Quantitative Models and Implicit Complexity

Ugo Dal Lago[1] and Martin Hofmann[2]

[1] Dipartimento di Scienze dell'Informazione
Università di Bologna
dallago@cs.unibo.it
[2] Institut für Informatik
Ludwig-Maximilians-Universität, München
mhofmann@informatik.uni-muenchen.de

Abstract. We give new proofs of soundness (all representable functions on base types lies in certain complexity classes) for Elementary Affine Logic, LFPL (a language for polytime computation close to realistic functional programming introduced by one of us), Light Affine Logic and Soft Affine Logic. The proofs are based on a common semantical framework which is merely instantiated in four different ways. The framework consists of an innovative modification of realizability which allows us to use resource-bounded computations as realisers as opposed to including all Turing computable functions as is usually the case in realizability constructions. For example, all realisers in the model for LFPL are polynomially bounded computations whence soundness holds by construction of the model. The work then lies in being able to interpret all the required constructs in the model. While being the first entirely semantical proof of polytime soundness for light logics, our proof also provides a notable simplification of the original already semantical proof of polytime soundness for LFPL. A new result made possible by the semantic framework is the addition of polymorphism and a modality to LFPL thus allowing for an internal definition of inductive datatypes.

1 Introduction

In recent years, a large number of characterizations of complexity classes based on logics and lambda calculi have appeared. At least three different principles have been exploited, namely linear types [3,10], restricted modalities in the context of linear logic [8,1,13] and non-size-increasing computation [9]. Although related one to the other, these systems have been studied with different, often unrelated methodologies and few results are known about relative intentional expressive power. We believe that this area of implicit computational complexity needs unifying frameworks for the analysis of quantitative properties of computation. This would help to improve the understanding of existing systems. More importantly, unifying frameworks can be used *themselves* as a foundation for controlling the use of resources inside programming languages.

In this paper, we introduce a new semantical framework which consists of an innovative modification of realizability. The main idea underlying our proposal

R. Ramanujam and S. Sen (Eds.): FSTTCS 2005, LNCS 3821, pp. 189–200, 2005.

lies in considering bounded-time algorithms as realizers instead of taking plain Turing Machines as is usually the case in realizability constructions. Bounds are expressed abstractly as elements of a monoid. We can define a model for a given (logical or type) system by choosing a monoid flexible enough to justify all the constructs in the system. The model can then be used to study the class of representable functions.

This allows us to give new proofs of soundness (all representable functions on base types lies in certain complexity classes) for Light Affine Logic (LAL, [1]), Elementary Affine Logic (EAL, [5]), LFPL [9] and Soft Affine Logic (SAL, [2]). While being the first entirely semantical proof of polytime soundness for light logics, our proof also provides a notable simplification of the original already semantical proof of polytime soundness for LFPL [9]. A new result made possible by the semantic framework is the addition of polymorphism and a modality to LFPL.

The rest of the paper is organized as follows. This Section is devoted to a brief description of related work and to preliminaries. In Section 2 we introduce length spaces and show they can be used to interpret multiplicative linear logic with free weakening. Sections 3 and 4 are devoted to present instances of the framework together with soundness results for elementary, soft and light affine logics. Section 5 presents a further specialization of length spaces and a new soundness theorem for LFPL based on it.

An extended version of this paper is available [7].

Related-Work. Realizability has been used in connection with resource-bounded computation in several places. The most prominent is Cook and Urquhart work (see [4]), where terms of a language called PV^ω are used to realize formulas of bounded arithmetic. The contribution of that paper is related to ours in that realizability is used to show "polytime soundness" of a logic. There are important differences though. First, realizers in Cook and Urquhart [4] are typed and very closely related to the logic that is being realized. Second, the language of realizers PV^ω only contains first order recursion and is therefore useless for systems like LFPL or LAL. In contrast, we use untyped realizers and interpret types as certain partial equivalence relations on those. This links our work to the untyped realizability model HEO (due to Kreisel [12]). This, in turn, has also been done by Crossley et al. [6]. There, however, one proves externally that untyped realizers (in this case of bounded arithmetic formulas) are polytime. In our work, and this happens for the first time, the untyped realizers are used to give meaning to the logic and obtain polytime soundness as a corollary. Thus, certain resource bounds are built into the untyped realizers by their very construction. Such a thing is not at all obvious, because untyped universes of realizers tend to be Turing complete from the beginning due to definability of fixed-point combinators. We get around this problem through our notion of a resource monoid and addition of a certain time bound to Kleene applications of realizers. Indeed, we consider this as the main innovation of our paper and hope it to be useful elsewhere.

Preliminaries. In this paper, we rely on an abstract computational framework rather than a concrete one like Turing Machines. This, in particular, will simplify proofs.

Let $L \subseteq \Sigma^*$ be a set of finite sequences over the alphabet Σ. We assume a pairing function $\langle \cdot, \cdot \rangle : L \times L \to L$ and a length function $| \cdot | : L \to \mathbb{N}$ such that $|\langle x, y \rangle| = |x| + |y| + cp$ and $|x| \leq \mathit{length}(x)$, where $\mathit{length}(x)$ is the number of symbols in x and cp is a fixed constant. We assume a reasonable encoding of algorithms as elements of L. We write $\{e\}(x)$ for the (possibly undefined) application of algorithm $e \in L$ to input $x \in L$. We furthermore assume an abstract time measure $\mathit{Time}(\{e\}(x)) \in \mathbb{N}$ such that $\mathit{Time}(\{e\}(x))$ is defined whenever $\{e\}(x)$ is and, moreover, there exists a fixed polynomial p such that $\{e\}(x)$ can be evaluated on a Turing machine in time bounded by $p(\mathit{Time}(\{e\}(x)) + |e| + |x|)$ (this is related to the so-called invariance thesis [14]). By "reasonable", we mean for example that for any $e, d \in L$ there exists $d \circ e \in L$ such that $|d \circ e| = |d| + |e| + O(1)$ and $\{d \circ e\}(x) = \{d\}(y)$ where $y = \{e\}(x)$ and moreover $\mathit{Time}(\{d \circ e\}(x)) = \mathit{Time}(\{e\}(x)) + \mathit{Time}(\{d\}(y)) + O(1)$. We furthermore assume that the abstract time needed to compute $d \circ e$ from $\langle d, e \rangle$ is constant. Likewise, we assume that "currying" and rewiring operations such as $\langle x, \langle y, z \rangle \rangle \mapsto \langle \langle y, z \rangle, x \rangle$ can be done in constant time. However, we do allow linear (in $|x|$) abstract time for copying operations such as $x \mapsto \langle x, x \rangle$.

There are a number of ways to instantiate this framework. In the full version of this paper [7], the precise form of the assumptions we make as well as one instance based on call-by-value lambda-calculus are described.

2 Length Spaces

In this section, we introduce the category of length spaces and study its properties. Lengths will not necessarily be numbers but rather elements of a commutative monoid.

A *resource monoid* is a quadruple $M = (|M|, +, \leq_M, \mathcal{D}_M)$ where
(i) $(|M|, +)$ is a commutative monoid;
(ii) $\leq_M$ is a pre-order on $|M|$ which is compatible with $+$;
(iii) $\mathcal{D}_M : \{(\alpha, \beta) \mid \alpha \leq_M \beta\} \to \mathbb{N}$ is a function such that for every α, β, γ

$$\mathcal{D}_M(\alpha, \beta) + \mathcal{D}_M(\beta, \gamma) \leq \mathcal{D}_M(\alpha, \gamma)$$
$$\mathcal{D}_M(\alpha, \beta) \leq \mathcal{D}_M(\alpha + \gamma, \beta + \gamma)$$

and, moreover, for every $n \in \mathbb{N}$ there is α such that $\mathcal{D}_M(0, \alpha) \geq n$.
Given a resource monoid $M = (|M|, +, \leq_M, \mathcal{D}_M)$, the function $\mathcal{F}_M : |M| \to \mathbb{N}$ is defined by putting $\mathcal{F}_M(\alpha) = \mathcal{D}_M(0, \alpha)$. We abbreviate $\sigma + \ldots + \sigma$ (n times) as $n.\sigma$.

Let us try to give some intuition about these axioms. We shall use elements of a resource monoid to bound data, algorithms, and runtimes in the following way: an element φ bounds an algorithm e if $\mathcal{F}_M(\varphi) \geq |e|$ and, more importantly, whenever α bounds an input x to e then there must be a bound $\beta \leq_M \varphi + \alpha$ for

the result $y = \{e\}(x)$ and, most importantly, the runtime of that computation must be bounded by $\mathcal{D}_M(\beta, \varphi + \alpha)$. So, in a sense, we have the option of either producing a large output fast or to take a long time for a small output. The "inverse triangular" law above ensures that the composition of two algorithms bounded by φ_1 and φ_2, respectively, can be bounded by $\varphi_1 + \varphi_2$ or a simple modification thereof. In particular, the contribution of the unknown intermediate result in a composition cancels out using that law. Another useful intuition is that $\mathcal{D}_M(\alpha, \beta)$ behaves like the difference $\beta - \alpha$, indeed, $(\beta - \alpha) + (\gamma - \beta) \leq \gamma - \alpha$.

A *length space* on a resource monoid $M = (|M|, +, \leq_M, \mathcal{D}_M)$ is a pair $A = (|A|, \Vdash_A)$, where $|A|$ is a set and $\Vdash_A \subseteq |M| \times L \times |A|$ is a (infix) relation satisfying the following conditions:

(i) If $\alpha, e \Vdash_A a$, then $\mathcal{F}_M(\alpha) \geq |e|$;
(ii) For every $a \in |A|$, there are α, e such that $\alpha, e \Vdash_A a$;
(iii) If $\alpha, e \Vdash_A a$ and $\alpha \leq_M \beta$, then $\beta, e \Vdash_A a$;
(iv) If $\alpha, e \Vdash_A a$ and $\alpha, e \Vdash_A b$, then $a = b$.

The last requirement implies that each element of $|A|$ is uniquely determined by the (nonempty) set of it realisers and in particular limits the cardinality of any length space to the number of partial equivalence relations on L.

A *morphism* from length space $A = (|A|, \Vdash_A)$ to length space $B = (|B|, \Vdash_B)$ (on the same resource monoid $M = (|M|, +, \leq_M, \mathcal{D}_M)$) is a function $f : |A| \to |B|$ such that there exist $e \in L \subseteq \Sigma^*$, $\varphi \in |M|$ with $\mathcal{F}_M(\varphi) \geq |e|$ and whenever $\alpha, d \Vdash_A a$, there must be β, c such that:

(i) $\beta, c \Vdash_B f(a)$;
(ii) $\beta \leq_M \varphi + \alpha$;
(iii) $\{e\}(d) = c$;
(iv) $\mathit{Time}(\{e\}(d)) \leq \mathcal{D}_M(\beta, \varphi + \alpha)$.

We call e a realizer of f and φ a majorizer of f. The set of all morphisms from A to B is denoted as $\mathit{Hom}(A, B)$. If f is a morphism from A to B realized by e and majorized by φ, then we will write $f : A \xrightarrow{e,\varphi} B$ or $\varphi, e \Vdash_{A \multimap B} f$.

Remark 1. It is possible to alter the time bound in the definition of a morphism to $\mathit{Time}(\{e\}(d)) \leq \mathcal{D}_M(\beta, \varphi + \alpha)\mathcal{F}_M(\alpha + \varphi)$. This allows one to accommodate linear time operations by padding the majorizer for the morphism. All the subsequent proofs go through with this alternative definition, at the expense of simplicity and ease of presentation,

Given two length spaces $A = (|A|, \Vdash_A)$ and $B = (|B|, \Vdash_B)$ on the same resource monoid M, we can build $A \otimes B = (|A| \times |B|, \Vdash_{A \otimes B})$ (on M) where $\alpha, e \Vdash_{A \otimes B} (a, b)$ iff $\mathcal{F}_M(\alpha) \geq |e|$ and there are f, g, β, γ with

$$\begin{gathered} \beta, f \Vdash_A a; \\ \gamma, g \Vdash_B b; \\ e = \langle f, g \rangle; \\ \alpha \geq_M \beta + \gamma. \end{gathered}$$

$A \otimes B$ is a well-defined length space due to the axioms on M.

Given A and B as above, we can build $A \multimap B = (\mathit{Hom}(A, B), \Vdash_{A \multimap B})$ where $\alpha, e \Vdash_{A \multimap B} f$ iff f is a morphism from A to B realized by e and majorized by α.

Lemma 1. *Length spaces and their morphisms form a symmetric monoidal closed category with tensor and linear implication given as above.*

A length space I is defined by $|I| = \{0\}$ and $\alpha, e \Vdash_A 0$ when $\mathcal{F}_M(\alpha) \geq |e|$. For each length space A there are isomorphisms $A \otimes I \simeq A$ and a unique morphism $A \to I$. The latter serves to justify full weakening.

For every resource monoid M, there is a length space $B_M = (\{0,1\}^*, \Vdash_{B_M})$ where $\alpha, e \Vdash_{B_M} t$ whenever e is a realizer for t and $\mathcal{F}_M(\alpha) \geq |e|$. The function s_0 (respectively, s_1) from $\{0,1\}^*$ to itself which appends 0 (respectively, 1) to the left of its argument can be computed in constant time in our computational model and, as a consequence, is a morphism from B_M to itself.

2.1 Interpreting Multiplicative Affine Logic

We can now formally show that second order multiplicative affine logic (i.e. multiplicative linear logic plus full weakening) can be interpreted inside the category of length spaces on any monoid M. Doing this will simplify the analysis of richer systems presented in following sections. Formulae of (intuitionistic) multiplicative affine logic are generated by the following productions:

$$A ::= \alpha \mid A \multimap A \mid A \otimes A \mid \forall \alpha.A$$

where α ranges over a countable set of atoms. Rules are reported in figure 1. A *realizability environment* is a partial function assigning length spaces (on the

Identity, Cut and Weakening.

$$\frac{}{A \vdash A}\ I \qquad \frac{\Gamma \vdash A \quad \Delta, A \vdash B}{\Gamma, \Delta \vdash B}\ U \qquad \frac{\Gamma \vdash A}{\Gamma, B \vdash A}\ W$$

Multiplicative Logical Rules.

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C}\ L_\otimes \qquad \frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B}\ R_\otimes \qquad \frac{\Gamma \vdash A \quad \Delta, B \vdash C}{\Gamma, \Delta, A \multimap B \vdash C}\ L_\multimap \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\ R_\multimap$$

Second Order Logical Rules.

$$\frac{\vdash \Gamma, A[C/\alpha] \vdash B}{\Gamma, \forall \alpha.A \vdash B}\ L^\forall \qquad \frac{\Gamma \vdash A \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash \forall \alpha.A}\ R^\forall$$

Fig. 1. Intuitionistic Multiplicative Affine Logic

same resource monoid) to atoms. Realizability semantics $[\![A]\!]^{\mathscr{R}}_\eta$ of a formula A on the realizability environment η is defined by induction on A:

$$\begin{aligned}
[\![\alpha]\!]^{\mathscr{R}}_{\eta} &= \eta(\alpha)\\
[\![A \otimes B]\!]^{\mathscr{R}}_{\eta} &= [\![A]\!]^{\mathscr{R}}_{\eta} \otimes [\![B]\!]^{\mathscr{R}}_{\eta}\\
[\![A \multimap B]\!]^{\mathscr{R}}_{\eta} &= [\![A]\!]^{\mathscr{R}}_{\eta} \multimap [\![B]\!]^{\mathscr{R}}_{\eta}\\
[\![\forall \alpha.A]\!]^{\mathscr{R}}_{\eta} &= (|[\![\forall \alpha.A]\!]^{\mathscr{R}}_{\eta}|, \Vdash_{[\![\forall \alpha.A]\!]^{\mathscr{R}}_{\eta}})
\end{aligned}$$

where

$$\begin{aligned}
|[\![\forall \alpha.A]\!]^{\mathscr{R}}_{\eta}| &= \prod_{C \in \mathscr{U}} |[\![A]\!]^{\mathscr{R}}_{\eta[\alpha \to C]}|\\
\alpha, e \Vdash_{[\![\forall \alpha.A]\!]^{\mathscr{R}}_{\eta}} a &\iff \forall C.\alpha, e \Vdash_{[\![A]\!]^{\mathscr{R}}_{\eta[\alpha \to C]}} a
\end{aligned}$$

Here $\mathscr{U}$ stands for the class of all length spaces. A little care is needed when defining the product since strictly speaking it does not exist for size reasons. The standard way out is to let the product range over those length spaces whose underlying set equals the set of equivalence classes of a partial equivalence relation on L. As already mentioned, every length space is isomorphic to one such. When working with the product one has to insert these isomorphisms in appropriate places which, however, we elide to increase readability.

If $n \geq 0$ and $A_1, \ldots, A_n$ are formulas, the expression $[\![A_1 \otimes \ldots \otimes A_n]\!]^{\mathscr{R}}_{\eta}$ stands for I if $n = 0$ and $[\![A_1 \otimes \ldots \otimes A_{n-1}]\!]^{\mathscr{R}}_{\eta} \otimes [\![A_n]\!]^{\mathscr{R}}_{\eta}$ if $n \geq 1$.

3 Elementary Length Spaces

In this section, we define a resource monoid $\mathcal{L}$ such that elementary affine logic can be interpreted in the category of length spaces on $\mathcal{L}$. We then (re)prove that functions representable in EAL are elementary time computable.

A *list* is either *empty* or $cons(n, l)$ where $n \in \mathbb{N}$ and l is itself a list. The sum $l + h$ of two lists l and h is defined as follows, by induction on l:

$$\begin{aligned}
empty + h &= h + empty = h;\\
cons(n, l) + cons(m, h) &= cons(n + m, l + h).
\end{aligned}$$

For every $e \in \mathbb{N}$, binary relations $\leq_e$ on lists can be defined as follows

- $empty \leq_e l$ for every e;
- $cons(n, l) \leq_e cons(m, h)$ iff there is $d \in \mathbb{N}$ such that
  (i) $n \leq 3^e(m + e) - d$;
  (ii) $l \leq_d h$.

For every e and for every lists l and h with $l \leq_e h$, we define the natural number $\mathcal{D}_e(l, h)$ as follows:

$$\begin{aligned}
&\mathcal{D}_e(empty, empty) = 0;\\
&\mathcal{D}_e(empty, cons(n, l)) = 3^e(n + e) + \mathcal{D}_{3^e(n+e)}(empty, l);\\
&\mathcal{D}_e(cons(n, l), cons(m, h)) = 3^e(m + e) - n + \mathcal{D}_{3^e(m+e)-n}(l, h).
\end{aligned}$$

Given a list l, $!l$ stands for the list $cons(0, l)$. The depth $depth(l)$ of a list l is defined by induction on l: $depth(empty) = 0$ while $depth(cons(n, l)) = depth(l) +$

1. $|l|$ stands for the maximum integer appearing inside l, i.e. $|empty| = 0$ and $|cons(n,l)| = \max\{n,|l|\}$. For every natural number n, $[n]_{\mathcal{L}}$ stands for $cons(n, empty)$.

Relation $\leq_0$ and function $\mathcal{D}_0$ can be used to build a resource monoid on lists. $|\mathcal{L}|$ will denote the set of all lists, while $\leq_{\mathcal{L}}, \mathcal{D}_{\mathcal{L}}$ will denote $\leq_0$ and $\mathcal{D}_0$, respectively.

Lemma 2. *$\mathcal{L} = (|\mathcal{L}|, +, \leq_{\mathcal{L}}, \mathcal{D}_{\mathcal{L}})$ is a resource monoid.*

An *elementary length space* is a length space on the resource monoid $(|\mathcal{L}|, +, \leq_{\mathcal{L}}, \mathcal{D}_{\mathcal{L}})$. Given an elementary length space $A = (|A|, \Vdash_A)$, we can build the length space $!A = (|A|, \Vdash_{!A})$, where $l, e \Vdash_{!A} a$ iff $h, e \Vdash_A a$ and $l \geq_{\mathcal{L}} !h$. The construction ! on elementary length spaces serves to capture the exponential modality of elementary affine logic. Indeed, the following two results prove the existence of morphisms and morphisms-forming rules precisely corresponding to axioms and rules from EAL.

Lemma 3 (Basic Maps). *Given elementary length spaces A, B, there are morphisms:*

$$contr : !A \multimap !A \otimes !A$$
$$distr : !A \otimes !B \multimap !(A \otimes B)$$

where $contr(a) = (a, a)$ and $distr(a, b) = (a, b)$

Lemma 4 (Functoriality). *If $f : A \xrightarrow{e,\varphi} B$, then there is ψ such that $f : !A \xrightarrow{e,\psi} !B$*

Elementary bounds can be given on $\mathcal{F}_{\mathcal{L}}(l)$ depending on $|l|$ and $depth(l)$:

Proposition 1. *For every $n \in \mathbb{N}$ there is an elementary function $p_n : \mathbb{N} \to \mathbb{N}$ such that $\mathcal{F}_{\mathcal{L}}(l) \leq p_{depth(l)}(|l|)$.*

We emphasize that Proposition 1 does not assert that the mapping $(n, m) \mapsto p_n(m)$ is elementary. This, indeed, cannot be true because we know EAL to be complete for the class of elementary functions. If, however, $A \subseteq \mathcal{L}$ is such that $l \in A$ implies $depth(l) \leq c$ for a fixed c, then $(l \in A) \mapsto p_{depth(l)}(|l|)$ is elementary and it is in this way that we will use the above proposition.

3.1 Interpreting Elementary Affine Logic

EAL can be obtained by endowing multiplicative affine logic with a restricted modality. The grammar of formulae is enriched with a new production $A ::= !A$, while modal rules are reported in figure 2. Realizability semantics is extended by $[\![!A]\!]^{\mathscr{R}}_{\eta} = ![\![A]\!]^{\mathscr{R}}_{\eta}$.

Theorem 1. *Elementary length spaces form a model of* EAL.

Exponential Rules and Contraction.

$$\frac{\Gamma \vdash A}{!\Gamma \vdash !A}\ P \qquad \frac{\Gamma, !A, !A \vdash B}{\Gamma, !A \vdash B}\ C$$

Fig. 2. Intuitionistic Elementary Affine Logic

Now, consider the formula

$$List_{\mathsf{EAL}} \equiv \forall\alpha.!(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha).$$

Binary lists can be represented as cut-free proofs with conclusion $List_{\mathsf{EAL}}$. Suppose you have a proof $\pi :!^j List_{\mathsf{EAL}} \multimap !^k List_{\mathsf{EAL}}$. From the denotation $[\![\pi]\!]^{\mathscr{R}}$ we can build a morphism g from $[\![List_{\mathsf{EAL}}]\!]^{\mathscr{R}}$ to $B_{\mathcal{L}}$ by internal application to ε, s_0, s_1. This map then induces a function $f : \{0,1\}^* \to \{0,1\}^*$ as follows: given $w \in \{0,1\}^*$, first compute a realizer for the closed proof corresponding to it, then apply g to the result.

Remark 2. Notice that elements of $B_{\mathcal{L}}$ can all be majorized by lists with unit depth. Similarly, elements of $[\![List_{\mathsf{EAL}}]\!]^{\mathscr{R}}$ corresponding to binary lists can be majorized by lists with bounded depth. This observation is essential to prove the following result.

Corollary 1 (Soundness). *Let π be an* EAL *proof with conclusion $!^j List_{\mathsf{EAL}} \multimap !^k List_{\mathsf{EAL}}$ and let $f : \{0,1\}^* \to \{0,1\}^*$ be the function induced by $[\![\pi]\!]^{\mathscr{R}}$. Then f is computable in elementary time.*

The function f in the previous result equals the function denoted by the proof π in the sense of [11]. This intuitively obvious fact can be proved straightforwardly but somewhat tediously using a logical relation or similar, see also [11].

4 Other Light Logics

Girard and Lafont have proposed other refinements of Linear Logic, namely Light Linear Logic and Soft Linear Logic, which capture polynomial time. We have succeeded in defining appropriate reource monoids for affine variants of these logics, too. In this way we can obtain proofs of "polytime soundness" by performing the same realizability interpretation as was exercised in the previous section. These instantiations of our framework are considerably more technical and difficult to find, but share the idea of the EAL interpretation which is why we have decided not to include them in this Extended Abstract. The interested reader may consult the full paper [7].

In the following section, we will elaborate in some more detail a rather different instantiation of our method.

5 Interpreting LFPL

In [9] one of us had introduced another language, LFPL, with the property that all definable functions on natural numbers are polynomial time computable. The key difference between LFPL and other systems is that a function defined by iteration or recursion is not marked as such using modalities or similar and can therefore be used as a step function of subsequent recursive definitions.

In this section we will describe a resource monoid $\mathcal{M}$ for LFPL, which will provide a proof of polytime soundness for that system. This is essentially the same as the proof from [9], but more structured and, hopefully, easier to understand.

The new approach also yields some new results, namely the justification of second-order quantification, a !-modality, and a new type of binary trees based on cartesian product which allows alternative but not simultaneous access to subtrees.

5.1 Overview of LFPL

LFPL is intuitionistic, affine linear logic, i.e., a linear functional language with $\otimes, \multimap, +, \times$. Unlike in the original presentation we also add polymorphic quantification here. In addition, LFPL has basic types for inductive datatypes, for example unary and binary natural numbers, lists, and trees. There is one more basic type, namely $\Diamond$, the resource type.

The recursive constructors for the inductive datatypes each take an additional argument of type $\Diamond$ which prevents one from invoking more constructor functions than one. Dually to the constructors one has iteration principles which make the $\Diamond$-resource available in the branches of a recursive definition. For example, the type $T(X)$ of X-labelled binary trees has constructors $\mathbf{leaf} : T(X)$ and $\mathbf{node} : \Diamond \multimap X \multimap T(X) \multimap T(X) \multimap T(X)$. The iteration principle allows one to define a function $T(X) \multimap A$ from closed terms A and $\Diamond \multimap X \multimap A \multimap A \multimap A$. In this paper we "internalise" the assumption of closedness using a !-modality.

5.2 A Resource Monoid for LFPL

The underlying set of $\mathcal{M}$ is the set of pairs (n, p) where $n \in \mathbb{N}$ is a natural number and p is a monotone polynomial in a single variable x. The addition is defined by $(n, p) + (m, r) = (n + m, p + r)$, accordingly, the neutral element is $0 = (0, 0)$. We have a submonoid $\mathcal{M}_0 = \{(n, p) \in \mathcal{M} \mid n = 0\}$.

To define the ordering we set $(n, p) \leq (m, r)$ iff $n \leq m$ and $(r - p)(x)$ is monotone and nonnegative for all $x \geq m$. For example, we have $(1, 42x) \leq (42, x^2)$, but $(1, 42x) \not\leq (41, x^2)$. The distance function is defined by

$$\mathcal{D}_{\mathcal{M}}((n, p), (m, r)) = (r - p)(m).$$

We can pad elements of $\mathcal{M}$ by adding a constant to the polynomial. The following is now obvious.

Lemma 5. *Both $\mathcal{M}$ and $\mathcal{M}_0$ are resource monoids.*

A simple inspection of the proofs in Section 2.1 shows that the realisers for all maps can be chosen from $\mathcal{M}_0$. This is actually the case for an arbitrary submonoid of a resource monoid. We note that realisers of elements may nevertheless be drawn from all of $\mathcal{M}$. We are thus led to the following definition.

Definition 1. *An* LFPL*-space is a length space over the resource monoid $\mathcal{M}$. A morphism from* LFPL *length space A to B is a morphism between length spaces which admits a majorizer from $\mathcal{M}_0$.*

Proposition 2. LFPL *length spaces with their maps form a symmetric monoidal closed category.*

Definition 2. *Let A be an* LFPL *space and $n \in \mathbb{N}$. The* LFPL *space A^n is defined by $|A^n| = |A|$ and $\alpha, e \Vdash_{A^n} a$ iff $\alpha \geq (2n-1).\beta$ for some β such that $\beta, e \Vdash_A a$.*

So, A^n corresponds to the subset of $A \otimes \ldots \otimes A$ consisting of those tuples with all n components equal to each other. The factor $2n-1$ ("modified difference") instead of just n is needed in order to justify the linear time needed to compute the copying involved in the obvious morphism from A^{m+n} to $A^m \otimes A^n$.

Let $\mathcal{I}$ be an index set and A_i, B_i be $\mathcal{I}$-indexed families of LFPL spaces. A uniform map from A_i to B_i consists of a family of maps $f_i : A_i \to B_i$ such that there exist α, e with the property that $\alpha, e \Vdash f_i$ for all i. Recall that, in particular, the denotations of proofs with free type variables are uniform maps.

Proposition 3. *For each A there is a uniform (in m, n) map $A^{m+n} \to A^m \otimes A^n$. Moreover, A^1 is isomorphic to A.*

The LFPL-space $\Diamond$ is defined by $|\Diamond| = \{\Diamond\}$ and put $\alpha, d \Vdash_\Diamond \Diamond$ if $\alpha \geq (1, 0)$.

For each LFPL-space A we define LFPL-space $!A$ by $|!A| = |A|$ and $\alpha, t \Vdash_{!A} a$ if there exists $\beta = (0, p) \in \mathcal{M}_0$ with $\beta, t \Vdash_A a$ and $\alpha \geq (0, (x+1)p)$.

Proposition 4. *There is an* LFPL *space $\Diamond$ and for each* LFPL *space A there is an* LFPL *space $!A$ with the following properties:*
(i) $|!A| = |A|$;
(ii) If $f : A \to B$ then $f : !A \to !B$;
(iii) $!(A \otimes B) \simeq !A \otimes !B$;
(iv) The obvious functions $!A \otimes \Diamond^n \to A^n \otimes \Diamond^n$ are a uniform map.
The last property means intuitively that with n "diamonds" we can extract n copies from an element of type $!A$ and get the n "diamonds" back for later use.

The proof of the last assertion relies on the fact that $(2n-1, (2n-1)p) \leq (2n-1, (x+1)p)$ for arbitrary n.

Definition 3. *Let T_i be a family of* LFPL *spaces such that $|T_i| = T$ independent of i. The* LFPL *space $\exists i.T_i$ is defined by $|\exists i.T_i| = |T|$ and $\alpha, e \Vdash_{\exists i.T_i} t$ iff $\alpha, e \Vdash_{T_i} t$ for some i.*

Note that if we have a uniform family of maps $T_i \to U$ where U does not depend on i then we obtain a map $\exists i.T_i \to U$ (existential elimination).

Conversely, if we have a uniform family of maps $U_i \to V_{f(i)}$ then we get a uniform family of maps $U_i \to \exists j.V_j$ (existential introduction). We will use an informal "internal language" to denote uniform maps which when formalised would amount to an extension of LFPL with indexed type dependency in the style of Dependent ML [15].

5.3 Inductive Datatypes

In order to interpret unary natural numbers, we define $N = \exists n.N_n$ where

$$N_n = \Diamond^n \otimes \forall\alpha.(\alpha \multimap \alpha)^n \multimap \alpha \multimap \alpha.$$

We can internally define a successor map $\Diamond \otimes N_n \to N_{n+1}$ as follows: starting from $d : \Diamond, c : \Diamond^n$ and $f : \forall\alpha.(\alpha \multimap \alpha)^n \multimap \alpha \multimap \alpha$ we obtain a member of $\Diamond^{n+1}$ (from d and c) and we define $g : \forall\alpha.(\alpha \multimap \alpha)^{n+1} \multimap \alpha \multimap \alpha$ as $\lambda(x^{A\multimap A}, y^{(A\multimap A)^n}).\lambda z^A.x(f\ y\ z)$. From this, we obtain a map $\Diamond \otimes N \to N$ by existential introduction and elimination.

Of course, we also have a constant zero $I \to N_0$ yielding a map $I \to N$ by existential introduction.

Finally, we can define an iteration map $N_n \multimap !(\Diamond \otimes A \multimap A) \multimap A \multimap A$ as follows: Given $(d, f) \in N_n$ and $t :!(\Diamond \otimes A \multimap A)$, we unpack t using Proposition 4 to yield $u \in ((\Diamond \otimes A) \multimap A)^n$ as well as $d \in \Diamond^n$. Feeding these "diamonds" one by one to the components of u we obtain $v \in (A \multimap A)^n$. But then $f\ v$ yields the required element of $A \multimap A$. Existential elimination now yields a single map $N \multimap !(\Diamond \otimes A \multimap A) \multimap A \multimap A$.

In the full version of this paper [7], we also show how to interpret two different kinds of binary trees.

6 Conclusion

We have given a unified semantic framework with which to establish soundness of various systems for capturing complexity classes by logic and programming. Most notably, our framework has all of second-order multiplicative linear logic built in, so that only the connectives and modalities going beyond this need to be verified explicitly.

While resulting in a considerable simplification of previous soundness proofs, in particular for LFPL and LAL, our method has also lead to new results, in particular polymorphism and a modality for LFPL.

The method proceeds by assiging both abstract resource bounds in the form of elements from a resource monoid and resource-bounded computations to proofs (respectively, programs). In this way, our method can be seen as a combination of traditional Kleene-style realisability (which only assigns computations) and polynomial and quasi interpretation known from term rewriting (which only

assigns resource bounds). An altogether new aspect is the introduction of more general notions of resource bounds than just numbers or polynomials as formalised in the concept of resource monoid. We thus believe that our methods can also be used to generalise polynomial interpretations to (linear) higher-order.

References

1. Andrea Asperti and Luca Roversi. Intuitionistic light affine logic. *ACM Transactions on Computational Logic*, 3(1):137–175, 2002.
2. Patrick Baillot and Virgile Mogbil. Soft lambda-calculus: a language for polynomial time computation. In *Proceedings of the 7th International Conference on Foundations of Software Science and Computational Structures*, 2004.
3. Stephen Bellantoni, Karl Heinz Niggl, and Helmut Schwichtenberg. Higher type recursion, ramification and polynomial time. *Annals of Pure and Applied Logic*, 104:17–30, 2000.
4. Stephen Cook and Alasdair Urquhart. Functional interpretations of feasible constructive arithmetic. *Annals of Pure and Applied Logic*, 63(2):103–200, 1993.
5. Paolo Coppola and Simone Martini. Typing lambda terms in elementary logic with linear constraints. In *Proceedings of the 6th International Conference on Typed Lambda-Calculus and Applications*, pages 76–90, 2001.
6. John Crossley, Gerald Mathai, and Robert Seely. A logical calculus for polynomial-time realizability. *Journal of Methods of Logic in Computer Science*, 3:279–298, 1994.
7. Ugo Dal Lago and Martin Hofmann. Quantitative models and implicit complexity. Arxiv Preprint. Available from http://arxiv.org/cs.LO/0506079, 2005.
8. Jean-Yves Girard. Light linear logic. *Information and Computation*, 143(2):175–204, 1998.
9. Martin Hofmann. Linear types and non-size-increasing polynomial time computation. In *Proceedings of the 14th IEEE Syposium on Logic in Computer Science*, pages 464–473, 1999.
10. Martin Hofmann. Safe recursion with higher types and BCK-algebra. *Annals of Pure and Applied Logic*, 104:113–166, 2000.
11. Martin Hofmann and Philip Scott. Realizability models for BLL-like languages. *Theoretical Computer Science*, 318(1-2):121–137, 2004.
12. Georg Kreisel. Interpretation of analysis by means of constructive functions of finite types. In Arend Heyting, editor, *Constructivity in Mathematics*, pages 101–128. North-Holland, 1959.
13. Yves Lafont. Soft linear logic and polynomial time. *Theoretical Computer Science*, 318:163–180, 2004.
14. Peter van Emde Boas. Machine models and simulation. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 1–66. Elsevier, 1990.
15. Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, 1999.