Implicit Computational Complexity of Subrecursive Definitions and Applications to Cryptographic Proofs (Long version)

Patrick Baillot, Gilles Barthe, Ugo Dal Lago

Implicit Computational Complexity of Subrecursive Definitions and Applications to Cryptographic Proofs*

Long version

Patrick Baillot[1], Gilles Barthe[2], and Ugo Dal Lago[3]

[1] CNRS, ENS de Lyon, INRIA, UCBL, Université de Lyon, LIP
[2] IMDEA Software Institute
[3] Università di Bologna & INRIA

Abstract. We define a call-by-value variant of Gödel's System T with references, and equip it with a linear dependent type and effect system, called dℓT, that can estimate the complexity of programs, as a function of the size of their inputs. We prove that the type system is intentionally sound, in the sense that it over-approximates the complexity of executing the programs on a variant of the CEK abstract machine. Moreover, we define a sound and complete type inference algorithm which critically exploits the subrecursive nature of dℓT. Finally, we demonstrate the usefulness of dℓT for analyzing the complexity of cryptographic reductions by providing an upper bound for the constructed adversary of the Goldreich-Levin theorem.

1 Introduction

The goal of modern cryptography is to design primitives and protocols that are provably secure against computationally bounded adversaries. Initially focused on the design of secure encryption schemes and key exchange protocols, the scope of modern cryptography has expanded to consider a wide range of primitives and protocols that achieve such goals as zero-knowledge proofs of knowledge, multi-party computation, verifiable computation, group key exchange, *etc*. To meet such goals, cryptographic constructions have become more complex, and this complexity is reflected in their security analysis. As a consequence, these proofs are lengthy, error-prone, and cannot easily be subjected to independent verification. As a consequence, a number of flaws have been discovered in published proofs. A plausible approach to eliminate such flaws and to ensure independent verifiability of proofs is to use computer-aided tools [16]. This approach has been explored with great success, leading to the development of several tools, like CryptoVerif [11], CertiCrypt [8], EasyCrypt [9, 5], or RF* [6] that have been used to reason about the security of several emblematic examples of cryptographic constructions. However, most of these tools only support partial verification of cryptographic reductions. Specifically, cryptographic reductions are proved in three steps: definition of a constructed adversary, proof of correctness of the reduction, and complexity analysis of the constructed adversary[4]; however, most tools consider only the first and second steps.

Implicit Computational Complexity (ICC) is a thriving field which develops foundational methods for estimating program complexity, often in the setting of pure higher-order languages. Many works in ICC use sophisticated type systems to estimate the complexity of programs, and these systems have many similarities with those used in computer-aided cryptography. In particular, there are close connections between dℓPCF [23, 24], an expressive type system for analyzing the complexity of PCF-expressions, and type systems for computer-aided cryptography, such as RF* [6], and its non-relational variant F* [28]. In particular, these type systems use restricted forms of linear dependent types, and programs can be type-checked by computing a set of verification conditions that can be discharged by SMT solvers. Thus, an integration of ICC type systems into tools for computer-aided cryptography might sound unproblematic at first sight. However, the match is less than ideal, even for closely related systems such as F*/RF* and dℓPCF. Indeed,

* This work has been partially supported by ANR Project ELICA ANR-14-CE25-0005.

[4] The goal of the complexity analysis is almost always to provide an upper bound on the execution time of the adversary. Only in few cases the analysis is concerned with the expected execution time. In this paper, we only focus on the first case.

the type inference algorithm of dℓPCF outputs a so-called equational program, which must be proved terminating, in order to guarantee the meaningfulness of the complexity bounds delivered by the algorithm. Proving the termination of equational programs can be extremely difficult, even for relatively simple expressions, and is specially frustrating when the expressions from which the programs are generated are clearly terminating. In particular, dℓPCF is overly expressive for cryptographic reductions, as constructed adversaries can be expressed in a subrecursive language, in the style of Gödel's system T, without the need to resort to the generality of fixpoint definitions. Because programs written in such a language are necessarily terminating, one can hope to develop an automated method which does not suffer from the drawback of dℓPCF, i.e. does not require to prove termination of an equational program. More generally, existing systems for ICC suffer one or several of the following shortcomings: they do not support stateful computations; they deliver asymptotic bounds, rather than concrete bounds which are useful for practice-oriented provable security; they can only analyze hereditarily polytime programs, i.e. programs whose sub-computations are also polytime; they lack sound and complete type inference algorithms—for instance, some systems have conditionally sound and complete type inference algorithms, i.e. a complex analysis of the output of the type inference algorithm is required to establish the validity of its results. As a consequence, these type systems are not appropriate for being used for computer-aided cryptography.

Contributions The main contribution of this paper is a type-based complexity analysis for a call-by-value, stateful, higher-order language with primitive recursion in the style of Gödel's system T. The language is sufficiently expressive to model constructed adversaries from the cryptographic literature, and yet sufficiently constrained to define a sound and complete type inference algorithm. Technically, our contributions are:

– we introduce in Section 2 a variant of Gödel's system T—a paradigmatic higher-order language with inductive types and higher-order primitive recursion—with references and use a variant of the CEK abstract machine to characterize the complexity of programs;

– we define in Section 3 a type system dℓT which conservatively approximates the cost of executing a program, and then prove its intensional soundness w.r.t. the cost of its execution using our variant of the CEK abstract machine. The key ingredients of our type system are: *linear types*, which we use to ensure that higher-order subexpressions cannot be duplicated; *indexed types*, which we use to keep track of the size of expressions—thus, our use of indexed types is somewhat different from other works on refinement types, which support finer-grained assertions about expressions, e.g. assertions about their values; *indexed effects*, which are used to track the size of references throughout computation;

– we show in Section 4 that dℓT is intensionally sound, i.e. correctly overapproximates the complexity of evaluating typable programs;

– we define in Section 5 a type inference algorithm and prove its soundness and completeness. Our algorithm critically exploits the constrained form of programs, in which all recursive definitions are performed through recursors, to deliver an equational program that is *provably* terminating;

– we demonstrate in Section 7 that dℓT captures plaintext extractors that arise in reduction proofs of padding-based encryption schemes, i.e. public-key encryption schemes built from one-way trapdoor permutations and random oracles, and constructed adversaries in the Goldreich-Levin theorem, which proves the existence of hardcore predicates. The latter example is particularly challenging, since it involves computations that are not hereditiraly polytime.

From a theoretical perspective, our work provides an answer to the challenging question of achieving sound and complete type inference algorithms for expressive ICC systems featuring both higher-order and references. On the other hand, the system is kept simple enough to avoid the necessity of checking equational programs for termination, as explained in Section 5.1. All this, in particular, means that dℓT is not merely a simplification on dℓPCF [23, 24]. From a practical point of view, our work opens the perspective to build tools that account for the correctness *and* complexity analysis of cryptographic reductions, without any additional cost for the user, other than specifying the cost of primitive operations. Implementation of dℓT and its potential integration in F*/RF* (or a related system) is left for future work, but we briefly discuss the principles of such an integration in Section 9.

2 Setting

We consider a simply typed λ-calculus with references and higher-order primitive recursion, with a call-by-value evaluation strategy. For the sake of readability, we consider a minimalistic language with natural numbers, booleans, and lists. For the sake of applicability, we allow the set of expressions to be parameterized by a set of function symbols; these functions can be used in cryptographic applications to model random sampling, one-way trapdoor permutations, oracles, *etc*.

The semantics of programs is defined by an abstract machine, which we use to characterize the complexity of programs. We assume that function symbols come equipped with a semantics, and with a cost; thus, the abstract machine and the associated cost of programs are parametrized by the semantics and cost functions. This is formalized through the notion of a *function setting*, which we defined below.

Note that it is also possible to define the semantics of programs using a reduction semantics; we define such a reduction semantics and prove subject reduction w.r.t. our linear dependent type system in Section 2.4.

2.1 Language

We assume given denumerable sets $\mathbb{V}$ of *variables* and $\mathbb{L}$ of locations, and a set $\mathcal{F}$ of function symbols. *Terms* and *values* are defined as follows in Figure 1. The constructions $\mathtt{map}(V)$ and $\mathtt{map}_2(V)$ are defined in a standard way, using $\mathtt{fold}(X, Z)$ for suitable X, Z. The following constructions used in the examples are also standard syntactic sugar: $\mathtt{if}\ M\ \mathtt{then}\ V\ \mathtt{else}\ W$, $\lambda\langle x, y\rangle.M$, $\mathtt{let}\ x = M\ \mathtt{in}\ N$, as well as $M; N$. The expression $\mathbf{M}$ stands for a sequence of terms.

$$
\begin{array}{rcll}
M & ::= & V & \text{(Value)} \\
 & \mid & x & \text{(Variable)} \\
 & \mid & \mathtt{succ}(M) & \text{(Succ)} \\
 & \mid & \mathtt{cons}(M, N) & \text{(Cons)} \\
 & \mid & \mathtt{let}\ M\ \mathtt{be}\ \langle x, y\rangle\ \mathtt{in}\ N & \text{(Let)} \\
 & \mid & \mathtt{f}(M) & \text{(Function)} \\
 & \mid & M\ N & \text{(Application)} \\
 & \mid & !\, r & \text{(Dereference)} \\
 & \mid & r := M & \text{(Assign)}
\end{array}
\qquad
\begin{array}{rcll}
V & ::= & * & \text{(Unit)} \\
 & \mid & \mathbf{zero} & \text{(Zero)} \\
 & \mid & \mathbf{tt} & \text{(True)} \\
 & \mid & \mathbf{ff} & \text{(False)} \\
 & \mid & \mathbf{nil} & \text{(Nil)} \\
 & \mid & \mathbf{succ}(V) & \text{(Succ)} \\
 & \mid & \mathbf{cons}(V, W) & \text{(Cons)} \\
 & \mid & \langle M, N\rangle & \text{(Pair)} \\
 & \mid & \lambda x.M & \text{(Abstraction)} \\
 & \mid & \mathtt{iter}(V, W) \mid \mathtt{fold}(V, W) & \text{(Iterators)} \\
 & \mid & \mathtt{if}(V, W) \mid \mathtt{ifz}(V, W) & \\
 & \mid & \mathtt{ifn}(V, W) & \text{(Selectors)}
\end{array}
$$

Fig. 1. Terms and Values

2.2 Linear Type System

We first equip the language with a (non-dependent) linear type system. The sets of base types and types are defined as follows:

$$T\ ::=\ \mathsf{unit} \mid \mathbf{B} \mid \mathbf{N} \mid \mathbf{L}(T); \qquad A, B\ ::=\ T \mid A \otimes A \mid A \overset{a}{\multimap} A;$$

where a ranges over sets of locations. The types $\mathbf{B}$, $\mathbf{N}$ and $\mathbf{L}(T)$ stand respectively for booleans, integers, and lists over the type T. If a is the empty set, we write $A \multimap B$ for $A \overset{a}{\multimap} B$. *First-order types* are types in which for any subformula $A \overset{a}{\multimap} B$, A does not contain any $\multimap$ connective. Each function symbol $\mathtt{f}$ is

assumed to have an *input type* $T_{\mathtt{f}}$ and an *output type* $S_{\mathtt{f}}$. The set $\mathcal{T}(T)$ of those values which can be given type T can be easily defined by induction on the structure of T.

Variable contexts (resp. *reference contexts*) are denoted as Γ (resp. Θ) and are of the shape $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ (resp. $\Theta = r_1 : A_1, \ldots, r_n : A_n$). A *ground variable context* is a variable context in the form $\{x_1 : T_1, \ldots, x_n : T_n\}$, and is denoted with metavariables like $\ell\Gamma$. *Ground reference contexts* are defined similarly and are denoted with metavariables like $\ell\Theta$.

Typing judgements are of the form $\Gamma; \Theta \vdash M : A; a$. This judgement means that when assigning to free variables types as from Γ and to references types as from Θ, the term M can be given type A, and during its evaluation the set of references that might be read is included in a. The union $\Gamma \uplus \Delta$ of variable contexts is defined only if the variables in common are attributed *the same* base type. Similarly the union $a \uplus b$ of sets of locations (in a judgement) is defined only if the locations in common are attributed the same base type in the reference context Θ of the judgement.

The typing rules are displayed on Figure 2, and Figure 3. Let us just comment on a few rules:

- *Assign and Dereference (Fig. 2)*: note that in the Dereference rule the reference r must belong to the set a, while in the Assign rule there is no condition on a but r and M must have the same type.
- *Pair (Fig. 2)*: note that the context $\Gamma \uplus \Delta$ implies that if M and N share a variable x, then this variable must have a base type; similarly if they can both read a reference r, then this reference must be in a, b and $a \uplus b$, and hence have a base type. The (Application) rule is similar, but one also needs to take into account the set a of references that can be read when M is applied to an argument.
- *Abstraction (Fig. 2)*: note how, reading the rule top-down, the set a is "moved" from the judgement to the type $A \xrightarrow{a} B$, and can in this way be used later in the derivation if $\lambda x.M$ is applied to an argument.
- *Iteration (Fig. 3)*: in the rules for $\mathtt{iter}(V, W)$ and $\mathtt{fold}(V, W)$ the variable context $\ell\Gamma$ and the reference context $\ell\Theta$ can only contain base types.

$$\frac{}{\Gamma, x : A; \Theta \vdash x : A; a} \qquad \frac{}{\Gamma; \Theta \vdash * : \mathsf{unit}; a} \qquad \frac{t \in \mathcal{T}(T)}{\Gamma; \Theta \vdash t : T; a} \qquad \frac{\Gamma; \Theta \vdash M : T_{\mathtt{f}}; a}{\Gamma; \Theta \vdash \mathtt{f}(M) : S_{\mathtt{f}}; a}$$

$$\frac{\Gamma; \Theta \vdash M : A; a \qquad \Delta; \Theta \vdash N : B; b}{\Gamma \uplus \Delta; \Theta \vdash \langle M, N\rangle : A \otimes B; a \uplus b} \qquad \frac{\Gamma; \Theta \vdash M : A \otimes B; a \qquad \Delta, x : A, y : B; \Theta \vdash N : C; b}{\Gamma \uplus \Delta; \Theta \vdash \mathtt{let}\ M\ \mathtt{be}\ \langle x, y\rangle\ \mathtt{in}\ N : C; a \uplus b}$$

$$\frac{\Gamma, x : A; \Theta \vdash M : B; a}{\Gamma : \Theta \vdash \lambda x.M : A \xrightarrow{a} B; b} \qquad \frac{\Gamma; \Xi \vdash M : A \xrightarrow{a} B; b \qquad \Delta; \Xi \vdash N : A; c}{\Gamma \uplus \Delta; \Xi \vdash MN : B; a \uplus b \uplus c}$$

$$\frac{r \in a}{\Gamma; \Theta, r : A \vdash\ !\, r : A; a} \qquad \frac{\Gamma; \Theta, r : A \vdash M : A; a}{\Gamma; \Theta, r : A \vdash r\mathtt{:=}M : \mathsf{unit}; a}$$

Fig. 2. Typing Rules, Part I

Consider the following two examples:

$$M = r\mathtt{:=}\mathbf{zero}; \mathbf{cons}(!\, r, \mathbf{cons}(!\, r, \mathbf{nil})); \qquad N = r\mathtt{:=}\lambda x.x;\ !\, r(!\, r*).$$

Both terms read a reference r twice, but M is typable, while N is not. Indeed, in M the reference r is read twice, but it is of base type $\mathbf{N}$; we can derive:

$$\emptyset; r : \mathbf{N} \vdash M : \mathbf{L}(\mathbf{N}); \{r\}$$

On the contrary an attempt to type N fails because of the rule for Application and the condition on the sets of locations, since r does not have a base type.

Let us now give another example for computing addition and multiplication on natural numbers, in an imperative style:

$$\begin{array}{lll}
incr_r & = \lambda x.r\mathtt{:=}\mathtt{succ}(!\, r) & \text{(increments the content of } r) \\
add_{r,r_1} & = \lambda x.\mathtt{iter}(incr_r, *)\, !\, r_1 & \text{(adds the content of } r_1 \text{ to that of } r) \\
mult_{r,r_1,r_2} & = \lambda x.(r\mathtt{:=}0; \mathtt{iter}(add_{r,r_1}, *)\, !\, r_2) & \text{(multiplies the contents of } r_1 \text{ and } r_2 \text{ and assigns the result to } r)
\end{array}$$

$$\frac{}{\Gamma;\Theta \vdash \mathtt{tt} : \mathbf{B}; a} \qquad \frac{}{\Gamma;\Theta \vdash \mathtt{ff} : \mathbf{B}; a} \qquad \frac{}{\Gamma;\Theta \vdash \mathbf{zero} : \mathbf{N}; a} \qquad \frac{}{\Gamma;\Theta \vdash \mathbf{nil} : \mathbf{L}(A); a}$$

$$\frac{\Gamma;\Theta \vdash M : \mathbf{N}; a}{\Gamma;\Theta \vdash \mathtt{succ}(M) : \mathbf{N}; a} \qquad \frac{\Gamma;\Theta \vdash M : T; a \qquad \Delta;\Theta \vdash N : \mathbf{L}(T); b}{\Gamma \uplus \Delta;\Theta \vdash \mathtt{cons}(M,N) : \mathbf{L}(T); a \uplus b}$$

$$\frac{\Gamma;\Theta \vdash W : A; a \qquad \Gamma;\Theta \vdash V : A; a}{\Gamma;\Theta \vdash \mathtt{if}(V,W) : \mathbf{B} \stackrel{a}{\multimap} A; b} \qquad \frac{\ell\Gamma;\ell\Theta \vdash W : A; a \qquad \ell\Gamma;\ell\Theta \vdash V : A \stackrel{c}{\multimap} A; a}{\ell\Gamma;\ell\Theta \vdash \mathtt{iter}(V,W) : \mathbf{N} \stackrel{c}{\multimap} A; b}$$

$$\frac{\Gamma;\Theta \vdash W : A; a \qquad \Gamma;\Theta \vdash V : \mathbf{N} \stackrel{c}{\multimap} A; a}{\Gamma;\Theta \vdash \mathtt{ifz}(V,W) : \mathbf{N} \stackrel{c}{\multimap} A; b} \qquad \frac{\ell\Gamma;\ell\Theta \vdash W : A; a \qquad \ell\Gamma;\ell\Theta \vdash V : T \stackrel{c}{\multimap} A \stackrel{c}{\multimap} A; a}{\ell\Gamma;\ell\Theta \vdash \mathtt{fold}(V,W) : \mathbf{L}(T) \stackrel{c}{\multimap} A; b}$$

$$\frac{\Gamma;\Theta \vdash W : A; a \qquad \Gamma;\Theta \vdash V : \mathbf{L}(T) \stackrel{c}{\multimap} T \stackrel{c}{\multimap} A; a}{\Gamma;\Theta \vdash \mathtt{ifn}(V,W) : \mathbf{L}(T) \stackrel{c}{\multimap} A; b}$$

Fig. 3. Typing Rules, Part II

The language we have just defined, then, can be seen as an affine variation on Gödel's T with pairs, references, and inductive types, called $\ell\mathsf{T}$. The just introduced type system, however, does not provide any guarantee as for the time complexity of typable program. Something much more refined is necessary.

2.3 Function Settings

The behavior of functions is not specified *a priori*, because such functions are meant to model calls to oracles in cryptography. Everything in the following, then, will be parametrized by a so-called *function setting*, which is a pair $(\{\mathbf{S}_\mathtt{f}\}_\mathtt{f}, \{\mathbf{C}_\mathtt{f}\}_\mathtt{f})$, where $\mathbf{S}_\mathtt{f}$ is a relation between base type values matching $\mathtt{f}$'s input and output types and modeling its (possibly probabilistic) extensional behaviour, while $\mathbf{C}_\mathtt{f}$ is a function from $\mathbb{N}$ to $\mathbb{N}$ expressing a bound to the cost evaluating $\mathtt{f}$ on arguments of a given length. In the rest of this paper, we assume that a function setting has been fixed, keeping in mind that type inference can be done independently on a specific function setting, as we will explain in Section 5.

2.4 Reduction Semantics

The simplest way of specifying how terms evaluate is to give a small-step semantics. Evaluation takes place in *evaluation contexts*, which are generated by the following grammar:

$$\begin{aligned}\mathsf{E} ::= {} & [\cdot] \mid \mathbf{succ}(\mathsf{E}) \mid \mathbf{cons}(\mathsf{E}, M) \mid \mathbf{cons}(V, \mathsf{E}) \mid \mathtt{f}(\mathsf{E}) \\ & \mid \mathtt{let}\ \mathsf{E}\ \mathtt{be}\ \langle x, y\rangle\ \mathtt{in}\ N \mid \mathtt{let}\ \langle \mathsf{E}, N\rangle\ \mathtt{be}\ \langle x, y\rangle\ \mathtt{in}\ N \\ & \mid \mathtt{let}\ \langle V, \mathsf{E}\rangle\ \mathtt{be}\ \langle x, y\rangle\ \mathtt{in}\ N \mid \mathsf{E}N \mid V\mathsf{E} \mid r\mathtt{:=}\mathsf{E},\end{aligned}$$

The language $\ell\mathsf{T}$ is not a purely functional language, due to the presence of assignments and references. Evaluation thus involves a term *and* a *store*, which is a function $\mathcal{S}$ assigning a value V to any reference r. The store which is equal to $\mathcal{S}$ except on r, to which it assigns V, is indicated with $\mathcal{S}\{r/V\}$. A *configuration* is a pair $(\mathcal{S}, M)$ where M is a closed term and $\mathcal{S}$ is a store. The *evaluation* relation is a binary relation $\longrightarrow$ on configurations defined following the rules in Figure 4, which are all standard except the third one, which allows to replace a call to an undefined symbol $\mathtt{f}$ with anything which can be put in correspondence with it by the semantics of $\mathtt{f}$.

2.5 Abstract Machine

We consider a variant of Felleisen and Friedman's CEK. As such, our machine will be given as a transition system on configurations, each of them keeping track of both the term being evaluated and the values

$$
\begin{aligned}
(\mathcal{S}, \mathsf{E}[\mathtt{succ}(t)]) &\longrightarrow (\mathcal{S}, \mathsf{E}[\mathbf{succ}(t)]);\\
(\mathcal{S}, \mathsf{E}[\mathtt{cons}(t,s)]) &\longrightarrow (\mathcal{S}, \mathsf{E}[\mathbf{cons}(t,s)]);\\
(\mathcal{S}, \mathsf{E}[\mathtt{f}(t)]) &\longrightarrow (\mathcal{S}, \mathsf{E}[\mathbf{S}_{\mathtt{f}}(t)]);\\
(\mathcal{S}, \mathsf{E}[\mathtt{let}\ \langle V, W\rangle\ \mathtt{be}\ \langle x, y\rangle\ \mathtt{in}\ M]) &\longrightarrow (\mathcal{S}, \mathsf{E}[M\{x, y/V, W\}]);\\
(\mathcal{S}, \mathsf{E}[(\lambda x.M)V]) &\longrightarrow (\mathcal{S}, \mathsf{E}[M\{x/V\}]);\\
(\mathcal{S}, \mathsf{E}[!\, r]) &\longrightarrow (\mathcal{S}, \mathsf{E}[\mathcal{S}(r)]);\\
(\mathcal{S}, \mathsf{E}[(r :=\! V)]) &\longrightarrow (\mathcal{S}\{r/V\}, \mathsf{E}[*]);\\
(\mathcal{S}, \mathsf{E}[\mathtt{iter}(M,N)\ \mathbf{succ}(t)]) &\longrightarrow (\mathcal{S}, \mathsf{E}[N\ (\mathtt{iter}(M,N)\ t)]);\\
(\mathcal{S}, \mathsf{E}[\mathtt{iter}(M,N)\ \mathbf{zero}]) &\longrightarrow (\mathcal{S}, \mathsf{E}[N]);\\
(\mathcal{S}, \mathsf{E}[\mathtt{ifz}(M,N)\ \mathbf{succ}(t)]) &\longrightarrow (\mathcal{S}, \mathsf{E}[N\ t]);\\
(\mathcal{S}, \mathsf{E}[\mathtt{ifz}(M,N)\ \mathbf{zero}]) &\longrightarrow (\mathcal{S}, \mathsf{E}[M]);\\
(\mathcal{S}, \mathsf{E}[\mathtt{if}(M,N)\ \mathbf{tt}]) &\longrightarrow (\mathcal{S}, \mathsf{E}[M]);\\
(\mathcal{S}, \mathsf{E}[\mathtt{if}(M,N)\ \mathbf{ff}]) &\longrightarrow (\mathcal{S}, \mathsf{E}[N]);\\
(\mathcal{S}, \mathsf{E}[\mathtt{fold}(M,N)\ \mathbf{cons}(t,s)]) &\longrightarrow (\mathcal{S}, \mathsf{E}[M\ (\mathtt{fold}(M,N)s)\ t]);\\
(\mathcal{S}, \mathsf{E}[\mathtt{fold}(M,N)\ \mathbf{nil}]) &\longrightarrow (\mathcal{S}, \mathsf{E}[N]);
\end{aligned}
$$

Fig. 4. Evaluation Rules.

locations map to. From now on, for the sake of simplicity, we consider natural numbers as the only base type, keeping in mind that all the other base types can be treated similarly. *Closures*, *environments*, and *stacks* are defined as follows, where M denotes a sequence of terms:

$$c\ ::=\ (\mathbf{M}, \xi); \qquad \xi\ ::=\ \varepsilon \ \Big|\ \xi \cdot (x \mapsto c); \qquad \pi\ ::=\ \varepsilon \ \Big|\ \delta \cdot \pi;$$

where δ ranges over *stack elements*:

$$\begin{aligned}\delta\ ::=\ &\mathtt{lft}(c) \ \Big|\ \mathtt{rgt}(c) \ \Big|\ \mathtt{let}(c,x,x) \ \Big|\ \mathtt{letlft}(c,c,x,x) \ \Big|\ \mathtt{letrgt}(c,c,x,x)\\ &\Big|\ \mathtt{fun}(c) \ \Big|\ \mathtt{arg}(c) \ \Big|\ \mathtt{succ} \ \Big|\ \mathtt{sel}(c) \ \Big|\ \mathtt{iter}(c) \ \Big|\ \mathtt{ufun}(\mathtt{f}) \ \Big|\ :=\!(r).\end{aligned}$$

Machine stores are finite, partial maps of locations to *value closures*, i.e., closures in the form (V, ξ). A machine store $\mathcal{S}$ is said to be *conformant* with a reference context Θ if the value closure $\mathcal{S}(r)$ can be given type A, where $r : A$ is in Θ. *Configurations* are triples in the form $\mathscr{C} = (c, \pi, \mathcal{S})$, where c is a closure, π is a stack and $\mathcal{S}$ is a machine store. Machine transitions are of the form $\mathscr{C} \succ^n \mathscr{D}$, where n is a natural number denoting the cost of the transition. This is always defined to be 1, except for function calls, which are attributed a cost depending on the underlying function setting:

$$((t, \xi), \mathtt{ufun}(\mathtt{f}) \cdot \pi, \mathcal{S}) \succ^{\mathbf{C}_{\mathtt{f}}(|t|)} ((\mathbf{S}_{\mathtt{f}}(t), \xi), \pi, \mathcal{S})$$

The other rules are given in Figure 5. The way we label machine transitions induces a cost model: the amount of time a program takes when executed is precisely the sum of the costs of the transitions the machine performs while evaluating it. This can be proved to be *invariant*, i.e. to correspond to the costs of ordinary models of computation (TMs, RAMs, etc.), modulo a polynomial overhead.

3 Linear Dependent Types

There is nothing in $\ell\mathsf{T}$ types which allows to induce complexity bounds for the programs; in fact, $\ell\mathsf{T}$ can express at least all the primitive recursive functions [22]. This is precisely the role played by linear

$$
\begin{array}{rcl}
((\texttt{let}\ M\ \texttt{be}\ \langle x,y\rangle\ \texttt{in}\ N,\xi),\pi,\mathcal{S}) & \succ^{1} & ((M,\xi),\texttt{let}((N,\xi),x,y)\cdot\pi,\mathcal{S}) \\
(((\langle M,N\rangle,\xi),\texttt{let}(c,x,y)\cdot\pi,\mathcal{S}) & \succ^{1} & ((M,\xi),\texttt{letlft}((N,\xi),c,x,y)\cdot\pi,\mathcal{S}); \\
((V,\xi),\texttt{letlft}(c,d,x,y)\cdot\pi,\mathcal{S}) & \succ^{1} & (c,\texttt{letrgt}((V,\xi),d,x,y)\cdot\pi,\mathcal{S}); \\
((V,\xi),\texttt{letrgt}((W,\theta),(N,\upsilon),x,y)\cdot\pi,\mathcal{S}) & \succ^{1} & ((N,\upsilon\cdot(x\mapsto(W,\theta))\cdot(y\mapsto(V,\xi))),\pi,\mathcal{S}); \\
((MN,\xi),\pi,\mathcal{S}) & \succ^{1} & ((N,\xi),\texttt{fun}(M,\xi)\cdot\pi,\mathcal{S}) \\
((V,\xi),\texttt{fun}(N,\theta)\cdot\pi,\mathcal{S}) & \succ^{1} & ((N,\theta),\texttt{arg}(V,\xi)\cdot\pi,\mathcal{S}) \\
((\lambda x.M,\xi),\texttt{arg}(V,\theta)\cdot\pi,\mathcal{S}) & \succ^{1} & ((M,\xi\cdot(x\mapsto(V,\theta))),\pi,\mathcal{S}); \\
((\texttt{succ}(M),\xi),\pi,\mathcal{S}) & \succ^{1} & ((M,\xi),\texttt{succ}\cdot\pi,\mathcal{S}); \\
((t,\xi),\texttt{succ}\cdot\pi,\mathcal{S}) & \succ^{1} & ((\mathbf{succ}(t),\xi),\pi,\mathcal{S}); \\
((\texttt{iter}(M,N),\xi),\texttt{fun}(L,\theta)\cdot\pi,\mathcal{S}) & \succ^{1} & ((L,\theta),\texttt{iter}(M,N,\xi)\cdot\pi,\mathcal{S}); \\
((\mathbf{succ}(t),\theta),\texttt{iter}(M,N,\xi)\cdot\pi,\mathcal{S}) & \succ^{1} & ((t,\theta),\texttt{iter}(M,N,\xi)\cdot\texttt{fun}(M,\xi)\cdot\pi,\mathcal{S}); \\
((\mathbf{zero},\theta),\texttt{iter}(M,N,\xi)\cdot\pi,\mathcal{S}) & \succ^{1} & ((N,\xi),\pi,\mathcal{S}); \\
((\mathtt{f}(M),\xi),\pi,\mathcal{S}) & \succ^{1} & ((M,\xi),\texttt{ufun}(\mathtt{f})\cdot\pi,\mathcal{S}); \\
((t,\xi),\texttt{ufun}(\mathtt{f})\cdot\pi,\mathcal{S}) & \succ^{\mathbf{C}_{\mathtt{f}}(|t|)} & ((\mathbf{S}_{\mathtt{f}}(t),\xi),\pi,\mathcal{S}); \\
((!\,r,\xi),\pi,\mathcal{S}) & \succ^{1} & (\mathcal{S}(r),\pi,\mathcal{S}); \\
((r:=M,\xi),\pi,\mathcal{S}) & \succ^{1} & ((M,\xi),:=(r)\cdot\pi,\mathcal{S}); \\
((V,\xi),:=(r)\cdot\pi,\mathcal{S}) & \succ^{1} & ((*,\xi),\pi,\mathcal{S}\{r/(V,\xi)\});
\end{array}
$$

Fig. 5. Transition Rules

dependency [23], whose underlying idea consists in decorating simple types with some information about the identity of objects that programs manipulate. This takes the form of so-called index terms, following in spirit Xi's DML [29]. Differently from [23], what indices keep track of here is the *length*, rather than the *value*, of ground-type objects. The fact that higher-order objects cannot be duplicated, on the other hand, greatly simplifies the type system.

Given a set $\mathcal{IV}$ of *index variables*, and a set $\mathcal{IF}$ of *index functions* (each with an arity), *index terms* over $\mathcal{IV}$ and $\mathcal{IF}$ are defined as follows $I \ ::= \ a \ \mid \ f(I,\ldots,I)$, where $a \in \mathcal{IV}$ and $f \in \mathcal{IF}$. Index functions are interpreted as *total* functions from (n-uples of) positive natural numbers to positive natural numbers, by way of an *equational program* $\mathcal{E}$, which can be specified as, e.g., an orthogonal, terminating, term rewriting system or a primitive recursive Herbrand-Gödel scheme. We often write $\models^{\mathcal{E}} I \leq J$, by which we mean that the semantics $\mathcal{E}$ assigns to I is smaller or equal to the semantics $\mathcal{E}$ assigns to J, this for every value of the index variables occurring in either I or J. We will assume that $\mathcal{IF}$ contains at least 0 (of arity 0), s (for the successor, of arity 1) and $+, \cdot$ (addition and multiplication, of arity 2, used with infix notation), with adequate equations in $\mathcal{E}$.

$$
\frac{\models^{\mathcal{E}} I \leq J}{\vdash^{\mathcal{E}} \mathbf{N}^I \sqsubseteq \mathbf{N}^J} \qquad \frac{}{\vdash^{\mathcal{E}} \mathsf{unit} \sqsubseteq \mathsf{unit}} \qquad \frac{\vdash^{\mathcal{E}} D \sqsubseteq E \qquad \vdash^{\mathcal{E}} F \sqsubseteq G}{\vdash^{\mathcal{E}} D \otimes F \sqsubseteq E \otimes G}
$$

$$
\frac{\vdash^{\mathcal{E}} E \sqsubseteq D \qquad \vdash^{\mathcal{E}} F \sqsubseteq G \qquad \vdash^{\mathcal{E}} \alpha \sqsubseteq \beta}{\vdash^{\mathcal{E}} D \stackrel{\alpha}{\multimap} F \sqsubseteq E \stackrel{\beta}{\multimap} G} \qquad \frac{\vdash^{\mathcal{E}} \Upsilon \sqsubseteq \Theta \quad \vdash^{\mathcal{E}} \Xi \sqsubseteq \Phi \quad \vdash^{\mathcal{E}} \Pi \sqsubseteq \Lambda}{\vdash^{\mathcal{E}} \Theta \Rightarrow \Xi \sqsubseteq \Upsilon, \Pi, \Psi \Rightarrow \Phi, \Lambda}
$$

$$
\frac{\vdash^{\mathcal{E}} D_1 \sqsubseteq E_1, \ldots, \vdash^{\mathcal{E}} D_n \sqsubseteq E_n}{\vdash^{\mathcal{E}} \{r_1 : D_1, \ldots, r_n : D_n\} \sqsubseteq \{r_1 : E_1, \ldots, r_n : E_n\}}
$$

Fig. 6. Subtyping Rules

We now define types and effects. *Indexed base types* U, *indexed reference contexts* Θ, *indexed types* D and *indexed effects* α are defined, respectively, as follows:

$$
\begin{array}{ll}
U \ ::= \ \mathsf{unit} \ \mid \ \mathbf{B} \ \mid \ \mathbf{N}^I \ \mid \ \mathbf{L}^I(U); & \Theta \ ::= \ \{r_1 : D_1, \ldots, r_n : D_n\}; \\
\alpha \ ::= \ \Theta \Rightarrow \Theta; & D \ ::= \ U \ \mid \ D \otimes D \ \mid \ D \stackrel{\alpha}{\multimap} D.
\end{array}
$$

where I ranges over index terms. The empty effect $\emptyset \Rightarrow \emptyset$ is denoted as $\mathbf{0}$. Given two effects $\alpha = \Theta \Rightarrow \Xi$ and $\beta = \Xi \Rightarrow \Upsilon$, their *composition* $\alpha; \beta$ is $\Theta \Rightarrow \Upsilon$. An effect $\Theta \Rightarrow \Xi$ is meant to describe how references are modified by terms: if the store is conformant to Θ *before* evaluation, it will be conformant to Ξ *after* evaluation. So in particular an effect $\Theta \Rightarrow \emptyset$ does not provide any information.

If D is an indexed type, $[D]$ is the *type* obtained from D by: (i) forgetting all the index information (ii) replacing on arrows $\overset{\alpha}{\multimap}$ the effect α by the set of locations that appear in it. E.g., if $D = \mathbf{N}^{I_1} \overset{\alpha}{\multimap} \mathbf{N}^{I_2}$ where $\alpha = \{r_1 : D_1, r_2 : D_2\} \Rightarrow \{r_1 : D_4, r_3 : D_3\}$, then $[D] = \mathbf{N} \overset{\{r_1, r_2, r_3\}}{\multimap} \mathbf{N}$.

Given $t \in \mathcal{T}(T)$, we write that $\models^{\mathcal{E}} t \in U$ iff $[U] = T$ and the size of t is bounded by the index terms in U, independently on the values of index variables. Similarly, $\models^{\mathcal{E}} \mathtt{f} \in U \multimap V$ stands for $\models^{\mathcal{E}} s \in V$ whenever $\models^{\mathcal{E}} t \in U$ and $(t, s) \in \mathbf{S}_{\mathtt{f}}$. As an example, any lists t whose three elements are natural numbers less or equal to 4 is such that $\models^{\mathcal{E}} t \in \mathbf{L}^7(\mathbf{N}^{4+a})$.

A subtyping relation $\sqsubseteq$ on indexed types and effects is defined in Figure 6. Note that we have $\vdash^{\mathcal{E}} \mathbf{0} \sqsubseteq \alpha$ iff α is of the shape $\alpha = \Xi_1, \Upsilon, \Xi_2 \Rightarrow \Pi$, where $\vdash^{\mathcal{E}} \Upsilon \sqsubseteq \Pi$. Suppose that r is a reference and that D is an indexed type. Then $\mathcal{ER}(r, D)$ is defined to be just $\{r : D\} \Rightarrow \{r : D\}$ if D is an indexed base type, and $\{r : D\} \Rightarrow \emptyset$ otherwise.

Typing contexts and terminology on them are the same as the one we used in the linear type system (Section 2) where, of course, *indexed types* plays the role of *types*. A typing judgement has the form $\Gamma \vdash^{\mathcal{E}} M : D; \alpha$. Let us denote $\alpha = \Theta \Rightarrow \Xi$. The intended meaning of the judgement is that if term variables are assigned types in Γ, then M can be typed with D, and if initially the contents of the references are typed as in Θ, then after evaluation of M the contents of the references can be typed as in Ξ. So while the former type system of Section 2 only provided information about *which* references might have been read during evaluation, this new system will also provide information about *how* the contents of references might have been modified, more specifically how the size of the contents might have changed.

Now, the typing rules are given in Figure 7 and Figure 8. A term M is dependently linearly typable if there exists a derivation of a judgement $\Gamma \vdash^{\mathcal{E}} M : D; \alpha$. Before analyzing the type system, let us make a

$$\frac{\vdash^{\mathcal{E}} D \sqsubseteq E \qquad \vdash^{\mathcal{E}} \mathbf{0} \sqsubseteq \alpha}{\Gamma, x : D \vdash^{\mathcal{E}} x : E; \alpha} \qquad \frac{\models^{\mathcal{E}} t \in U \qquad \vdash^{\mathcal{E}} \mathbf{0} \sqsubseteq \alpha}{\Gamma \vdash^{\mathcal{E}} t : U; \alpha} \qquad \frac{\vdash^{\mathcal{E}} \mathbf{0} \sqsubseteq \alpha}{\Gamma \vdash^{\mathcal{E}} * : \mathsf{unit}; \alpha}$$

$$\frac{\models^{\mathcal{E}} \mathtt{f} \in U \multimap V \qquad \Gamma \vdash^{\mathcal{E}} M : U; \alpha}{\Gamma \vdash^{\mathcal{E}} \mathtt{f}(M) : V; \alpha} \qquad \frac{\Gamma \vdash^{\mathcal{E}} M : D; \alpha \qquad \Delta \vdash^{\mathcal{E}} N : E; \beta}{\Gamma \uplus \Delta \vdash^{\mathcal{E}} \langle M, N \rangle : D \otimes E; \alpha; \beta}$$

$$\frac{\Gamma \vdash^{\mathcal{E}} M : D \otimes E; \alpha \qquad \Delta, x : D, y : E \vdash^{\mathcal{E}} N : F; \beta}{\Gamma \uplus \Delta \vdash^{\mathcal{E}} \mathtt{let}\ M\ \mathtt{be}\ \langle x, y \rangle\ \mathtt{in}\ N : F; \alpha; \beta}$$

$$\frac{\Gamma, x : D \vdash^{\mathcal{E}} M : E; \alpha \qquad \vdash^{\mathcal{E}} \mathbf{0} \sqsubseteq \beta}{\Gamma \vdash^{\mathcal{E}} \lambda x.M : D \overset{\alpha}{\multimap} E; \beta} \qquad \frac{\Gamma \vdash^{\mathcal{E}} M : D \overset{\gamma}{\multimap} E; \alpha \qquad \Delta \vdash^{\mathcal{E}} N : D; \beta}{\Gamma \uplus \Delta \vdash^{\mathcal{E}} MN : E; \alpha; \beta; \gamma}$$

$$\frac{\vdash^{\mathcal{E}} \mathcal{ER}(r, D) \sqsubseteq \alpha}{\Gamma \vdash^{\mathcal{E}} !r : D; \alpha} \qquad \frac{\Gamma \vdash^{\mathcal{E}} M : D; \Theta \Rightarrow \Xi, \{r : E\}}{\Gamma \vdash^{\mathcal{E}} r := M : \mathsf{unit}; \Theta \Rightarrow \Xi, \{r : D\}}$$

Fig. 7. Typing Rules, Part I

few comments:

- As in the linear type system, all rules treat variables of ground types differently than variables of higher-order types: the former can occur free an arbitrary number of times, while the latter can occur free at most once. Similarly for references. As a consequence, if a term is dependently linearly typable, then it is linearly typable.
- We should also take note of the fact that values can all be typed with the $\mathbf{0}$ effect. This is quite intuitive, since values are meant to be terms which need not be further evaluated.

$$\frac{\models^{\mathcal{E}} I+1 \leq J \quad \Gamma \vdash^{\mathcal{E}} M : \mathbf{N}^I; \alpha}{\Gamma \vdash^{\mathcal{E}} \mathtt{succ}(M) : \mathbf{N}^J; \alpha}$$

$$\frac{\begin{array}{c}\ell\Gamma \vdash^{\mathcal{E}} W : D\{a/1\}; \mathbf{0} \qquad \ell\Gamma \vdash^{\mathcal{E}} V : \left(D \overset{\Xi \Rightarrow \Xi\{a/a+1\}}{\multimap} D\{a/a+1\}\right); \mathbf{0} \\ \vdash^{\mathcal{E}} \mathbf{0} \sqsubseteq \alpha \qquad \vdash^{\mathcal{E}} D \sqsubseteq E \qquad \vdash^{\mathcal{E}} E \sqsubseteq E\{a/a+1\} \qquad \vdash^{\mathcal{E}} E\{a/I\} \sqsubseteq F \\ \vdash^{\mathcal{E}} \Theta \sqsubseteq \Xi\{a/1\} \qquad \vdash^{\mathcal{E}} \Xi \sqsubseteq \Pi \qquad \vdash^{\mathcal{E}} \Pi \sqsubseteq \Pi\{a/a+1\} \qquad \vdash^{\mathcal{E}} \Pi\{a/I\} \sqsubseteq \Phi\end{array}}{\ell\Gamma \vdash^{\mathcal{E}} \mathtt{iter}(V,W) : \mathbf{N}^I \overset{\Theta \Rightarrow \Phi}{\multimap} F; \alpha}$$

$$\frac{\begin{array}{c}\ell\Gamma \vdash^{\mathcal{E}} V : D\{a/1\}; \mathbf{0} \qquad \ell\Gamma \vdash^{\mathcal{E}} W : \left(\mathbf{N}^a \overset{\beta\{a/a+1\}}{\multimap} D\{a/a+1\}\right); \mathbf{0} \\ \vdash^{\mathcal{E}} \mathbf{0} \sqsubseteq \alpha \qquad \vdash^{\mathcal{E}} D \sqsubseteq D\{a/a+1\} \qquad \vdash^{\mathcal{E}} D\{a/I\} \sqsubseteq E \\ \vdash^{\mathcal{E}} \mathbf{0} \sqsubseteq \beta\{a/1\} \qquad \vdash^{\mathcal{E}} \beta \sqsubseteq \beta\{a/a+1\} \qquad \vdash^{\mathcal{E}} \beta\{a/I\} \sqsubseteq \gamma\end{array}}{\ell\Gamma \vdash^{\mathcal{E}} \mathtt{ifz}(V,W) : \mathbf{N}^I \overset{\gamma}{\multimap} E; \alpha}$$

Fig. 8. Typing Rules, Part II

– The rules typing assignments and location references (two bottom rules in Figure 7) show how higher-order references are treated. While an assignment simply overwrites the type attributed to the assigned location, a reference is typed with the location effect of the referenced location.

As an example, we can derive for the term M of Sect. 2:

$$\vdash^{\mathcal{E}} M : \mathbf{L}^{a+3}(\mathbf{N}^{b+1}); r : \mathbf{N}^c \Rightarrow r : \mathbf{N}^{b+1}$$

As to the terms of Sect.2 for increment, addition and multiplication, we obtain:

$$\vdash^{\mathcal{E}} incr_r : \mathsf{unit} \overset{r:\mathbf{N}^a \Rightarrow r:\mathbf{N}^{a+1}}{\multimap} \mathsf{unit}; \mathbf{0} \quad \vdash^{\mathcal{E}} add_{r,r_1} : \mathsf{unit} \overset{r:\mathbf{N}^a, r_1:\mathbf{N}^b \Rightarrow r:\mathbf{N}^{a+b}, r_1:\mathbf{N}^b}{\multimap} \mathsf{unit}; \mathbf{0}$$
$$\vdash^{\mathcal{E}} mult_{r,r_1,r_2} : \mathsf{unit} \overset{\alpha}{\multimap} \mathsf{unit}; \mathbf{0} \text{ where}$$
$$\alpha = (r : \mathbf{N}^c, r_1 : \mathbf{N}^a, r_2 : \mathbf{N}^b) \Rightarrow (r : \mathbf{N}^{a \cdot b}, r_1 : \mathbf{N}^a, r_2 : \mathbf{N}^b)$$

4 Intensional Soundness

Once a term M has been typed by a derivation π, one can assign a *weight* $\mathbb{W}(\pi)$ to π (and thus indirectly to M) in the form of an index term I. It is meant to estimate the time complexity of M. $\mathbb{W}(\pi)$ is defined by induction on the structure of π; interesting cases are those for iteration and function:

$$\frac{\rho \triangleright \Gamma \vdash^{\mathcal{E}} M : \mathbf{N}^I; \alpha}{\pi \triangleright \Gamma \vdash^{\mathcal{E}} \mathtt{f}(M) : U; \alpha} \qquad \mathbb{W}(\pi) = \mathbb{W}(\rho) + \mathbf{C}_{\mathtt{f}}(I)$$

$$\frac{\begin{array}{c}\rho \triangleright \ell\Gamma \vdash^{\mathcal{E}} W : D\{a/1\}; \mathbf{0} \\ \sigma \triangleright \ell\Gamma \vdash^{\mathcal{E}} V : \left(D \overset{\Xi \Rightarrow \Xi\{a/a+1\}}{\multimap} D\{a/a+1\}\right); \mathbf{0}\end{array}}{\pi \triangleright \ell\Gamma \vdash^{\mathcal{E}} \mathtt{iter}(V,W) : \mathbf{N}^I \overset{\Theta \Rightarrow \Phi}{\multimap} F; \alpha} \quad \mathbb{W}(\pi) = \mathbb{W}(\rho) + \textstyle\sum_{1 \leq a < I} \mathbb{W}(\sigma) + I$$

It is important to note that the definition of $\mathbb{W}(\pi)$ in the case of $\mathtt{iter}(V,W)$ involves a summation, but $\mathbb{W}(\sigma)$ is computed only once and the summation itself is not evaluated. Other cases are defined in the standard way, i.e. the weight of a derivation is the weight of the sum of its subderivations plus 1.

The full definition is displayed on Figure 9. Note that the weight $\mathbb{W}(\pi)$ can be easily computed from π, i.e., in time linear in the size of π.

The weight $\mathbb{W}(\pi)$ of π does *not* necessarily decrease along reduction as defined in Section 2.4.

The weight of a derivation decreases along *machine* transition rules, as defined in Section 2. This can be proved by generalizing $\mathsf{d}\ell\mathsf{T}$ to a type system for machine configurations. This extension is in Figure 10.

$$\pi \triangleright \frac{}{\Gamma, x : D \vdash^{\mathcal{E}} x : E; \alpha} \qquad \mathbb{W}(\pi) = 1$$

$$\pi \triangleright \frac{}{\Gamma \vdash^{\mathcal{E}} t : U; \alpha} \qquad \mathbb{W}(\pi) = 0$$

$$\pi \triangleright \frac{}{\Gamma \vdash^{\mathcal{E}} * : \mathsf{unit}; \alpha} \qquad \mathbb{W}(\pi) = 1$$

$$\pi \triangleright \frac{\rho \triangleright \Gamma \vdash^{\mathcal{E}} M : \mathbf{N}^I; \alpha}{\Gamma \vdash^{\mathcal{E}} \mathtt{f}(M) : U; \alpha} \qquad \mathbb{W}(\pi) = \mathbb{W}(\rho) + \mathbf{C}_{\mathtt{f}}(I)$$

$$\pi \triangleright \frac{\rho \triangleright \Gamma \vdash^{\mathcal{E}} M : D; \alpha \qquad \sigma \triangleright \Delta \vdash^{\mathcal{E}} N : E; \beta}{\Gamma \uplus \Delta \vdash^{\mathcal{E}} \langle M, N \rangle : D \otimes E; \alpha; \beta} \qquad \mathbb{W}(\pi) = \mathbb{W}(\rho) + \mathbb{W}(\sigma) + 1$$

$$\pi \triangleright \frac{\rho \triangleright \Gamma \vdash^{\mathcal{E}} M : D \otimes E; \alpha \qquad \sigma \triangleright \Delta, x : D, y : E \vdash^{\mathcal{E}} N : F; \beta}{\Gamma \uplus \Delta \vdash^{\mathcal{E}} \mathtt{let}\ M\ \mathtt{be}\ \langle x, y \rangle\ \mathtt{in}\ N : F; \alpha; \beta} \qquad \mathbb{W}(\pi) = \mathbb{W}(\rho) + \mathbb{W}(\sigma) + 1$$

$$\pi \triangleright \frac{\rho \triangleright \Gamma, x : D \vdash^{\mathcal{E}} M : E; \alpha \qquad \vdash^{\mathcal{E}} \mathbf{0} \sqsubseteq \beta}{\Gamma \vdash^{\mathcal{E}} \lambda x.M : D \stackrel{\alpha}{\multimap} E; \beta} \qquad \mathbb{W}(\pi) = \mathbb{W}(\rho) + 1$$

$$\pi \triangleright \frac{\rho \triangleright \Gamma \vdash^{\mathcal{E}} M : D \stackrel{\gamma}{\multimap} E; \alpha \qquad \sigma \triangleright \Delta \vdash^{\mathcal{E}} N : D; \beta}{\Gamma \uplus \Delta \vdash^{\mathcal{E}} MN : E; \alpha; \beta; \gamma} \qquad \mathbb{W}(\pi) = \mathbb{W}(\rho) + \mathbb{W}(\sigma) + 1$$

$$\pi \triangleright \frac{}{\Gamma \vdash^{\mathcal{E}} !r : D; \alpha} \qquad \mathbb{W}(\pi) = 1$$

$$\pi \triangleright \frac{\rho \triangleright \Gamma \vdash^{\mathcal{E}} M : D; \Theta \Rightarrow \Xi, \{r : E\}}{\Gamma \vdash^{\mathcal{E}} r \mathtt{:=} M : \mathsf{unit}; \Theta \Rightarrow \Xi, \{r : D\}} \qquad \mathbb{W}(\pi) = 1$$

$$\pi \triangleright \frac{\Gamma \vdash^{\mathcal{E}} M : \mathbf{N}^I; \alpha}{\Gamma \vdash^{\mathcal{E}} \mathtt{succ}(M) : \mathbf{N}^J; \alpha} \qquad \mathbb{W}(\pi) = \mathbb{W}(\rho) + 1$$

$$\pi \triangleright \frac{\begin{array}{c}\rho \triangleright \ell\Gamma \vdash^{\mathcal{E}} W : D\{a/1\}; \mathbf{0} \\ \sigma \triangleright \ell\Gamma \vdash^{\mathcal{E}} V : \left(D \stackrel{\Xi \Rightarrow \Xi\{a/a+1\}}{\multimap} D\{a/a+1\}\right); \mathbf{0}\end{array}}{\ell\Gamma \vdash^{\mathcal{E}} \mathtt{iter}(V, W) : \mathbf{N}^I \stackrel{\Theta \Rightarrow \Phi}{\multimap} F; \alpha} \qquad \mathbb{W}(\pi) = \mathbb{W}(\rho) + \sum_{1 \leq a < I} \mathbb{W}(\sigma) + I$$

$$\pi \triangleright \frac{\begin{array}{c}\rho \triangleright \Gamma \vdash^{\mathcal{E}} V : D\{a/1\}; \mathbf{0} \\ \sigma \triangleright \Gamma \vdash^{\mathcal{E}} W : \left(\mathbf{N}^a \stackrel{\beta\{a/a+1\}}{\multimap} D\{a/a+1\}\right); \mathbf{0}\end{array}}{\Gamma \vdash^{\mathcal{E}} \mathtt{ifz}(V, W) : \mathbf{N}^I \stackrel{\gamma}{\multimap} E; \alpha} \qquad \mathbb{W}(\pi) = \mathbb{W}(\rho) + \mathbb{W}(\sigma) + 1$$

Fig. 9. The Weight of Type Derivations

Closures

$$\pi \triangleright \left(\frac{\begin{array}{c} \rho \triangleright \Gamma, \Delta \vdash^{\mathcal{E}} M : D; \alpha \\ \sigma \triangleright \vdash^{\mathcal{E}} \xi : \Gamma \\ \upsilon \triangleright \vdash^{\mathcal{E}} \theta : \Omega \end{array}}{\Delta \vdash^{\mathcal{E}} (M, \xi \cdot \theta) : D; \alpha} \right) \qquad \mathbb{W}(\pi) = \mathbb{W}(\rho) + \mathbb{W}(\sigma)$$

Environments

$$\pi \triangleright \frac{}{\vdash^{\mathcal{E}} \varepsilon : \cdot} \qquad \mathbb{W}(\pi) = 0$$

$$\pi \triangleright \frac{\rho \triangleright \vdash^{\mathcal{E}} \xi : \Gamma \qquad \sigma \triangleright \vdash^{\mathcal{E}} v : D}{\vdash^{\mathcal{E}} \xi \cdot (x \mapsto v) : \Gamma, x : D} \qquad \mathbb{W}(\pi) = \mathbb{W}(\rho) + \mathbb{W}(\sigma)$$

Stacks

$$\pi \triangleright \frac{}{\vdash^{\mathcal{E}} \varepsilon : D \rightsquigarrow D; \mathbf{0}} \qquad \mathbb{W}(\pi) = 0$$

$$\pi \triangleright \frac{\rho \triangleright \vdash^{\mathcal{E}} \pi : D \rightsquigarrow E; \alpha \qquad \sigma \triangleright x : F, y : G \vdash^{\mathcal{E}} c : D; \beta}{\vdash^{\mathcal{E}} \mathtt{let}(c, x, y) \cdot \pi : F \otimes G \rightsquigarrow E; \beta; \alpha} \qquad \mathbb{W}(\pi) = \mathbb{W}(\rho) + \mathbb{W}(\sigma) + 1$$

$$\pi \triangleright \left(\frac{\begin{array}{c} \rho \triangleright \vdash^{\mathcal{E}} \pi : D \rightsquigarrow E; \alpha \\ \sigma \triangleright x : F, y : G \vdash^{\mathcal{E}} c : D; \beta \\ \upsilon \triangleright \vdash^{\mathcal{E}} d : G; \gamma \end{array}}{\vdash^{\mathcal{E}} \mathtt{letlft}(c, d, x, y) \cdot \pi : F \rightsquigarrow E; \beta; \gamma; \alpha} \right) \qquad \mathbb{W}(\pi) = \mathbb{W}(\rho) + \mathbb{W}(\sigma) + \mathbb{W}(\upsilon) + 2$$

$$\pi \triangleright \left(\frac{\begin{array}{c} \rho \triangleright \vdash^{\mathcal{E}} \pi : D \rightsquigarrow E; \alpha \\ \sigma \triangleright x : F, y : G \vdash^{\mathcal{E}} c : D; \beta \\ \upsilon \triangleright \vdash^{\mathcal{E}} v : F \end{array}}{\vdash^{\mathcal{E}} \mathtt{letrgt}(c, v, x, y) \cdot \pi : F \rightsquigarrow E; \beta; \alpha} \right) \qquad \mathbb{W}(\pi) = \mathbb{W}(\rho) + \mathbb{W}(\sigma) + \mathbb{W}(\upsilon) + 1$$

$$\pi \triangleright \frac{\rho \triangleright \vdash^{\mathcal{E}} \pi : D \rightsquigarrow E; \alpha \qquad \sigma \triangleright \vdash^{\mathcal{E}} d : F; \beta}{\vdash^{\mathcal{E}} \mathtt{fun}(c) \cdot \pi : (F \overset{\gamma}{\multimap} D) \rightsquigarrow E; \beta; \gamma; \alpha} \qquad \mathbb{W}(\pi) = \mathbb{W}(\rho) + \mathbb{W}(\sigma) + 1$$

$$\pi \triangleright \frac{\rho \triangleright \vdash^{\mathcal{E}} \pi : D \rightsquigarrow E; \alpha \qquad \sigma \triangleright \vdash^{\mathcal{E}} v : F \overset{\beta}{\multimap} D}{\vdash^{\mathcal{E}} \mathtt{arg}(v) \cdot \pi : F \rightsquigarrow E; \beta; \gamma; \alpha} \qquad \mathbb{W}(\pi) = \mathbb{W}(\rho) + \mathbb{W}(\sigma) + 1$$

$$\pi \triangleright \frac{\rho \triangleright \vdash^{\mathcal{E}} \pi : \mathbf{N}^I \rightsquigarrow D; \alpha \qquad \models^{\mathcal{E}} J \leq I + 1}{\vdash^{\mathcal{E}} \mathtt{succ} \cdot \pi : \mathbf{N}^J \rightsquigarrow D; \alpha} \qquad \mathbb{W}(\pi) = \mathbb{W}(\rho) + 1$$

$$\pi \triangleright \frac{\rho \triangleright \vdash^{\mathcal{E}} \pi : \mathsf{unit} \rightsquigarrow D; \{r : E\} \cup \Theta \Rightarrow \Xi}{\sigma \triangleright \vdash^{\mathcal{E}} :=(r) \cdot \pi : E \rightsquigarrow D; \Theta \Rightarrow \Xi} \qquad \mathbb{W}(\pi) = \mathbb{W}(\rho) + 1$$

Configurations

$$\pi \triangleright \frac{\rho \triangleright \vdash^{\mathcal{E}} c : D; \alpha \qquad \sigma \triangleright \vdash^{\mathcal{E}} \pi : D \rightsquigarrow E; \beta}{\vdash^{\mathcal{E}} (c, \pi) : E; \alpha; \beta} \qquad \mathbb{W}(\pi) = \mathbb{W}(\rho) + \mathbb{W}(\sigma)$$

Fig. 10. Typing and Weighting Configurations

Following the same ideas, one can also define the weight $\mathbb{W}(\mathcal{S})$ of any conformant store $\mathcal{S}$ (also on Fig. 10). By a careful analysis of the reduction rules, one gets Subject Reduction for configurations:

Lemma 1. *If $\vdash^{\mathcal{E}} (c, \pi) : U; \Theta \Rightarrow \Xi$, the store $\mathcal{S}$ is conformant with Θ, and $(c, \pi, \mathcal{S}) \succ^n (d, \rho, \mathcal{R})$, then $\vdash^{\mathcal{E}} (d, \rho) : U; \Upsilon \Rightarrow \Xi$, the store $\mathcal{R}$ is conformant with Υ, and $\mathbb{W}(\pi) + \mathbb{W}(\mathcal{S}) \geq \mathbb{W}(\rho) + \mathbb{W}(\mathcal{R}) + n$.*

Proof. One only needs to carefully analyze each of the machine transition rules from Figure 5. Indeed, the way the type system has been extended to configurations and the way the weight of a type derivation has been defined make the task easy. □

As an easy consequence, we obtain intensional soundness.

Theorem 1 (Intensional Soundness). *If $\pi\triangleright \vdash^{\mathcal{E}} M : U; \Theta \Rightarrow \Xi$, the store $\mathcal{S}$ is conformant with Θ, and $(M, \varepsilon, \mathcal{S}) \succ^n \mathscr{C}$, then $n \leq \mathbb{W}(\pi) + \mathbb{W}(\mathcal{S})$.*

Proof. This is an induction on n, making essential use of Lemma 1.

So this theorem shows that the weight of a $\mathsf{d}\ell\mathsf{T}$ program is indeed a bound on its evaluation time. But how can we type an $\ell\mathsf{T}$ program so as to obtain a $\mathsf{d}\ell\mathsf{T}$ type derivation? This is the subject of the next Section.

5 Type Inference

The type inference procedure is defined quite similarly to the one in [24], where the language at hand, called $\mathsf{d}\ell\mathsf{PCF}$, is more general than the one described here (apart from effects). As a consequence, we will only describe the general scheme of the algorithm, together with the most important cases (noticeably, iteration). For simplification we will also omit the effects, as anyway they don't present specific difficulties

A tree whose nodes are labelled by typing judgements, but which is not necessarily built according to our type system is said to be a *pseudo-derivation*. This is to be contrasted with a proper type derivation. Similarly, an incomplete set of rewrite rules which, contrarily to proper equational programs, does not univocally define all the function symbols that occur in them is said to be a *pseudo-program*.

The type inference algorithm $\mathtt{TI}$ takes in input a linearly typable term M, together with a finite sequence of index variables $\phi = a_1, \ldots, a_n$, and returns:

- A pseudo-derivation π with conclusion $\Gamma \vdash M : D; \alpha$ where the types in Γ and D match the ones of M.
- A pseudo-program $\mathcal{E}$.

In other words, $\mathtt{TI}(M^A, \phi) = (\pi, \mathcal{E})$. The understanding here is that the undefinded function symbols in $\mathcal{E}$ are those which occur in *negative* positions in the conclusion of π, and the termination of symbols in $\mathcal{E}$ is necessary and sufficient for π to be a correct type derivation (once symbols not defined in $\mathcal{E}$ are properly defined).

The algorithm $\mathtt{TI}$ is recursive, and proceeds by pattern matching on its first argument. The way we define $\mathtt{TI}$, and in particular the fact that the output program is only a *pseudo*-program and not a proper equational program, has the consequence of allowing $\mathtt{TI}$ to be defineable by recursion.

Before describing the algorithm $\mathtt{TI}$, it is necessary to give some preliminary definitions:

- An index term is said to be *primitive* iff it is in the form $f(\phi)$, where ϕ is a sequence of index variables. An index type is *primitive* iff all index terms occurring in it are primitive. Similarly for typing contexts. Two primitive index terms, types or contexts are said to be *homogeneous* iff all terms in them are on the *same* sequence of index variables.
- Given two primitive index types D, E such that $[D] = [E]$, the expression $\mathtt{cut}(D, E)$ is defined as the unique equational program satisfying the following equations:

$$\begin{aligned}
\mathtt{cut}(\mathbf{N}^{f(\phi)}, \mathbf{N}^{g(\psi)}) &= \{g(\psi) \to f(\phi);\}; \\
\mathtt{cut}(D \otimes E, F \otimes G) &= \mathtt{cut}(D, F) \cup \mathtt{cut}(E, G); \\
\mathtt{cut}(D \multimap E, F \multimap G) &= \mathtt{cut}(F, D) \cup \mathtt{cut}(E, G).
\end{aligned}$$

– Given a type A and a sequence ϕ of index variables, $\texttt{fresh}(A, \phi)$ stands for the indexed type obtained by "decorating" A with fresh primitive index terms on ϕ.
– Suppose that Γ and Δ are two homogeneous primitive typing contexts that only share variables of base type (to which they do not necessarily assign the same index term). Then $\texttt{merge}(\Gamma, \Delta)$ is the pair $(\Omega, \mathcal{E})$ where $\Omega, \mathcal{E}$ are the smallest objects satisfying the following conditions:
  – All type assignments from *either* Γ or Δ (but not both), are in Ω;
  – For all variables to which *both* Γ and Δ assign types, say $\mathbf{N}^{f(\phi)}$ and $\mathbf{N}^{g(\phi)}$, the typing context Ω contains $x : \mathbf{N}^{j(\phi)}$ and $\mathcal{E}$ contains $\{f(\phi) \to j(\phi), g(\phi) \to j(\phi)\}$.
– Suppose that D, E, F are indexed types such that $[D] = [E] = [F]$. Moreover, suppose that f is an index function, and that $\mathbf{p} \in \{+, -\}$ is a polarity, then

$$\texttt{itercut}_{\mathbf{p}}(D, E, F, f) = (\mathcal{E}, G, H)$$

where the following equations hold:

$$\texttt{itercut}_{+}((\mathbf{N}^{g(\phi,a,b)}, \mathbf{N}^{j(\phi,a,b)}, \mathbf{N}^{h(\phi,b)}, f)) = (\mathcal{E}, \mathbf{N}^{k(\phi,b)}, \mathbf{N}^{p(\phi)})$$
where $\mathcal{E}$ is
$$\{g(\phi, a, b+1) \to j(\phi, a+1, b);$$
$$g(\phi, a, 0) \to h(\phi, a);$$
$$k(\phi, b+1) \to j(\phi, 0, b);$$
$$k(\phi, 0) \to h(\phi, a);$$
$$p(\phi) \to \max_{c \le f(\phi)} k(\phi, c)\}$$

$$\texttt{itercut}_{-}((\mathbf{N}^{g(\phi,a,b)}, \mathbf{N}^{j(\phi,a,b)}, \mathbf{N}^{h(\phi,b)}, f)) = (\mathcal{E}, \mathbf{N}^{k(\phi,b)}, \mathbf{N}^{p(\phi)})$$
where $\mathcal{E}$ is
$$\{j(\phi, a+1, b) \to g(\phi, a, b+1);$$
$$j(\phi, 0, b) \to h(\phi, b);$$
$$h(\phi, b+1) \to j(\phi, 0, b);$$
$$h(\phi, 0) \to k(\phi, 0);$$
$$k(\phi, b) \to p(\phi);\}$$

$$\texttt{itercut}_{\mathbf{p}}(D_L \otimes D_R, E_L \otimes E_R, F_L \otimes F_R, f) = (\mathcal{E}_L \cup \mathcal{E}_R, G_L \otimes G_R, H_L \otimes H_R)$$
where
$$\texttt{itercut}_{\mathbf{p}}((D_L, E_L, F_L, f)) = (\mathcal{E}_L, G_L, H_L)$$
$$\texttt{itercut}_{\mathbf{p}}((D_R, E_R, F_R, f)) = (\mathcal{E}_R, G_R, H_R)$$

$$\texttt{itercut}_{\mathbf{p}}(D_L \multimap D_R, E_L \multimap E_R, F_L \multimap F_R, f) = (\mathcal{E}_L \cup \mathcal{E}_R, G_L \multimap G_R, H_L \multimap H_R)$$
where
$$\texttt{itercut}_{\neg\mathbf{p}}((D_L, E_L, F_L, f)) = (\mathcal{E}_L, G_L, H_L)$$
$$\texttt{itercut}_{\mathbf{p}}((D_R, E_R, F_R, f)) = (\mathcal{E}_R, G_R, H_R)$$

Finally, we are now able to formally define the algorithm TI. It is given in Figure 11. What TI produces in output, however, is *not* a type derivation but a *pseudo*-derivation: $\mathcal{E}$ does not give meaning to all function symbols, and in particular not to the symbols occurring in negative position. Getting a proper type derivation, then, requires giving meaning to those symbols. This is the purpose of the algorithm CTI which, given in input a term M, proceeds as follows:
– It calls $\texttt{TI}(M, \phi)$, where the variables in ϕ are in bijective correspondence to the negative occurrences of base types in M.
– Once obtained $(\pi, \mathcal{E})$ in output, it complements $\mathcal{E}$ with equations in the form $f_i(\phi) = a_i$ where a_i is the variable corresponding to f_i.
– In the conclusion of π, replace $f_i(\phi)$ by just a_i.

$$\mathtt{TI}(x^A, \phi) = \left(\overline{\ x : D \vdash^{\mathcal{E}} x : E\ }, \mathtt{cut}(D, E)\right)$$
where $D = \mathtt{fresh}(A, \phi) \wedge E = \mathtt{fresh}(A, \phi)$;

$$\mathtt{TI}(t^{\mathbf{N}}, \phi) = \left(\overline{\ \vdash^{\mathcal{E}} t : \mathbf{N}^{f(\phi)}\ }, \{f(\phi) \to |t|\}\right)$$
where f is fresh;

$$\mathtt{TI}(M^{A \multimap B} N^A, \phi) = \left(\frac{\pi \triangleright \Gamma \vdash^{\mathcal{E}} M : D \multimap E \qquad \mathtt{sty}(\rho) \triangleright \Delta \vdash^{\mathcal{E}} N : D}{\Omega \vdash^{\mathcal{E}} MN : E}, \mathcal{E} \cup \mathcal{F} \cup \mathcal{G} \cup \mathcal{H}\right)$$
where
$\mathtt{TI}(M^{A \multimap B}, \phi) = (\pi, \mathcal{E}) \wedge \mathtt{TI}(N^A, \phi) = (\rho \triangleright \Delta \vdash^{\mathcal{E}} N : F, \mathcal{F}) \wedge$
$\mathtt{cut}(D, F) = \mathcal{G} \wedge \mathtt{merge}(\Gamma, \Delta) = (\Omega, \mathcal{H})$.

$$\mathtt{TI}(\langle M^A, N^B \rangle, \phi) = \left(\frac{\pi \triangleright \Gamma \vdash^{\mathcal{E}} M : D \qquad \rho \triangleright \Delta \vdash^{\mathcal{E}} N : E}{\Omega \vdash^{\mathcal{E}} \langle M, N \rangle : D \otimes E}, \mathcal{E} \cup \mathcal{F} \cup \mathcal{G}\right)$$
where
$\mathtt{TI}(M^{A \multimap B}, \phi) = (\pi, \mathcal{E}) \wedge \mathtt{TI}(N^A, \phi) = (\rho, \mathcal{F}) \wedge$
$\mathtt{merge}(\Gamma, \Delta) = (\Omega, \mathcal{H})$.

$$\mathtt{TI}(\lambda x^A . M^B, \phi) = \left(\frac{\rho \triangleright \Gamma, x : D \vdash^{\mathcal{E}} M : E}{\Gamma \vdash^{\mathcal{E}} \lambda x . M : D \multimap E}, \mathcal{E} \cup \mathcal{F}\right)$$
where
$\mathtt{TI}(M^A, \phi) = (\pi, \mathcal{E}) \wedge \mathtt{weak}(\pi, x) = (\rho, \mathcal{F})$.

$$\mathtt{TI}(\mathtt{let}\ M^{A \otimes B}\ \mathtt{be}\ \langle x, y \rangle\ \mathtt{in}\ N^C, \phi) = \left(\frac{\begin{array}{c}\pi \triangleright \Gamma \vdash^{\mathcal{E}} M : D \otimes E \\ \sigma \triangleright \Delta, x : F, y : G \vdash^{\mathcal{E}} N : H\end{array}}{\Omega \vdash^{\mathcal{E}} \langle M, N \rangle : D \otimes E}, \mathcal{E} \cup \mathcal{F} \cup \mathcal{G} \cup \mathcal{H} \cup \mathcal{J} \cup \mathcal{K}\right)$$
where
$\mathtt{TI}(M^{A \otimes B}, \phi) = (\pi, \mathcal{E}) \wedge \mathtt{TI}(N^C, \phi) = (\rho, \mathcal{F}) \wedge$
$\mathtt{weak}(\rho, x, y) = (\sigma, \mathcal{G}) \wedge \mathtt{cut}(D, F) = \mathcal{H} \wedge$
$\mathtt{cut}(E, G) = \mathcal{J} \wedge \mathtt{merge}(\Gamma, \Delta) = (\Omega, \mathcal{H})$.

Fig. 11. The Type Inference Algorithm.

Soundness and Completeness. The way the type-inference algorithm CTI is defined makes its output correct mostly by definition.

Theorem 2 (Soundness). *If* $\texttt{CTI}(M) = (\pi, \mathcal{E})$*, then* $\mathcal{E}$ *is completely specified and* π *is a correct type derivation, i.e., all proof obligations in* π *are true in* $\mathcal{E}$*.*

Proof. First of all, one can prove, by induction on M, that any completion of the equational program obtained from $\texttt{TI}(M, \phi)$ turns the obtained pseudo-derivation into a correct type derivation. Then, one can observe that CTI simply completes TI in the obvious way. □

Since the CTI algorithm, contrarily to what happens traditionally in the context of type-inference, *never* fails when fed with terms which can be linearly typed as from Section 2, it means that it is also complete by design:

Theorem 3 (Completeness). *The algorithm* CTI *is total.*

5.1 Termination

We now prove that the equational program $\mathcal{E}$ produced in output by type inference (i.e. $\texttt{CTI}(M) = (\pi, \mathcal{E})$) is indeed *terminating*. Importantly, this cannot be proved directly, i.e. by induction on M, merely following the way TI is defined. Indeed, a reducibility-like argument is needed, which goes as follows: given an equational program $\mathcal{E}$ and an assignment ρ of natural numbers to some free index variables, we write $\mathcal{E}, \rho \models D$ if for the value of variables as in ρ, $\mathcal{E}$ is *reducible* in D, where *reducible* is a concept defined by induction on D. As an example, if D is just $\mathbf{N}^I$, then I is given a meaning by the equational program $\mathcal{E}$ when the variables occurring free in I are given values according to ρ. Then, as usual, one proves that all equational programs output by TI are reducible, but *for all* assignments ρ.

Theorem 4 (Termination). *If* $\texttt{CTI}(M) = (\pi, \mathcal{E})$*, then* $\mathcal{E}$ *is terminating.*

Proof. The proof is structured as follows:

- On the one hand, one needs to prove that any equational program $\mathcal{E}$ that $\texttt{TI}(M, \phi)$ produces in output is indeed reducible for every assignment ρ over ϕ. This property can indeed be proved by induction on the structure of M.
- On the other hand, one also proves that all *reducible* programs, when completed like in CTI, are terminating.

6 Application to Cryptographic Proofs

This section presents an application of dℓT to cryptographic proofs. Typically, such proofs reduce the security of a cryptographic construction to computational assumption(s), and consist of three steps. The first step is the definition of an algorithm $\mathcal{B}$, hereby called the constructed adversary, that breaks the computational assumption(s), using as a subroutine the adversary $\mathcal{A}$ against the cryptographic construction. The second step exhibits and formally justifies upper bounds on the winning probability of the constructed adversary $\mathcal{B}$, as an expression of the winning probability of the adversary $\mathcal{A}$ against the cryptographic construction. This step can be carried out formally using tools such as e.g. CryptoVerif [11] or EasyCrypt [9]. Finally, the third step formally justifies upper bounds for the execution time of the constructed adversary $\mathcal{B}$ as a function of the execution time of the adversary $\mathcal{A}$. We use dℓT for the third step.

We consider two examples. Our first example deals with padding-based encryption schemes, i.e. public key encryption schemes built from a one-way trapdoor permutation f and one or several hash functions, modelled as random oracles. The constructed adversaries for such schemes are relatively easy to analyze, as they typically search in the lists of adversarial calls to the random oracles for values that satisfy some predicate. Our second example deals with hardcore predicates; such predicates characterize the information leaked by a one-way function. We consider the constructed adversary from the Goldreich-Levin theorem, which shows the existence of hardcore predicates for a class of one-way functions. This example is particularly challenging, because some of the intermediate computations are not polytime w.r.t. the size of their inputs.

Notation In dℓT we will freely use the combinators map and fold and map_2 on lists. Combinators map and fold are defined in the usual way, whereas $\mathsf{map}_2\ f\ l\ l'$ returns the list $(f\ a_1\ a'_1, \ldots, f\ a_n\ a'_n)$ where $n = \min(|l|, |l'|)$ and $|\cdot|$ denotes the length operator. Moreover, we let app denote concatenation of lists, and $*^k$ denote the list that repeats k times the constant $*$.

Furthermore, we model bits and bitstrings as booleans and lists of booleans, respectively. In order to increase readability, we often use $\{0,1\}$ as a synonym for $\mathbf{B}$ and $\{0,1\}^k$ to denote the set of bitstrings of length k. Moreover, we use standard notations for bitstrings: we let $\oplus$ and $\otimes$ respectively denote the exclusive or operator and multiplication operators (both of type $\{0,1\} \to \{0,1\} \to \{0,1\}$), and 0^k denote the 0-bitstring of length k. Using operators on maps, one can define exclusive or on bitstrings, and scalar multiplication of a bitstring by a bit. Moreover, we assume given a probabilistic operator $\mathsf{flip} : \mathsf{unit} \to \{0,1\}$ that samples a bit uniformly at random. Again using standard operators on maps, one can define an operator $\mathsf{flip}_k : \mathsf{unit}^k \to \{0,1\}^k$.

Finally, we can define an operator pow_0 which takes as input a natural number k and outputs the list of non-empty subsets of $\{1, \ldots, k\}$—we model each subset as a list of bitstrings of length k.

6.1 Padding-based Encryption

The BR93 encryption scheme [10] is an example of a public-key encryption scheme built from a one-way trapdoor permutation $(\mathcal{K}, \mathsf{f}, \mathsf{f}^{-1})$ and a random oracle H. A one-way trapdoor permutation is given by a triple of algorithms $(\mathcal{K}, \mathsf{f}_{pk}, \mathsf{f}_{sk}^{-1})$ consisting of an algorithm $\mathcal{K}$ that generates valid key pairs (sk, pk) and of two indexed families of functions $\mathsf{f}_{pk}, \mathsf{f}_{sk}^{-1} : \{0,1\}^\ell \to \{0,1\}^\ell$ such that for every pair of keys (sk, pk) generated by $\mathcal{K}$, the functions f_{pk} and f_{sk}^{-1} are mutually inverse. The informal requirement for a one-way trapdoor permutation is that f_{pk} and f_{sk}^{-1} can be computed efficiently by parties that have knowledge of the keys (in the first case, pk is assumed to be public and hence can be computed by all parties), but f_{sk}^{-1} is infeasible to compute without knowledge of the key. Formally, the security of one-way trapdoor permutation is measured by the probability ϵ that an inverter executing in time t can invert f_{pk} on a randomly sampled value from its input domain. A random oracle is a stateful procedure $\mathcal{H}$ that lazily computes a random function: it takes as input a bitstring of size ℓ and returns as output a uniformly distributed bitstring of size k, and maintains a table to return the same value when queried twice on the same input. The encryption algorithm proceeds as follows: it first samples a bitstring r of length ℓ and computes $\mathsf{f}_{pk}(r)$ and $h(r)$; finally, it masks the message m (a bitstring of length k) by xoring it with $h(r)$ and concatenates the resulting bitstring to $\mathsf{f}_{pk}(r)$. The BR93 encryption scheme achieves indistinguishability against chosen-plaintext attacks, or IND-CPA security for short, assuming that the hash function is modelled as a random oracle, i.e. a stateful function that samples a uniformly distributed value on the output space when given a fresh input—and returns consistent results in case of repeated inputs. Informally, IND-CPA security states that an adversary has negligible probability to distinguish between two encrypted messages for plaintexts of its choice.

In an asymptotic setting, the IND-CPA security of BR93 is captured by the following statement: for every adversary $\mathcal{A}$ with a non-negligible advantage of guessing the bit b (the advantage of guessing is the probability of guessing minus $\frac{1}{2}$), there exists an inverter $\mathcal{I}$ with a non-negligible probability of inverting f on a random input. In a concrete setting, the statement states that the advantage of the IND-CPA adversary is upper bounded by the probability of $\mathcal{I}$ inverting f, and that the execution time of $\mathcal{I}$ is upper bounded by $t_{\mathcal{A}} + q_H \cdot T_{\mathsf{f}}$, where $t_{\mathcal{A}}$ denotes the execution time of $\mathcal{A}$, q_H is the maximal number of adversarial queries to the random oracle $\mathcal{H}$, and T_{f} is an upper bound on the time to compute f_{pk} on an input. In the remainder of this paragraph, we define the inverter $\mathcal{I}$ and provide an informal analysis of its execution time.

Given as input a bitstring y of length ℓ, the inverter $\mathcal{I}$ outputs another bistring x of length ℓ as follows:

1. the key generation algorithm is invoked to generate a valid pair of keys (sk, pk);
2. the first adversary $\mathcal{A}_1$ is invoked with pk, and returns a pair of messages (m_0, m_1);
3. it samples uniformly at random a bistring s of length k;
4. the second adversary $\mathcal{A}_2$ is invoked with input $y \parallel s$, and obtains in return a bit b;
5. it traverses the list of adversarial queries to the random oracle $\mathcal{H}$, and tests for each input z whether it satisfies the equality $\mathsf{f}_{pk}(z) = y$; if such a z is found, then it returns its value, else it returns a uniformly sampled bitstring.

The formal definition of the inverter $\mathcal{I}$ is given in Figure 12. Here r_L is the reference to the list of adversarial queries to the random oracle. We use a function `equal` for testing equality of bitstrings. The complexity of $\mathcal{I}$ can be derived readily from the complexity of each individual step. The key step here is the last, in which the adversary does a list traversal. Loosely speaking, the cost of the traversal is the length of the list, which is upper bounded by the maximal number q_H of allowed adversarial queries to the random oracle, and by the cost of the test, which is upper bounded by the time T_{f} to compute f_{pk} on an input. Overall, the execution time $t_{\mathcal{I}}$ of $\mathcal{I}$ verifies $t_{\mathcal{I}} \sim t_{\mathcal{A}} + q_H \cdot T_{\mathsf{f}}$, where $t_{\mathcal{A}}$ denotes the execution time of the adversary $\mathcal{A}$.

$\lambda y.$ — *argument of the inverter*
`let` $\langle sk, pk\rangle = \mathcal{K}(*)$ `in` — *generation of the keys*
`let` $\langle m_0, m_1\rangle = \mathcal{A}_1(pk)$ `in` — *$\mathcal{A}_1$ returns pair of messages*
`let` $s = \mathsf{map}\ (\mathsf{flip})\ *^k$ `in` — *sampling bitstring of length k*
`let` $b = \mathcal{A}_2(\mathsf{app}\ y\ s)$ `in` — *$\mathcal{A}_2$ returns bit b*
`let` $\langle flag, x\rangle = \mathtt{fold}\ (step, \langle \mathtt{ff}, 0^\ell\rangle)\ !r_L$ `in` — *traversal of list $!r_L$*
`if` $flag$ `then` x `else` $\mathsf{map}\ (\mathsf{flip})\ *^\ell$ — *return result*

where :
$step = \lambda z.\lambda\langle flag, temp\rangle.$`if` $flag$ `then` $\langle \mathtt{tt}, temp\rangle$ `else` `if` $\mathtt{equal}(\mathsf{f}(pk, z), y)$ `then` $\langle \mathtt{tt}, z\rangle$ `else` $\langle flag, temp\rangle$ — *function for traversal, with equality test updating $flag$*

Fig. 12. Inverter for BR93

Discussion The BR93 encryption scheme is the simplest instance of so-called padding-based encryption schemes, which include widely used schemes like OAEP. Many of these schemes either achieve chosen-plaintext security or the stronger notion of chosen-ciphertext security, in which the adversary has access to a decryption oracle; however, security is sometimes achieved at the cost of a stronger assumption on the one-way function, such as partial-domain one-wayness. Irrespective of the security property and of the assumption, the constructed inverter will traverse multiple lists of adversarial queries to oracles, testing for each possible tuple of elements from the lists whether it satisfies some appropriate test. The cost of such trasversals is upper bounded by $(\prod_{i\in I} q_i) \cdot t$, where q_i is the maximal length of the different lists traversed, and t is an upper bound for the cost of the test. In some cases, the inverter tests lists sequentially, in which case the cost is upper bounded by $(\prod_{i\in I} q_i \cdot t_i)$, where t_i is an upper bound for the cost of each test.

6.2 Hardcore Predicates for One-Way Functions

Recall that a one-way function is a function that is easy to compute but hard to invert. Although it seemingly contradicts the definition, one-way functions can also leak information about their inputs. Thus, a natural question is to characterize the amount of information that one-way functions hide from their inputs. This hiding property is captured by the notion of hardcore predicate; informally, a predicate p is a hardcore predicate for a function f if p can be computed efficiently and an efficient adversary with access to $\mathsf{f}\ x$ has a small probability to guess correctly whether $\mathsf{p}\ x$ holds, where the value x is sampled uniformly over the domain of f. The existence of a hardcore predicate for every one-way function is a long-standing open problem in cryptography. However, the celebrated Goldreich-Levin theorem proves that for every one-way function f over bitstrings of length n, there exists a hardcore predicate p for the one-way function g over bitstrings of length $2n$, where g is defined by the clause $\mathsf{g}(\mathsf{app}\ x\ y) = \mathsf{app}\ (\mathsf{f}\ x)\ y$, and p is defined by the clause:

$$\mathsf{p}(\mathsf{app}\ x\ y) = \bigoplus_{i=1}^{n} x_i \otimes y_i$$

where x_i denotes the i-th bit of x and we recall that app is the concatenation of strings.

The theorem is proved by showing that for every adversary $\mathcal{A}$ with a non-negligible probability of guessing the value of the hardcore predicate on a randomly chosen value x, there exists a probabilistic

polytime inverter $\mathcal{I}$ with a non-negligible probability of guessing the pre-image of f on a randomly sampled value x. Informally, the inverter is given as input a bitstring y of length n, and outputs a bitstring x of length n as follows:

1. it sets ℓ to $\lceil \log(n+1) \rceil$;
2. it defines zerobut as the function that takes as inputs a natural number i and a bitstring w of length n and returns the bitstring z such that $z_j = 0$ for $j \neq i$ and $z_i = w_i$ (note that z also has length n);
3. it samples uniformly at random ℓ bitstrings $z_1 \dots z_\ell$ of length n, and ℓ bits $r_1 \dots r_\ell$;
4. for every non-empty subset X of $\{1 \dots \ell\}$, it computes the bitstrings z^X and the bit r^X, respectively as $\bigoplus_{i \in X} z_i$ and $\oplus_{i \in X} r_i$;
5. for every non-empty subset X of $\{1 \dots \ell\}$ and for every $i \in \{1 \dots n\}$, it computes the bit $x_i^X = r^X \oplus \mathcal{A}(\mathsf{app}\ y\ (\mathsf{zerobut}\ i\ z^X))$, and sets $x_i = \mathsf{majority}(x_i^X)$, where X is drawn from the set of non-empty subsets of $\{1 \dots \ell\}$.

The formal definition of the inverter $\mathcal{I}$ in $\ell\mathsf{T}$ is given in Figure 13. It uses two functions, both defined recursively in the expected way: the majority function $\mathsf{majority} : \mathbf{L}(\{0,1\}) \to \{0,1\}$ which returns the most frequent bit from a list, and the function $\mathsf{zerobut} : \mathbf{N} \to \mathbf{L}(\{0,1\}) \to \mathbf{L}(\{0,1\})$ which takes as input a natural number n and a list l and zeroes all elements of l but the nth one.

We refer to e.g. [21, §6.3] for a proof of the validity of the reduction, and focus on analyzing the complexity of the inverter $\mathcal{I}$. Let us first carry an informal analysis. For each function $\mathsf{zerobut}(i, \cdot)$, the inverter invokes the adversary $2^\ell - 1$ times, since the lists R^P and Z^P both have length $2^\ell - 1$. So the inverter calls the adversary $n \cdot (2^\ell - 1)$ times, i.e. $t_{\mathcal{I}} \sim n^2 \cdot t_{\mathcal{A}}$. Hence $\mathcal{I}$ executes in polytime, assuming that $\mathcal{A}$ does. This example is particularly interesting because it is not hereditarily polytime: the function pow_0 has an output of exponential size (and so is not polytime), but in the program $\mathcal{I}$ it is applied only to a small input (ℓ, of logarithmic size). It does not use references, but it illustrates how higher-order can be used to write concise code.

$\lambda y.$ *argument of the inverter*
$\mathtt{let}\ \ell = \lceil \log(n+1) \rceil\ \mathtt{in}$ *defines* ℓ
$\mathtt{let}\ \mathbf{P} = \mathsf{pow}_0\ \ell\ \mathtt{in}$ *enumerates all non-empty subsets of* $\{1, \dots, \ell\}$
$\mathtt{let}\ R = \mathsf{map}\ (\mathsf{flip})\ *^\ell\ \mathtt{in}$ *samples uniformly at random* $(r_1, \dots, r_\ell)$
$\mathtt{let}\ Z = \mathsf{map}\ (\lambda\_.\ \mathsf{map}\ (\mathsf{flip})\ *^n)\ *^\ell\ \mathtt{in}$ *samples uniformly at random* $(z_1, \dots, z_\ell)$
$\mathtt{let}\ R^P = \mathsf{map}\ (g_r)\ \mathbf{P}\ \mathtt{in}$ *computes the list* $(r^X)_{X \in \mathbf{P}}$
$\mathtt{let}\ Z^P = \mathsf{map}\ (g_z)\ \mathbf{P}\ \mathtt{in}$ *computes the list* $(z^X)_{X \in \mathbf{P}}$
$\mathsf{map}\ (G)(1, \dots, n)$

where :
$\bigotimes = \lambda x.\mathsf{map}\ (\otimes\ x)$
$\bigoplus = \mathsf{map}_2\ (\oplus)$
$g_r = \lambda X.\mathsf{fold}\ (\oplus, 0)\ (\mathsf{map}_2(\otimes)\ X\ R)$
$g_z = \lambda X.\mathsf{fold}\ (\bigoplus, 0^n)\ (\mathsf{map}_2\ (\bigotimes)\ X\ Z)$
$G = \lambda i.\ \mathsf{majority}\ (\mathsf{map}_2\ (\lambda r\ z.\ \oplus\ r\ (\mathcal{A}\ (\mathsf{app}\ y\ (\mathsf{zerobut}\ i\ z))))\ R^P\ Z^P)$

Fig. 13. Inverter $\mathcal{I}$ against hardcore predicate, with helper functions

Now let us sketch how we can type the inverter $\mathcal{I}$. We need for that to extend $\mathcal{IF}$ with two new function symbols $\log$, e with the following equations in $\mathcal{E}$:

$\log(1) \to 0, \quad \log(2 \cdot a) \to s\log(a), \quad e(0) \to 1, \quad e(sa) \to 2 \cdot e(a).$

As examples of types for subterms we obtain, where $J = \log(sn)$:

$$\mathsf{pow}_0 : \mathbf{N}^a \multimap \mathbf{L}^{e(a)}(\mathbf{L}^a(\mathbf{B})), \quad P : \mathbf{L}^{e(J)}(\mathbf{L}^J(\mathbf{B})), \quad g_r : \mathbf{L}^J(\mathbf{B}) \multimap \mathbf{B}$$

Finally the inverter $\mathcal{I}$ can be given a type derivation π of conclusion $\mathbf{L}^n(\mathbf{B}) \multimap \mathbf{L}^n(\mathbf{B})$. If we denote as g the index function representing the time complexity of the adversary $\mathcal{A}$, we obtain as weight for the derivation: $\mathbb{W}(\pi) = O(n^2 g(1+2n) + n^2 \log(n+1))$. As we can assume that $\log(n+1)$ is dominated by $g(1+2n)$, we finally obtain $\mathbb{W}(\pi) = O(n^2 g(1+2n))$. By Theorem 1 this yields a bound for the running time of $\mathcal{I}$, so this result confirms the informal analysis carried out above.

7 Examples, Revisited

We now illustrate the use of $\mathsf{d}\ell\mathsf{T}$ by typing the examples of Sect. 6. We consider that $\mathcal{IF}$ contains at least 0, a successor function s and binary functions $+$ (addition) and $\cdot$ (multiplication), for which we will use an infix notation. We write 1 for $s(0)$. We will extend the set $\mathcal{IF}$ when needed.

Consider the BR93 inverter of Fig. 12. We will use an index function f of arity 2 (resp. g_1, g_2 of arity 1) for representing the computation time of the function f (resp. adversaries $\mathcal{A}_1$, $\mathcal{A}_2$). Here are some types we can assign to intermediate terms of the program:

$$\begin{aligned}
y &: \mathbf{N}^\ell \\
\mathcal{K}(*) &: \mathbf{L}^m(\mathbf{B}) \otimes \mathbf{L}^m(\mathbf{B}) \\
\mathcal{A}_1 &: \mathbf{L}^m(\mathbf{B}) \multimap \\
&\qquad \mathbf{L}^n(\mathbf{B}) \otimes \mathbf{L}^n(\mathbf{B}) \\
\mathsf{map}\ (\mathsf{flip})\ *^k &: \mathbf{L}^k(\mathbf{B}) \\
\mathcal{A}_2(\mathtt{app}\ y\ s) &: \mathbf{B} \\
\mathit{step} &: \mathbf{L}^\ell(\mathbf{B}) \multimap \\
&\qquad \mathbf{B} \otimes \mathbf{L}^\ell(\mathbf{B}) \multimap \mathbf{B} \otimes \mathbf{L}^\ell(\mathbf{B}) \\
!\, r_L &: \mathbf{L}^p(\mathbf{L}^\ell(\mathbf{B})) \\
\mathtt{fold}\ (\mathit{step}, \langle \mathtt{ff}, 0^\ell \rangle) &: \mathbf{L}^p(\mathbf{L}^\ell(\mathbf{B})) \multimap \mathbf{B} \otimes \mathbf{L}^\ell(\mathbf{B})
\end{aligned}$$

This leads to a type derivation π of conclusion $\mathbf{L}^\ell(\mathbf{B}) \multimap \mathbf{L}^\ell(\mathbf{B})$. Observe on Fig. 9 that the only case where $\mathbb{W}(\rho)$ actually depends on the typing information and not just on the term is the case of the $\mathtt{iter}(V, W)$ (and similarly $\mathtt{fold}(V, W)$) construction. By abuse of notation we will denote by $\mathbb{W}(M)$ the weight $\mathbb{W}(\rho)$ of the subderivation ρ of π typing the term M. We can compute the following weights:

$$\begin{aligned}
\mathbb{W}(\mathcal{A}_1(pk)) &= g_1(m) \\
\mathbb{W}(\mathsf{map}\ (\mathsf{flip})\ *^k) &= 3k \\
\mathbb{W}(\mathcal{A}_2(\mathtt{app}\ y\ s)) &= g_2(k+l) \\
\mathbb{W}(\mathtt{equal}(\mathsf{f}(pk, z), y)\ ) &= f(m, \ell) + \ell \\
\mathbb{W}(\mathit{step}) &= \mathit{cst} + f(m, \ell) + \ell \quad \text{where} \\
&\qquad \mathit{cst} \text{ is a constant} \\
\mathbb{W}(\mathtt{fold}\ (\mathit{step}, \langle \mathtt{ff}, 0^\ell \rangle)) &= \mathbb{W}(\langle \mathtt{ff}, 0^\ell \rangle) + p\mathbb{W}(\mathit{step}) + p \\
&= \mathit{cst} + \ell + p \cdot f(m, \ell) + p\ell + p \\
\mathbb{W}(\mathsf{map}\ (\mathsf{flip})\ *^\ell) &= 3\ell
\end{aligned}$$

So we obtain $\mathbb{W}(\pi) = \mathit{cst} + g_1(m) + 3k + g_2(k + \ell) + \ell + pf(m, \ell) + p\ell + p + 3\ell$, where we recall that ℓ is the length of the input bitstring, k is the length of the bistring sampled and p is the length of the list r_L of adversarial queries to the random oracle. So if we consider as parameters k and ℓ we get $\mathbb{W}(\pi) = O(g_2(k+\ell) + pf(m, \ell) + (p+3)\ell)$. By Theorem 1 this gives us a complexity time bound on the execution of this program on the abstract machine.

Let us now examine the program of Fig. 13. We use an index function f for the computation time of the adversary $\mathcal{A}$ and extend $\mathcal{IF}$ with two new function symbols $\log$, e and the following equations in $\mathcal{E}$:

$$\begin{aligned}
&\log(1) \to 0, \quad \log(2 \cdot a) \to s \log(a), \\
&e(0) \to 1, \quad e(sa) \to 2 \cdot e(a)
\end{aligned}$$

We can assign the following types, where we use $J = \log(n+1)$:

$$
\begin{aligned}
\ell &: \mathbf{N}^J \\
\mathsf{pow}_0 &: \mathbf{N}^a \multimap \mathbf{L}^{e(a)}(\mathbf{L}^a(\mathbf{B})) \\
P &: \mathbf{L}^{e(J)}(\mathbf{L}^J(\mathbf{B})) \\
\mathsf{zerobut} &: \mathbf{N}^n \multimap \mathbf{L}^n(\mathbf{B}) \multimap \mathbf{L}^n(\mathbf{B}) \\
R &: \mathbf{L}^J(\mathbf{B}) \\
Z &: \mathbf{L}^J(\mathbf{L}^n(\mathbf{B})) \\
\mathsf{map}_2\ (\otimes)\ X\ R &: \mathbf{L}^J(\mathbf{B}) \\
g_r &: \mathbf{L}^J(\mathbf{B}) \multimap \mathbf{B} \\
\bigotimes &: \mathbf{B} \multimap \mathbf{L}^n(\mathbf{B}) \multimap \mathbf{L}^n(\mathbf{B}) \\
\bigoplus &: \mathbf{L}^n(\mathbf{B}) \multimap \\
&\qquad \mathbf{L}^n(\mathbf{B}) \multimap \mathbf{L}^n(\mathbf{B}) \\
\mathsf{map}_2\ (\bigotimes)\ X\ Z &: \mathbf{L}^J(\mathbf{L}^n(\mathbf{B})) \\
g_z &: \mathbf{L}^J(\mathbf{B}) \multimap \mathbf{L}^n(\mathbf{B}) \\
R^P &: \mathbf{L}^{e(J)}(\mathbf{B}) \\
Z^P &: \mathbf{L}^{e(J)}(\mathbf{L}^n(\mathbf{B})) \\
M = \lambda r\ z.\ \oplus\ r\ (\mathcal{A}\ (\mathsf{app}\ y\ (\mathsf{zerobut}\ i\ z)))) & \\
&: \mathbf{B} \multimap \mathbf{L}^n(\mathbf{B})) \multimap \mathbf{B} \\
\mathsf{map}_2\ (M) &: \mathbf{L}^n(\mathbf{B}) \multimap \\
&\qquad \mathbf{L}^n(\mathbf{L}^n(\mathbf{B})) \multimap \mathbf{L}^n(\mathbf{B}) \\
\mathsf{majority} &: \mathbf{L}^n(\mathbf{B}) \multimap \mathbf{B} \\
G &: \mathbf{N}^n \multimap \mathbf{B} \\
\mathsf{map}\ (G)(1,\ldots,n) &: \mathbf{L}^n(\mathbf{B})
\end{aligned}
$$

Call π the corresponding type derivation for the inverter $\mathcal{I}$. We want to bound $\mathbb{W}(\pi)$. To improve readability we will carry the following computations up to additive and multiplicative constants. So all statements of the form $\mathbb{W}(M) = I$ will actually stand for $\mathbb{W}(M) = O(I)$.

Assume we know the following weights for helper functions:

$$
\begin{aligned}
\mathbb{W}(\lceil \log(n+1) \rceil) &= \log(sn) \\
\mathbb{W}(\mathsf{pow}_0) &= e(a) \\
\mathbb{W}(\mathsf{zerobut}) &= n \\
\mathbb{W}(\mathsf{majority}) &= n \\
\mathbb{W}(\mathsf{app} M\ N) &= \mathbb{W}(M) + n, \text{ if } N \text{ is} \\
&\qquad \text{a list of type } \mathbf{L}^n(\mathbf{B}) \\
\mathbb{W}((1,\ldots,n)) &= n
\end{aligned}
$$

Then we obtain the following weights for intermediary derivations (where $J = \log(n+1)$):

$$
\begin{aligned}
\mathbb{W}(\mathbf{P}) &= e(J)\\
\mathbb{W}(\mathsf{map}\ (\mathsf{flip})\ *^{\ell}) &= J\\
\mathbb{W}(Z) &= J + nJ\\
\mathbb{W}(\mathsf{map}_2\ (\otimes)\ X\ R) &= J\\
\mathbb{W}(g_r) &= J\\
\mathbb{W}(\mathsf{map}_2(\bigotimes)\ X\ Z) &= nJ\\
\mathbb{W}(g_z) &= nJ\\
\mathbb{W}(R^P) &= nJ\\
\mathbb{W}(Z^P) &= n^2 J\\
\mathbb{W}(M) &= f(1+2n)\\
\mathbb{W}(\mathsf{map}_2\ (M)) &= nf(1+2n)\\
\mathbb{W}(G) &= nf(1+2n)\\
\mathbb{W}(\mathsf{map}\ (G)(1,\ldots,n)) &= n^2 f(1+2n)
\end{aligned}
$$

Finally we get $\mathbb{W}(\pi) = O(n^2 f(1+2n) + n^2 \log(n+1))$, and as we can assume that $\log(n+1)$ is dominated by $f(1+2n)$, $\mathbb{W}(\pi) = O(n^2 f(1+2n))$. By Theorem 1 we thus get a time complexity bound for this inverter on the abstract machine, and we can observe that this bound is consistent with our informal complexity analysis of sect. 6.2.

8 Related Work

In this section, we review existing works on complexity analysis, and existing tools for computer-aided cryptography. For the latter, we mostly concentrate on aspects that are relevant to analyzing the complexity of constructed adversaries.

8.1 Complexity Analysis

There exist many verification techniques for analysing the complexity of programs. What is specific about our proposal is the presence of both higher-order functions and imperative features, which allows a reasonable degree of flexibility, coupled with a nice way to accomodate probabilistic effects and oracles.

Type Systems. To our best knowledge, none of the (many) type systems characterizing polytime from the literature (e.g. [25, 18, 2, 17]) are able to capture non-hereditarily polytime programs. Technically, our type system can be seen as a variation and a simplification of linear dependent types [23, 24]. The main novelty is the presence of effects, which allow to deal with imperative features. Duplication of higher-order values is restricted, and this renders the type system simpler. Subrecursivity, in turn, enforces termination of equational programs obtained through type inference. All these aspects were simply missing in previous works on linear dependent types. Other related works are [12, 14], which however only deal with linear bounds or with first-order definitions.

Static Analysis. Among the static analysis methodologies for complexity analysis, those based on abstract interpretation [1, 15] deserve to be cited. These can be very effective on imperative programs, but are not able to handle higher-order features. It is moreover not clear whether relatively complicated examples like the ones we presented here could be handled. This is even more evident in, e.g., matrix-based calculi for imperative programs [19].

8.2 Computer-aided Cryptography.

Most tools for computer-aided cryptography adhere to the code-based game-playing approach. In this approach, cryptographic reductions are decomposed into a series of "hops", in which intermediate programs are introduced and related to their adjacent programs in the sequence of hops. There are two points worth noting about game-based proofs. First, constructed adversaries may be described explicitly or implicitly; of course, our method only applies to proofs in which the constructed adversary is explicitly described. Second, game-based proofs involve many constructed adversaries, typically one for each intermediate game in the proof. In order to measure the strength of the reduction, it is sufficient to analyze the complexity of the final constructed adversary, and there is no need to compare the complexity of adversaries in two adjacent games; more technically, we do not need to carry relational reasoning about complexity.

CryptoVerif The earliest tool to support computer-aided cryptographic proof is CryptoVerif [11], which can be used automatically or interactively for reasoning about the security of cryptographic constructions written in a probabilistic process calculus. In order to prove relational properties between two processes, CryptoVerif uses an approximate notion of probabilistic equivalence and a set of transformations that preserve (up to some approximation factor) the semantics of processes. CryptoVerif does not explicitly provide the constructed adversary.

EasyCrypt and CertiCrypt EasyCrypt [9] is an interactive framework which allows to reason about probabilistic imperative programs with adversarial code, using a combination of probabilistic Hoare logic and probablistic relational Hoare logic. EasyCrypt explicitly provides the constructed adversary, but its complexity analysis has to be performed by hand. Its predecessor CertiCrypt [8] formalizes an instrumented semantics that tracks the execution time of probabilistic program, and allows users to reason about the complexity of programs directly at the level of the instrumented operational semantics. Such reasoning is naturally cumbersome.

CIL Computational Indistinguishability Logic (CIL) [4] is a general framework to reason about cryptographic reductions. Contrary to other tools, cryptographic constructions in CIL are written in the usual style of mathematics, making it impossible to carry a type-based complexity analysis. Instead, CIL carries an implicit complexity analysis in its judgments.

FCF Foundational Cryptographic Framework (FCF) [27] is a machine-checked framework for proving the security of cryptographic constructions. Probabilistic computations are modelled in FCF using an embedded domain-specific language. FCF does not use an instrumented semantics to model the cost of computations; instead, the cost of computations is axiomatized. FCF provides an explicit characterization of the constructed adversary, and hence its complexity can be analyzed using this axiomatization. It may be possible to apply some of our techniques to FCF, using computational reflection as a bridge between our type system and the shallow embedding used by FCF.

Higher-order languages F* [28] is a refinement type system for a stateful, higher-order λ-calculus with a call-by-value strategy. F* has been used to verify implementations of cryptographic protocols and security of JavaScript implementations. F* cannot model cryptographic reductions faithfully, because it lacks support for relational reasoning. However, several works have considered extensions or variants of F* with relational refinement types; in particular, RF* [6] is a relational refinement type system for a probabilistic extension of F*; it has been used to verify simple examples of reductions. HoaRe^2 [7] is a more general system of relational refinements for a probabilistic higher-order language; it features refinements at higher types and a polymonad for approximate relational properties of probabilistic computations, and can be used to reason about cryptographic reductions—as well as for differential privacy, and mechanism design. None of these tools offer direct support to reason about the complexity of computations.

9 Conclusion

We have introduced an expressive type and effect system dℓT that can be used for analyzing the complexity of many cryptographic reductions automatically. A natural step for future work is to implement dℓT and

integrate it in a system for computer-aided cryptography, such as F*/RF* or a similar system such as HoaRe^2. Since the expressions of dℓT can be construed as a sublanguage of these systems, which support full recursion, it would be sufficient that the constructed adversary in game-based proofs is written in dℓT. It should also be possible to adapt the type system to other settings and implement it on top of systems like EasyCrypt.

Another direction for future work is to develop automated approaches to reason about expected complexity of programs. Several noteworthy reductions in cryptography are based on constructed adversaries that execute in expected, rather than strict, probabilistic polynomial time; the main challenge here is not only to come up with a type system for expected complexity, but also a definitional one; see [13] for a recent account of the subtleties with existing definitions.

More generally, there are numerous opportunities to connect formalisms used for computer-aided cryptography and formal systems for analyzing the complexity of programs. One important direction fur future work is to certify the complexity of attacks. These attacks are generally quite intricate and therefore analyzing their complexity is difficult and error-prone, making formal verification an appealing approach to guarantee that the published bounds are indeed correct. Moreover, many of these attacks are infeasible, in which case the sole measure of their strength is an upper bound on their complexity. One very ambitious goal would be to verify formally the complexity of recent algorithms for computing the discrete logarithm [20, 3].

In a different line of work, it would be interesting to apply dℓT for verifying other examples from algorithmic game theory and mechanism design [26], whose formal verification is performed in systems that are closely related to the ones used for computer-aided cryptography.

References

1. Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. COSTA: design and implementation of a cost and termination analyzer for java bytecode. In *Formal Methods for Components and Objects, 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures*, pages 113–132, 2007.
2. Patrick Baillot and Kazushige Terui. Light types for polynomial time computation in lambda calculus. *Inf. Comput.*, 207(1):41–62, 2009.
3. Razvan Barbulescu, Pierrick Gaudry, Antoine Joux, and Emmanuel Thomé. A heuristic quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic. In *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques. Proceedings*, volume 8441 of *LNCS*, pages 1–16. Springer, 2014.
4. Gilles Barthe, Marion Daubignard, Bruce Kapron, and Yassine Lakhnech. Computational indistinguishability logic. In *Computer and Communications Security, CCS 2010*, pages 375–386, New York, 2010. ACM.
5. Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. EasyCrypt: A tutorial. In Alessandro Aldini, Javier Lopez, and Fabio Martinelli, editors, *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer, 2013.
6. Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella-Béguelin. Probabilistic relational verification for cryptographic implementations. In *Principles of Programming Languages*, pages 193–206, 2014.
7. Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, and Pierre-Yves Strub. Higher-order approximate relational refinement types for mechanism design and differential privacy. In *Principles of Programming Languages*, pages 55–68. ACM, 2015.
8. Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *POPL*, pages 90–101, 2009.
9. Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *CRYPTO*, pages 71–90, 2011.
10. Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Computer and Communications Security*, pages 62–73, 1993.
11. Bruno Blanchet. A computationally sound mechanized prover for security protocols. In *IEEE Symposium on Security and Privacy*, pages 140–154, 2006.
12. Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *Proceedings of POPL 2008*, pages 133–144, 2008.
13. Oded Goldreich. On expected probabilistic polynomial-time adversaries: A suggestion for restricted definitions and their benefits. In *Theory of Cryptography*, pages 174–193. Springer, 2007.
14. Bernd Grobauer. Cost recurrences for DML programs. In *International Conference on Functional Programming (ICFP '01)*, pages 253–264, 2001.
15. Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139, 2009.
16. Shai Halevi. A plausible approach to computer-aided cryptographic proofs. *IACR Cryptology ePrint Archive*, page 181, 2005.
17. Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial potential. In *ESOP*, pages 287–306, 2010.
18. Martin Hofmann. Safe recursion with higher types and bck-algebra. *Ann. Pure Appl. Logic*, 104(1-3):113–166, 2000.
19. Neil D. Jones and Lars Kristiansen. A flow calculus of *mwp*-bounds for complexity analysis. *ACM Trans. Comput. Log.*, 10(4), 2009.
20. Antoine Joux. A new index calculus algorithm with complexity $l(1/4 + o(1))$ in very small characteristic. Cryptology ePrint Archive, Report 2013/095, 2013. `http://eprint.iacr.org/`.
21. Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman & Hall Cryptography and Network Security Series. Chapman & Hall, 2007.
22. Ugo Dal Lago. The geometry of linear higher-order recursion. *ACM Trans. Comput. Log.*, 10(2), 2009.
23. Ugo Dal Lago and Marco Gaboardi. Linear dependent types and relative completeness. *Logical Methods in Computer Science*, 8(4), 2011.
24. Ugo Dal Lago and Barbara Petit. The geometry of types. In *POPL*, pages 167–178, 2013.
25. Daniel Leivant and Jean-Yves Marion. Lambda calculus characterizations of poly-time. *Fundam. Inform.*, 19(1/2):167–184, 1993.

26. Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay V Vazirani. *Algorithmic game theory*. Cambridge University Press, 2007.
27. Adam Petcher and Greg Morrisett. The foundational cryptography framework. 2015. To appear.
28. Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In *International Conference on Functional Programming*, 2011.
29. Hongwei Xi. Dependent types for program termination verification. *Higher-Order and Symbolic Computation*, 15(1):91–131, 2002.