtt{@}_{\mathtt{p2}})}{y : [\mathtt{v}, \mathtt{v}] \vdash^{(1,2)} \lambda x.(xx)(yy) : [[\mathtt{a}] \Rightarrow \mathtt{a}, \mathtt{a}] \Rightarrow \mathtt{n}}\,(\lambda)$$

Then, we can build the following tight typing derivation Φ_t for t:

$$\dfrac{\Psi \qquad \dfrac{\dfrac{\dfrac{}{z : [\mathtt{a}] \vdash^{(0,0)} z : \mathtt{a}}\,(\lambda_{\mathtt{p}})}{\vdash^{(0,0)} \lambda z.z : [\mathtt{a}] \Rightarrow \mathtt{a}}\,(\lambda) \qquad \dfrac{}{\vdash^{(0,0)} \lambda z.z : \mathtt{a}}\,(\lambda_{\mathtt{p}})}{\vdash^{(0,0)} \lambda z.z : [[\mathtt{a}] \Rightarrow \mathtt{a}, \mathtt{a}]}\,(\mathtt{m})}{y : [\mathtt{v}, \mathtt{v}] \vdash^{(2,2)} (\lambda x.(xx)(yy))(\lambda z.z) : \mathtt{n}}\,(\mathtt{@})$$

The type system $\mathcal{O}$ can be shown to be *sound* and *complete* w.r.t. the operational semantics $\rightarrow$ introduced in Sec. 2.1. Soundness means that not only a *tightly* typable term t is terminating, but also that the *tight* type derivation of t gives exact and split measures concerning the reduction sequence from t to normal form. More precisely, if $\Phi \triangleright \Gamma \vdash^{(b,s)} t : \tau$ is tight, then there exists $u \in \mathtt{no}$ such that $t \twoheadrightarrow^b u$ with $|u| = s$. Dually for *completeness*. Because we are going to show this kind of properties for the more sophisticated language with global state (Sec. 3.3), we do not give here technical details of them. However, we highlight these properties on our previous example. Consider again term t in Ex. 1 and its tight derivation Φ_t with counters $(b, s) = (2, 2)$. Counter b is different from 0, so $t \notin \mathtt{no}$, but t normalizes in two β_v-steps ($b = 2$) to a normal form having size $s = 2$. Indeed, $(\lambda x.(xx)(yy))(\lambda z.z) \rightarrow_{\beta_v} ((\lambda z.z)(\lambda z.z))(yy) \rightarrow_{\beta_v} (\lambda z.z)(yy)$ and $|(\lambda z.z)(yy)| = 2$.

3 A λ-Calculus with Global State

Based on the preliminary presentation of Sec. 2, we now introduce a λ-calculus with global state following a CBV strategy. Sec. 3.1 defines the syntax and operational semantics of the λ-calculus with global state. Sec. 3.2 presents the tight typing system $\mathcal{P}$, and Sec. 3.3 shows soundness and completeness.

3.1 Syntax and Operational Semantics

Let l be a location drawn from some set of location names. **Values**, **terms**, **states** and **configurations** of $\lambda_{\mathtt{gs}}$ are defined respectively as follows:

$$\begin{array}{rlrl} v, w ::= & x \mid \lambda x.t & t, u, p ::= & v \mid vt \mid \mathtt{get}_l(\lambda x.t) \mid \mathtt{set}_l(v, t) \\ s, q ::= & \epsilon \mid \mathtt{upd}_l(v, s) & c ::= & (t, s) \end{array}$$

Notice that applications are restricted to the form vt. This, combined with the use of a deterministic reduction strategy based on weak contexts, ensures that composition of effects is well behaved. Indeed, this kind of restriction is usual in computational calculi [30,32,16,19].

Intuitively, operation $\mathtt{get}_l(\lambda x.t)$ interacts with the global state by retrieving the value stored in location l and binding it to variable x of the continuation t. And operation $\mathtt{set}_l(v, t)$ interacts with the state by storing value v in location l and (possibly) overwriting whatever was previously stored there, and then returns t.

The size function is extended to states and configurations: $|s| := 0$, and $|(t, s)| := |t|$. The update constructor is commutative in the following sense:

$$\mathtt{upd}_l(v, \mathtt{upd}_{l'}(w, s)) \equiv_{\mathtt{c}} \mathtt{upd}_{l'}(w, \mathtt{upd}_l(v, s)) \text{ if } l \neq l'$$

We denote by $\equiv$ the equivalence relation generated by the axiom $\equiv_{\mathtt{c}}$. We write $l \in \mathtt{dom}(s)$, if $s \equiv \mathtt{upd}_l(v, q)$, for some value v and state q. Moreover, these v and q are *unique*. For example, if $l_1 \neq l_2$, then $s_1 = \mathtt{upd}_{l_1}(v_1, \mathtt{upd}_{l_2}(v_2, q)) \equiv \mathtt{upd}_{l_2}(v_2, \mathtt{upd}_{l_1}(v_1, q)) = s_2$, but $\mathtt{upd}_{l_1}(v_1, \mathtt{upd}_{l_1}(v_2, s)) \not\equiv \mathtt{upd}_{l_1}(v_2, \mathtt{upd}_{l_1}(v_1, s))$. As a consequence, whenever we want to access the content of a particular location in a state, we can simply assume that the location is at the top of the state.

The operational semantics of the $\lambda_{\mathtt{gs}}$-calculus is given on configurations. The **deterministic reduction relation** $\rightarrow$ is defined to be the union of the rules $\rightarrow_{\mathtt{r}}$ ($\mathtt{r} \in \{\beta_v, \mathtt{g}, \mathtt{s}\}$) below. We write $(t, s) \twoheadrightarrow^{(b,m)} (u, q)$ if (t, s) reduces to (u, q) in b β_v-steps and m $\mathtt{g}/\mathtt{s}$-steps.

$$\frac{}{((\lambda x.t)v, s) \rightarrow_{\beta_v} (t\{x \backslash v\}, s)}\ (\beta_{\mathtt{v}}) \qquad \frac{s \equiv \mathtt{upd}_l(v, q)}{(\mathtt{get}_l(\lambda x.t), s) \rightarrow_{\mathtt{g}} (t\{x \backslash v\}, s)}\ (\mathtt{get})$$

$$\frac{(t, s) \rightarrow_{\mathtt{r}} (u, q) \quad \mathtt{r} \in \{\beta_v, \mathtt{g}, \mathtt{s}\}}{(vt, s) \rightarrow_{\mathtt{r}} (vu, q)}\ (\mathtt{appR}) \qquad \frac{}{(\mathtt{set}_l(v, t), s) \rightarrow_{\mathtt{s}} (t, \mathtt{upd}_l(v, s))}\ (\mathtt{set})$$

Note that in reduction rule ($\mathtt{appR}$), the $\mathtt{r}$ appearing as the name of the reduction rule in the premise is the same as the one appearing in the reduction rule in the conclusion.

Example 2. Consider the configuration $c_0 = ((\lambda x.\texttt{get}_l(\lambda y.yx))(\texttt{set}_l(\texttt{I}, z)), \epsilon)$. Then we can reach an irreducible configuration as follows:

$$\begin{aligned}&((\lambda x.\texttt{get}_l(\lambda y.yx))(\texttt{set}_l(\texttt{I}, z)), \epsilon) \to_{\texttt{g}} ((\lambda x.\texttt{get}_l(\lambda y.yx))z, \texttt{upd}_l(\texttt{I}, \epsilon))\\ &\to_{\beta_v} (\texttt{get}_l(\lambda y.yz), \texttt{upd}_l(\texttt{I}, \epsilon)) \to_{\texttt{g}} (\texttt{I}z, \texttt{upd}_l(\texttt{I}, \epsilon)) \to_{\beta_v} (z, \texttt{upd}_l(\texttt{I}, \epsilon))\end{aligned}$$

A configuration (t, s) is said to be **blocked** if either $t = \texttt{get}_l(\lambda x.u)$ and $l \notin \texttt{dom}(s)$; or $t = vu$ and (u, s) is blocked. A configuration is **unblocked** if it is not blocked. As an example, $(\texttt{get}_l(\lambda x.x), \epsilon)$ is obviously blocked. As a consequence, the following configuration reduces to a blocked one: $((\lambda y.y\ \texttt{get}_l(\lambda x.x))z, \epsilon) \to (z\ \texttt{get}_l(\lambda x.x), \epsilon)$. This suggests a notion of **final configuration**: (t, s) is **final** if either (t, s) is blocked; or $t \in \texttt{no}$, where **neutral** and **normal** terms are given respectively by the grammars $\texttt{ne} ::= x\ \texttt{no} \mid (\lambda x.t)\ \texttt{ne}$ and $\texttt{no} ::= \texttt{Val} \mid \texttt{ne}$.

Proposition 2. *Let (t, s) be a configuration. Then (t, s) is final iff $(t, s) \not\to$.*

Notice that when (t, s) is an unblocked final configuration, then $t \in \texttt{no}$. These are the configurations captured by the typing system $\mathcal{P}$ in Sec. 3.2. Consider the final configurations $c_0 = (\texttt{get}_l(\lambda x.x), \epsilon)$, $c_1 = (z\ \texttt{get}_l(\lambda x.x), \epsilon)$, $c_2 = (y, s)$ and $c_3 = ((\lambda x.x)(yz), s)$. Then c_0 and c_1 are blocked, while c_2 and c_3 are unblocked.

3.2 A Quantitative Type System for the $\lambda_{\texttt{gs}}$-calculus

We now introduce the quantitative type system $\mathcal{P}$ for $\lambda_{\texttt{gs}}$. To deal with global states, we extend the language of types with the notions of state, configuration and monadic types. To do this, we translate linear arrow types according to Moggi's [30] CBV interpretation of reflexive objects in the category of λ_c-models: $D = !D \multimap !D$ becomes $D = !D \multimap T(!D)$, where T is a monad. Type system $\mathcal{P}$ was built having this equation in mind, similarly to what was done in [21].

The **set of types** is given by the following grammar:

(Tight Constants)	$\texttt{tt} ::= \texttt{v} \mid \texttt{a} \mid \texttt{n}$
(Value Types)	$\sigma ::= \texttt{v} \mid \texttt{a} \mid \mathcal{M} \mid \mathcal{M} \Rightarrow \delta$
(Multi-types)	$\mathcal{M} ::= [\sigma_i]_{i \in I}$ where I is a finite set
(Liftable Types)	$\mu ::= \texttt{v} \mid \texttt{a} \mid \mathcal{M}$
(Types)	$\tau ::= \texttt{n} \mid \sigma$
(State Types)	$\mathcal{S} ::= \{(l_i : \mathcal{M}_i)\}_{i \in I}$ where all l_i are distinct
(Configuration Types)	$\kappa ::= \tau \times \mathcal{S}$
(Monadic Types)	$\delta ::= \mathcal{S} \gg \kappa$

In system $\mathcal{P}$, the minimal types to be assigned to normal forms distinguish between variables ($\texttt{v}$), abstractions ($\texttt{a}$), and neutral terms ($\texttt{n}$). A **multi-type** is a multi-set of value types. A **state type** is a partial function mapping labels to (possibly empty) multi-types. A **configuration type** is a product type, where the first component is a type and the second is a state type. A **monadic type** associates a state type to a configuration type. We use symbol $\mathcal{T}$ to denote a value type or a monadic type. **Typing environments** and operations over types are defined in the same way as in system $\mathcal{O}$.

The **domain** of a state type $\mathcal{S}$ is the set of all its labels, *i.e.* $\mathtt{dom}(\mathcal{S}) := \{l \mid (l : \mathcal{M}) \in \mathcal{S}\}$. Also, when $l \in \mathtt{dom}(\mathcal{S})$, *i.e.* $(l : \mathcal{M}) \in \mathcal{S}$, we write $\mathcal{S}(l)$ to denote $\mathcal{M}$. The **union of state types** is defined as follows:

$$(\mathcal{S} \Cup \mathcal{S}')(l) = \text{if } (l : \mathcal{M}) \in \mathcal{S} \text{ then } (\text{if } (l : \mathcal{M}') \in \mathcal{S}' \text{ then } \mathcal{M} \sqcup \mathcal{M}' \text{ else } \mathcal{M})$$
$$\text{else } (\text{if } (l : \mathcal{M}') \in \mathcal{S}' \text{ then } \mathcal{M}' \text{ else } \mathtt{undefined})$$

Example 3. Let $\mathcal{S} = \{(l_1 : [\sigma_1, \sigma_2]), (l_2 : [\sigma_1])\} \Cup \{(l_2 : [\sigma_1, \sigma_2]), (l_3 : [\sigma_3])\}$. Then, $\mathcal{S}(l_1) = [\sigma_1, \sigma_2]$, $\mathcal{S}(l_2) = [\sigma_1, \sigma_1, \sigma_2]$, $\mathcal{S}(l_3) = [\sigma_3]$, and $\mathcal{S}(l) = \mathtt{undefined}$, assuming $l \neq l_i$, for $i \in \{1, 2, 3\}$.

Notice that $\mathtt{dom}(\mathcal{S} \Cup \mathcal{S}') = \mathtt{dom}(\mathcal{S}) \cup \mathtt{dom}(\mathcal{S}')$. Also $\{(l : [\,])\} \Cup \mathcal{S} \neq \mathcal{S}$, if $l \notin \mathtt{dom}(\mathcal{S})$, while $x : [\,]; \Gamma = \Gamma$. Indeed, typing environments are total functions, where variables mapped to $[\,]$ do not occur in typed programs. In contrast, states are partial functions, where labels mapped to $[\,]$ correspond to positions in memory that are accessed (by get or set), but ignored/discarded by the typed program. We use $\{(l : \mathcal{M})\}; \mathcal{S}$ for $\{(l : \mathcal{M})\} \Cup \mathcal{S}$ if $l \notin \mathtt{dom}(\mathcal{S})$.

A **term type judgement** (resp. **state type judgment** and **configuration type judgment**) has the form $\Gamma \vdash^{(b,m,d)} t : \mathcal{T}$ (resp. $\Gamma \vdash^{(b,m,d)} s : \mathcal{S}$ and $\Gamma \vdash^{(b,m,d)} (t, s) : \kappa$) where b, m, d are three natural numbers, the first and second representing, respectively, the number of β-steps and $\mathtt{g}/\mathtt{s}$-steps needed to normalize t, and the third representing the size of the normal form of t. The **typing system** $\mathcal{P}$ is defined by the rules in Fig. 2. We write $\triangleright \mathcal{J}$ if there is a type derivation of the judgement $\mathcal{J}$ using the rules of system $\mathcal{P}$. The term t (resp. state s, configuration (t, s)) is **$\mathcal{P}$-typable** iff there is an environment Γ, a type $\mathcal{T}$ (resp. $\mathcal{S}$, κ) and counters (b, m, d) such that $\triangleright \Gamma \vdash^{(b,m,d)} t : \mathcal{T}$ (resp. $\triangleright \Gamma \vdash^{(b,m,d)} s : \mathcal{S}$, $\triangleright \Gamma \vdash^{(b,m,d)} (t, s) : \kappa$). As before, we use letters $\Phi, \Psi, \ldots$ to name type derivations.

Rules $(\mathtt{ax})$, (λ), $(\mathtt{m})$, and $(\mathtt{@})$ are essentially the same as in Fig. 1, but with types lifted to monadic types (*i.e. decorated* with state types). Rule $(\mathtt{@})$ assumes a value type associated to a value v on the left premise and a monadic type associated to a term t on the right premise. To type the application vt, it is necessary to match both the value type $\mathcal{M}$ inside the type of t with the input value type of v, and the output state type $\mathcal{S}'$ of t with the input state type of v. Rule $(\uparrow)$ is used to lift multi-types or tight constants $\mathtt{v}$ and $\mathtt{a}$ (the type of values) to monadic types. Rules $(\mathtt{get})$ and $(\mathtt{set})$ are used to type operations over the state. Rule $(\mathtt{emp})$ types empty states, rule $(\mathtt{upd})$ types states, and $(\mathtt{conf})$ types configurations.

A **type** τ is **tight**, if $\tau \in \mathtt{tt}$. We write $\mathtt{tight}(\mathcal{M})$ if every $\sigma \in \mathcal{M}$ is tight. A **state type** $\mathcal{S}$ is **tight** if $\mathtt{tight}(\mathcal{S}(l))$ holds for all $l \in \mathtt{dom}(\mathcal{S})$. A **configuration type** $\tau \times \mathcal{S}$ is **tight**, if τ and $\mathcal{S}$ are tight. A monadic type $\mathcal{S} \gg \kappa$ is **tight**, if κ is tight. The notion of tightness of type derivations is defined in the same way as in system $\mathcal{O}$, *i.e.* a **type derivation** Φ is **tight** if the type environment and the type of the conclusion of Φ are tight.

Rules for Terms

$$\frac{}{x : [\sigma] \vdash^{(0,0,0)} x : \sigma}\ (\mathtt{ax}) \qquad \frac{\Gamma \vdash^{(b,m,d)} v : \mu}{\Gamma \vdash^{(b,m,d)} v : \mathcal{S} \gg (\mu \times \mathcal{S})}\ (\uparrow)$$

$$\frac{\Gamma \vdash^{(b,m,d)} t : \mathcal{S} \gg \kappa}{\Gamma \setminus\!\!\setminus x \vdash^{(b,m,d)} \lambda x.t : \Gamma(x) \Rightarrow (\mathcal{S} \gg \kappa)}\ (\lambda) \qquad \frac{(\Gamma_i \vdash^{(b_i,m_i,d_i)} v : \sigma_i)_{i \in I}}{+_{i \in I} \Gamma_i \vdash^{(+_{i \in I} b_i, +_{i \in I} m_i, +_{i \in I} d_i)} v : [\sigma_i]_{i \in I}}\ (\mathtt{m})$$

$$\frac{\Gamma \vdash^{(b,m,d)} v : \mathcal{M} \Rightarrow (\mathcal{S}' \gg \kappa) \qquad \Delta \vdash^{(b',m',d')} t : \mathcal{S} \gg (\mathcal{M} \times \mathcal{S}')}{\Gamma + \Delta \vdash^{(1+b+b',m+m',d+d')} vt : \mathcal{S} \gg \kappa}\ (\mathtt{@})$$

$$\frac{\Gamma \vdash^{(b,m,d)} t : \mathcal{S} \gg \kappa}{\Gamma \setminus\!\!\setminus x \vdash^{(b,1+m,d)} \mathtt{get}_l(\lambda x.t) : \{(l : \Gamma(x))\} \Cup \mathcal{S} \gg \kappa}\ (\mathtt{get})$$

$$\frac{\Gamma \vdash^{(b,m,d)} v : \mathcal{M} \qquad \Delta \vdash^{(b',m',d')} t : \{(l : \mathcal{M})\}; \mathcal{S} \gg \kappa}{\Gamma + \Delta \vdash^{(b+b',1+m+m',d+d')} \mathtt{set}_l(v,t) : \mathcal{S} \gg \kappa}\ (\mathtt{set})$$

$$\frac{}{\vdash^{(0,0,0)} \lambda x.t : \mathtt{a}}\ (\lambda_{\mathtt{p}})$$

$$\frac{\Gamma \vdash^{(b,m,d)} t : \mathcal{S} \gg (\mathtt{tt} \times \mathcal{S}')}{(x : [\mathtt{v}]) + \Gamma \vdash^{(b,m,1+d)} xt : \mathcal{S} \gg (\mathtt{n} \times \mathcal{S}')}\ (\mathtt{@}_{\mathtt{p1}}) \qquad \frac{\Gamma \vdash^{(b,m,d)} u : \mathcal{S} \gg (\mathtt{n} \times \mathcal{S}')}{\Gamma \vdash^{(b,m,1+d)} (\lambda x.t)u : \mathcal{S} \gg (\mathtt{n} \times \mathcal{S}')}\ (\mathtt{@}_{\mathtt{p2}})$$

Rules for States

$$\frac{}{\vdash^{(0,0,0)} \epsilon : \emptyset}\ (\mathtt{emp}) \qquad \frac{\Gamma \vdash^{(b,m,d)} v : \mathcal{M} \qquad \Delta \vdash^{(b',m',d')} s : \mathcal{S}}{\Gamma + \Delta \vdash^{(b+b',m+m',d+d')} \mathtt{upd}_l(v,s) : \{(l : \mathcal{M})\}; \mathcal{S}}\ (\mathtt{upd})$$

Rule for Configurations

$$\frac{\Gamma \vdash^{(b,m,d)} t : \mathcal{S} \gg \kappa \qquad \Delta \vdash^{(b',m',d')} s : \mathcal{S}}{\Gamma + \Delta \vdash^{(b+b',m+m',d+d')} (t,s) : \kappa}\ (\mathtt{conf})$$

Fig. 2. Typing Rules for $\lambda_{\mathtt{gs}}$.

Example 4. Consider configuration c_0 from Ex. 2. Let $\mathcal{M} = [[\mathtt{v}] \Rightarrow \emptyset \gg (\mathtt{v} \times \emptyset)]$, and Φ be the following typing derivation:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{y : \mathcal{M} \vdash^{(0,0,0)} y : [\mathtt{v}] \Rightarrow \emptyset \gg (\mathtt{v} \times \emptyset)}\ (\mathtt{ax}) \qquad \cfrac{\cfrac{\cfrac{}{x : [\mathtt{v}] \vdash^{(0,0,0)} x : \mathtt{v}}\ (\mathtt{ax})}{x : [\mathtt{v}] \vdash^{(0,0,0)} x : [\mathtt{v}]}\ (\mathtt{m})}{x : [\mathtt{v}] \vdash^{(0,0,0)} x : \emptyset \gg ([\mathtt{v}] \times \emptyset)}\ (\uparrow)}{y : \mathcal{M}, x : [\mathtt{v}] \vdash^{(1,0,0)} yx : \emptyset \gg (\mathtt{v} \times \emptyset)}\ (\mathtt{@})}{x : [\mathtt{v}] \vdash^{(1,1,0)} \mathtt{get}_l(\lambda y.yx) : \{(l : \mathcal{M})\} \gg (\mathtt{v} \times \emptyset)}\ (\mathtt{get})}{\vdash^{(1,1,0)} \lambda x.\mathtt{get}_l(\lambda y.yx) : [\mathtt{v}] \Rightarrow (\{(l : \mathcal{M})\} \gg (\mathtt{v} \times \emptyset))}\ (\lambda)$$

And Φ' be the following typing derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : [\mathtt{v}] \vdash^{(0,0,0)} x : \mathtt{v}}\,(\mathtt{ax})}{x : [\mathtt{v}] \vdash^{(0,0,0)} x : \emptyset \gg (\mathtt{v} \times \emptyset)}\,(\uparrow)}{\vdash^{(0,0,0)} \mathtt{I} : [\mathtt{v}] \Rightarrow \emptyset \gg (\mathtt{v} \times \emptyset)}\,(\lambda)}{\vdash^{(0,0,0)} \mathtt{I} : \mathcal{M}}\,(\mathtt{m}) \qquad \dfrac{\dfrac{\dfrac{}{z : [\mathtt{v}] \vdash^{(0,0,0)} z : \mathtt{v}}\,(\mathtt{ax})}{z : [\mathtt{v}] \vdash^{(0,0,0)} z : [\mathtt{v}]}\,(\mathtt{m})}{z : [\mathtt{v}] \vdash^{(0,0,0)} z : \{(l : \mathcal{M})\} \gg ([\mathtt{v}] \times \{(l : \mathcal{M})\})}\,(\uparrow)$$
$$\overline{\qquad z : [\mathtt{v}] \vdash^{(0,1,0)} \mathtt{set}_l(\mathtt{I}, z) : \emptyset \gg ([\mathtt{v}] \times \{(l : \mathcal{M})\}) \qquad}\,(\mathtt{set})$$

Then we can build the following tight typing derivation Φ_c for c:

$$\dfrac{\dfrac{\Phi \qquad \Phi'}{z : [\mathtt{v}] \vdash^{(1,2,0)} (\lambda x.\mathtt{get}_l(\lambda y.yx))(\mathtt{set}_l(\mathtt{I}, z)) : \emptyset \gg (\mathtt{v} \times \emptyset)}\,(@) \qquad \dfrac{}{\vdash^{(0,0,0)} \epsilon : \emptyset}\,(\mathtt{emp})}{z : [\mathtt{v}] \vdash^{(1,2,0)} ((\lambda x.\mathtt{get}_l(\lambda y.yx))(\mathtt{set}_l(\mathtt{I}, z)), \epsilon) : \mathtt{v} \times \emptyset}\,(\mathtt{conf})$$

We will come back to this example at the end of Sec. 3.3.

3.3 Soundness and Completeness

In this section, we show the main properties of the type system $\mathcal{P}$ with respect to the operational semantics of the λ-calculus with global state introduced in Sec. 3.1. The properties of type system $\mathcal{P}$ are similar to the ones for $\mathcal{O}$, but now with respect to configurations instead of terms. *Soundness* does not only state that a (tightly) typable configuration (t, s) is terminating, but also gives exact (and split) measures concerning the reduction sequence from (t, s) to a final form. *Completeness* guarantees that a terminating configuration (t, s) is tightly typable, where the measures of the associated reduction sequence of (t, s) to final form are reflected in the counters of the resulting type derivation of (t, s). This is the first work providing a model for a language with global memory being able to count the number of memory accesses.

We start by noting that type system $\mathcal{P}$ does not type blocked configurations, which is exactly the notion that we want to capture.

Proposition 3. *If $\Phi \triangleright \Gamma \vdash^{(b,m,d)} (t, s) : \kappa$, then (t, s) is unblocked.*

We also show that counters capture the notion of normal form correctly, both for terms and states.

Lemma 1.

1. *Let $\Phi \triangleright \Gamma \vdash^{(0,0,d)} t : \delta$ be tight. Then, (1) $t \in \mathtt{no}$ and (2) $d = |t|$.*
2. *Let $\Phi \triangleright \Delta \vdash^{(0,0,d)} s : \mathcal{S}$ be tight. Then $d = 0$.*

In fact, we can show the following stronger property with respect to the counters for the number of β_v- and $\mathtt{g}/\mathtt{s}$-steps.

Lemma 2. *Let $\Phi \triangleright \Gamma \vdash^{(b,m,d)} t : \delta$ be tight. Then, $b = m = 0$ iff $t \in \mathtt{no}$.*

The following property is essential for tight type systems [2], and it shows that tightness of types spreads throughout type derivations of neutral terms, just as long as the environments are tight.

Lemma 3 (Tight Spreading). *Let $\Phi \triangleright \Gamma \vdash^{(b,m,d)} t : \mathcal{S} \gg (\tau \times \mathcal{S}')$, such that Γ is tight. If $t \in \mathtt{ne}$, then $\tau \in \mathtt{tt}$.*

The two following properties ensure tight typability of final configurations. For that we need to be able to *tightly* type any state, as well as any normal form. In fact, normal forms do not depend on a particular state since they are irreducible, so we can type them with any state type.

Lemma 4 (Typability of States and Normal Forms).

1. *Let s be a state. Then, there exists $\Phi \triangleright \vdash^{(0,0,0)} s : \mathcal{S}$ tight.*
2. *Let $t \in \mathtt{no}$. Then for any tight $\mathcal{S}$ there exists $\Phi \triangleright \Gamma \vdash^{(0,0,d)} t : \mathcal{S} \gg (\mathtt{tt} \times \mathcal{S})$ tight s.t. $d = |t|$.*

Finally, we state the usual basic properties.

Lemma 5 (Substitution and Anti-Substitution).

1. **(Substitution)** *If $\Phi_t \triangleright \Gamma_t; x : \mathcal{M} \vdash^{(b_t,m_t,d_t)} t : \delta$ and $\Phi_v \triangleright \Gamma_v \vdash^{(b_v,m_v,d_v)} v : \mathcal{M}$, then $\Phi_{t\{x\backslash v\}} \triangleright \Gamma_t + \Gamma_v \vdash^{(b_t+b_v,m_t+m_v,d_t+d_v)} t\{x\backslash v\} : \delta$.*
2. **(Anti-Substitution)** *If $\Phi_{t\{x\backslash v\}} \triangleright \Gamma_{t\{x\backslash v\}} \vdash^{(b,m,d)} t\{x\backslash v\} : \delta$, then $\Phi_t \triangleright \Gamma_t; x : \mathcal{M} \vdash^{(b_t,m_t,d_t)} t : \delta$ and $\Phi_v \triangleright \Gamma_v \vdash^{(b_v,m_v,d_v)} v : \mathcal{M}$, such that $\Gamma_{t\{x\backslash v\}} = \Gamma_t + \Gamma_v$, $b = b_t + b_v$, $m = m_t + m_v$, and $d = d_t + d_v$.*

Lemma 6 (Split Exact Subject Reduction and Expansion).

1. **(Subject Reduction)** *Let $(t, s) \rightarrow_{\mathtt{r}} (u, q)$. If $\Phi \triangleright \Gamma \vdash^{(b,m,d)} (t, s) : \kappa$ is tight, then $\Phi' \triangleright \Gamma \vdash^{(b',m',d)} (u, q) : \kappa$, where $\mathtt{r} = \beta$ implies $b' = b - 1$ and $m' = m$, while $\mathtt{r} \in \{\mathtt{g}, \mathtt{s}\}$ implies $b' = b$ and $m' = m - 1$.*
2. **(Subject Expansion)** *Let $(t, s) \rightarrow_{\mathtt{r}} (u, q)$. If $\Phi' \triangleright \Gamma \vdash^{(b',m',d)} (u, q) : \kappa$ is tight, then $\Phi \triangleright \Gamma \vdash^{(b,m,d)} (t, s) : \kappa$, where $\mathtt{r} = \beta$ implies $b' = b - 1$ and $m' = m$, while $\mathtt{r} \in \{\mathtt{g}, \mathtt{s}\}$ implies $b' = b$ and $m' = m - 1$.*

Soundness (resp. completeness) is based on exact subject reduction (resp. expansion), in turn based on the previous substitution (resp. anti-substitution) lemma.

Theorem 1 (Quantitative Soundness and Completeness).

1. **(Soundness)** *If $\Phi \triangleright \Gamma \vdash^{(b,m,d)} (t, s) : \kappa$ tight, then there exists (u, q) such that $u \in \mathtt{no}$ and $(t, s) \twoheadrightarrow^{(b,m)} (u, q)$ with b β-steps, m $\mathtt{g}/\mathtt{s}$-steps, and $|(u, q)| = d$.*
2. **(Completeness)** *If $(t, s) \twoheadrightarrow^{(b,m,d)} (u, q)$ and $u \in \mathtt{no}$, then there exists $\Phi \triangleright \Gamma \vdash^{(b,m,|(u,q)|)} (t, s) : \kappa$ tight.*

Example 5. Consider again configuration c_0 from Ex. 2 and its associated tight derivation Φ_{c_0}. The first two counters of Φ_c are different from 0: this means that c is not a final configuration, but normalizes in one β_v-step ($b = 1$) and two $\mathtt{g}/\mathtt{s}$-steps ($m = 2$), to a final configuration having size $d = 0 = |z| = |(z, \mathtt{upd}_l(I, \epsilon))|$.

4 Conclusion and Related Work

This paper provides a foundational step into the development of quantitative models for programming languages with effects. We focus on a simple language with global memory access capabilities. Due to the inherent lack of confluence in such framework we fix a particular evaluation strategy following a (weak) CBV approach. We provide a type system for our language that is able to (both) extract and discriminate between (exact) measures for the length of evaluation, number of memory accesses and size of normal forms. This study provides a valuable insight into time and space analysis of languages with global memory, with respect to length of evaluation and the size of normal forms, respectively.

In future work we would like to explore effectful computations involving global memory in a more general framework being able to capture different models of computation, such as the CBPV [28] or the bang calculus [9]. Furthermore, we would like to apply our quantitative techniques to other effects that can be found in programming languages, such as non-termination, exceptions, non-determinism, and I/O.

Related Work. Several papers proposed quantitative approaches for different notions of CBV (without effects). But none of them exploits the idea of exact *and* split tight typing. Indeed, the first non-idempotent intersection type system for Plotkin's CBV is [18], where reduction is allowed under abstractions, and terms are considered to be closed. This work was further extended to [11], where commutation rules are added to the calculus. None of these contributions extracts quantitative bounds from the type derivations. A calculus for open CBV is proposed in [3], where *fireball* –normal forms– can be either erased or duplicated. Quantitative results are obtained, but no split measures. Other similar approaches appear in [23]. A logical characterization of CBV solvability is given in [4], the resulting non-idempotent system gives quantitative information of the *solvable* associated reduction relation. A similar notion of solvability for CBV for generalized applications was studied in [26], together with a logical characterization provided by a quantitative system. Other non-idempotent systems for CBV were proposed [29,25], but they are defective in the sense that they do not enjoy subject reduction and expansion. Split measures for (strong) open CBV are developed in [27].

In [17], a system with universally quantified intersection and reference types is introduced for a language belonging to the ML-family. However, intersections are idempotent and only (qualitative) soundness is proved.

More recently, there has been a lot of work involving probabilistic versions of the lambda calculus. In [20], extensions of the lambda calculus with a probabilistic choice operator are introduced. However, no quantitative results are provided. In [8], monadic intersection types are used to obtain a (non-exact) quantitative model for a probabilistic calculus identical to the one in [20].

Concerning (exact) quantitative models for programming languages with global state, the state of the art is still underexplored. Some sound but not complete approaches are given in [7,15], and quantitative results are not provided. Our work is inspired by a recent idempotent (thus only qualitative and not quantitative) model for CBV with global memory proposed by [16]. This work was further extended in [21] to a more generic framework of algebraic effectful computation, still the model does not provide any quantitative information about the evaluation of programs and the size of their results.

References

1. Accattoli, B.: Proof nets and the call-by-value λ-calculus. Theor. Comput. Sci. **606**, 2–24 (2015). `https://doi.org/10.1016/j.tcs.2015.08.006`
2. Accattoli, B., Graham-Lengrand, S., Kesner, D.: Tight typings and split bounds, fully developed. J. Funct. Program. **30**(e14), 1–101 (2020). `https://doi.org/10.1017/S095679682000012X`
3. Accattoli, B., Guerrieri, G.: Types of fireballs. In: 16th Asian Symposium Programming Languages and Systems, (APLAS), 2018, Wellington, New Zealand. Lecture Notes in Computer Science, vol. 11275, pp. 45–66. Springer (2018). `https://doi.org/10.1007/978-3-030-02768-1_3`
4. Accattoli, B., Guerrieri, G.: The theory of call-by-value solvability. Proc. ACM Program. Lang. **6**(ICFP), 855–885 (2022). `https://doi.org/10.1145/3547652`
5. Alves, S., Kesner, D., Ramos, M.: Quantitative global memory (2023), `https://arxiv.org/abs/2303.08940`
6. Atkey, R.: Syntax and semantics of quantitative type theory. In: Dawar, A., Grädel, E. (eds.) Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, (LICS), 2018, Oxford, UK. pp. 56–65. ACM (2018). `https://doi.org/10.1145/3209108.3209189`
7. Benton, N., Kennedy, A., Beringer, L., Hofmann, M.: Relational semantics for effect-based program transformations: higher-order store. In: Porto, A., López-Fraguas, F.J. (eds.) 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, (PPDP), 2009, Coimbra, Portugal. pp. 301–312. ACM (2009). `https://doi.org/10.1145/1599410.1599447`, `https://doi.org/10.1145/1599410.1599447`
8. Breuvart, F., Lago, U.D.: On intersection types and probabilistic lambda calculi. In: Sabel, D., Thiemann, P. (eds.) Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, (PPDP), 2018, Frankfurt am Main, Germany. pp. 8:1–8:13. ACM (2018). `https://doi.org/10.1145/3236950.3236968`
9. Bucciarelli, A., Kesner, D., Ríos, A., Viso, A.: The bang calculus revisited. In: Functional and Logic Programming, 15th International Symposium, (FLOPS), 2020, Akita, Japan, Proceedings. Lecture Notes in Computer Science, vol. 12073, pp. 13–32. Springer (2020). `https://doi.org/10.1007/978-3-030-59025-3_2`
10. Bucciarelli, A., Kesner, D., Ventura, D.: Non-idempotent intersection types for the lambda-calculus. Log. J. (IGPL) **25**(4), 431–464 (2017). `https://doi.org/10.1093/jigpal/jzx018`
11. Carraro, A., Guerrieri, G.: A semantical and operational account of call-by-value solvability. In: 17th International Conference on Foundations of Software Science and Computation Structures, (FOSSACS), 2014, Grenoble, France. Lecture Notes in Computer Science, vol. 8412, pp. 103–118. Springer (2014). `https://doi.org/10.1007/978-3-642-54830-7_7`
12. de Carvalho, D.: Sémantiques de la logique linéaire et temps de calcul. These de doctorat, Université Aix-Marseille II (2007)

13. de Carvalho, D.: Execution time of λ-terms via denotational semantics and intersection types. Math. Struct. Comput. Sci. **28**(7), 1169–1203 (2018). `https://doi.org/10.1017/S0960129516000396`
14. Coppo, M., Dezani-Ciancaglini, M.: A new type assignment for lambda-terms. Archiv für Math. Logik **19**, 139–156 (1978)
15. Davies, R., Pfenning, F.: Intersection types and computational effects. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, (ICFP), 2000, Montreal, Canada. pp. 198–208. ACM (2000). `https://doi.org/10.1145/351240.351259`
16. de'Liguoro, U., Treglia, R.: Intersection types for a λ-calculus with global store. In: Veltri, N., Benton, N., Ghilezan, S. (eds.) 23rd International Symposium on Principles and Practice of Declarative Programming, (PPDP), 2021, Tallinn, Estonia. pp. 5:1–5:11. ACM (2021). `https://doi.org/10.1145/3479394.3479400`
17. Dezani-Ciancaglini, M., Giannini, P., Rocca, S.R.D.: Intersection, universally quantified, and reference types. In: 18th Annual Conference of the EACSL on Computer Science Logic, 23rd international Workshop, (CSL), 2009, Coimbra, Portugal. Proceedings. Lecture Notes in Computer Science, vol. 5771, pp. 209–224. Springer (2009). `https://doi.org/10.1007/978-3-642-04027-6_17`
18. Ehrhard, T.: Collapsing non-idempotent intersection types. In: 26th International Workshop/21st Annual Conference of the EACSL on Computer Science Logic, (CSL), 2012, Fontainebleau, France. LIPIcs, vol. 16, pp. 259–273. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2012). `https://doi.org/10.4230/LIPIcs.CSL.2012.259`
19. Faggian, C., Guerrieri, G., de'Liguoro, U., Treglia, R.: On reduction and normalization in the computational core. Math. Struct. Comput. Sci. **32**(7), 934–981 (2022). `https://doi.org/10.1017/S0960129522000433`
20. Faggian, C., Rocca, S.R.D.: Lambda calculus and probabilistic computation. In: 34th Annual ACM/IEEE Symposium on Logic in Computer Science, (LICS), 2019, Vancouver, BC, Canada. pp. 1–13. IEEE (2019). `https://doi.org/10.1109/LICS.2019.8785699`
21. Gavazzo, F., Vanoni, G., Treglia, R.: On monadic intersection types (2023), draft
22. Girard, J.: Linear logic. Theor. Comput. Sci. **50**, 1–102 (1987). `https://doi.org/10.1016/0304-3975(87)90045-4`
23. Guerrieri, G.: Towards a semantic measure of the execution time in call-by-value lambda-calculus. In: 12th Workshop on Developments in Computational Models and Ninth Workshop on Intersection Types and Related Systems, (DCM/ITRS), 2018, Oxford, UK. EPTCS, vol. 293, pp. 57–72 (2018). `https://doi.org/10.4204/EPTCS.293.5`
24. Jones, S.L.P., Wadler, P.: Imperative functional programming. In: Conference Record of the Twentieth Annual (ACM) (SIGPLAN-SIGACT) Symposium on Principles of Programming Languages, (POPL), 1993, Charleston, South Carolina, USA. pp. 71–84. ACM Press (1993). `https://doi.org/10.1145/158511.158524`
25. Kerinec, A., Manzonetto, G., Rocca, S.R.D.: Call-by-value, again! In: 6th International Conference on Formal Structures for Computation and Deduction, (FSCD), 2021, Buenos Aires, Argentina (Virtual Conference). LIPIcs, vol. 195, pp. 7:1–7:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). `https://doi.org/10.4230/LIPIcs.FSCD.2021.7`
26. Kesner, D., Peyrot, L.: Solvability for generalized applications. In: 7th International Conference on Formal Structures for Computation and Deduction, (FSCD), 2022, Haifa, Israel. LIPIcs, vol. 228, pp. 18:1–18:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). `https://doi.org/10.4230/LIPIcs.FSCD.2022.18`

27. Kesner, D., Viso, A.: Encoding tight typing in a unified framework. In: 30th EACSL Annual Conference on Computer Science Logic, (CSL), 2022, Göttingen, Germany (Virtual Conference). LIPIcs, vol. 216, pp. 27:1–27:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). `https://doi.org/10.4230/LIPIcs.CSL.2022.27`
28. Levy, P.B.: Call-by-push-value: A subsuming paradigm. In: Typed Lambda Calculi and Applications, 4th International Conference, (TLCA'99), 1999, L'Aquila, Italy, Proceedings. Lecture Notes in Computer Science, vol. 1581, pp. 228–242. Springer (1999). `https://doi.org/10.1007/3-540-48959-2_17`
29. Manzonetto, G., Pagani, M., Rocca, S.R.D.: New semantical insights into call-by-value λ-calculus. Fundamenta Informaticae **170**(1-3), 241–265 (2019). `https://doi.org/10.3233/fi-2019-1862`
30. Moggi, E.: Computational lambda-calculus and monads. In: 4th Annual Symposium on Logic in Computer Science, (LICS), 1989, Pacific Grove, California, USA. pp. 14–23. IEEE Computer Society (1989). `https://doi.org/10.1109/LICS.1989.39155`
31. Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. Theor. Comput. Sci. **1**(2), 125–159 (1975). `https://doi.org/10.1016/0304-3975(75)90017-1`
32. Sabry, A., Wadler, P.: A reflection on call-by-value. ACM Trans. Program. Lang. Syst. **19**(6), 916–941 (1997). `https://doi.org/10.1145/267959.269968`
33. Treglia, R.: The computational core: reduction theory and intersection type discipline. Phd thesis, Università di Torino (2022)
34. Wadler, P.: Monads for functional programming. In: 1st International Spring School on Advanced Functional Programming Techniques on Advanced Functional Programming, (AFP), 1995, Båstad, Sweden, Tutorial Text. Lecture Notes in Computer Science, vol. 925, pp. 24–52. Springer (1995). `https://doi.org/10.1007/3-540-59451-5_2`