Implicit Computational Complexity of Subrecursive Definitions and Applications to Cryptographic Proofs

Patrick Baillot, Gilles Barthe, Ugo Dal Lago

Implicit Computational Complexity of Subrecursive Definitions and Applications to Cryptographic Proofs

Patrick Baillot* Gilles Barthe† Ugo Dal Lago‡

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 time 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

Developing automated analyses that accurately over-approximate the complexity of (stateful) higher-order programs is a challenging task. Sophisticated type systems such as dℓPCF [13, 14] *almost* achieve this goal for a (pure and call-by-value) higher-order language with unbounded recursion, using a combination of linear dependent types and constraint solving. Informally, the type inference algorithm of dℓPCF outputs equational programs from which one can infer the complexity bounds of expressions. However, the bound is *conditional*, in the sense that its validity depends on the equational program being terminating. Proving the termination of equational programs can be extremely difficult, even for relatively simple expressions. One possible way to overcome this problem would be to perform an automated termination analysis of the equational program. However, this approach is impractical, because the type inference algorithm of dℓPCF generates complex equational programs. Indeed the size of these equational programs quickly grows with the size of the original program and of its types, and they do not present any obvious regularity, for instance one cannot simply check their termination with easy methods like e.g. termination orderings. This approach is also frustrating, because one would like to avoid verifying termination of equational programs when the original terms are themselves clearly terminating. Hence, it is natural to consider the following question: *is there a normalizing higher-order language for which one can automatically compute accurate and unconditional complexity bounds?* By accurate here we mean complexity bounds which are close to the real worst-case complexity bound of the program. The question is not only of theoretical interest. Indeed, there are many applications which require to analyze the complexity of expressions that clearly terminate. For instance, cryptography is an application domain where systems like dℓPCF is overly expressive. In a typical cryptographic proof, one reduces the security of a cryptographic construction to the hardness of one or more assumptions, by showing that every adversary that can be used to break the security of a cryptographic construction can be used to build an adversary against one or more assumption. One important criterion is that the reduction must be tight, in the sense that the complexity of the constructed adversary must be close to the complexity of the original adversary.

*CNRS & ENS Lyon, patrick.baillot@ens-lyon.fr
†IMDEA Software Institute, gilles.barthe@imdea.org
‡Università di Bologna & INRIA, dallago@cs.unibo.it

For most examples, 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 main drawback of $\mathsf{d}\ell\mathsf{PCF}$, i.e. does not require to prove termination of an equational program.

More generally, $\mathsf{d}\ell\mathsf{PCF}$ and related systems for implicit computational complexity suffer from one or several of the following shortcomings, which limit their applicability to cryptography: 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 [14]—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.

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, we make the following contributions:

- 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[1] to characterize the complexity of programs;
- we define in Section 3 a type system $\mathsf{d}\ell\mathsf{T}$ 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 $\mathsf{d}\ell\mathsf{T}$ 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. Here soundness of the type inference algorithm means that it outputs a correct type derivation for the program. 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 6 that $\mathsf{d}\ell\mathsf{T}$ 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* . A hardcore predicate p for a function f is a predicate which can be computed efficiently but for which an efficient adversary with access to $\mathsf{f}\ x$ only 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 example of hardcore predicates is particularly challenging, since it involves computations that are not hereditarily polynomial-time.

From a theoretical perspective, our work provides a sound and complete type inference algorithm for an expressive ICC system 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 5.6. All this, in particular, means that $\mathsf{d}\ell\mathsf{T}$ is not merely a simplification on $\mathsf{d}\ell\mathsf{PCF}$ [13, 14]. From a practical point of view, our work opens the perspective to

[1]The original CEK machine comes from [16] and its name comes from the fact that its states have three components, a term, an environment and a continuation.

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 $\mathsf{d}\ell\mathsf{T}$ is left for future work, but we briefly discuss the principles of such an integration in Section 9.

This paper is a revised and extended version of the conference paper [3]. With respect to this conference version the $\mathsf{d}\ell\mathsf{T}$ type system and the type inference algorithm have been slightly revised so as to make it easier to establish the soundness of the type inference procedure. The proofs of intensional soundness (Theorem 1), soundness of type inference (Theorem 2), as well as termination of the generated equational programs (Theorem 4) had to be omitted in the short version and are provided here.

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; however, it is possible to extend the language to arbitrary (strictly positive) inductive types. 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 define 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. Variables are denoted as x, y ..., locations as r, q ..., function symbols as $\mathtt{f}$, $\mathtt{g}$ ... *Terms*, *values*, and *base values* are defined as in Figure 1. Two operational semantics will be given later for (a subset of) this language, a reduction semantics in Sect. 2.4 and an abstract machine in Sect. 5, but let us here already introduce some notations and explain the intended meaning of some of the constructions:

- We denote as $\underline{n}$ the term $\mathbf{succ}(\dots\mathbf{succ}(\mathbf{zero})\dots)$, with n occurrences of **succ**, representing the integer n.
- We denote as $(M_1, \dots M_n)$ the term $\mathbf{cons}(M_1, \dots \mathbf{cons}(M_n, \mathbf{nil})\dots)$, with n occurrences of **cons**, representing the list composed of $M_1, \dots M_n$.
- Pairs can be created with the construction $\langle V, W\rangle$ and used with $\mathtt{let}\ M\ \mathtt{be}\ \langle x, y\rangle\ \mathtt{in}\ N$;
- Note that there are two versions of successors, for terms $\mathtt{succ}(\cdot)$ and for values $\mathbf{succ}(\cdot)$ (similarly for $\mathtt{cons}(\cdot)$ and $\mathbf{cons}(\cdot)$): this is harmless for expressivity and there is a technical reason, related to the proof of the forthcoming intensional soundness Theorem (Lemma 2 and Theorem 1).
- The iterator for integers $\mathtt{iter}$ is meant to evaluate $\mathtt{iter}(V, W)\ \underline{n}$ to $V(\dots V(W))$ with n occurrences of V.
- The iterator for lists $\mathtt{fold}$ is meant to evaluate $\mathtt{fold}(V, W)\ (M_1, \dots M_n)$ to $V(M_1(V(M_2 \dots V\ M_n\ W)\dots)$ with n occurrences of V.
- There are three conditionals or selectors, to test respectively booleans, integers and lists; they are meant to evaluate in the following ways:
  $\mathtt{if}(V, W)\ \mathbf{tt}$ and $\mathtt{if}(V, W)\ \mathbf{ff}$ resp. to V and W,
  $\mathtt{ifz}(V, W)\ \mathbf{zero}$ and $\mathtt{ifz}(V, W)\ \underline{n+1}$ resp. to V and $W\ \underline{n}$,
  $\mathtt{ifn}(V, W)\ \mathbf{nil}$ and $\mathtt{ifz}(V, W)\ \mathbf{cons}(M_1, M_2)$ resp. to V and $W\ M_2$.

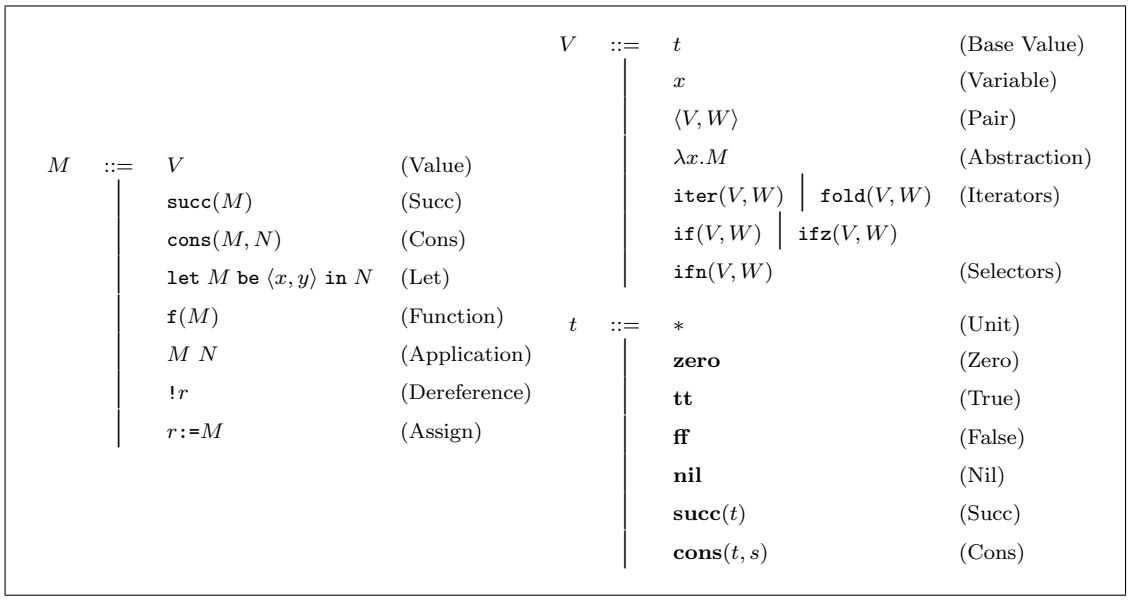

Figure 1: Terms, Values, and Base Values

- References are managed with the two operations of assignment $r\texttt{:=}M$ and dereferencing $\texttt{!}r$. An example of usage will be given in Sect. 2.2, Example 1.

We also defined the following derived constructions:

- denote $\texttt{if}\ M\ \texttt{then}\ V\ \texttt{else}\ W$ for $\texttt{if}(V, W)\ M$,
- denote $\lambda\langle x, y\rangle.N$ for $\lambda z.\texttt{let}\ z\ \texttt{be}\ \langle x, y\rangle\ \texttt{in}\ N$,
- denote $\texttt{let}\ x = M\ \texttt{in}\ N$ for $(\lambda x.N)M$,
- denote $\texttt{let}\ \langle x, y\rangle = M\ \texttt{in}\ N$ for $\texttt{let}\ M\ \texttt{be}\ \langle x, y\rangle\ \texttt{in}\ N$,
- denote $M; N$ for $(\lambda x.N)M$ (sequential composition),
- denote $\texttt{map}(V)$ for $\texttt{fold}(\lambda x.\lambda y.\mathbf{cons}(Vx, y))$.

Note that the intended meaning of $\texttt{map}(V)$ is thus that $\texttt{map}(V)\ (M_1, \dots M_n)$ should evaluate to $(V\ M_1, \dots, V\ M_n)$. The expression $\mathbf{M}$ will stand for a sequence of terms.

2.2 Linear Type System

We first equip the language with a (non-dependent) linear type system. Our goal is to define by means of this type system a linear-affine variant of Gödel's T with pairs, references, and inductive types, that we call $\ell\mathsf{T}$.

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 finite sets of locations. The types $\mathbf{B}$, $\mathbf{N}$ and $\mathbf{L}(T)$ stand respectively for booleans, integers, and lists over the base type T. If a is the empty set $\emptyset$, we write $A \multimap B$ for $A \overset{\emptyset}{\multimap} B$. *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 closed values which can be given base type T can be easily defined by induction on the structure of T. As an example, $\mathcal{T}(\mathbf{N}) = \{\underline{n} \mid n \in \mathbb{N}\}$.

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

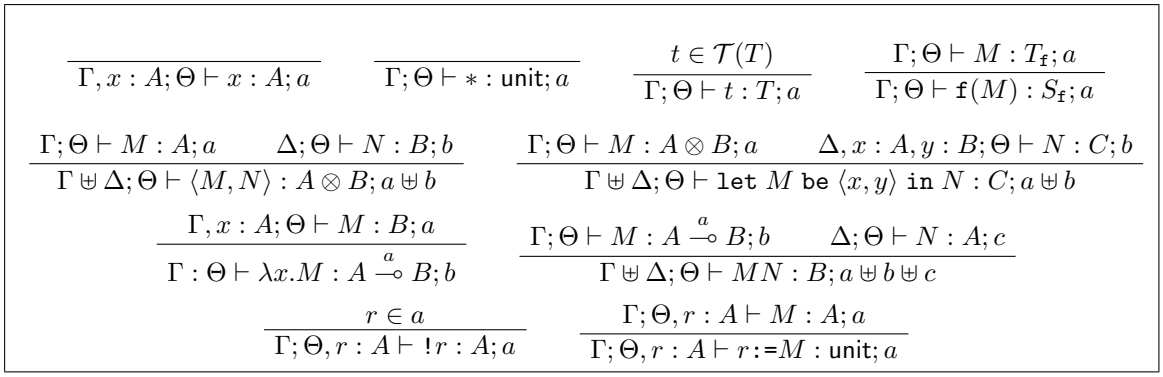

Figure 2: Linear Typing Rules, Part I

The $\ell\mathsf{T}$ typing judgements are of the form $\Gamma; \Theta \vdash M : A; a$. This judgement means that when assigning to free variables types given in Γ and to references types given in 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 $\ell\mathsf{T}$ typing rules are displayed on Figure 2 and Figure 3. Let us just comment on a few rules:

- *Assign and Dereference (Figure 2)*: note that in the Dereference rule (rule for $!r$) the reference r must belong to the set a, while in the Assign rule (rule for $r := M$) there is no condition on a but r and M must have the same type.
- *Pair (Figure 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 (Figure 2)*: note how, reading the rule top-down, the set a is "moved" from the judgement to the type $A \overset{a}{\multimap} B$, and can in this way be used later in the derivation if $\lambda x.M$ is applied to an argument. Observe that the b in the conclusion judgement is arbitrary.
- *Functions (Figure 2)*: note here that each undefined function symbol $\mathtt{f}$ is attributed an input base type $T_{\mathtt{f}}$ and an output base type $S_{\mathtt{f}}$.
- *Iteration (Figure 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.

Derivations will be denoted as $\pi, \rho \ldots$, and we will write $\pi \triangleright \Gamma; \Theta \vdash M : A; a$ to mean that π is a type derivation of conclusion $\Gamma; \Theta \vdash M : A; a$. This notation will also be kept for the other type systems in this paper. We say that a term M is an $\ell\mathsf{T}$ term if there exists an $\ell\mathsf{T}$ type derivation of a judgement $\Gamma; \Theta \vdash M : A; a$.

Example 1 *In order to illustrate the properties of this type system, consider the two following examples of terms:*

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

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:

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

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

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

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

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

Figure 3: Linear Typing Rules, Part II

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.

Example 2 *Let us now give another example for computing addition and multiplication on natural numbers, in an imperative style:*

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

We can then derive:

$$\begin{array}{l} \emptyset; r : \mathbf{N} \vdash incr_r : \mathsf{unit} \overset{r}{\multimap} \mathsf{unit}; \emptyset \\ \emptyset; r : \mathbf{N}, q : \mathbf{N} \vdash add_{r,q} : \mathsf{unit} \overset{r,q}{\multimap} \mathsf{unit}; \emptyset \\ \emptyset; r : \mathbf{N}, q : \mathbf{N}, s : \mathbf{N} \vdash mult_{r,q,s} : \mathsf{unit}; \{r,q,s\} \end{array}$$

Observe that the abstraction λx in the $incr_r$ term is jut needed for this term to have a type of the shape $A \overset{a}{\multimap} A$ so that it can be iterated.

Example 3 *Finally we now give a toy example of a higher-order iteration term. This example will be useful later when we will want to illustrate the type inference procedure for the forthcoming type system, called* $\mathsf{d}\ell\mathsf{T}$.

$$\begin{array}{rcl} BASE & = & \lambda x.\mathtt{succ}(x) \\ STEP & = & \lambda f.\lambda x.(f\ (\mathtt{succ}(\mathtt{succ}(x)))) \\ M & = & \mathtt{iter}(STEP, BASE) \end{array}$$

As this term does not use any reference we omit effects in types (they are all equal to $\emptyset$). We have the following types:

$$\vdash BASE : \mathbf{N} \multimap \mathbf{N}, \quad \vdash STEP : (\mathbf{N} \multimap \mathbf{N}) \multimap (\mathbf{N} \multimap \mathbf{N}), \quad \vdash M : \mathbf{N} \multimap \mathbf{N} \multimap \mathbf{N}.$$

Note that the type system $\ell\mathsf{T}$ that we have introduced actually 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 $\mathbf{S}_\mathtt{f} \subseteq \mathcal{T}(T_\mathtt{f}) \times \mathcal{T}(S_\mathtt{f})$ 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 of 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 of $\ell\mathsf{T}$ evaluate is to give a small-step operational semantics. Actually this small-step operational semantics will not be used in the rest of the paper because we will use instead the abstract machine semantics to be introduced in the following section. We nevertheless define this small-step operational semantics here, for a subset of the language $\ell\mathsf{T}$, because it is somehow more standard: reduction semantics is the standard way to give computational meaning to calculi in the style of λ-calculus. However this section can be skipped without any harm.

Evaluation takes place in *evaluation contexts*, which are generated by the following grammar:

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

Given a term M, $\mathsf{E}[M]$ will denote the term obtained by replacing the hole $[\cdot]$ in E by M.

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 for 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 $\mathtt{f}$.

Example 4 *One can check with Example 3 that* $(\mathcal{S}, M\ \underline{n}\ \underline{m})$ *reduces to* $(\mathcal{S}, \underline{n+2m+1})$.

2.5 Abstract Machine

Reduction semantics as introduced in Section 2.4 is completely satisfactory from an extensional point of view, because it allows to reduce any typable term to its normal form, at the same time appropriately modeling the interaction between the term and the store. Certain rules in the reduction semantics, in particular those for β-reduction and for $\mathtt{fold}$-iteration are not atomic: the amount of computational work an actual *machine* needs to perform to implement them is not constant, although one might hope to get at least a (low-rank) polynomial overhead [1]. For these reasons, an *abstract machine* for our calculus is introduced in this Section. In this abstract machine, reduction rules are replaced by more atomic operations in which even the process of *looking for* a redex is appropriately taken into account.

We consider a variant of Felleisen and Friedman's CEK [16], called $\mathsf{CEK}_{\ell\mathsf{T}}$. 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 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, respectively:

$$c ::= (M, \xi); \qquad \xi ::= \varepsilon \mid \xi \cdot (x \mapsto c); \qquad \pi ::= \varepsilon \mid \delta \cdot \pi;$$

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

Figure 4: Evaluation Rules.

where δ ranges over *stack elements*:

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

We will call *value closures* and denote as v, closures of the form (V, ξ), where V is a value.

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, this whenever $r : A$ is in Θ. *Machine 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. We will call *machine preconfigurations* pairs of the form (c, π). Machine transition 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:

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

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 correpond to the costs of ordinary models of computation (TMs, RAMs, etc.), modulo a polynomial overhead. Moreover, the induced overhead is arguably lower than that induced by reduction itself: applying machine transitions amounts to checking whether the current configuration has a certain shape, and move some of the closures and information around. In some cases, environments need to be copied, but this can be implemented through sharing.

$$
\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 V, W\rangle, \xi), \texttt{let}((M, \theta), x, y) \cdot \pi, \mathcal{S}) & \succ^{1} & ((M, \xi \cdot (x \mapsto (V, \theta)) \cdot (y \mapsto (W, \theta)), \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}(V, W), \xi), \texttt{arg}(X, \theta) \cdot \pi, \mathcal{S}) & \succ^{1} & ((X, \theta), \texttt{iter}(V, W, \xi) \cdot \pi, \mathcal{S}); \\
((\mathbf{succ}(t), \theta), \texttt{iter}(V, W, \xi) \cdot \pi, \mathcal{S}) & \succ^{1} & ((t, \theta), \texttt{iter}(V, W, \xi) \cdot \texttt{fun}(V, \xi) \cdot \pi, \mathcal{S}); \\
((\mathbf{zero}, \theta), \texttt{iter}(V, W, \xi) \cdot \pi, \mathcal{S}) & \succ^{1} & ((W, \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|)} & ((s, \xi), \pi, \mathcal{S}) \text{ if } (t, s) \in \mathbf{S}_{\mathtt{f}}; \\
((!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}
$$

Figure 5: Abstract Machine Transition Rules

3 Linear Dependent Types

There is nothing in $\ell\mathsf{T}$ types which allows to enforce complexity bounds for typable programs; in fact, $\ell\mathsf{T}$ can express at least all the primitive recursive functions [12]. This is precisely the role played by linear dependency [13], whose underlying idea consists in decorating simple types with some information about the identity of objects programs manipulate. This takes the form of so-called index terms, following in spirit Xi's DML [29]. Differently from [13], 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, ruling out types of the form $!_{x<p}D$. In the following we will therefore define a typed language called $\mathsf{d}\ell\mathsf{T}$, delineating a subset of the $\ell\mathsf{T}$ programs.

First, 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

$$I \ ::= \ a \ \Big| \ f(I, \ldots, I),$$

where $a \in \mathcal{IV}$ and $f \in \mathcal{IF}$. Index functions f are interpreted as *total* functions $[\![f]\!]$ from n-uples of positive natural numbers to positive natural numbers, where n is the arity of f. An *assignment* ρ is a map of index variables to natural numbers. Given an assignment, we can interpret in a standard way any index term I by a natural number $[\![I]\!]_\rho$, where ρ is an assignment. The interpretation $[\![f]\!]$ will actually be defined 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. In the present paper concretely we use term rewriting systems, but we will continue to call them equational programs.

A *constraint* is any expression of the form $I \leq J$. A finite set of constraints will be indicated with metariables like ϕ, ψ. The constraints set $\phi\{a/I\}$ is defined from ϕ by substituting each occurrence of a in an index term by I. We say that an assignment ρ of index variables to integers *satisfies a constraint* $I \leq J$ if we have $[\![I]\!]_\rho \leq [\![J]\!]_\rho$. We say that ρ satisfies the set ϕ if it satisfies all the constraints of ϕ. We often write $\models^{\mathcal{E}} I \leq J$, by which we mean that $[\![I]\!]_\rho \leq [\![J]\!]_\rho$ for all assignments ρ of index variables to integers, If the same is required only for those assignments which satisfy the constraints in ϕ, then we write $\phi \models^{\mathcal{E}} I \leq J$.

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 $\mathcal{E}$. With a slight abuse of notation, we will denote k for $s(\ldots s(0)\ldots)$ with k occurrences of s.

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

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

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

Figure 6: Subtyping Rules

We now define types and effects. *Indexed base types* are defined as follows:

$$U \ ::= \ \mathsf{unit} \ \Big| \ \mathbf{B} \ \Big| \ \mathbf{N}^I \ \Big| \ \mathbf{L}^I(U)$$

where I ranges over index terms. *Indexed reference contexts* Θ and *indexed effects* α are defined, respectively, as follows:

$$\Theta \ ::= \ \{r_1 : D_1, \ldots, r_n : D_n\}; \qquad\qquad \alpha \ ::= \ \Theta \Rightarrow \Theta.$$

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 $\Theta \Rightarrow \Upsilon$. We denote as Θ, Ξ the union of two indexed reference contexts Θ and Ξ. Finally, *indexed types* are given by the following grammar:

$$D \ ::= \ U \ \Big| \ D \otimes D \ \Big| \ D \overset{\alpha}{\multimap} D.$$

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. For instance: 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 $\phi \models^{\mathcal{E}} t \in U$ iff $[U] = T$ and for any assignment ρ satisfying ϕ, the size of t is boundes by $[\![I]\!]_\rho$. Similarly, $\phi \models^{\mathcal{E}} \mathtt{f} \in U \multimap V$ stands for $\phi \models^{\mathcal{E}} s \in V$ whenever $\phi \models^{\mathcal{E}} t \in U$ and $(t, s) \in \mathbf{S}_{\mathtt{f}}$.

A subtyping relation $\sqsubseteq$ on indexed types and effects is defined in Figure 6. Note that we have $\phi \vdash^{\mathcal{E}} \mathbf{0} \sqsubseteq \alpha$ iff α is of the shape $\alpha = \Xi, \Upsilon \Rightarrow \Pi$, where $\phi \vdash^{\mathcal{E}} \Upsilon \sqsubseteq \Pi$. 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 ones we used in the linear type system (Section 2) where, of course, *indexed types* now play the role of *types*. A $\mathsf{d}\ell\mathsf{T}$ typing judgement has the form $\phi; \Gamma \vdash^{\mathcal{E}} M : D; \alpha$, where ϕ is a finite (possibly empty) set of constraints. If $\phi = \emptyset$ we will simply write the judgement as $\Gamma \vdash^{\mathcal{E}} M : D; \alpha$.

Let us denote $\alpha = \Theta \Rightarrow \Xi$. The intended meaning of the judgement $\phi; \Gamma \vdash^{\mathcal{E}} M : D; \alpha$ is that if term variables are assigned types in Γ and assuming the constraints ϕ are satisfied, 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 content of references might have been modified, more specifically about how the size of the contents might have changed. If Γ (resp. Δ) is a sequence $D_1, \ldots, D_n$ (resp. $E_1, \ldots, E_n$) then the notation $\phi \vdash^{\mathcal{E}} \Gamma \sqsubseteq \Delta$ stands for $\{\phi \vdash^{\mathcal{E}} D_i \sqsubseteq E_i\}_{1 \leq i \leq n}$.

Now, the $\mathsf{d}\ell\mathsf{T}$ typing rules for a subset of this language are given in Figure 7 and Figure 8. We say that a term M is *dependently linearly typable*, or that it is a $\mathsf{d}\ell\mathsf{T}$ term, iff there exists a derivation of a judgement $\phi; \Gamma \vdash^{\mathcal{E}} M : D; \alpha$. Before stating the properties of the $\mathsf{d}\ell\mathsf{T}$ type system, let us make a few comments:

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

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

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

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

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

Figure 7: *dlT* Typing Rules, Part I

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

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

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

Figure 8: *dlT* Typing Rules, Part II

- 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 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 **0** effect. This is quite intuitive, since values are meant to be terms which need not be further evaluated.
- Observe that the constraints ϕ in judgements only play a rôle in the typing rules for constructors (for instance $\mathtt{succ}(M)$), iterators ($\mathtt{iter}(V,W)$) and selectors ($\mathtt{ifz}(V,W)$) (see Figure 8).
- 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.

Remark 1 *We opt for a syntax-directed presentation of the type system: there is a single rule per construct, and each rule embodies subtyping constraints. A common alternative is to use a subsumption rule, which states that a term M of type D also has type E if D is a subtype of E. The choice of a syntax-directed presentation simplifies the proof of decidability of type inference.*

Remark 2 *Actually, the constraints ϕ occurring in typing judgements $\phi; \Gamma \vdash^{\mathcal{E}} M : D; \alpha$ are not an essential feature of the linear dependent type system. All the important properties of this paper could be stated without using these constraints in judgements, and indeed the former paper [3] did not mention them. The reason for including them is that they are convenient for proving some properties, in particular the fact that the type inference algorithm that we will define in Sect. 5 is sound (see the proof of Theorem 2). So on a first read the reader might just as well ignore these constraints. We will also omit them in the examples to come, for improving readability.*

In the following we will sometimes need to perform some substitutions in derivations. Given a derivation π, an index I and an index variable a, the tree $\pi\{a/I\}$ (which is a tentative derivation) is obtained from π by replacing in each judgement and statement any index J by $J\{a/I\}$, any effect α by $\alpha\{a/I\}$ and any type D by $D\{a/I\}$. We then have the following properties:

Lemma 1 *1. If $\phi \models^{\mathcal{E}} J_1 \leq J_2$, then $\phi\{a/I\} \models^{\mathcal{E}} J_1\{a/I\} \leq J_2\{a/I\}$;*
2. If $\phi \vdash^{\mathcal{E}} D \sqsubseteq E$, then $\phi\{a/I\} \vdash^{\mathcal{E}} D\{a/I\} \sqsubseteq E\{a/I\}$;
3. If π is a linear dependent type derivation, then so is $\pi\{a/I\}$.

Proof. Point 1. is a consequence of the definition of $\phi \models^{\mathcal{E}} J_1 \leq J_2$. Point 2 is proved by the definition of subtyping, by induction on D. Point 3. is proved by induction on π, by showing that each rule in $\pi\{a/I\}$ is valid, by using points 1. and 2. in order to show that the side conditions are satisfied.

Let us now turn to some examples. We will illustrate various aspects of the $\mathsf{d}\ell\mathsf{T}$ type system, first examining the case of functional terms (without effects), and then examining the case where effects are used.

Example 5 *Consider the terms BASE and STEP of Example 3.*

$$BASE = \lambda x.\mathtt{succ}(x), \qquad STEP = \lambda f.\lambda x.(f\ (\mathtt{succ}(\mathbf{succ}(x))))$$

One can get the following type judgements in $\mathsf{d}\ell\mathsf{T}$, where we omit effects since they are all $\emptyset$:

$$\vdash^{\mathcal{E}} BASE : \mathbf{N}^{r(a)} \multimap \mathbf{N}^{s(a)}$$
$$\vdash^{\mathcal{E}} STEP : (\mathbf{N}^{h(a,b)} \multimap \mathbf{N}^{g(a,b)}) \multimap (\mathbf{N}^{p(a,b)} \multimap \mathbf{N}^{q(a,b)})$$

with $\mathcal{E}$ containing the following equations:

$$\begin{array}{lcl} s(a) & \to & r(a)+1 \\ q(a,b) & \to & g(a,b) \\ h(a,b) & \to & p(a,b)+2 \end{array}$$

The choice of having one argument for functions s, r, and two for functions p, q, g, h, is somewhat conventional at this stage, but in general we will choose as number of arguments the number of occurrences of base types in negative position in the type (hence 1 in $\mathbf{N} \multimap \mathbf{N}$ and 2 in $(\mathbf{N} \multimap \mathbf{N}) \multimap (\mathbf{N} \multimap \mathbf{N})$). Now, notice that there are no equations defining index functions r, g and p, that is to say index functions corresponding to occurrences of base types in negative position. This is a general situation. Hence if we are interested only in typing these terms individually, we can pick the particular types obtained by setting the values of these functions as projections:

$$r(a) = a, \quad g(a,b) = a, \quad p(a,b) = b.$$

We then obtain the following specific types, where we have replaced all index functions by their explicit value for more clarity:

$$\vdash^{\mathcal{E}} BASE : \mathbf{N}^a \multimap \mathbf{N}^{a+1}$$
$$\vdash^{\mathcal{E}} STEP : (\mathbf{N}^{b+2} \multimap \mathbf{N}^a) \multimap (\mathbf{N}^b \multimap \mathbf{N}^a)$$

Note however that if we wanted to use the types of BASE and STEP for typing a larger term, for instance the term $M = \mathtt{iter}(STEP, BASE)$ of Example 3, then we would need to keep the initial general types with undefined (negative) index functions r, g, p, because the additional typing conditions for M will bring more equations defining some of these functions.

Actually we will not provide here a type derivation for $M = \mathtt{iter}(STEP, BASE)$, because it is not straightforward. Indeed the difficulty for applying the $\mathtt{iter}(V,W)$ typing rule of $\mathsf{d}\ell\mathsf{T}$ here (see Figure 8) is to find suitable types E and F satisfying the subtyping conditions in premises of the rule. This is a situation where the type inference algorithm is useful, and we will come back to this example in Sect. 5.

Example 6 *Let us examine a particular case of iteration on a base type. Assume that we have two closed terms M_{BASE} and M_{STEP} of respective type $\mathbf{N}$ and $\mathbf{N} \multimap \mathbf{N}$ and that we have obtained $\mathsf{d}\ell\mathsf{T}$ type judgements of the following form:*

$$\vdash^{\mathcal{E}} M_{BASE} : \mathbf{N}^k$$
$$\vdash^{\mathcal{E}} M_{STEP} : \mathbf{N}^a \multimap \mathbf{N}^{a+k}$$

where we recall that k is the index term $s(\ldots s(0))\ldots$ (k applications of s). Let us try to type the term $\mathtt{iter}(M_{STEP}, M_{BASE})$. For that, considering the typing rule of Figure 8, we need to replace the type $\mathbf{N}^a \multimap \mathbf{N}^{a+k}$ by a type of the form $D \multimap D\{b/b+1\}$ and to find a type E such that $\vdash^{\mathcal{E}} D \sqsubseteq E$ and $\vdash^{\mathcal{E}} E \sqsubseteq E\{b/b+1\}$.

But by Lemma 1, taking $I = k \cdot b$ he have $\vdash^{\mathcal{E}} M_{STEP} : \mathbf{N}^{k \cdot b} \multimap \mathbf{N}^{k \cdot b + k}$. So we can take $D = \mathbf{N}^{k \cdot b} = E$. As $\vdash^{\mathcal{E}} \mathbf{N}^{k \cdot b} \sqsubseteq \mathbf{N}^{k \cdot (b+1)}$, we have that $\vdash^{\mathcal{E}} E \sqsubseteq E\{b/b+1\}$. Therefore we can apply the $\mathtt{iter}(V,W)$ typing rule, choosing $F = E\{b/I\}$, for an arbitrary I, and we obtain:

$$\vdash^{\mathcal{E}} \mathtt{iter}(M_{STEP}, M_{BASE}) : \mathbf{N}^I \multimap \mathbf{N}^{k \cdot I}.$$

Example 7 *Consider now the term M of Example 1 given in Sect. 2, which uses references. We can derive the following typing judgement:*

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

Example 8 *Let us now examine the typing of the terms of Example 2 in Sect. 2 for increment, addition and multiplication. We obtain:*

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

4 Intensional Soundness

Once a term M has been typed by a $\mathsf{d}\ell\mathsf{T}$ derivation π, we will assign a *weight* $\mathbb{W}(\pi)$ to π (and thus indirectly to M) in the form of an index term I. The quantity $\mathbb{W}(\pi)$ will strictly decrease at earch transition step, this way representing an upper bound to the length of transition sequences.

The weight $\mathbb{W}(\pi)$ will be defined by induction on the structure of π. Actually not all the information carried by the derivation π will be useful for defining the weight: note that many premises of the rules in Figure 7 and 8 are actually *proof obligations* consisting in subtyping conditions (e.g. $\phi \vdash^{\mathcal{E}} D \sqsubseteq E$ in the $\mathtt{iter}(V,W)$ rule) or constraints conditions (e.g. $\phi \models^{\mathcal{E}} I{+}1 \leq J$ in the $\mathtt{succ}$ rule). Let us call a *bare derivation* π a $\mathsf{d}\ell\mathsf{T}$ derivation where all proof obligations premises have been omitted, and where in judgements the constraints ϕ have also been dropped (note Remark 2); for instance in a bare derivation the typing rule for $\mathtt{iter}(V,W)$ becomes:

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

So the weight $\mathbb{W}(\pi)$ will actually be defined by induction on the structure of the bare derivation π; interesting cases are those for iteration and function:

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

$$\pi \triangleright \frac{\begin{array}{c}\rho \triangleright \ell\Gamma \vdash^{\mathcal{E}} W : D\{a/1\}; \mathbf{0} \\ \sigma \triangleright \ell\Gamma \vdash^{\mathcal{E}} V : \left(D \overset{\Xi \Rightarrow \Xi\{a/a+1\}}{\multimap} D\{a/a+1\}\right); \mathbf{0}\end{array}}{\ell\Gamma \vdash^{\mathcal{E}} \mathtt{iter}(V,W) : \mathbf{N}^I \overset{\Theta \Rightarrow \Phi}{\multimap} F; \alpha} \qquad \mathbb{W}(\pi) = \mathbb{W}(\rho) + \sum_{1 \leq a < I} \mathbb{W}(\sigma) + 4I$$

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. The overhead $4I$ accounts for the transition steps needed to unfold the iteration. 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 or 2. 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.

Please observe how (the type derivation for) any ground value t is attributed *null* weight. Indeed, since we want ground values to be duplicable, they *must* have null weight, otherwise the weight could not decrease along a computation. Ultimately, this is why we need to have two instances of cons and succ in the language of terms: the former needs to have a *non-null* weight because computing basic operations on lists and natural numbers takes a bit of time anyway, while the others can have *null* weight.

The weight $\mathbb{W}(\pi)$ of π does *not* necessarily decrease along reduction as defined in Section 2.4. But the weight of a derivation *does* decrease along *machine* transition rules, as defined in Section 2.5. Indeed, this is one technical reason why the $\mathsf{CEK}_{\ell\mathsf{T}}$ abstract machine is convenient in this context, the other one being that its transition rules are computationally less heavy to be carried out than rewriting on terms (see again Setion 2.5 for a more thorough discussion about these issues). This can be proved by generalizing $\mathsf{d}\ell\mathsf{T}$ to a type system for machine preconfigurations. This extension is in Figure 10. The typing rules for closures deserves to be explained. The typing context Ω is disjoint from both Γ and Δ. Observe that the substitution θ does not "affect" the term M, because the variables in Ω cannot be free in M. The reason why we need to give weight to closures in this complex way is that the third transition rule of the Abstract Machine (Figure 5) *duplicates* environments. And this is of course incompatible with weight decreasing if we do not handle this issue with great care. About typing rules for stacks, the treatment of higher-order functions needs to be commented. The two stack elements dealing with it, namely $\mathtt{fun}(\cdot)$ and $\mathtt{arg}(\cdot)$, are given different weights, the first one being 1 and the second one being 0: again, this

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

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

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

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

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

$$\pi \triangleright \frac{\rho \triangleright \Gamma \vdash^{\mathcal{E}} M : D \otimes E; \alpha \qquad \sigma \triangleright \Delta, x : D, y : E \vdash^{\mathcal{E}} N : F; \beta}{\Gamma \uplus \Delta \vdash^{\mathcal{E}} \mathtt{let}\ M\ \mathtt{be}\ \langle x, y \rangle\ \mathtt{in}\ N : F; \alpha; \beta} \qquad \mathbb{W}(\pi) = \mathbb{W}(\rho) + \mathbb{W}(\sigma) + 2$$

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

$$\pi \triangleright \frac{\rho \triangleright \Gamma \vdash^{\mathcal{E}} M : D \overset{\gamma}{\multimap} E; \alpha \qquad \sigma \triangleright \Delta \vdash^{\mathcal{E}} N : D; \beta}{\Gamma \uplus \Delta \vdash^{\mathcal{E}} MN : E; \alpha; \beta; \gamma} \qquad \mathbb{W}(\pi) = \mathbb{W}(\rho) + \mathbb{W}(\sigma) + 2$$

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

$$\pi \triangleright \frac{\rho \triangleright \Gamma \vdash^{\mathcal{E}} M : D; \Theta \Rightarrow \Xi, \{r : E\}}{\Gamma \vdash^{\mathcal{E}} r\mathtt{:=}M : \mathsf{unit}; \Theta \Rightarrow \Xi, \{r : D\}} \qquad \mathbb{W}(\pi) = \mathbb{W}(\rho) + 2$$

$$\pi \triangleright \frac{\Gamma \vdash^{\mathcal{E}} M : \mathbf{N}^I; \alpha}{\Gamma \vdash^{\mathcal{E}} \mathtt{succ}(M) : \mathbf{N}^J; \alpha} \qquad \mathbb{W}(\pi) = \mathbb{W}(\rho) + 2$$

$$\pi \triangleright \frac{\begin{array}{c}\rho \triangleright \ell\Gamma \vdash^{\mathcal{E}} W : D\{a/1\}; \mathbf{0} \\ \sigma \triangleright \ell\Gamma \vdash^{\mathcal{E}} V : \left(D \overset{\Xi \Rightarrow \Xi\{a/a+1\}}{\multimap} D\{a/a+1\}\right); \mathbf{0}\end{array}}{\ell\Gamma \vdash^{\mathcal{E}} \mathtt{iter}(V, W) : \mathbf{N}^I \overset{\Theta \Rightarrow \Phi}{\multimap} F; \alpha} \qquad \mathbb{W}(\pi) = \mathbb{W}(\rho) + \sum_{1 \leq a < I} \mathbb{W}(\sigma) + 4I$$

$$\pi \triangleright \frac{\begin{array}{c}\rho \triangleright \Gamma \vdash^{\mathcal{E}} V : D\{a/1\}; \mathbf{0} \\ \sigma \triangleright \Gamma \vdash^{\mathcal{E}} W : \left(\mathbf{N}^a \overset{\beta\{a/a+1\}}{\multimap} D\{a/a+1\}\right); \mathbf{0}\end{array}}{\Gamma \vdash^{\mathcal{E}} \mathtt{ifz}(V, W) : \mathbf{N}^I \overset{\gamma}{\multimap} E; \alpha} \qquad \mathbb{W}(\pi) = \mathbb{W}(\rho) + \mathbb{W}(\sigma) + 2$$

Figure 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

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

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

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

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

$$\pi \triangleright \frac{\models^{\mathcal{E}} \mathtt{f} \in \mathbf{N}^I \multimap U \qquad \rho \triangleright \vdash^{\mathcal{E}} \pi : U \rightsquigarrow D; \alpha}{\vdash^{\mathcal{E}} \mathtt{ufun}(\mathtt{f}) \cdot \pi : \mathbf{N}^I \rightsquigarrow D; \alpha} \qquad \mathbb{W}(\pi) = \mathbb{W}(\rho) + \mathbf{C}_{\mathtt{f}}(I)$$

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

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

Machine Preconfigurations

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

Figure 10: Typing and Weighting Machine Preconfigurations

is due to weight-decreasing, which we need to guarantee for any machine transition step. Please contrast this with $\mathtt{ufun}(\cdot)$ which, instead, only deals with first-order undefined symbols.

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

Lemma 2 *If $\pi\triangleright \vdash^{\mathcal{E}} (c,\pi) : U;\Theta \Rightarrow \Xi$, the store $\mathcal{S}$ is conformant with Θ, and $(c,\pi,\mathcal{S}) \succ^n (d,\rho,\mathcal{R})$, then $\rho\triangleright \vdash^{\mathcal{E}} (d,\rho) : U;\Upsilon \Rightarrow \Xi$, 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. Remind the remark made in Section 2.1 on the two versions of constructors, $\mathtt{succ}(\cdot)$ for terms and $\mathbf{succ}(\cdot)$ for values: the reason for this distinction is that on the one hand we need ground-values to have null weight, on the other we want every transition of the abstract machine to make the weight of the underlying term to strictly decrease. Now here are some interesting cases of the proof:

- One case which is worth being analysed is the one in which

$$((MN,\xi),\pi,\mathcal{S}) \succ^1 ((N,\xi),\mathtt{fun}(M,\xi)\cdot\pi,\mathcal{S})$$

This implies the following:

$$\begin{aligned}\sigma\triangleright &\vdash^{\mathcal{E}} ((MN),\xi) : D;\alpha.\\ \upsilon\triangleright &\vdash^{\mathcal{E}} \pi : D \rightsquigarrow U;\beta,\\ \alpha;\beta &= \Theta \Rightarrow \Xi,\end{aligned}$$

and that $\mathcal{S}$ is conformant with Θ. In turn, this implies that

$$\begin{aligned}\Gamma &\vdash^{\mathcal{E}} MN : D;\alpha,\\ &\vdash^{\mathcal{E}} \theta : \Gamma,\\ &\vdash^{\mathcal{E}} \upsilon : \Omega,\end{aligned}$$

where $\xi = \theta\cdot\upsilon$. By the typing rule for applications, it must be that $\Gamma = \Gamma_M \uplus \Gamma_N$, that $\alpha = \gamma_N;\gamma_M;\delta$ and

$$\begin{aligned}\Gamma_M &\vdash^{\mathcal{E}} M : E \stackrel{\delta}{\multimap} F;\gamma_M,\\ \Gamma_N &\vdash^{\mathcal{E}} N : E;\gamma_N.\end{aligned}$$

Now, observe that Γ can be partitioned into its ground part, call it $\ell\Gamma$, and two higher-order parts, call them Δ_M and Δ_N, in such a way that θ can be itself partitioned into $\ell\theta$, υ_M, and υ_N, in such a way that

$$\begin{aligned}\Gamma_M &= \Delta_M,\ell\Gamma, & \Gamma_N &= \Delta_N,\ell\Gamma,\\ &\vdash^{\mathcal{E}} \ell\theta,\upsilon_M : \Gamma_M, & &\vdash^{\mathcal{E}} \ell\theta,\upsilon_N : \Gamma_N.\end{aligned}$$

We can now assemble the information we have gathered, and conclude that there are type derivations

$$\begin{aligned}\theta\triangleright &\vdash^{\mathcal{E}} (M,\xi) : E \stackrel{\delta}{\multimap} D;\gamma_M,\\ \xi\triangleright &\vdash^{\mathcal{E}} (N,\xi) : E;\gamma_N\end{aligned}$$

from which it easily follows that there is ρ such that

$$\rho \triangleright \vdash^{\mathcal{E}} ((N, \xi), \mathtt{fun}(M, \xi) \cdot \pi) : D; \alpha$$

and that, moreover,

$$\mathbb{W}(\rho) + 1 = \mathbb{W}(\theta) + \mathbb{W}(\xi) + 2 + \mathbb{W}(\upsilon) + 1 \leq \mathbb{W}(\sigma) + \mathbb{W}(\upsilon) = \mathbb{W}(\pi).$$

- Another case which is worth studying is the one in which

$$((\mathtt{f}(M), \xi), \pi, \mathcal{S}) \succ^{1} ((M, \xi), \mathtt{ufun}(\mathtt{f}) \cdot \pi, \mathcal{S}),$$

This implies the following:

$$\begin{aligned} &\sigma \triangleright \vdash^{\mathcal{E}} (\mathtt{f}(M), \xi) : V; \alpha, \\ &\upsilon \triangleright \vdash^{\mathcal{E}} \pi : V \rightsquigarrow U; \beta, \\ &\alpha; \beta = \Theta \Rightarrow \Xi, \end{aligned}$$

This, in turn, implies that

$$\begin{aligned} \Gamma \vdash^{\mathcal{E}} &\mathtt{f}(M) : V; \alpha, \\ \vdash^{\mathcal{E}} &\theta : \Gamma, \\ \vdash^{\mathcal{E}} &\upsilon : \Omega, \end{aligned}$$

where $\Gamma = \Delta \cdot \Omega$. By the typing rule for function application, it must be that the following holds:

$$\models^{\mathcal{E}} \mathtt{f} \in \mathbf{N}^I \multimap V \qquad\qquad \Gamma \vdash^{\mathcal{E}} M : \mathbf{N}^I; \alpha$$

and that

$$\theta \triangleright \vdash^{\mathcal{E}} (M, \xi) : \mathbf{N}^I; \alpha$$

where $\mathbb{W}(\theta) = \mathbb{W}(\sigma) + \mathbb{W}(\upsilon)$. Summing up, there is a derivation ρ of

$$\vdash^{\mathcal{E}} ((M, \xi), \mathtt{ufun}(\mathtt{f}) \cdot \pi) : D; \alpha$$

such that $\mathbb{W}(\rho) + 1 \leq \mathbb{W}(\pi)$.

□

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 easy corollary of Lemma 2.

So this theorem shows that computing the weight of of a $\mathsf{d}\ell\mathsf{T}$ program allows to establish statically a time bound on its execution. 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 $\mathsf{d}\ell\mathsf{T}$ type inference procedure is defined quite similarly to the one in [14], where the language at hand, called $\mathsf{d}\ell\mathsf{PCF}$, is more general than the one described here (apart from the absence of imperative features). 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 do not present specific difficulties.

We call *pseudo-derivation* a tree whose nodes are labelled by typing judgements, but which is not necessarily built according to our type system. More specifically, proof obligations (for instance the subtyping conditions) which are not necessarily satisfied could indeed occur in pseudo-derivations. 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*. Proper equational programs are sometime said to be *completely specified*.

We will proceed in two steps when defining the type inference procedure:

- we will first define a type inference algorithm for linearly typable terms, called $\mathtt{TI}$, which produces in output a pseudo-derivation and a pseudo-program,
- in a second step we will define a second algorithm $\mathtt{CTI}$ using $\mathtt{TI}$ as a subroutine, and we will prove that its output is not only a pseudo-derivation but in fact a valid $\mathsf{d}\ell\mathsf{T}$ type derivation (see Theorem 2).

The type inference algorithm $\mathtt{TI}$ takes in input a linearly typable term M, together with a finite sequence of index variables $\overline{a} = a_1, \ldots, a_n$, 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, $\mathtt{TI}(M^A, \overline{a}) = (\pi, \mathcal{E})$. The understanding here is that the function symbols which occur in $\mathcal{E}$, but which are undefined, are those which occur in *negative* position 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).

As we just mentioned, the input to $\mathtt{TI}$ is a linearly typed term, i.e. a term M together with a linear type derivation for M. For the sake of simplicity, we will rather assume $\mathtt{TI}$ to take in input a term M in *Church-style*, namely a term in which every bound variable is labelled with an affine type A. The information provided by these labelled will be essential at certain stages of the algorithm, e.g., when defining the procedure $\mathtt{weak}(\cdot, \cdot, \cdot)$.

The algorithm $\mathtt{TI}$ is recursive, and proceeds by traversing 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 definable by recursion. Indeed, if we insisted the output of the type inference algorithm to be a *proper* equational program, then we would not have any hope to define the algorithm itself purely by recursion on the input term: there are certain terms, namely those of higher-order type, which can hardly be mapped to proper equational programs in a genuinely inductive way. It is thus convenient to make the class of possible outputs slightly broader, so as to be able to get a purely recursive algorithms.

Before describing the algorithm $\mathtt{TI}$, it is necessary to give some preliminary definitions. We will give them in the following subsections.

5.1 Primitive Index terms, Cutting, and Merging

An index term is said to be *primitive* iff it is in the form $f(\overline{a})$, where $\overline{a}$ 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:

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

The fact that $\mathtt{cut}(D, F)$ is replaced with $\mathtt{cut}(F, D)$ when $\multimap$ takes the place of $\otimes$ is due to the fact that D and F occur in negative position in the former case, and in positive position in the second case.

Given a type A and a sequence $\overline{a}$ of index variables, $\texttt{fresh}(A, \overline{a})$ stands for the indexed type obtained by "decorating" A with fresh primitive index terms on $\overline{a}$. This can be easily defined by induction on A.

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 $\mathbf{N}^{f(\overline{a})}$ and $\mathbf{N}^{g(\overline{a})}$, the typing context Ω contains $x : \mathbf{N}^{j(\overline{a})}$ and $\mathcal{E}$ contains $\{f(\overline{a}) \to j(\overline{a}), g(\overline{a}) \to j(\overline{a})\}$.

5.2 Weakening

We define a partially defined ternary function $\texttt{weak}(\cdot, \cdot, \cdot)$ which takes as inputs tuples (π, x, A) of a derivation $\pi \triangleright \Gamma \vdash^{\mathcal{E}} M : E$, a variable x and a type A, and when it is defined returns a pair $(\rho, \mathcal{F})$ of a derivation ρ and an equational program $\mathcal{F}$.

The result $\texttt{weak}(\pi, x, A)$ is defined or undefined in the following way:

- if x belongs to Γ and $[\Gamma(x)] \neq A$ then $\texttt{weak}(\pi, x, A)$ is undefined,
- if x belongs to Γ and $[\Gamma(x)] = A$ then $\texttt{weak}(\pi, x, A) = (\pi, \emptyset)$;
- otherwise let $D = \texttt{fresh}(A, \overline{a})$; then $\texttt{weak}(\pi, x, A) = (\rho, \mathcal{F})$, where ρ, $\mathcal{F}$ are defined as follows:
  - ρ is obtained from π by adding $x : D$ to the context of the judgements of one branch of the tree representing the derivation;
- $\mathcal{F}$ contains equations $f(\overline{a}) \to 0$, for all index terms $f(\overline{a})$ in positive position in D.

Observe that when $\texttt{weak}(\pi, x, A)$ is defined and equal to $(\rho, \mathcal{F})$, then ρ is indeed a valid derivation. The definition of $\mathcal{F}$ has been done in such a way to maintain the invariant that functions in positive position in the types of the judgement are defined by the equational program (the choice of the value 0 is arbitrary). Now, given two variables x, y and types A, B, we define $\texttt{weak}(\pi, x, A, y, B)$ in the following way:

- if $\texttt{weak}(\pi, x, A) = (\rho_1, \mathcal{F}_1)$ and $\texttt{weak}(\rho_1, y, B) = (\rho_2, \mathcal{F}_2)$, then $\texttt{weak}(\pi, x, A, y, B) = (\rho_2, \mathcal{F}_1 \cup \mathcal{F}_2)$,
- otherwise $\texttt{weak}(\pi, x, A, y, B)$ is undefined.

5.3 Dealing with Iteration

The most delicate case for type inference is probably that of higher-type iteration. For that we will define the procedure $\texttt{itercut}_{\mathbf{p}}(\cdot)$. Some explanation about the intuitions underlying this definition will be given in the forthcoming Example 9. What $\texttt{itercut}_{\mathbf{p}}(\cdot)$ does is to allow to type terms of the form $\texttt{iter}(\mathit{STEP}, \mathit{BASE})$ given type derivations for STEP and BASE. This is a place in which many new index equations need to be added to the underlying equational program, and $\texttt{itercut}_{\mathbf{p}}(\cdot)$ is there to collect all these equations.

In order to define the $\texttt{itercut}_{\mathbf{p}}(\cdot)$ procedure, we will need to define an additional index function $-$, by the following equations which are included in $\mathcal{E}$ (in infix notation):

$$\begin{array}{lcl} 0 - b & \to & 0 \\ s(a) - s(b) & \to & a - b \\ a - 0 & \to & a \end{array}$$

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 we define

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

by the equations in Figure 11, where $\neg : \{+, -\} \to \{+, -\}$ returns the polarity opposite to the input one. The role of the $\texttt{itercut}_{\mathbf{p}}(\cdot)$ can be described as follows. If D and $E \multimap F$ are the types of BASE and STEP, respecitly, then $\texttt{iter}(\mathit{STEP}, \mathit{BASE})$ can receive type G whenever

$\texttt{itercut}_{+}(\mathbf{N}^{g(\overline{a},a,b)}, \mathbf{N}^{j(\overline{a},a,b)}, \mathbf{N}^{h(\overline{a},a)}, f)) = (\mathcal{E}, \mathbf{N}^{g(\overline{a},f(\overline{a})-b,b)}, \mathbf{N}^{k(\overline{a},b)})$

where k is fresh and $\mathcal{E}$ is

$\{g(\overline{a}, a, b+1) \to \max\{j(\overline{a}, a+1, b), h(\overline{a}, a)\},$

$g(\overline{a}, a, 0) \to 0,$

$k(\overline{a}, b) \to \max_{c \leq b} g(\overline{a}, f(\overline{a}) - c, c)\ \}$

(note the use of the $-$ index function in the 3rd equation)

$\texttt{itercut}_{-}(\mathbf{N}^{g(\overline{a},a,b)}, \mathbf{N}^{j(\overline{a},a,b)}, \mathbf{N}^{h(\overline{a},a)}, f)) = (\mathcal{E}, \mathbf{N}^{\max\{g(\overline{a},f(\overline{a})-b,b),k(\overline{a})\}}, \mathbf{N}^{k(\overline{a})})$

where k is fresh and $\mathcal{E}$ is

$\{j(\overline{a}, a+1, b) \to \max\{g(\overline{a}, a, b+1), k(\overline{a})\},$

$j(\overline{a}, 0, b) \to k(\overline{a}),$

$h(\overline{a}, a) \to \max\{g(\overline{a}, a, 1), k(\overline{a})\}\ \}$

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

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

$\texttt{itercut}_{\neg\mathbf{p}}(D_L, E_L, F_L, f) = (\mathcal{E}_L, G_L, H_L),$

$\texttt{itercut}_{\mathbf{p}}(D_R, E_R, F_R, f) = (\mathcal{E}_R, G_R, H_R);$

Figure 11: Equations defining $\texttt{itercut}_{\mathbf{p}}(\cdot)$.

$\texttt{itercut}_+(D,E,F,f) = (\mathcal{E},G,H)$. In other words, $\texttt{itercut}_+(\cdot)$ provides precisely the information one needs to get such typing. The component H is there only to ensure $\texttt{itercut}_{\mathbf{p}}(\cdot)$ to definable by induction.

5.4 The Algorithm

Finally, we are now able to formally define the algorithm TI. It is given in Figure 12. 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 $\texttt{TI}(M,\overline{a})$, where the variables in $\overline{a}$ 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 $f_i(\overline{a}) \to a_i$ where a_i is the variable corresponding to f_i
- In the conclusion of π, replace $f_i(\overline{a})$ by just a_i.

Example 9 . *In order to illustrate the definition of* $\texttt{itercut}_{\mathbf{p}}(\cdot)$ *let us consider the particular case of a term which is obtained by an iteration on the type* $\mathbf{N} \multimap \mathbf{N}$ *(as in Example 3):*

$$M = \texttt{iter}(STEP, BASE).$$

So assuming given $\mathsf{d}\ell\mathsf{T}$ *derivations for BASE, STEP, we want to build a* $\mathsf{d}\ell\mathsf{T}$ *derivation for* M*. As the type of* M *is* $\mathbf{N} \multimap \mathbf{N} \multimap \mathbf{N}$*, there are two negative occurrences of base type* $\mathbf{N}$ *in it, and so we will take* $\overline{a}$ *containing two variables, say* $\overline{a} = a_1, a_2$*.*

Let us denote the $\mathsf{d}\ell\mathsf{T}$ *type judgements for BASE and STEP as follows:*

$$\vdash^{\mathcal{E}} BASE : \mathbf{N}^{r(\overline{a},a)} \multimap \mathbf{N}^{s(\overline{a},a)}$$
$$\vdash^{\mathcal{E}} STEP : (\mathbf{N}^{h(\overline{a},a,b)} \multimap \mathbf{N}^{g(\overline{a},a,b)}) \multimap (\mathbf{N}^{p(\overline{a},a,b)} \multimap \mathbf{N}^{q(\overline{a},a,b)})$$

$$D_1 = \mathbf{N}^{r(\overline{a},a)} \multimap \mathbf{N}^{s(\overline{a},a)},\ D_2 = \mathbf{N}^{h(\overline{a},a,b)} \multimap \mathbf{N}^{g(\overline{a},a,b)},\ D_3 = \mathbf{N}^{p(\overline{a},a,b)} \multimap \mathbf{N}^{q(\overline{a},a,b)}$$

By applying the definition we get:

$$\texttt{itercut}_+(D_1,D_2,D_3,f) = (\mathcal{E}_L \cup \mathcal{E}_R, G_L \multimap G_R, H_L \multimap H_R)$$

with

$$\texttt{itercut}_-(\mathbf{N}^{h(\overline{a},a,b)}, \mathbf{N}^{p(\overline{a},a,b)}, \mathbf{N}^{r(\overline{a},a,b)}, f) = (\mathcal{E}_L, G_L, H_L) \quad (1)$$
$$\texttt{itercut}_+(\mathbf{N}^{g(\overline{a},a,b)}, \mathbf{N}^{q(\overline{a},a,b)}, \mathbf{N}^{q(\overline{a},a,b)}, f) = (\mathcal{E}_R, G_R, H_R) \quad (2)$$

where

$$\begin{array}{rrcl}
\mathcal{E}_L = & \{p(\overline{a},a+1,b) & \to & \max\{h(\overline{a},a,b+1), k(\overline{a})\}, \\
& p(\overline{a},0,b) & \to & k(\overline{a}), \\
& r(\overline{a},a) & \to & \max\{h(\overline{a},a,1), k(\overline{a})\}; \quad \} \\
\mathcal{E}_R = & \{g(\overline{a},a,b+1) & \to & \max\{q(\overline{a},a+1,b), r(\overline{a},a)\}, \\
& g(\overline{a},a,0) & \to & 0, \\
& k'(\overline{a},b) & \to & \max_{c\le b} g(\overline{a}, f(\overline{a})-c, c), \quad \}
\end{array}$$

and

$$\begin{array}{rclrcl}
G_L & = & \mathbf{N}^{\max\{h(\overline{a},f(\overline{a})-b,b),k(\overline{a})\}}, & H_L & = & \mathbf{N}^{k(\overline{a})}, \\
G_R & = & \mathbf{N}^{g(\overline{a},f(\overline{a})-b,b)}, & H_R & = & \mathbf{N}^{k'(\overline{a},b)}.
\end{array}$$

Hence

$$\texttt{itercut}_+(D_1,D_2,D_3,f) = (\mathcal{E}_L \cup \mathcal{E}_R, \quad \mathbf{N}^{\max\{h(\overline{a},f(\overline{a})-b,b),k(\overline{a})\}} \multimap \mathbf{N}^{g(\overline{a},f(\overline{a})-b,b)}, \quad \mathbf{N}^{k(\overline{a})} \multimap \mathbf{N}^{k'(\overline{a},b)}).$$

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

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

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

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

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

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

$$\mathtt{TI}(\mathtt{iter}(V^{A \multimap A}, W^A), \overline{a}) = \left(\frac{\pi \triangleright \Gamma \vdash V : D_1 \multimap D_2 \qquad \rho \triangleright \Delta \vdash W : D_3}{\Omega \vdash \mathtt{iter}(V, W) : \mathbf{N}^{f(\overline{a})} \multimap E\{f(\overline{a})/b\}}, \mathcal{E} \cup \mathcal{F} \cup \mathcal{G} \cup \mathcal{H}\right)$$
where
$\mathtt{TI}(V^{A \multimap A}, \overline{a}) = (\pi, \mathcal{E}) \wedge \mathtt{TI}(W^A, \overline{a}) = (\rho, \mathcal{F}) \wedge$
$\mathtt{itercut}_+(D_1, D_2, D_3, f) = (\mathcal{G}, D, E) \wedge$
$\mathtt{merge}(\Gamma\{a/f(\overline{a}) - b\}, \Delta\{a/f(\overline{a}) - 1\}) = (\Omega, \mathcal{H})$.

Figure 12: The Type Inference Algorithm.

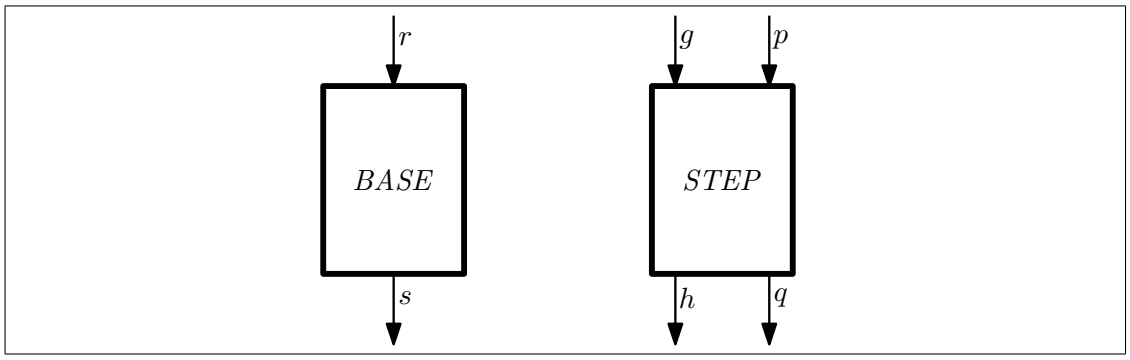

Figure 13: Example: *BASE* and *STEP*

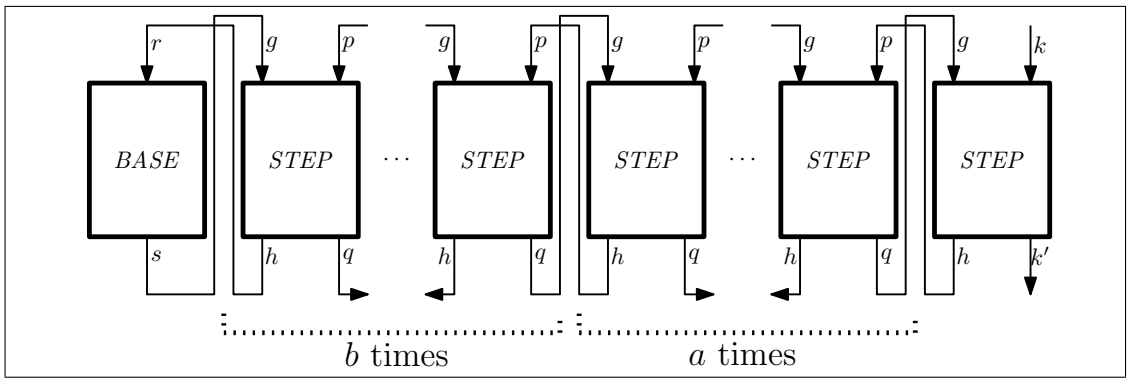

Figure 14: Example: $\texttt{itercut}_{\mathbf{p}}()$

Hence following the definition of TI *from Figure 12 the type obtained by* $\texttt{TI}(\texttt{iter}(STEP, BASE))$ *is* $\mathbf{N}^{f(\overline{a})} \multimap \mathbf{N}^{k(\overline{a})} \multimap \mathbf{N}^{k'(\overline{a},f(\overline{a}))}$.

The definition of CTI *adds equations* $f(a_1, a_2) \to a_1$ *and* $k(a_1, a_2) \to a_2$*, and the type obtained by* $\texttt{CTI}(\texttt{iter}(STEP, BASE))$ *is thus* $\mathbf{N}^{a_1} \multimap \mathbf{N}^{a_2} \multimap \mathbf{N}^{k'(a_1)}$.

Let us now give some of the intuitions underlying the definition of $\mathcal{E}_L$ *and* $\mathcal{E}_R$*. We represent graphically STEP and BASE on Figure 13 as boxes with inputs the negative index functions and outputs the positive index functions. Then the* $\texttt{itercut}_{\mathbf{p}}()$ *construction on type* $\mathbf{N} \multimap \mathbf{N}$ *can be seen as a way to build a box with input function* k *and output function* k'*, and where functions* g, p, h, q, r, s *are used internally. This is done by* $\mathcal{E}_L$ *and* $\mathcal{E}_R$ *in a way which is depicted pictorially on Figure 14. Note that on this picture:*

- *the horizontal axis corresponds to the values of* a *(increasing from right to left) and* b *(increasing from left to right);*
- *the* max *operations on the r.h.s. of equations of* $\mathcal{E}_L \cup \mathcal{E}_R$ *are not represented, for simplicity. For instance from the first equation of* $\mathcal{E}_L$ *we only retain that "*$p(a+1, b)$ *is defined using* $h(a, b+1)$*", and from the third equation that "*$r(a)$ *is defined using* $h(a, 1)$*".*

Now, let us specialize our example to the case of Example 5 where $BASE = \lambda x.\mathbf{succ}(x)$ *and* $STEP = \lambda f.\lambda x.(f\ (\mathbf{succ}(\mathbf{succ}(x))))$*, and admit that the equations obtained by* $\texttt{TI}(BASE)$ *and* $\texttt{TI}(STEP)$ *are the ones described in Example 5 (three equations). Then the full set of equations*

obtained by $\texttt{TI}(\texttt{iter}(STEP, BASE))$ *is the set* $\mathcal{E}'$ *given by:*

$$\begin{array}{rrcl}
 & s(\overline{a}, a) & \to & r(\overline{a}, a) + 1 \\
 & q(\overline{a}, a, b) & \to & g(\overline{a}, a, b) \\
 & h(\overline{a}, a, b) & \to & p(\overline{a}, a, b) + 2 \\
(\mathcal{E}_L) & p(\overline{a}, a+1, b) & \to & \max\{h(\overline{a}, a, b+1), k(\overline{a})\} \\
 & p(\overline{a}, 0, b) & \to & k(\overline{a}) \\
 & r(\overline{a}, a) & \to & \max\{h(\overline{a}, a, 1), k(\overline{a})\} \\
(\mathcal{E}_R) & g(\overline{a}, a, b+1) & \to & \max\{q(\overline{a}, a+1, b), r(\overline{a}, a)\} \\
 & g(\overline{a}, a, 0) & \to & 0 \\
 & k'(\overline{a}, b) & \to & \max_{c \leq b} g(\overline{a}, f(\overline{a}) - c, c)
\end{array}$$

Observe that here the only undefined functions are k *and* f*.*

After this example let us now come back to the properties of the type-inference algorithm.

5.5 Soundness and Completeness.

The soundness proof of the type-inference algorithm will proceed by structural induction on the term M. For each term construction we will essentially need to show how to build a valid (last) typing rule step, by using the pseudo-derivation π provided by $\texttt{TI}(M^A, \overline{a}) = (\pi, \mathcal{E})$. For that we will first need some intermediary results.

The first lemma shows that the subtyping rule is admissible:

Lemma 3 *If the judgement* $\phi; \Gamma \vdash^{\mathcal{E}} M : D$ *is derivable and if we have*

$$\phi \vdash^{\mathcal{E}} D \sqsubseteq E \text{ and } \phi \vdash^{\mathcal{E}} \Delta \sqsubseteq \Gamma,$$

then $\phi; \Delta \vdash^{\mathcal{E}} M : E$ *is derivable.*

For all binary rules we need to merge the contexts, and the following result shows that $\texttt{merge}(.,.)$ will do the job:

Lemma 4 *If the judgement* $\phi; \Gamma \vdash^{\mathcal{E}} M : D$ *is derivable and if we have* $\texttt{merge}(\Gamma, \Delta) = (\Omega, \mathcal{F})$ *for some* Δ*, where* $\mathcal{F} \subseteq \mathcal{E}$*, then* $\phi; \Omega \vdash^{\mathcal{E}} M : D$ *is derivable.*

Then we need a property for the $\texttt{cut}(.,.)$ procedure, used in the application and pair elimination cases:

Lemma 5 *If the equational program* $\mathcal{E}$ *is such that* $\texttt{cut}(D, E) \subseteq \mathcal{E}$*, then we have*

$$\vdash^{\mathcal{E}} D \sqsubseteq E \text{ and } \vdash^{\mathcal{E}} E \sqsubseteq D.$$

Finally, the most delicate case of the soundness proof will be for dealing with the `iter` construct; for that we first need a lemma listing several properties about the $\texttt{itercut}_{\mathbf{p}}(D, E, F, f)$ procedure:

Lemma 6 *Let us denote* $\texttt{itercut}_{\mathbf{p}}(D, E, F, f) = (\mathcal{E}_{\mathbf{p}}, G_{\mathbf{p}}, H_{\mathbf{p}})$*, for any polarity* $\mathbf{p} \in \{+, -\}$*. Then we have:*

1. $\vdash^{\mathcal{E}_+} G_+ \sqsubseteq H_+$ *and* $\vdash^{\mathcal{E}_-} H_- \sqsubseteq G_-$*,*
2. $\vdash^{\mathcal{E}_+} H_+ \sqsubseteq H_+\{b/b+1\}$ *and* $\vdash^{\mathcal{E}_-} H_-\{b/b+1\} \sqsubseteq H_-$*,*
3. *Denote* ψ *the constraint* $\{b + 1 \leq f(\overline{a})\}$*, then*

$$(i) \quad \psi \vdash^{\mathcal{E}_+} G_+ \sqsubseteq D\{a/f(\overline{a}) - b\} \qquad (ii) \quad \psi \vdash^{\mathcal{E}_+} E\{a/f(\overline{a}) - b\} \sqsubseteq G_+\{b/b+1\}$$

$$(iii) \quad \psi \vdash^{\mathcal{E}_-} D\{a/f(\overline{a}) - b\} \sqsubseteq G_- \qquad (iv) \quad \psi \vdash^{\mathcal{E}_-} G_-\{b/b+1\} \sqsubseteq E\{a/f(\overline{a}) - b\}$$

4. $\vdash^{\mathcal{E}} F\{a/f(\overline{a}) - 1\} \sqsubseteq G_+\{b/1\}$ *and* $\vdash^{\mathcal{E}} G_-\{b/1\} \sqsubseteq F\{a/f(\overline{a}) - 1\}$*.*

Proof. [of Lemma 6] For each item the proof will proceed by induction on the structure of $[D]$. The base cases are thus those where $[D] = \mathbf{N}$ and $\mathbf{p}$ is respectively equal to $+$ and $-$. Then one needs to examine the inductive cases where $[D]$ is of the shape $A \multimap B$ and $A \otimes B$. The most delicate point is point 3.

1. For $[D] = \mathbf{N}$ this follows from the definition of $\mathcal{E}$, because for $\mathbf{p} = +$ we have that $\models^{\mathcal{E}} g(\overline{a}, f(\overline{a}) - b, b) \leq k(\overline{a}, b)$, and for $\mathbf{p} = -$ we have that $\models^{\mathcal{E}} k(\overline{a}) \leq \max\{g(\overline{a}, f(\overline{a}) - b, b), k(\overline{a})\}$. The inductive cases are straightforward.
2. For $[D] = \mathbf{N}$: if $\mathbf{p} = +$, we have that $\max_{c \leq b} g(\overline{a}, f(\overline{a}) - c, c)$ is monotone w.r.t. b, and thus so is $k(\overline{a}, b)$; if $\mathbf{p} = -$ the property holds because $k(\overline{a})$ simply does not depend on b. As before the inductive cases are straightforward.
3. We prove the four claims by a simultaneous induction on $[D]$. Consider the base case $[D] = \mathbf{N}$, so we have $D = \mathbf{N}^{g(\overline{a},a,b)}$, $E = \mathbf{N}^{j(\overline{a},a,b)}$, and $G_+ = \mathbf{N}^{g(\overline{a},f(\overline{a})-b,b)}$. Claim (i) is straightforward, and we do not need to use the constraint ψ. For (ii) observe that by definition of $\mathcal{E}_+$ (first equation):
$$\models^{\mathcal{E}_+} j(\overline{a}, a+1, b) \leq g(\overline{a}, a, b+1)$$
So in particular in the case where $a = f(\overline{a}) - (b+1)$ we get that:
$$\models^{\mathcal{E}_+} j(\overline{a}, (f(\overline{a}) - (b+1)) + 1, b) \leq g(\overline{a}, f(\overline{a}) - (b+1), b+1)$$
Beware that we do not have in general that $(I - (b+1)) + 1$ is equal to $(I - b)$, because this does not hold if $I \leq b$. But thanks to the constraint ψ we do have here:
$$\psi \models^{\mathcal{E}_+} (f(\overline{a}) - (b+1)) + 1 = f(\overline{a}) - b$$
So we can deduce that
$$\phi \models^{\mathcal{E}_+} j(\overline{a}, (f(\overline{a}) - b, b) \leq g(\overline{a}, f(\overline{a}) - (b+1), b+1)$$
So (ii) also holds.
For claims (iii) and (iv) we have $G_- = \mathbf{N}^{\max\{g(\overline{a},f(\overline{a})-b,b),k(\overline{a})\}}$. As $\models^{\mathcal{E}} g(\overline{a}, f(\overline{a}) - b, b) \leq \max\{g(\overline{a}, f(\overline{a}) - b, b), k(\overline{a})\}$ we get that (iii) holds (again without using constraint ψ).
Now by the first equation of $\mathcal{E}_-$ we have that (for $a = f(\overline{a}) - (b+1)$):
$$\models^{\mathcal{E}_-} \max\{g(\overline{a}, f(\overline{a}) - (b+1), b+1), k(\overline{a})\} \leq j(\overline{a}, (f(\overline{a}) - (b+1)) + 1, b)$$
Again we have
$$\psi \models^{\mathcal{E}_+} (f(\overline{a}) - (b+1)) + 1 = f(\overline{a}) - b$$
and so we can deduce:
$$\psi \models^{\mathcal{E}_-} \max\{g(\overline{a}, f(\overline{a}) - (b+1), b+1), k(\overline{a})\} \leq j(\overline{a}, f(\overline{a}) - b, b)$$
So (iv) is proved. So the base case of all four claims has been established.
Let us now consider the inductive case where $[D] = A \multimap B$. Let us denote:

$$\begin{array}{lrcl}
(L+) & \texttt{itercut}_+(D_L, E_L, F_L, f) & = & (\mathcal{E}_{L+}, G_{L+}, H_{L+}) \\
(L-) & \texttt{itercut}_-(D_L, E_L, F_L, f) & = & (\mathcal{E}_{L-}, G_{L-}, H_{L-}) \\
(R+) & \texttt{itercut}_+(D_R, E_R, F_R, f) & = & (\mathcal{E}_{R+}, G_{R+}, H_{R+}) \\
(R-) & \texttt{itercut}_-(D_R, E_R, F_R, f) & = & (\mathcal{E}_{R-}, G_{R-}, H_{R-})
\end{array}$$

By applying the induction hypothesis to these four cases we get:

$$\begin{array}{llll}
(i.L+) & \psi \vdash^{\mathcal{E}_{L+}} G_{L+} \sqsubseteq D_L\{a/f(\overline{a}) - b\} & (ii.L+) & \psi \vdash^{\mathcal{E}_{L+}} E_L\{a/f(\overline{a}) - b\} \sqsubseteq G_{L+}\{b/b+1\} \\
(iii.L-) & \psi \vdash^{\mathcal{E}_{L-}} D_L\{a/f(\overline{a}) - b\} \sqsubseteq G_{L-} & (iv.L-) & \psi \vdash^{\mathcal{E}_{L-}} G_{L-}\{b/b+1\} \sqsubseteq E_L\{a/f(\overline{a}) - b\} \\
(i.R+) & \psi \vdash^{\mathcal{E}_{R+}} G_{R+} \sqsubseteq D_R\{a/f(\overline{a}) - b\} & (ii.R+) & \psi \vdash^{\mathcal{E}_{R+}} E_R\{a/f(\overline{a}) - b\} \sqsubseteq G_{R+}\{b/b+1\} \\
(iii.R-) & \psi \vdash^{\mathcal{E}_{R-}} D_R\{a/f(\overline{a}) - b\} \sqsubseteq G_{R-} & (iv.R-) & \psi \vdash^{\mathcal{E}_{R-}} G_{R-}\{b/b+1\} \sqsubseteq E_R\{a/f(\overline{a}) - b\}
\end{array}$$

We then deduce the following properties, by using the definition of subtyping:

$$\begin{array}{lll}
(i) & \psi \vdash^{\mathcal{E}_+} (G_{L-} \multimap G_{R+}) \sqsubseteq (D_L \multimap D_R)\{a/f(\overline{a}) - b\}, & \text{by } (i.R+) \text{ and } (iii.L-) \\
(ii) & \psi \vdash^{\mathcal{E}_+} (E_L \multimap E_R)\{a/f(\overline{a}) - b\} \sqsubseteq (G_{L-} \multimap G_{R+})\{b/b+1\}, & \text{by } (ii.R+) \text{ and } (iv.L-) \\
(iii) & \psi \vdash^{\mathcal{E}_-} (D_L \multimap D_R)\{a/f(\overline{a}) - b\} \sqsubseteq (G_{L+} \multimap G_{R-}), & \text{by } (iii.R-) \text{ and } (i.L+) \\
(iv) & \psi \vdash^{\mathcal{E}_-} (G_{L+} \multimap G_{R-})\{b/b+1\} \sqsubseteq (E_L \multimap E_R)\{a/f(\overline{a}) - b\}, & \text{by } (iv.R-) \text{ and } (ii.L+)
\end{array}$$

where $\mathcal{E}_+ = \mathcal{E}_{L-} \cup \mathcal{E}_{R+}$ and $\mathcal{E}_- = \mathcal{E}_{L+} \cup \mathcal{E}_{R-}$.
Besides as we have by definition:

$$\begin{array}{rcl}
\texttt{itercut}_+((D_L \multimap D_R), (E_L \multimap E_R), (F_L \multimap F_R), f) & = & (\mathcal{E}_{L-} \cup \mathcal{E}_{R+}, (G_{L-} \multimap G_{R+}), (H_{L-} \multimap H_{R+})) \\
\texttt{itercut}_-((D_L \multimap D_R), (E_L \multimap E_R), (F_L \multimap F_R), f) & = & (\mathcal{E}_{L+} \cup \mathcal{E}_{R-}, (G_{L+} \multimap G_{R-}), (H_{L+} \multimap H_{R-}))
\end{array}$$

we see that the inductive hypothesis corresponding to this case indeed correspond to the statements (i), (ii), (iii), (iv) above, and thus have been proven.
We omit the inductive case for $[D] = A \otimes B$ because it is similar to $A \multimap B$ (and somewhat easier). The proof is thus completed.

4. As before we prove the two claims by induction on $[D]$.
In the base case, where $[D] = \mathbf{N}$, we have by definition:

$$\begin{array}{rcl}
F & = & \mathbf{N}^{h(\overline{a},a)} \\
G_+ & = & \mathbf{N}^{g(\overline{a},f(\overline{a})-b,b)}, \\
G_- & = & \mathbf{N}^{\max\{g(\overline{a},f(\overline{a})-b,b),k(\overline{a})\}}
\end{array}$$

By using the first rule of $\mathcal{E}_+$ in the case where $a = f(\overline{a}) - 1$ and $b = 0$ we have:

$$\models^{\mathcal{E}_+} g(\overline{a}, f(\overline{a}) - 1, 1) = \max\{j(\overline{a}, (f(\overline{a}) - 1) + 1, 0), h(\overline{a}, f(\overline{a}) - 1)\}$$

So we can deduce that:

$$\models^{\mathcal{E}_+} h(\overline{a}, f(\overline{a}) - 1) \leq g(\overline{a}, f(\overline{a}) - 1, 1),$$

which proves the first claim.
As to the second claim, by the 3rd rule of $\mathcal{E}_-$ we have, for $a = f(\overline{a}) - 1$:

$$\models^{\mathcal{E}_+} h(\overline{a}, f(\overline{a}) - 1) = \max\{g(\overline{a}, f(\overline{a}) - 1, 1), k(\overline{a})\}$$

So this proves $\vdash^{\mathcal{E}} G_-\{b/1\} \sqsubseteq F\{a/f(\overline{a}) - 1\}$, that is the second claim.
The inductive cases of $[D] = A \multimap B$ and $[D] = A \otimes B$ are proven in a way similar to point 3.

Now we can state and prove the soundness result:

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

Proof. We will first prove a property on $\texttt{TI}(M, \overline{a})$, by induction on the structure of M:
IH(M): *we have* $\texttt{TI}(M, \overline{a}) = (\pi, \mathcal{E})$*, where* $\mathcal{E}$ *is a pseudo-program, and for any* $\mathcal{E}'$ *extending* $\mathcal{E}$*, which is orthogonal and has all its function symbols terminating, we have that* $(\pi, \mathcal{E}')$ *is a derivation.*
Consider the following cases:

- If $M = x^A$: then by definition we have $\mathcal{E} = \texttt{cut}(D, E)$ and by Lemma 5 we deduce that $\vdash^{\mathcal{E}} D \sqsubseteq E$, and thus the last (and only) rule of the pseudo-derivation is a valid rule instantiation (following the first rule of Figure 7).
- If $M = {M_1}^{A \multimap B} {M_2}^A$:
following the definition in Figure 12 we have:

$$\begin{array}{rcl}
\texttt{TI}(M_1, \overline{a}) & = & (\pi \triangleright \Gamma \vdash M_1 : D \multimap E, \mathcal{E}_1) \\
\texttt{TI}(M_2, \overline{a}) & = & (\rho \triangleright \Delta \vdash M_2 : F, \mathcal{E}_2) \\
\texttt{cut}(D, F) & = & \mathcal{G} \\
\texttt{merge}(\Gamma, \Delta) & = & (\Omega, \mathcal{H})
\end{array}$$

Denote $\mathcal{E}_0 = \mathcal{E}_1 \cup \mathcal{E}_2 \cup \mathcal{G} \cup \mathcal{H}$ and consider an extension $\mathcal{E}'$ as in the assumption. Then $\mathcal{E}'$ is also an extension of $\mathcal{E}_1$ (resp. $\mathcal{E}_2$) and thus by induction hypothesis we know that $(\pi, \mathcal{E}')$ (resp. $(\rho, \mathcal{E}')$) is a derivation.
Denote as Γ' (resp. Δ') the context obtained from Γ (resp. Δ) by: for each variable x to which both Γ and Δ assign types, resp. denoted $\mathbf{N}^{f(\overline{a})}$ and $\mathbf{N}^{g(\overline{a})}$, Γ' assigns the type of Ω, $x : \mathbf{N}^{j(\overline{a})}$, and to each variable x to which only Γ (resp. only Δ) assigns a type, Γ' (resp. Δ') assigns the same type. We thus have $\Omega = \Gamma' \uplus \Delta'$, as well as:

$$\vdash^{\mathcal{E}'} \Gamma' \sqsubseteq \Gamma \quad \vdash^{\mathcal{E}'} \Delta' \sqsubseteq \Delta$$

By Lemma 3 we get that the two following judgements are derivable, with $\mathcal{E}'$:

$$\Gamma' \vdash^{\mathcal{E}'} M_1 : D \multimap E$$
$$\Delta' \vdash^{\mathcal{E}'} M_2 : F,$$

Moreover as $\mathtt{cut}(D, F) \subseteq \mathcal{E}'$, by Lemma 5 we have that $\vdash^{\mathcal{E}'} F \sqsubseteq D$. Therefore by Lemma 3 again we have a derivation of:

$$\Delta' \vdash^{\mathcal{E}'} M_2 : D.$$

Hence we can apply the application typing rule of Figure 7 and obtain a derivation of:

$$\Omega \vdash^{\mathcal{E}'} M_1 M_2 : E.$$

- If $M = \mathtt{iter}(V^{A \multimap A}, W^A)$:
following the definition in Figure 12 we have:

$$\begin{aligned}
\mathtt{TI}(V^{A\multimap A}, \overline{a}) &= (\pi \triangleright \Gamma \vdash^{\mathcal{E}} V : D_1 \multimap D_2, \mathcal{E}) \\
\mathtt{TI}(W^A, \overline{a}) &= (\rho \triangleright \Delta \vdash^{\mathcal{E}} W : D_3, \mathcal{F}) \\
\mathtt{itercut}_+(D_1, D_2, D_3, f) &= (\mathcal{G}, D, E) \\
\mathtt{merge}(\Gamma\{a/f(\overline{a}) - b\}, \Delta\{a/f(\overline{a}) - 1\}) &= (\Omega, \mathcal{H}).
\end{aligned}$$

Denote $\mathcal{E}_0 = \mathcal{E} \cup \mathcal{F} \cup \mathcal{G} \cup \mathcal{H}$ and consider an extension $\mathcal{E}'$ as in the assumption.
By induction hypothesis we know that $(\pi, \mathcal{E}')$ and $(\rho, \mathcal{E}')$ are derivations. By using Lemma 1 we get that $(\pi\{a/f(\overline{a}) - b\}, \mathcal{E}')$ and $(\rho\{a/f(\overline{a}) - 1\}, \mathcal{E}')$ also are derivations.
By Lemma 4 we thus get that the two following judgements are derivable:

$$\Omega \vdash^{\mathcal{E}} V : (D_1 \multimap D_2)\{a/f(\overline{a}) - b\}$$
$$\Omega \vdash^{\mathcal{E}} W : D_3\{a/f(\overline{a}) - 1\},$$

Moreover by Lemma 6, items 3. and 4. we have that, if $\psi = \{b + 1 \leq f(\overline{a})\}$:

$$\psi \vdash^{\mathcal{E}} (D_1 \multimap D_2)\{a/f(\overline{a}) - b\} \sqsubseteq D \multimap (D\{b/b + 1)\}$$

$$\vdash^{\mathcal{E}} D_3\{a/f(\overline{a}) - 1\} \sqsubseteq D\{b/1\}$$

By applying Lemma 3 we obtain that the following judgements are derivable:

$$\{b + 1 \leq f(\overline{a})\}; \Omega \vdash^{\mathcal{E}} V : D \multimap (D\{b/b + 1)\}$$
$$\emptyset; \Omega \vdash^{\mathcal{E}} W : D\{b/1\},$$

We can then apply the `iter` typing rule of Figure 8, by observing that the following side conditions are satisfied, by Lemma 6 1. and 2. :

$$\vdash^{\mathcal{E}} D \sqsubseteq E \quad \{b + 1 \leq f(\overline{a})\} \vdash^{\mathcal{E}} E \sqsubseteq E\{b/b + 1\}$$

and we thus obtain a derivation of

$$\Omega \vdash^{\mathcal{E}} \mathtt{iter}(V, W) : \mathbf{N}^{f(\overline{a})} \multimap E\{b/f(\overline{a})\}.$$

This completes this case.

- The other inductive cases are similar and thus omitted here.

□

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.*

5.6 Termination

We now prove that the equational program $\mathcal{E}$ produced in output by type inference (i.e. $\mathtt{CTI}(M) = (\pi, \mathcal{E})$) 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.

The first auxiliary concept we need, then, is that of a *reducible* equational program. This goes as follows: given an equational program $\mathcal{E}$, a finite sequence $\overline{a} = a_1, \ldots, a_n$ of index variables and an assignment ρ of natural numbers to the index variables in $\overline{a}$, we write $\mathcal{E}, \overline{a}, \rho \Vdash D$ iff the following holds:

- If $D = \mathbf{N}^{f(\overline{a})}$, then $\mathcal{E}, \overline{a}, \rho \Vdash D$ iff the evaluation of $f(\overline{a})\rho$ according to the equations from $\mathcal{E}$ terminates.
- If $D = E \multimap F$, then $\mathcal{E}, \overline{a}, \rho \Vdash D$ iff whenever $\mathcal{F}, \overline{a}, \rho \Vdash G$ and $\mathcal{E} \cup \mathcal{F} \cup \mathtt{cut}(E, G)$ is well defined, it holds that $\mathcal{E} \cup \mathcal{F} \cup \mathtt{cut}(E, G), \overline{a}, \rho \Vdash F$.
- If $D = E \otimes F$, then $\mathcal{E}, \overline{a}, \rho \Vdash D$ iff $\mathcal{E}$ is equivalent to an equational program $\mathcal{F} \cup \mathcal{G}$ such that $\mathcal{F}, \overline{a}, \rho \Vdash E$ and $\mathcal{G}, \overline{a}, \rho \Vdash F$. (Here the notion of program equivalence is the natural one: we consider two programs as equivalent if they define the same function symbols, and they attribute the same meaning, i.e. the same function on the natural numbers, to each function symbol.

As usual in reducibility arguments, one needs to prove that all equational programs output by TI are reducible, but *for all* assignments ρ.

Theorem 4 (Termination) *Suppose that M is a closed term having simple type $\mathbf{N} \multimap \mathbf{N}$, and that $\mathtt{CTI}(M) = (\pi, \mathcal{E})$. Then $\pi \triangleright \emptyset \vdash^{\mathcal{E}} M : \mathbf{N}^a \multimap \mathbf{N}^{f(a)}$, where the evaluation of $f(a)$ and of $\mathbb{W}(\pi)$ are terminating.*

Proof. The proof is structured as follows:

- On the one hand, one needs to prove that any equational program $\mathcal{E}$ that $\mathtt{TI}(M, \overline{a})$ produces in output is indeed reducible for every assignment ρ over $\overline{a}$. This property can indeed be proved by induction on the structure of M. More formally, we need to prove the following strengthening of the claim above. If $\mathtt{TI}(M, \overline{a}) = (\pi, \mathcal{E})$ where $\pi \triangleright x_1 : D_1, \ldots, x_n, D_n \vdash^{\mathcal{E}} M : E$, $\mathcal{F}_i, \overline{a}, \rho \Vdash F_i$ for every i and the equational program

$$\mathcal{G} = \mathcal{E} \cup \bigcup_{1 \leq i \leq n} \mathcal{F}_i \cup \bigcup_{1 \leq i \leq n} \mathtt{cut}(F_i, D_i)$$

  is well-defined, then $\mathcal{G}, \overline{a}, \rho \Vdash E$ and $\mathbb{W}(\pi)$ terminates whenever the undefined symbols in it themselves give rise to terminating computations. Some interesting cases:
  - If M is an abstraction, the thesis follows from the induction hypothesis.
  - If M is a variable, then, again, the thesis follows easily from the induction hypothesis
  - If M is $\mathtt{iter}(V, W)$, then one can derive the thesis by observing that the obtained equational program $\mathcal{E}$ is equivalent to one obtained by iterating that of V with itself starting from the equational program for W.
- On the other hand, one also proves that all *reducible* programs, when completed like in CTI, give rise to terminating computations. More specifically, if $\mathtt{TI}(M) = (\pi, \mathcal{E})$ where $\pi \triangleright \emptyset \vdash^{\mathcal{E}} M : \mathbf{N}^{f(a)} \multimap \mathbf{N}^{g(a)}$ then $g(a)$ terminates in the equational program $\mathcal{E}$ endowed with the equation $f(a) = a$. This, in fact, easily follows from the definition of reducibility for the type $\mathbf{N}^{f(a)} \multimap \mathbf{N}^{g(a)}$.

□

Actually, the statement of Theorem 4 could be slightly generalized by allowing functions depending on more than one natural number parameter. Going beyond first-order functions, however, would not be so easy. And after all, first-order programs (possibly being defined by way of higher-order constructions) are those whose complexity we are interested in analyzing, ultimately.

6 Application to Cryptographic Proofs

In this section we present examples of cryptographic proofs, which we will analyze in $\mathsf{d}\ell\mathsf{T}$ in the next section. 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 $\mathsf{d}\ell\mathsf{T}$ 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 We freely use the combinators map and fold and map_2 on lists. Combinators map and fold have been defined in Section 2.1, whereas $\mathsf{map}_2\ f\ l\ l'$ returns the list $\mathsf{cons}\ (f\ a_1\ a'_1) \ldots (\mathsf{cons}\ (f\ a_n\ a'_n)\ (\mathsf{nil}) \ldots)$ where $n = \min(|l|, |l'|)$, a_i and a'_i respectively denote the ith element of l and 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 (as a special case of the notation $\star^k$) 0^k denote the 0-bitstring of length k. Using 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 operations 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 $(\mathcal{K}, \mathsf{f}_{pk}, \mathsf{f}^{-1}_{sk})$ 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 ℓ, 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 s of length ℓ and computes $\mathsf{f}_{pk}(s)$ and $\mathcal{H}(s)$; finally, it masks the message m (a bitstring of length ℓ) by xoring it with $\mathcal{H}(s)$ and concatenates the resulting bitstring to $\mathsf{f}_{pk}(s)$. 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 $\mathcal{A}$ (in this setting we let $\mathcal{A}$ denote a pair of adversaries $\mathcal{A}_1$ and $\mathcal{A}_2$ with shared state) 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 says 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 $t_\mathcal{A} + q_\mathcal{H} \cdot T_\mathsf{f}$, where $t_\mathcal{A}$ denotes the execution time of $\mathcal{A}$, $q_\mathcal{H}$ 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 ℓ;
4. the second adversary $\mathcal{A}_2$ is invoked with input $\mathsf{f}_{pk}(s) \parallel \mathcal{H}(s) \oplus y$, 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 15. 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 $\mathcal{A}$.

Typing We now illustrate the use of $\mathsf{d}\ell\mathsf{T}$ by typing the BR93 inverter of Figure 15. Recall 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.

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

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

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

Figure 15: Inverter for BR93

$$
\begin{array}{rcl}
y & : & \mathbf{N}^{\ell} \\
\mathcal{K}(*) & : & \mathbf{L}^{m}(\mathbf{B}) \otimes \mathbf{L}^{m}(\mathbf{B}) \\
\mathcal{A}_1 & : & \mathbf{L}^{m}(\mathbf{B}) \multimap \mathbf{L}^{n}(\mathbf{B}) \otimes \mathbf{L}^{n}(\mathbf{B}) \\
\mathsf{map}\ (\mathsf{flip})\ *^{\ell} & : & \mathbf{L}^{\ell}(\mathbf{B}) \\
\mathcal{A}_2(\mathsf{app}\ \mathsf{f}(pk, s)\ (\oplus\ \mathcal{H}(s)\ y)) & : & \mathbf{B} \\
step & : & \mathbf{L}^{\ell}(\mathbf{B}) \multimap \mathbf{B} \otimes \mathbf{L}^{\ell}(\mathbf{B}) \multimap \mathbf{B} \otimes \mathbf{L}^{\ell}(\mathbf{B}) \\
!r_L & : & \mathbf{L}^{p}(\mathbf{L}^{\ell}(\mathbf{B})) \\
\texttt{fold } (step, \langle \texttt{ff}, 0^{\ell}\rangle) & : & \mathbf{L}^{p}(\mathbf{L}^{\ell}(\mathbf{B})) \multimap \mathbf{B} \otimes \mathbf{L}^{\ell}(\mathbf{B})
\end{array}
$$

This leads to a type derivation π of conclusion $\mathbf{L}^{\ell}(\mathbf{B}) \multimap \mathbf{L}^{\ell}(\mathbf{B})$. Observe on Figure 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 $\texttt{iter}(V, W)$ (and similarly $\texttt{fold}(V, W)$) 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:

$$
\begin{array}{rcll}
\mathbb{W}(\mathcal{A}_1(pk)) & = & g_1(m) & \\
\mathbb{W}(\mathsf{map}\ (\mathsf{flip})\ *^{\ell}) & = & 3\ell & \\
\mathbb{W}(\mathcal{A}_2(\mathsf{app}\ \mathsf{f}(pk, s)\ (\oplus\ \mathcal{H}(s)\ y))) & = & g_2(2\ell) + f(m, \ell) + h(\ell) + \ell & \\
\mathbb{W}(\texttt{equal}(\mathsf{f}(pk, z), y)\ ) & = & f(m, \ell) + \ell & \\
\mathbb{W}(step) & = & cst + f(m, \ell) + \ell & \text{where } cst \text{ is a constant} \\
\mathbb{W}(\texttt{fold } (step, \langle \texttt{ff}, 0^{\ell}\rangle)) & = & \mathbb{W}(\langle \texttt{ff}, 0^{\ell}\rangle) + p\mathbb{W}(step) + p & \\
 & = & cst + \ell + p \cdot f(m, \ell) + p\ell + p & \\
\mathbb{W}(\mathsf{map}\ (\mathsf{flip})\ *^{\ell}) & = & 3\ell &
\end{array}
$$

So we obtain $\mathbb{W}(\pi) = cst + g_1(m) + 3\ell + g_2(2\ell) + f(m, \ell) + h(\ell) + 2\ell + pf(m, \ell) + p\ell + p + 3\ell$, where we recall that ℓ is the length of the input bitstring, p is the length of the list r_L of adversarial queries to the random oracle. So if we consider as parameter ℓ we get

$$\mathbb{W}(\pi) = O(g_2(2\ell) + (p+1)f(m, \ell) + h(\ell) + (p+8)\ell).$$

By Theorem 1 this gives us a complexity time bound on the execution of this program on the abstract machine. Moreover it is consistent with the informal complexity analysis carried out previously, where we obtained $t_{\mathcal{I}} \sim t_{\mathcal{A}} + q_H \cdot T_{\mathsf{f}}$, because $t_{\mathcal{A}}$ corresponds here to $g_2(2\ell)$, q_H to $(p+1)$, and T_{f} to $f(m, \ell)$.

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 $\mathsf{g}(x \parallel y) = \mathsf{f}\ x \parallel y$, and p is defined by the clause:

$$\mathsf{p}(x \parallel y) = \bigoplus_{i=1}^{l} x_i \cdot y_i$$

where x_i denotes the i-th bit of x and $\cdot$ denotes the product of two bits.

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-polynomial time 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 h 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 $x_i^X = r^X \oplus \mathcal{A}(y \parallel h(i, z^X))$, 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 16. 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. [24, §6.3] for a proof of the validity of the reduction. For an informal analysis of the complexity of the inverter, observe that for each function h_i, 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 polynomial-time, assuming that $\mathcal{A}$ does. This example is particularly interesting because it is not hereditarily polytime: the function pow_0

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

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

Figure 16: Inverter $\mathcal{I}$ against hardcore predicate, with helper funtions

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).

Typing Let us now examine the inverter $\mathcal{I}$ of Figure 16 from the Goldreich-Levin theorem. We use an index function g 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 $\mathcal{E}$:

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

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

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

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:

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

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

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

Finally we get $\mathbb{W}(\pi) = O(n^2 g(1+2n) + n^2 \log(n+1))$, and as we can assume that $\log(n+1)$ is dominated by $g(1+2n)$, $\mathbb{W}(\pi) = O(n^2 g(1+2n))$. 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.

7 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.

7.1 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, 22, 4, 21]) 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 [13, 14]. With respect to these systems it adds the possibility 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 [15, 18], 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 [2, 19] 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 [23].

7.2 Computer-aided Cryptography.

Computer-aided cryptography is an area at the intersection of formal methods and cryptography; that aims to help building and verifying cryptographic proofs using computer-aided tools. The benefits of such an approach are discussed for instance in [20]. 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 probabilistic 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) [5] 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.

8 Discussion

We now discuss some of the advances and limitations of our work.

We have introduced an expressive type and effect system $\mathsf{d}\ell\mathsf{T}$ that can be used for analyzing the complexity of higher-order programs. We have stressed the fact that it can be used to analyze the complexity of some non-hereditarily polytime programs. One advantage of our system is that the type inference can be performed automatically, and then produce a time bound. However there are of course some limitations to our approach.

The first limitation is related to the language itself. One can wonder how natural it is to write algorithms from the cryptography literature in the language $\ell\mathsf{T}$. We think that the functional style of this language and the use of functional combinators like map and fold allow for a concise description of some algorithms. On the few examples we presented in this paper the implementation of the algorithm in $\ell\mathsf{T}$ was rather straightforward and did not require any inventiveness. One could also try to define a small imperative for-loop-language which could have a translation in $\ell\mathsf{T}$. We did not investigate yet though whether this could give sensible complexity bounds on algorithms written in the source imperative language, through the translation and the $\mathsf{d}\ell\mathsf{T}$ type-based complexity analysis. Another question is whether the linearity constraints of $\ell\mathsf{T}$ are not too severe and do not rule out too many natural algorithms. We are rather optimistic on this question but further work is needed to examine a larger range of examples.

Probably the most important question however is about the equational programs: our type inference is indeed automatic, but the types produced are annotated with functions defined by equational programs. In the examples described in this paper we have given some asymptotic bounds on these functions but this second step has been done manually, not automatically, and this was in the case of relatively simple examples. In general however these equational programs could be very involved and extracting an explicit bound, like say a composition of exponentials and polynomials, should be a difficult task. This problem is often described in the literature as finding closed forms of functions defined by equations, or closed forms of over-approximations of these functions. In our setting this would be necessary in general for obtaining understandable indexed types in an automated way. This work remains to be done. Note however that the methodology consisting in separating the constraints/equational programs generation (first phase) from the search for closed forms (second phase) is common in such static analysis approaches (see e.g. [2]). We leave for future work the exploration of automatic or interactive methods for obtaining closed forms for index functions. Some first steps in this direction, including automatic generation of equational programs and partial methods for finding closed forms have been obtained in [26]. We hope that we might be able to adapt or combine some already existing methods for finding closed forms developed in other settings of static analysis.

From a practical point of view, it would be desirable to implement our type system, preferably by integrating it into a a system for computer-aided cryptography, such as F*/RF* or a similar system such as HoaRe^2. For systems such as F*/RF*, since the expressions of dℓT can be construed as a sublanguage of this system, which supports 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. Such an implementation would also help to evaluate the practicality of our approach on a broader range of examples from the cryptographic literature.

9 Conclusion

We have introduced an expressive type and effect system dℓT that can be used for analyzing the complexity of higher-order programs. We have described the type inference procedure producing indexed types defined by means of equational programs. We have illustrated our language on some examples of cryptographic proofs and in particular have shown that it can be used to analyze some non-hereditarily polytime programs.

We have not yet though explored the problem, given an indexed type with equational programs defining the index functions, to search for closed forms for these functions, that is to say explicit over-approximations of these functions. We think however that as for some other static analysis problems this second step is rather distinct from the first one, and we leave it for future work.

An interesting 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 [17] for a recent account of the subtleties with existing definitions.

References

[1] Beniamino Accattoli and Ugo Dal Lago. (leftmost-outermost) beta reduction is invariant, indeed. *Logical Methods in Computer Science*, 12(1), 2016.

[2] 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*, pages 113–132, 2007.

[3] Patrick Baillot, Gilles Barthe, and Ugo Dal Lago. Implicit computational complexity of subrecursive definitions and applications to cryptographic proofs. In *Proceedings of LPAR 2015*, volume 9450 of *LNCS*, pages 203–218. Springer, 2015.

[4] Patrick Baillot and Kazushige Terui. Light types for polynomial time computation in lambda calculus. *Inf. Comput.*, 207(1):41–62, 2009.

[5] 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.

[6] Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella-Béguelin. Probabilistic relational verification for cryptographic implementations. In *Proceedings of POPL 2015*, 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 *Proceedings of POPL 2015*, 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] Ugo Dal Lago. The geometry of linear higher-order recursion. *ACM Trans. Comput. Log.*, 10(2), 2009.

[13] Ugo Dal Lago and Marco Gaboardi. Linear dependent types and relative completeness. *Logical Methods in Computer Science*, 8(4), 2011.

[14] Ugo Dal Lago and Barbara Petit. The geometry of types. In *POPL*, pages 167–178, 2013.

[15] Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *POPL*, pages 133–144, 2008.

[16] Matthias Felleisen and Daniel P. Friedman. Control operators, the secd-machine, and the lambda-calculus. In *3rd Working Conference on the Formal Description of Programming Concepts*, august 1986.

[17] 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.

[18] Bernd Grobauer. Cost recurrences for DML programs. In *International Conference on Functional Programming (ICFP '01)*, pages 253–264, 2001.

[19] 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.

[20] Shai Halevi. A plausible approach to computer-aided cryptographic proofs. *IACR Cryptology ePrint Archive*, page 181, 2005.

[21] Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial potential. In *ESOP*, pages 287–306, 2010.

[22] Martin Hofmann. Safe recursion with higher types and bck-algebra. *Ann. Pure Appl. Logic*, 104(1-3):113–166, 2000.

[23] Neil D. Jones and Lars Kristiansen. A flow calculus of *mwp*-bounds for complexity analysis. *ACM Trans. Comput. Log.*, 10(4), 2009.

[24] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman & Hall Cryptography and Network Security Series. Chapman & Hall, 2007.

[25] D. Leivant and J.-Y. Marion. Lambda calculus characterizations of poly-time. *Fundam. Inform.*, 19(1/2):167–184, 1993.

[26] Maxime Lesourd. Type inference for complexity analysis of functional programs, 2016. Master Thesis, ENS Lyon, https://ilsordo.github.io/research/rapport2016.pdf.

[27] Adam Petcher and Greg Morrisett. The foundational cryptography framework. In *Proceedings of POST 2015*, volume 9036 of *LNCS*, pages 203–218, 2015.

[28] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In *Proceedings of ICFP 2011*, pages 266–278, 2011.

[29] Hongwei Xi. Dependent types for program termination verification. *Higher-Order and Symbolic Computation*, 15(1):91–131, 2002.