Theoretical Computer Science 412 (2011) 2029–2047

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Realizability models and implicit complexity

Ugo Dal Lago [a,*], Martin Hofmann [b]

[a] Dipartimento di Scienze dell'Informazione, Università di Bologna, Italy
[b] Institut für Informatik, Ludwig-Maximilians-Universität, München, Germany

ARTICLE INFO

Keywords:
Implicit computational complexity
Realizability
Linear logic

ABSTRACT

New, simple, proofs of soundness (every representable function lies in a given complexity class) for Elementary Affine Logic, LFPL and Soft Affine Logic are presented. The proofs are obtained by instantiating a semantic framework previously introduced by the authors and based on an innovative modification of realizability. The proof is a notable simplification on the original already semantic proof of soundness for the above mentioned logical systems and programming languages. 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. The methodology presented proceeds by assigning both abstract resource bounds in the form of elements from a resource monoid and resource-bounded computations to proofs (respectively, programs).

1. Introduction

Implicit computational complexity is an active research area lying in the intersection of logic and computer science whose goal is to characterize complexity classes as classes of functions or predicates definable in logical systems or lambda calculi. A question that has attracted particular interest in the last two decades is how to tame systems with higher order functions and recursion so as to capture small complexity classes, polynomial time in particular. At least three different principles have been used when characterizing complexity classes by languages with higher order functions, namely linear types [4,15], restricted modalities in the context of linear logic [12,1,20] and non-size-increasing computation [16]. Although related to each other, these systems have been studied with different, often unrelated methodologies and few results are known about their relative intensional expressive power. By intensional expressive power we mean the ability to represent natural algorithms as opposed to just extensionally capture classes of functions. We believe that this is one of the main reasons that there has been relatively little progress towards the main challenge in the field, namely finding systems capturing small complexity classes while being at the same time intensionally expressive.

In a recent paper [9], the authors introduced a new semantic framework based upon an innovative modification of realizability. The main idea underlying the proposal consists 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 *resource monoid*. Given a resource monoid M, the notions of a *length space* on M and of a *morphism* between length spaces (on M) can both be defined. Noticeably, this is a symmetric monoidal closed category, independently of M.

But our goal here is not limited to defining a new realizability model for certain logical systems or programming languages. Given a (logical or type) system L, we define a model for L by choosing a resource monoid which is both:

- *flexible* enough to justify all the constructs or rules in L.
- *restricted* enough to induce proper bounds on the running time of the underlying realizers.

The model can then be *used* as a powerful tool for the analysis of the class of functions representable in L.

* Corresponding author. Tel.: +39 0512094991; fax: +39 0512094510.
E-mail addresses: dallago@cs.unibo.it (U. Dal Lago), mhofmann@informatik.uni-muenchen.de (M. Hofmann).

0304-3975/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2010.12.025

Quite remarkably, second order multiplicative affine logic (MAL) can be interpreted in the presented framework, *independently* on the underlying resource monoid. As a consequence, the flexibility requirement should only be checked for constructs which are not in MAL.

A logical system (or a programming language) is said to be *sound* with respect to a given complexity class iff the class of functions which can be represented in the logical system is included in the complexity class. In [9], we presented proofs of soundness theorems for the following systems: Light Affine Logic (LAL, [1]), Elementary Affine Logic (EAL, [6]), LFPL [16] and Soft Affine Logic (SAL, [2]). The one in [9] was the first entirely semantic proof of polytime soundness for light logics, providing a notable simplification on the original already semantic proof of polytime soundness for LFPL. On the other hand, the resource monoids for LAL, EAL and SAL were complicated compared to the one for LFPL. The latter was a *functional* monoid: elements of the carrier are pairs (n, f), where n is a natural number and f is a function from natural numbers to natural numbers bounded by a polynomial. The first three were not functional models and, more importantly, their definition was complex; as a consequence, proof of soundness for LAL was relatively long and could not be presented in the extended abstract [9].

In this paper, we introduce the semantic framework in full detail, together with concrete instances for EAL, SAL and LFPL. The three resource monoids are all functional. A companion paper by the authors [10] presents a new, simple, functional model for LAL.

Related work. Realizability has been used in connection with resource-bounded computation in several places. The most prominent is Cook and Urquhart's work [5], 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 [5] are typed and very closely related to the logic that is being realized. Second, the language of realizers PV^ω are terms of the simply-typed lambda calculus (endowed with 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 [19]). This, in turn, has also been done by Crossley et al. [8]. 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, due to definability of fixed-point combinators. We get around this problem through our notion of a resource monoid and the addition of certain time bounds to Kleene applications of realizers. Indeed, we consider this the main innovation of our paper and hope that it proves useful elsewhere. Similar ideas were already present in some previous works by the second author [16,14,17]. The presented techniques, however, were designed with one particular system in mind and could not be easily adapted to other systems. Our presentation style is particularly similar to the one adopted in [17].

2. A computational model

In this paper, we adopt the lambda calculus [3] as the language of realizers. More precisely, realizers will be closed values of the pure, untyped, weak and call-by-value lambda calculus. This section summarizes those properties of the calculus which will be relevant in the rest of the paper. For more information, one can consult a recent paper by the first author and Simone Martini [11].

Λ denotes the set of *lambda terms*, defined inductively as follows:

$$M, N ::= x \mid \lambda x.M \mid MM$$

where x ranges over a denumerable set of *variables*. Given lambda terms M, N and a variable x, $M\{x/N\}$ is the lambda term obtained by substituting N for every free occurrence of x in M (see [3] for more details). The *size* $|M|$ of a term M is defined by induction on M: $|x| = 1$, $|\lambda x.M| = |M| + 1$ and $|MN| = |M| + |N|$. *Values* are abstractions and variables. Capital letters like V, U, W range over values. We consider weak call-by-value reduction on lambda terms, i.e. we take $\to$ as the closure of

$$(\lambda x.M)V \to M\{x/V\}$$

under all applicative contexts, i.e. reduction is governed by the following rules:

$$\frac{}{(\lambda x.M)V \to M\{x/V\}} \qquad \frac{M \to N}{ML \to NL} \qquad \frac{M \to N}{LM \to LN}.$$

A *realizer* is simply a closed value, i.e. an abstraction without free occurrences of variables. Realizers are ranged over by letters like e, f, g. $\mathscr{L}$ is the set of all realizers. The application $\{e\}(f)$ of two realizers is the normal form of ef relative to weak call-by-value reduction (if such a normal form exists). Observe that $\{e\}(f)$, if it exists, is always a realizer.

$\mathscr{B} = \{0, 1\}^*$ is the set of binary strings. Letters like s, t, u range over $\mathscr{B}$. The map $\Phi : \mathscr{B} \to \mathscr{L}$ is defined by induction as follows:

$$\Phi(\varepsilon) = \lambda x.\lambda y.\lambda z.z$$