Information and Computation 248 (2016) 150–194

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Computation by interaction for space-bounded functional programming

Ugo Dal Lago [a,b,1], Ulrich Schöpp [c,*]

[a] Università di Bologna, Italy
[b] INRIA Sophia Antipolis, France
[c] Ludwig-Maximilians-Universität München, Germany

ARTICLE INFO

Article history:
Received 13 November 2013
Available online 6 January 2016

Keywords:
Implicit computational complexity
Logarithmic space
Type system
Geometry of interaction
Functional programming

ABSTRACT

When programming with sublinear space constraints one often needs to use special implementation techniques even for simple tasks, such as function composition. In this paper, we study how such implementation techniques can be supported in a functional programming language. Our approach is based on modelling computation by interaction using the Int construction of Joyal, Street & Verity. We apply this construction to a term model of a first-order programming language and use the resulting structure to derive the functional programming language INTML. INTML can be understood as a programming language simplification of Stratified Bounded Affine Logic. We formulate INTML by means of a type system inspired by Baillot & Terui's Dual Light Affine Logic. We show that it captures the complexity classes FLOGSPACE and NFLOGSPACE. We illustrate its expressiveness by showing how typical graph algorithms, such a test for acyclicity in undirected graphs, can be represented.

1. Introduction

A central goal in programming language theory is to design programming languages that allow a programmer to express efficient algorithms in a convenient way. The programmer should be able to focus on algorithmic issues as much as possible and the programming language should give him or her the means to delegate inessential implementation details to the machine.

In this paper we study how a programming language can support the implementation of algorithms with sublinear space usage. Sublinear space algorithms are characterised by having less memory than they would need to store their input. They are useful for computing with large data that may not fit into memory. Examples of concrete application scenarios are streaming algorithms [35], web crawling, GPU computing, and statistical data mining.

We focus in particular on algorithms with logarithmic space usage. A common intuition of such algorithms is that they access externally stored large data using only a fixed number of pointers. A typical example is that of large graphs, where the pointers are references to graph nodes. To encode a pointer to a node of a graph with n nodes, one needs space $O(\log n)$. Logarithmic space graph algorithms are therefore often thought of as algorithms that work with only a constant number

* Corresponding author.
E-mail addresses: ugo.dallago@unibo.it (U. Dal Lago), ulrich.schoepp@ifi.lmu.de (U. Schöpp).

[1] Partially supported by ANR project 14CE250005 ELICA.

http://dx.doi.org/10.1016/j.ic.2015.04.006
0890-5401/

of pointers to graph nodes. Another typical example of large data are long strings. In this case the pointers are character positions.

While logarithmic space algorithms can only store a fixed number of pointers, they can produce large output that does not fit into memory. They do so by producing the output bit by bit. A program that outputs a graph would output the graph nodes one after the other, followed by a stream of the pairs of nodes that are connected by an edge. The program may not have enough memory to store the whole output all at once, but it can produce it bit by bit.

When writing programs with logarithmic space usage, one must often use special techniques for tasks that would normally be simple. Consider for example the composition of two programs. In order to remain in logarithmic space, one cannot run the two algorithms one after the other, as there may not be enough space to store the intermediate result. Instead, one needs to implement composition without storing the intermediate value at all. To compose two programs, one begins by running the second program and executes it until it tries to read a bit from the output of the first program. Only then does one start the first program to compute the requested bit, so that the execution of the second program can be continued. Running the first program may produce more output than just the required bit, but this output is simply discarded. In this way, the two programs work as coroutines and the output of the first program is only computed on demand, one bit at a time.

Implementing such on-demand recomputation is tedious and error-prone. We believe that programming language support should be very useful for the implementation of algorithms that rely on on-demand recomputation of intermediate values. Instead of implementing composition with on-demand recomputation by hand, the programmer should be able to write function composition in the usual way and have a compiler generate a program that does not store the whole intermediate value.

In this paper we ask: How can the implementation of functions with logarithmic space usage be supported in a programming language? We follow the approach of viewing logarithmic space computation as computation with a fixed number of pointers. Algorithms that use a fixed number of pointer values are not hard to capture by first-order programming languages. Suppose we use a type $\mathtt{int}$ of unsigned integers to represent pointers. Then computation with a fixed number of pointers simply becomes computation with a fixed number of $\mathtt{int}$-values. In imperative programming, this may be captured by a while-language [26]; in functional programming it may be captured by programs that are restricted to first-order data and tail recursion [27].

But how should we represent large data like externally stored graphs or large intermediate results that should only be recomputed in small parts on demand? First, we need some way to use the pointers to actually access some large data. If we work with a large graph, for example, and pointers are just $\mathtt{int}$-values, then we need some way of finding out whether or not there is an edge between the nodes reference by two pointers. It would be easy to extend a programming language with primitive operations for this purpose if we only had a single fixed large input graph. But we would like to treat large data much like first-class values that can be passed around and transformed in the program. Therefore, we must explain how to access large data and how to construct it, not just for a the input and output, but also for intermediate results computed by the program itself.

In this paper we propose to model large data as interactive entities. The idea is to formalise the intuition that large data is never computed all at once, but that pieces of data are requested when they are needed. For example, a large graph may be represented by an entity that takes pairs of pointers to nodes as requests and that answers with true or false depending on whether or not there is an edge in the graph. Large strings can be represented similarly as entities that take requests for character positions and that answer with the character at that position.

Such interactive entities may be presented to the programmer as functions that map requests to answers. A large string can be represented by its length together with a function of type $\mathtt{int} \to \mathtt{char}$ that returns the character at the given position (if it is smaller than the length). Graphs can be represented by the number of nodes together with a function $\mathtt{int} \times \mathtt{int} \to \mathtt{bool}$ that tells whether there is an edge between two nodes. Such functional representations of data are suitable for on-demand recomputation. In practical implementations of functional programming languages, the values of type $\mathtt{int} \to \mathtt{char}$ are not stored by the whole graph of the function. Rather, they are given by code that can compute a character for any given integer when required.

In a higher-order functional programming language, such functional representations of data can be used as first-class values. To represent strings, we may use the type $\mathtt{String} := \mathtt{int} \times (\mathtt{int} \to \mathtt{char})$ of pairs of the word length and a function mapping positions to characters. A predicate on strings then becomes a higher-order function of type $\mathtt{String} \to \mathtt{bool}$. To work with strings, one may define basic operations directly, e.g. $\mathtt{cons}_c \colon \mathtt{String} \to \mathtt{String}$ for prepending a character c:

$$\mathtt{cons}_c \;=\; \lambda x.\, \mathtt{let}\ \langle n, w\rangle = x\ \mathtt{in}\ \langle n+1, \lambda i.\, \mathtt{if}\ i = 0\ \mathtt{then}\ c\ \mathtt{else}\ w\ (i-1)\rangle \ .$$

Having defined a suitable set of such operations, one does not need to know the encoding details of the string type anymore and can use it as if it were a normal data type with operations. It is then not visible that strings may be too large to fit into memory.

In this paper we show that this higher-order functional programming approach can be used for implementing algorithms with logarithmic space usage. We argue that the higher-order approach captures typical algorithms with logarithmic space in a natural way. Suppose we have a program p of type $\mathtt{String} \to \mathtt{String}$; a simple example would be $\lambda x.\, \mathtt{cons}_c\ (\mathtt{cons}_d\ x)$.