Type Systems for Resource-Bounded Programming and Compilation
Case for Support

Martin Hofmann David Aspinall

1 Introduction

Recent decades have seen a gradual move from low-level programming languages such as assembler, Basic, and COBOL to high-level languages such as C++, Haskell, Java, and ML. High-level languages provide abstraction layers and powerful programming idioms, making it much easier to implement and analyse complicated algorithms. At the beginning, lack of efficiency prevented the use of these languages in real-life applications, but nowadays powerful hardware and efficient compilers make high-level programming languages the method of choice for most applications. Nonetheless, efficiency concerns are still paramount when computing resources are limited, such as in embedded and real-time systems, or for applications to be run across the internet.

For embedded and real-time systems, programmers use assembler code (or assembler-close fragments of 'C') to ensure a close control over resource consumption. For internet applications, high-level languages such as Java *are* used, but since it is a new application area, malfunction due to violation of resource bounds is accepted as normal. This is one reason why computation over the web is currently limited to applications where reliability is not important. With more serious applications, resource awareness will become a crucial asset.

Even for more general application areas, certified efficiency may still be locally important, of course, when dealing with very large amounts of data or very time-consuming calculations.

Therefore it is an important challenge for programming language researchers to develop high-level languages that express efficiency notions. One way to do this is with an enhanced *type system* for the programming language, so that assertions about the resource usage of functions or expressions are made manifest in their types.

Indeed, the challenge has been taken up in the last few years and there is now a rapidly emerging field of type systems for resource-aware computation, with several international players, including one of us (Hofmann). The ultimate aim of our work is to enhance new and existing programming languages with our type systems, and to integrate them with production-quality compilers.

State of the art

Some of the work so far has focused on establishing complexity bounds by, in essence, bookkeeping of resources such as clock ticks [7] or memory cells [21]. We feel, however, that such low-level approaches run against the spirit of high-level programming. Rather than analysing each program or component from scratch, it is useful to identify common patterns which are known to fulfil particular resource bounds.

Consider linear tree recursion with a size-bounded step function. That is to say, a pattern of the form:

$$\begin{aligned} &f(\mathbf{leaf}) = g \\ &f(\mathbf{node}(l, r)) = h(f(l), f(r)) \end{aligned}$$

Suppose that: (1) the size, in some suitable sense, of $h(x, y)$ is a constant c plus the sizes of x and y together; (2) h is polynomial time computable, and (3) $h(x, y)$ uses x, y at most once (this final point only plays a role if x, y are of functional type). Then f is polynomial time computable and the size of $f(t)$ is linear in the size of t. From explicitly given polynomials for the runtime and space usage of h we can reconstruct explicit (not merely asymptotic) bounds for h.

It is the thesis of this proposal that a type system is the best means of describing and enforcing conditions such as (1), (2), and (3) above, and their generalisations for extra parameters. In a system based on formalised bookkeeping such as [7], such analysis has to be carried out each time anew.

Advances in our direction (albeit with rather different motivation) have come from researchers who study logical systems and computation formalisms with inherent complexity bounds, such as bounded arithmetic [2], polynomial time subsytems of Gödel's T [6, 1], and systems for other complexity classes [4].

In our own work [14, 13, 18, 17, 16] we substantially extended these systems to encompass higher-order functions and datatypes such as lists and trees. The key idea is not to try and define notions of complexity for higher-order computation per se (as has been done e.g., in [29, 27, 5]), but rather to study the impact of higher-order concepts on complexity in the first-order fragment, where the usual notions and complexity classes apply. This ties in with the familiar auxiliary use of higher-order concepts like functionals or exact real numbers, which appear as *subterms* of eventually first-order functions.

There is growing evidence that these results can be used in programming languages for resource sensitive applications. For example, in [19] we have shown how a slight variation of the type system from [16] can be employed to write functional programs operating on lists and trees guaranteed to run in linearly bounded heap space and thus admit a compilation into the malloc-free fragment of 'C' while still adhering to the intuitive applicative semantics which enables, e.g., proof by induction and equational reasoning. [1]

Another piece of evidence is the use by Mitchell, Mitchell, and Scedrov of an extension of Hofmann's type system from [14] in the context of verification of cryptographic protocols [23].

The time is now right to pursue the advance and application of this technology.

2 Programme & methodology

The overall aim of this project is to design languages and tools to aid resource-aware computation in high-level programming. While the emphasis is on programming language design, we also expect feedback towards the mathematical-logical foundations from which the work started.

Our specific objectives are as follows:

1. To design new type systems encompassing algorithms and uses of data structures which fall into desirable feasible complexity classes, but which are prohibited by current systems.

2. To extract explicit resource bounds and *certificates* from typing derivations. It should be possible to independently verify certificates, to integrate with the *proof-carrying code* protocol [24].

3. To investigate applications of the new type systems for compiler technology. The idea is to show that particular optimisations are guaranteed to apply to all well-typed programs, to enforce better resource bounds.

4. To integrate the new type systems with those of full-scale functional programming languages such as ML or Haskell. We aim for a smooth transition between the systems going beyond mere juxtaposition. This topic includes pragmatic aspects such as type inference and pattern matching.

5. To explore applications to non-functional programming, e.g., OOP.

6. To investigate and implement type-checking, compilation, and certification algorithms for these systems, ultimately within a production-quality compiler such as OCaml.

Each objective is described in detail below.

2.1 New type systems

The goal here is twofold. On the one hand, we want type systems capable of accepting and properly analysing more and more obviously feasible *algorithms*, as opposed to extensional functions. On the other hand, we need to encompass new data and control structures, to get as close as possible to a full size functional language.

2.1.1 More algorithms

At present we have two rather independent type systems for resource control. The first one, called SLR and described in [14, 18], captures exactly the class PTIME of polynomial time computable functions by imposing linearity and modality restrictions on structural recursion. The basic idea is that a recursive definition can make only one recursive call and that the result of this recursive call can be processed only by "tame functions" which do not feed it into another recursive definition.

While SLR allows all polynomial time functions to be expressed, its rather drastic discipline rules out many naturally occurring polynomial-time *algorithms* such as the usual sorting algorithms on lists or trees. To repair this problem we have in [16] introduced another system, called ICFP, which keeps the linearity restriction but allows for arbitrary nesting of recursive definitions. This is done by maintaining the additional invariant that definable functions are non size-increasing. This system therefore does not cover all of PTIME, but a different and more refined syntax-free characterisation is possible. Moreover, we have shown in [19] that the same system enriched with general recursion and restricted to first-order recursion captures exactly linear space and allows for a natural compilation into the malloc-free fragment of 'C'.

What is needed now is a smooth integration of these two systems in the sense of [26] which would for example allow variables stemming from the richer system SLR to appear in ICFP terms as long as this happens within the guard of a conditional or similar.

Another avenue worth investigating in this context would be a (perhaps dependently-typed) variation and extension of *bounded linear logic* [11].

For the described applications, control of space complexity is more essential than ensuring runtime bounds. The first step towards controlling space made in [19] is promising but more work

[1] Note that this possibility of intuitive extensional reasoning is lost in approaches based on finite model theory such as [12] where the language is not sound for the usual semantics but only for a finite one in which, e.g., all numbers are smaller than a certain maximum (and so the successor function cuts off to prevent boundary violation).

is needed, for example, we need a system akin to SLR capturing, say, polynomial or linear space and hence capable of generating corresponding resource bounds. Such a system would presumably provide general recursion instead of structural recursion along datatypes.

Also, we would like a system which relaxes the modality restrictions in SLR while maintaining the same expressive power on the level of functions. What we need to do is to shrink the gap between the semantic invariant (a certain limit on growth rate) needed to guarantee the desired resource bound and what the type system actually ensures (not only limit on growth but also limit on syntactic complexity). The type system ensures limited growth rate of step functions in recursive definitions by stipulating that these inspect their arguments only up to a fixed depth. It ought to be possible to allow arbitrary-depth inspection as long as this does not lead to non-constant growth. In general, of course, determining this is undecidable. But it could be done statically in some cases: for instance, under the additional assumption that argument inspection occurs within a subterm of finite-size type. We believe that ideas from O'Hearn-Reddy's active / passive type system [25] could be useful here.

An example of this situation would be the following recursive definition of a function which removes duplicate elements from a list:

$$
\begin{aligned}
&\mathbf{rem}([\,]) = [\,] \\
&\mathbf{rem}(y :: l) = \textbf{if } \mathbf{member}(y, \mathbf{rem}(l)) \\
&\quad \textbf{then } \mathbf{rem}(l) \textbf{ else } y :: \mathbf{rem}(l)
\end{aligned}
$$

Presently, SLR would reject this clearly polynomial definition. In fact the ICFP system would accept this (with some extra annotation) because **rem** is non size-increasing; however it would reject a similarly defined function which returns a list containing every element of the input list exactly twice.

2.1.2 New data structures

Another improvement is to add new data structures to our type systems. Take for example binary trees. In a linear setting we have two kinds of trees, one corresponding to trees laid out in full in memory ($\otimes$-trees), the other corresponding more to an object-oriented representation ($\times$-trees) under which a tree can be sent messages asking it to return the outermost constructor or to evolve into one of its subtrees. In ordinary functional programming these two are extensionally equivalent; in the presence of linearity constraints they differ considerably. The $\otimes$-trees allow for rich elimination rules encompassing e.g., computing the list of leaf labellings, whereas access to the $\times$-trees is restricted to essentially search operations. Conversely, $\otimes$-trees are more difficult to construct; we must ensure that their overall size is polynomially bounded which precludes in particular the definition of a function which constructs the full binary tree of *depth* n. This can be done for $\times$-trees.

The novelty here is that we reflect in the type system the kind of choices that a programmer would normally make in selecting the best data representation for a purpose.

Some preliminary ideas concerning these different data representations exist in [17]. We need to generalise them to arbitrary inductive datatypes and include dag-like representations, which should be more efficient than the closures used in *loc. cit.* Recent unpublished work by Jean-Yves Marion (presented at ICC'99 in Trento) should be useful here. The dichotomy between $\otimes$- and $\times$-trees should also be considered in the context of the ICFP system.

We also propose to investigate in the context of resource certification linear functional front-ends to traditionally imperative data structures such as queues, graphs, storage variables, and arrays. A first attempt has been made in the full version of [19] (see `www.dcs.ed.ac.uk/home/mxh/papers/malloc.ps.gz`) where a linear ADT formulation of queues is compiled into 'C' linked lists with a pointer to their tail.

The methodology for achieving these goals will be as before: analyse the common patterns of representative case studies, identify appropriate invariants, cast them into an abstract (usually category-theoretic) model, and finally describe the model by a type system.

2.2 Explicit resource bounds

The aim here is to extract explicit resource information from well-typed programs. One use for this would be to provide the user with estimates for time or space usage for a given first-order function or concrete computation. Perhaps more interestingly, another use is to endow programs with independently verifiable certificates on their resource consumption.

A general framework for managing programs equipped with certificates of safety properties has recently been put forward in [24] under the name of the *proof carrying code (PCC) protocol.* In a nutshell the idea is as follows. A *certifying compiler* generates mobile code with special annotations, e.g., loop invariants. These annotations enable independent verification of the alleged correctness property by a small and trusted piece of software, the *proof checker*, which may be run by a potential recipient.

This provides security against failures in implementation and, indeed, in our proofs of type soundness. (In fact, we could even opt for a "partially sound" type system if the logic or the implementation becomes overly complex.) It has the added benefit that a recipient of code generated from well typed programs will not need to read and understand our research papers in order to be satisified that the resource bounds are valid!

The soundness proofs for the existing systems state that every well-typed function of first-order type is extensionally equal to a polynomial time or linear space computable function. Each proof is constructive, so in principle it provides a method for both compiling programs and generating resource bounds. However, the proofs have not been carried out with this application in mind and provide rather generous bounds. To obtain more accurate bounds, we need to strengthen the proofs or even use entirely new techniques, for example proofs based on normalisation and subject reduction rather than abstract interpretation. An attempt in this direction has been made in H. Schwichtenberg's group (personal communication) by providing a syntactic soundness proof for Hofmann's system from [16].

A benchmark here would be a reconstruction in our context of systems for automatic derivation of complexity bounds of first-order recursive programs and term rewriting like [3, 10].

Integration with PCC requires the design of a suitable target language in the form of annotated assembler code or bytecode. Perhaps existing typed assembly languages [9] can be reused for this purpose. Once this has been done we need a compiler which translates programs in our type systems into annotated machine code, and a proof checker. We will content ourselves with the formal foundations and experimental implementations of this framework.

2.3 Compilation

We anticipate that well-typed programs will be guaranteed to allow certain optimisation techniques to apply successfully.

For example, the type sytem in [19] ensures that dynamic memory allocation, hence garbage collection, is not needed during evaluation of well-typed functions. This is because these functions can be evaluated within the heap space occupied by their argument. The run-time systems of existing compilers also realise similar space efficiencies, by the use of sophisticated caching during garbage collection. However, to absolutely guarantee good space performance as would be required for embedded systems programming, such reliance on runtime systems is not enough. We need to be explicit about the circumstances under which and how the optimisation occurs.

Other examples of optimisations include generalisations of tail recursion and layout of partial results in the heap rather than the call stack. Consider for example the naive recursive definition of list concatenation with copying:

$$\mathbf{append}([\,], r) = r$$
$$\mathbf{append}(a :: l, r) = a :: \mathbf{append}(l, r)$$

Although it is not prima facie tail recursive we can implement it iteratively in a stack-free manner if we place the partial result into the heap position it belongs at rather than keeping it on the stack. We want to investigate if similar reasoning applies to a well-circumscribed subclass of the recursive definitions allowed by our type systems.

We emphasize that the type system needs to be rather explicit about such possible optimisations so that it is possible to statically guarantee their applicability and the resulting effect on certified resource bounds. Preliminary experiments with the existing linear type system and the particular semantic interpretation described in [15] are promising. We also plan to systematically study existing optimising compilers (such as Ocamlopt [22]) to identify more instances of this.

2.4 Full-scale type systems

Integrating our type systems and their implementations with those of a full-scale programming language is an engineering problem outside the scope of this project. Before it can be undertaken, however, there are some research-level problems which need to be addressed. We need to consider how to combine our type systems with features such as polymorphism, modules, pattern matching, and type inference. We propose to look at these problems.

The existing type systems rely on explicit annotations of functional abstractions and can only infer the "aspect" of a function, i.e., whether it is linear and whether it recurses on its argument. Clearly, some sort of type inference mechanism is needed to make the system usable in practice. Ideally, one might want to aim for full-blown Damas-Milner style type inference [8] which requires no type annotations at all in programs. A step in this direction for a purely linear type system has been undertaken in [30]. We plan to investigate to what extent these results can be extended to the present case. Alternatively, we might follow Pierce and Turner [28] who have argued convincingly that type annotations in definitions might be acceptable and even desirable as they provide some documentation. Based on these extra assumptions they were able to give a much simplified inference algorithm for ML.

Another pragmatic aspect is that at present SLR is based on System T-style recursion operators. These are somewhat awkward to use in practice and should be replaced by pattern matching.

We have to merge our type systems with the existing ML type system in such a way that SLR-typable programs receive a certain type and non SLR-typable terms receive their usual ML type. Once this has been achieved it becomes possible to integrate the type system and algorithms for compilation and complexity bound derivation with an existing ML compiler. This task, however, goes beyond the scope of our project and will be left for future work.

Even though our project is of a foundational nature a satisfactory treatment of these pragmatic issues is of great importance because it allows for larger examples which in turn will suggest new theoretical questions.

2.5 Non-functional languages

Strong-typing disciplines have a traditional connection with functional programming languages, but their influence has spread to languages of other paradigms, including imperative languages like MODULA-2 and object-oriented languages such as Java.

We believe that our type systems should be adaptable to these too. In the final stage of the project we intend to carry out exploratory research towards this goal. We expect that purely functional OOP could be accommodated by redoing one of the existing functional encodings of OOP [20] in our setting. More challenging would be to account for imperative aspects bringing in issues of aliasing and sharing of heap allocated data structures.

2.6 Implementation

We plan to implement the various software deliverables such as type checkers, compilers and evaluators, resource bound generators and certifiers, and possibly proof checkers. To begin with, we will build prototypes (with the help of student projects) to demonstrate feasibilty of our approach and to enable experiments. Ultimately, perhaps in a successor project, we would like to make these tools part of an Open Source production-quality compiler such as Ocaml.

In fact, we have some experience of writing such prototype implementations, and we have some skeletons for type checkers and compilers which can quickly be adapted to new situations. A fourth year student is currently working on an implementation of the ICFP system and its translation into malloc-free 'C'.

3 Beneficiaries

In the long to medium term this research will be beneficial to programmers and users of resource-sensitive applications as outlined in the introduction. We expect that after sufficient maturation the results of this research will become part of the common knowledge of this field.

To achieve this goal we rely on the means of dissemination described below. When the foundational questions are largely settled and assuming the expected progress we might in a possible second round of this project seek involvement of an industrial partner.

In the short term the research results will be profitable for researchers in the same and related areas such as complexity theory at higher types and application of related type systems such as [23] and [25].

Finally, the research will also have educational benefit. We will involve students with interesting final-year projects, helping to implement prototypes of our systems. This will provide them with invaluable experience of formal foundations, which is important with the growing industrial need for formal specification and verification.

4 Dissemination

We will disseminate our research results in the usual scholarly ways, presenting partial results at international conferences and workshops and publicizing mature material as journal articles.

We will make software prototypes available for others to test, via the Internet. As has been done with the system from [14] we will provide "live" web pages which make the software prototypes available for testing without installation, through CGI scripts. This encourages other workers to quickly try out the systems.

5 Justification of resources

Staff We request one RA AR1 post (pt 9) for Dr. Aspinall, employed as a post doctoral research associate to work full-time on the project for two years. Depending on progress, we will then consider application for another two years. Aspinall's experience with compiler technology, type systems, and large-scale software projects makes him an ideal candidate to undertake the proposed research. Hofmann will devote an average of 9hrs/week to the project.

Travel To present partial results and keep abreast of the latest developments we seek support for attendance of five international conferences or workshops (Hofmann: 3, Aspinall: 2), e.g., CSL, Dagstuhl, ETAPS, ICALP, ICC, LICS, POPL, TLCA. In order to maintain and extend our national and international contacts we request travel and subsistence support for four one-week visits to European universities, e.g. Aarhus, Darmstadt, Gothenburg, Marseille, Munich, Paris, Sophia Antipolis, one 10-14 day visit (Hofmann) to a US university, e.g. U. Pennsylvania (Pierce, Scedrov) or Carnegie Mellon (Reynolds, Pfenning) and four 2-3 day visits to UK universities (e.g. Cambridge, Leeds, QMW).

Equipment A workstation (Aspinall) and a portable computer (Hofmann) maintained over the project period, are requested to support implementation of type checkers and compilers, as well as for publishing. We also request an appropriate contribution to consumables, shared networking, and server provision (filesystem, printing, computing).

References

[1] Stephen Bellantoni and Stephen Cook. New recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.

[2] Samuel R. Buss. *Bounded Arithmetic*. Bibliopolis, 1986.

[3] Christine Choppy, Stéphane Kaplan, and Michèle Soria. Complexity analysis of term rewriting systems. *Theoretical Computer Science*, 67:261–282, 1989.

[4] Peter Clote. Computation models and function algebras. available electronically under http://thelonius.tcs.informatik.uni-muenchen.de/~clote/Survey.ps.gz, 1996.

[5] S. Cook and B. Kapron. Characterisations of the basic feasible functionals at all finite types. In S. Buss and P. Scott, editors, *Feasible Mathematics*, pages 154–159. Birkhäuser, 1990.

[6] S. Cook and A. Urquhart. Functional interpretations of feasibly constructive arithmetic. *Annals of Pure and Applied Logic*, 63:103–200, 1993.

[7] K. Crary and S. Weirich. Resource bound certification. In *Proc. 27th Symp. Principles of Prog. Lang. (POPL)*. ACM, 2000. to appear.

[8] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Proceedings of the 9th ACM Symposium on the Principles of Programming Languages*, pages 207–212, 1982.

[9] Greg Morrisett et al. Talx86: A realistic typed assembly language. In *Second Workshop on Compiler Support for System Software*, 1999.

[10] P. Flajolet, B. Salvy, and P. Zimmermann. Lambda-Upsilon-Omega: An assistant algorithms analyzer. In T. Mora, editor, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, volume 357 of *Lecture Notes in Computer Science*, pages 201–212, 1989. Proceedings AAECC'6, Rome, July 1988.

[11] J.-Y. Girard, A. Scedrov, and P. Scott. Bounded linear logic. *Theoretical Computer Science*, 97(1):1–66, 1992.

[12] Andreas Goerdt. Characterizing complexity classes by higher type primitive recursive definitions. *Theoretical Computer Science*, 100:45–66, 1992.

[13] Martin Hofmann. An application of category-theoretic semantics to the characterisation of complexity classes using higher-order function algebras. *Bulletin of Symbolic Logic*, 3(4):469–485, 1997.

[14] Martin Hofmann. A mixed modal/linear lambda calculus with applications to Bellantoni-Cook safe recursion. In *Proceedings of CSL '97, Aarhus. Springer LNCS 1414*, pages 275–294, 1998.

[15] Martin Hofmann. Semantics of linear/modal lambda calculus. To appear in *Journal of Functional Programming*, 1998.

[16] Martin Hofmann. Linear types and non size-increasing polynomial time computation. In *Logic in Computer Science (LICS)*. IEEE, Computer Society Press, 1999. to appear.

[17] Martin Hofmann. Typed lambda calculi for polynomial-time computation, 1999. Habilitation thesis, TU Darmstadt, Germany. Edinburgh University LFCS Technical Report, ECS-LFCS-99-406.

[18] Martin Hofmann. Safe recursion with higher types and bck-algebra. *Journal of Symbolic Logic*, 2000. to appear.

[19] Martin Hofmann. A type system for bounded space and functional in-place update. In *European Symposium on Programming (ESOP)*. Springer, 2000. to appear.

[20] Martin Hofmann and Benjamin Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–635, 1995. Extended Abstract in Proc. TAPSOFT '94.

[21] J. Hughes and L. Pareto. Recursion and dynamic data structures in bounded space: towards embedded ml programming. In *Proc. International Conference on Functional Programming (ACM). Paris, September '99.*, pages 70–81, 1999.

[22] Xavier Leroy. The Objective Caml System, documentation and user's guide. Release 2.02. http://pauillac.inria.fr/ocaml/htmlman, 1999.

[23] J. Mitchell, M. Mitchell, and A. Scedrov. A linguistic characterisation of bounded oracle computation and probabilistic polynomial time. In *39th IEEE Symposium on Foundations of Computer Science (FOCS)*, 1998.

[24] George Necula. Proof-carrying code. In *Proc. 24th Symp. Principles of Prog. Lang. (POPL)*. ACM, 1997.

[25] P. W. O'Hearn, M. Takeyama, A. J. Power, and R. D. Tennent. Syntactic control of interference revisited. In *MFPS XI, conference on Mathematical Foundations of Program Semantics*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995.

[26] Peter O'Hearn. Doubly closed categories , resource interpretations, and the $\alpha\lambda$-calculus. In *Typed Lambda Calculi and Applications (TLCA '99)*. Springer LNCS, 1999.

[27] E. Pezzoli. On the computational complexity of type two functionals. In *Proc. Conf.* Computer Science Logic, *Aarhus*, 1997. Springer LNCS.

[28] Benjamin C. Pierce and David N. Turner. Local type inference. In *Conference Record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998. Full version available as Indiana University CSCI technical report #493.

[29] Anil Seth. Turing machine characterisations of feasible functionals of all finite types. In P. Clote and J. Remmel, editors, *Feasible Mathematics II*, pages 407–428. Birkhäuser, 1995.

[30] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *7'th International Conference on Functional Programming and Computer Architecture*, San Diego, California, June 1995.