Computer Languages, Systems & Structures 36 (2010) 34 – 59

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

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

Cost-driven autonomous mobility

Xiao Yan Deng*, Greg Michaelson, Phil Trinder

School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, EH14 4AS Scotland, UK

ARTICLE INFO

Article history:
Received 10 November 2008
Accepted 26 January 2009

Keywords:
Autonomous systems
Load balancing
Cost models
Jocaml

ABSTRACT

Autonomous mobile programs (AMPs) offer a novel decentralised load management technology where periodic use is made of cost models to decide where to execute in a network. In this paper we demonstrate how sequential programs can be automatically converted into AMPs. The AMPs are generated by an *automatic continuation cost analyser* that replaces iterations with costed autonomous mobility skeletons (CAMS) that encapsulate autonomous mobility. The CAMS cost model uses an entirely novel *continuation cost semantics* to predict both the cost of the current iteration and the continuation cost of the remainder of the program. We show that CAMS convey significant performance advantages, e.g. reducing execution time by up to 53%; that the continuation cost models are consistent with the existing AMP cost models; and that the overheads of collecting and utilising the continuation costs are relatively small. We discuss example AMPs generated by the analyser and demonstrate that they have very similar performance to hand-costed CAMS programs.

1. Introduction

The explosive growth in wired and wireless networks enables the construction of substantial distributed systems based on shared interconnected clusters. However, the effective use of such systems raises pressing problems for the optimal utilisation of resources in the presence of dynamically changing and unpredictable demand. The simplest approach is to *statically allocate* new jobs to available resources, either blindly based on snapshots of local cluster loads, or through various strategies for balancing user predicted *demand* against available resource. However, such approaches can quickly become suboptimal, especially if demand is predicted inaccurately, leading to over or under resource allocation, or to resources being freed or retained unpredictably.

An alternative is to try to *dynamically manage* resource *consumption* by moving live jobs across processors or clusters at run time to maintain balance. Load management and analysis of patterns of resource use may be either *centralised* at a single location or, in larger networks, *decentralised* across a number of locations. Such load monitoring can incur significant local and global housekeeping overheads. More problematic, dynamic load management is *reactive* and is driven by the need to continually recover from imbalance.

We are exploring a novel approach to decentralised load management, where decisions about when and where to execute are devolved to individual programs. That is, we develop *autonomous mobile programs* (AMPs) [1] where, instead of some external system managing load, *the program itself* decides whether its resource needs would be better served by movement to another location. Furthermore, rather than simplistic movement-based solely on identifying the most lightly loaded location, our AMPs are aware of their *future* resource needs and hence can make informed decisions about whether those needs are best served locally or by movement elsewhere. We have shown that collections of AMPs, while not aware of each other individually, will nonetheless move to maintain optimal balance collectively [1].

* Corresponding author. Tel.: +44 131 451 4162.
E-mail addresses: xyd3@macs.hw.ac.uk (X.Y. Deng), greg@macs.hw.ac.uk (G. Michaelson), trinder@macs.hw.ac.uk (P. Trinder).

1477-8424/$ - see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cl.2009.01.003

The novelty in our approach lies in each program bearing its own *cost model* which is parameterised on the remaining execution time and data sizes. However, constructing AMP cost models by hand is a skilled task, so we have been exploring common patterns of mobility we term *autonomous mobility skeletons* (AMSs) [2] with standard cost models. Anticipating that the most effective locus of mobility control lies in top-level iterations, our AMSs generalise standard iterative forms. For example *auto_iter* evaluates the cost model to assess the benefits of moving periodically during a Java iteration. Likewise *auto_map* periodically considers moving while applying a function to each element of a sequence.

While AMSs greatly simplify the construction of AMPs, nonetheless they still require considerable proficiency in cost model construction. Hence, we have been investigating the *automatic generation of cost models from programs* using AMS as loci [3]. Our results suggest that even simple automatically generated cost models can be highly effective in enabling mobile programs to adapt sanely to dynamically changing environments.

1.1. Novelty

The paper situates the work (Section 2) and outlines earlier work (Section 3) before making the following four research contributions.

- We present a new *continuation cost semantics* for a core mobile functional programming language $\mathscr{J}$ that predicts the cost of computing the remainder of a program at arbitrary program points. We believe our cost semantics is the first to cost continuations rather than entire programs. The continuation cost equations are generated statically but are designed to be parameterised dynamically to more accurately predict the time to evaluate the remainder of the program (Section 4).
- The continuation costs are incorporated into a cost model for high-level abstractions of autonomously mobile iterations over collections, called costed autonomous mobility skeletons (CAMS), and a Jocaml implementation of a CAMS is exhibited (Section 5).
- We evaluate the continuation cost semantics and CAMS using six pairs of programs to show, *inter alia*, the following. The continuation cost models are consistent with the existing AMP cost models. The overheads of collecting and utilising the continuation costs are relatively small. Most significantly, utilising the predicted continuation costs can convey significant performance advantages compared with both static and AMS programs (Section 6).
- We show that sequential programs can be automatically converted into AMPs that move to better exploit computational resources on a network. We do so by exhibiting an *automatic continuation cost analyser* that implements the continuation cost semantics to supply cost equations to a translator that replaces iterating higher-order functions with the corresponding CAMS. We show example AMPs generated by the analyser and demonstrate that they have very similar performance to hand-costed CAMS programs (Section 7).

The continuation cost analyser and CAMS were outlined in [3]. Here we present the underlying indexing, cost semantics, and continuation cost semantics. We likewise present the CAMS cost model and implementation for the first time, evaluate CAMS performance against more AMPs and make a deeper performance analysis. Finally we elaborate the architecture of the continuation cost analyser, and demonstrate it against further, and more substantial, examples.

2. Related work

The idea of relocating a process during execution has existed for some time, and is termed migration, rescheduling or strong mobility by different communities. Strong mobility is discussed in Section 2.1. Much work was done on load management using task migration in distributed operating systems in the 1970s [4], and some well known examples are Mach [5] and MOSIX [6]. Sophisticated distributed memory implementations of parallel programming languages support task migration, for example the Charm parallel C++ [7]. However, where parallel languages are typically designed for homogeneous dedicated architectures, AMPs operate on heterogeneous shared architectures. Moreover, both distributed operating systems and parallel programming languages differ from AMPs as the tasks are passive, and the scheduling is typically centralised.

Grid workflow reschedulers are more closely related to AMPs, and are currently the focus of considerable research effort. An excellent taxonomy of Grid workflow management systems can be found in [8]. Like AMPs Grid workflow reschedulers operate on heterogeneous shared networks, and many make decentralised scheduling decisions, use performance prediction to inform scheduling decisions, and reschedule after periodic reassessment of system status. However, our AMP approach is novel in automating the performance prediction process as a one-off, compile-time program analysis, and in devolving the rescheduling decisions to individual programs. Effective load management is only derived as an emergent behaviour from collections of AMPs.

Although loop scheduling mechanisms [9], many algorithmic skeletons [10], and AMSs all operate on iterations, they do so for very different purposes. Both algorithmic skeletons and loop scheduling mechanisms parallelise their own iterations. In contrast, AMS do not parallelise iterations but rather use them to predict work and to determine whether to move.

The remainder of this section surveys related work in the three core AMP technologies: mobile computations, autonomous systems and cost analysis.

2.1. Mobility and mobile languages

Mobile computations can move between locations in a network and potentially enable better use of shared computational resources [11]. Mobile programming languages like Jocaml [12] or Java Voyager [13] give programmers control over the placement of code or active computations across the network. Basically a mobile program can transport its state and code to another location in a network, where it resumes execution [14].

Fuggetta et al. distinguish two forms of mobility supported by mobile languages [15]. *Weak mobility* is the ability to move only code from one location to another. *Strong mobility* is the ability to move both code and its current execution state [2]. Jocaml is a strict functional programming language with strong mobility, and while JavaGo also supports strong mobility, Voyager supports only weak mobility. While strong mobility is required for arbitrary AMPs, they may be constructed in a language with weak mobility if the locus of movement is a function without external states beyond parameter values. Thus, AMPs [1] and AMSs [2] have been developed in Jocaml, Java Voyager, and JavaGo. However, the cost semantics we define, automate and evaluate in Sections 4–7 is for substantial subset of Jocaml only.

AMPs are relatively unusual mobile programs. AMPs relocate to obtain computational resource and hence collections of AMPs manage load. In contrast most strongly mobile programs relocate to obtain other resources, e.g. access to a specific repository. Moreover, where movement control is explicit in most strongly mobile programs, the purpose of our cost analysis and of AMS and CAMS is to make the movement as implicit as possible.

2.2. Agents and autonomous systems

Agent technology is a high-level, implementation independent approach to developing software as collections of distinct but interacting entities which cooperate to achieve some common goal. With the continuing decline in price and increase in speed of both processors and networks, it has become feasible to apply agent technology to problems involving cooperation in distributed environments, in particular, where agents may change location, typically to manipulate resources in varying locations.

An agent is "an encapsulated computer system that is situated in some environment, and that is capable of flexible, autonomous action in that environment in order to meet its design objectives" [16,17]. An agent with mobility is called a *mobile agent* [18], and AMPs are mobile agents.

Autonomous systems are also called autonomic computing systems, and a definition has been given by IBM: "autonomic computing system can manage themselves given high-level objectives from administrators" [19,20] and maintenance. Autonomic systems will maintain and adjust their operation in the face of changing components, workloads, demands, and external conditions and in the face of hardware or software failures. Four aspects of self-management are *self-configuration*, *self-optimisation*, *self-healing*, and *self-protection*. Different autonomic systems may have some or all these four aspects. AMPs are primarily self-optimisation systems. They are aware of their processing resource needs and sensitive to the environment in which they execute, and are able to dynamically relocate themselves to minimise processing time in the presence of varying external loads on shared locations.

Most distributed environments are shared by multiple users. In particular, distributed agent-based systems must also contend with external competition for resources, not least for the locations they share. The agents community has focused on autonomous problem solving, which can act flexibly in uncertain and dynamic environments. Mobile languages provide efficient tools to make the agent move more flexibly in the large scale network, which make it possible to build self-management systems (autonomous systems) for resource sharing using agent technology. So many autonomous systems are based on mobile agents [16].

AMPs have strong connections with both agents and autonomous systems, but they also have important differences. Firstly, unlike previous mobile agents approaches, AMPs have cost models and are autonomous, making decision themselves when and where to move according to the cost model. Furthermore, unlike traditional autonomous systems [19,21,22], which use schedulers to decide whether to move, AMPs themselves can make the decision when and where to move according to the cost model [1].

2.3. Cost analysis

Cost models estimate the resource consumption of a program, typically its execution time or memory consumption [23]. Although the cost of an arbitrary program cannot be accurately modelled, as this would imply solving the halting problem, useful predictions can be obtained for many programs. Costs may be modelled *statically* prior to execution, or *dynamically* during execution. Our work focuses primarily on generating static models of *computation*, *communication*, and *coordination* costs: the latter being the cost of determining where best to execute in a network. Here, statically generated models are dynamically instantiated to predict and adapt program behaviour.

Predicting resource consumption is an important problem and a range of static computation cost models have been constructed. Early work includes that of Cohen and Zuckerman, who consider cost analysis of Algol-60 programs [24]; Wegbreit, whose pioneering work on cost analysis of Lisp programs addressed the treatment of recursion [25]; and Ramshaw [26] and Wegbreit [27], who discuss the formal verification of cost specifications. Many of the cost analyses use non-standard semantics, e.g. Rosendahl [28] uses abstract interpretation for cost analysis, and Wadler [29] uses projection analysis.

Recent approaches use type inference to model time and space costs. In their influential paper, Hughes and Pareto [30] combine a sized type system with region-based memory management to semi-automate the prediction of space costs in Embeded ML. Hammond et al. [31] have extended the approach to develop a sized time system for a model higher-order functional language. In complementary work, Hofmann and Jost [32,33] have been exploring amortised cost models for heap use, which Hermann et al. [34] have extended to stack, heap and time analysis. Finally, Brady and Hammond [35] have been using dependent types to support static cost analysis.

Cost models that incorporate communication costs are well developed for parallel programming languages. For example many parallel languages use algorithmic skeletons which encapsulate the expression of parallelism, communication, synchronisation and embedding, and often have an associated cost model. Thus, Skillicorn and Cai have developed a cost calculus for the Bird Meertens Formalism (BMF) [36], and Rangaswami has developed the *HOPP* skeleton-based parallel programming language with an associated cost model [37].

We have developed a generic cost model for AMPS that incorporates not only computation and communication cost, but also coordination costs. We believe this is one of the first cost models for a mobile language. This generic model is instantiated for specific instances, e.g. for matrix multiplication [1,2]. The continuation cost semantics we present in Section 4 uses an approach similar to [38], but calculates costs using a non-standard semantics rather than type inference.

The cost semantics is extremely novel in being the first semantics to cost *continuations*. Additional novelty is provided by costing coordination alongside the classical computation and communication costs.

3. Previous work

3.1. Autonomous mobile programs

To manage load on large and dynamic networks we have developed what we term AMPs, which are aware of their processing resource needs and sensitive to the environment in which they execute [1]. Unlike autonomous mobile agents that move to change their function or *computation*, an AMP always performs the same computation, but move to change *coordination*, i.e. to improve performance. AMPs are able to dynamically relocate themselves to minimise execution time in the presence of varying external loads on a network of shared locations. The advantages of an AMP architecture are as follows.

- Mobility is truly autonomous as the AMPs themselves use local and external load information to determine when and where to move rather than relying on a central scheduler.
- AMPs combine analytic cost models with empirical observation of their own behaviours to determine their current progress. The generic AMP cost model is reprised in Section 5.1.
- The cost of movement can be kept to a very small proportion of overall execution time, under the assumption that location performance does not change radically immediately after a move, see discussion of Eq. (20) in Section 5.1.

A limitation of the cost model is that the parameterisation assumes that the computation is regular in the sense that the computational cost of each iteration is similar to those of the preceding iterations. This is formalised and discussed further in Section 5.

3.2. Single AMP performance

AMPs may dramatically reduce execution time. Fig. 1 compares the execution times of static and mobile matrix multiplication programs. Our test environment is based on three locations with CPU speed 534, 933 and 1894 MHz. The loads on these three computers are almost zero. We launch both static and mobile programs on the first location. The execution time of the static and mobile programs is very similar up to matrix sizes of 500×500 as both programs execute on the original location. For matrix sizes above 500×500 the cost of moving is outweighed by the speed of the fastest location and the mobile program moves, successfully reducing execution time for all larger matrices.

Fig. 2 shows the movement of the AMP matrix multiplication during successive execution time periods with CPU speeds normalised by the local loads. We launch the AMP in time period 0 on *Loc*1. In time period 1 it moves to the fastest processor currently available *Loc*3. When *Loc*3 becomes more heavily loaded in period 2 the AMP moves to the new fastest processor *Loc*5. In time period 3, *Loc*4 becomes less loaded and hence fastest location, so the AMP moves to it. Similarly for the other moves.

We draw the following conclusions from Fig. 2:

- The program may move repeatedly to adapt to changing loads and always finds the fastest location in a single step.
- Move (1) shows that if there is a faster location then the AMP moves to it.
- Move (2) shows that AMPs can respond to changes in current location.
- Move (3) shows that AMPs can respond to changes in other locations.
- Move (4) shows that even if the speed differential is small, the AMP moves.

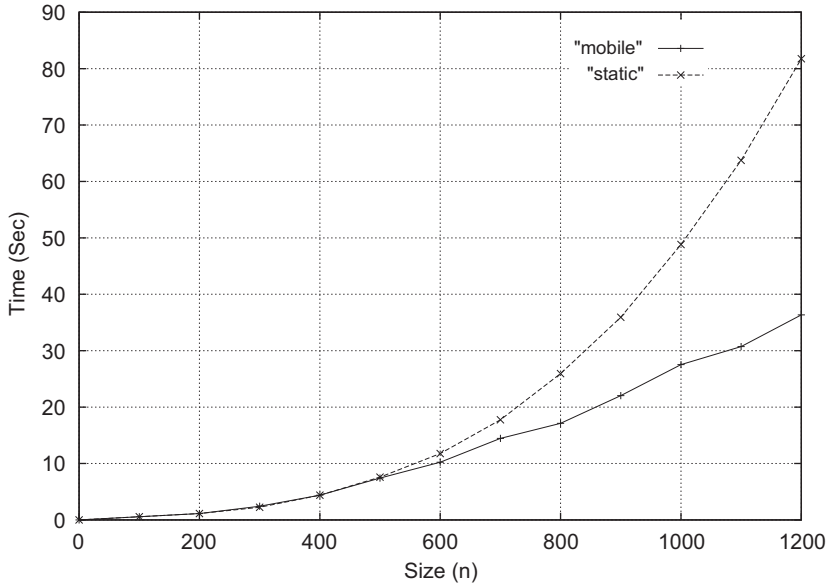

Fig. 1. AMP and static matrix multiplication execution time.

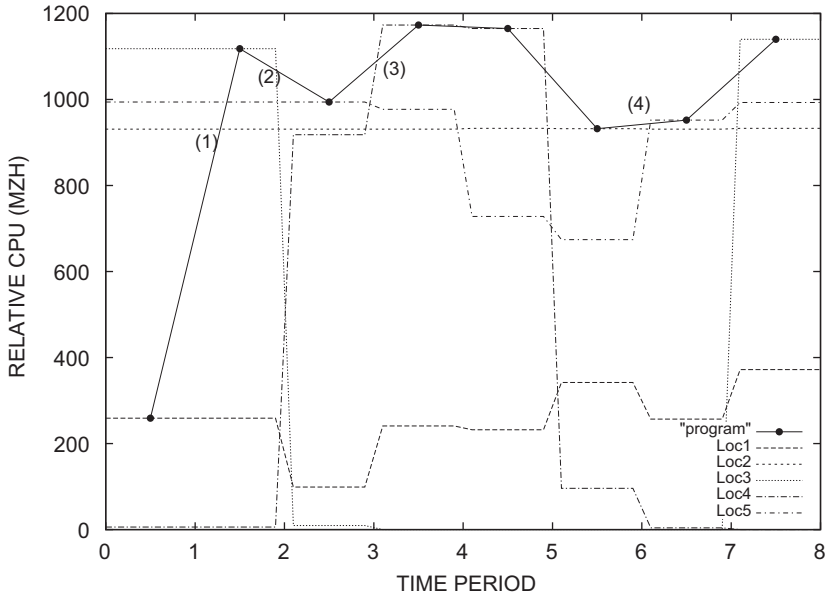

Fig. 2. Single AMP matrix multiplication movement.

3.3. Collections of AMPs

A collection of AMPs is balanced if every AMP has similar *relative CPU speed*, i.e. CPU speed divided by load, available to it. Experiments on both homogeneous and heterogeneous networks show that collections of AMPs quickly obtain and maintain balanced loads on the locations of a network [1,2].

For illustration Fig. 3 shows the movement of 25 AMPs on a network of 15 locations with CPU speeds 3193 MHz (Loc1–Loc5), 2168 MHz (Loc6–Loc10), and 1793 MHz (Loc11–Loc15). All the AMPs are launched on Loc1, and after some movements the AMPs achieve a balance (denoted **B** in the figure) in time period "k" with one AMP on each of the slower machines (Loc11–Loc15), two AMPs on each of the faster machines (Loc2–Loc10), excepting that one of the faster machines has 3 AMPs and the launch location (Loc1) is a communication bottleneck and has just 1 AMP. Thereafter the AMPs remain statically balanced until an AMP on the

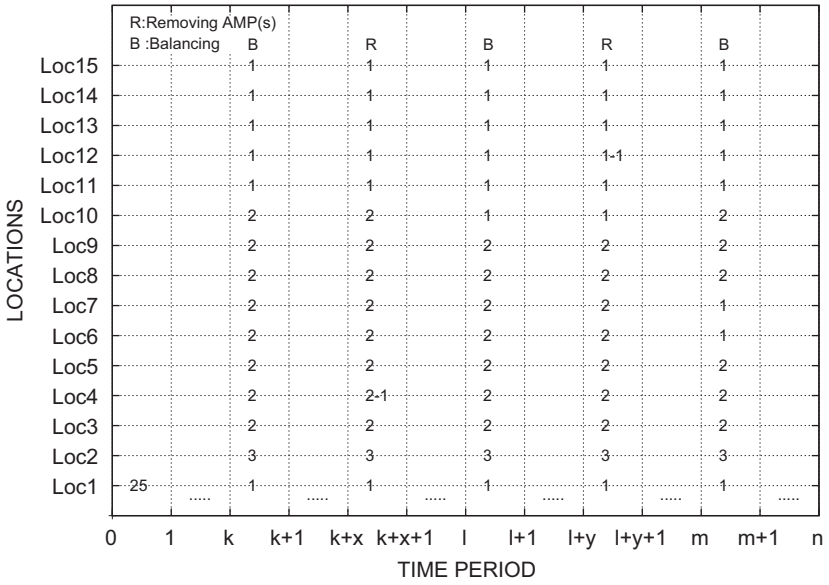

Fig. 3. Twenty-five AMPs on a heterogeneous network (15 locations).

fast Loc4 terminates at period "k + x", inducing a rebalancing period (denoted **R** in the figure) until period "l". The AMPs remain statically balanced until an AMP on the slow Loc12 terminates at period "l + y", inducing a rebalancing period until period "m".

3.4. Autonomous mobility skeletons

A disadvantage of directly programming AMPs is that the cost model, mobility decision function, and network interrogation are all explicit in the program. We have defined and evaluated AMSs that encapsulate autonomous mobility for common patterns of computation over collections [2]. AMSs are polymorphic higher-order functions, such as that make mobility decisions by combining generic and task specific cost models. We have built AMSs for the classic higher-order functions `map` and `fold` and for the object-oriented `Iterator` interface [39] decisions.

The `automap` AMS, performs the same computation as the map high order function, but may cause the program to migrate to a faster location. The standard Jocaml map, `map f [a1;...;an]` applies function `f` to each list element `a1,...,an`, building the list `[f a1;...; f an]`. automap, `automap cur f [a1;...;an]` computes the same value but takes another argument `cur`, recording current location information, e.g. CPU speed and load. The standard left fold, `fold f a [b1; ...; bn]`, computes `f (... (f (f a b1) b2) ...) bn`. `autofold f a [b1;...;bn]` computes the same value but may migrate to a faster location. The `automap` and `autofold` AMSs have also been constructed in both Jocaml and Java Voyager.

In the object-oriented as opposed to the functional paradigm, the Java `Iterator` specifies a generic mechanism to enumerate the elements of a collection. The `AutoIterator` AMS class implements the `Iterator` methods (`hasNext, next` and `remove`), and extends it with `autonext`, which has the same functionality as `next` but can make autonomous mobility.

4. Continuation cost calculus

AMSs only consider the costs of a single collection iteration. This is adequate only if a single collection iteration dominates the computational cost of the program. To deploy autonomous mobility effectively more generally, it is necessary to know the cost of the remainder of the program in addition to the cost of the current iteration. The cost of the remainder of the program is precisely the cost of the program continuation in denotational semantics [40] and we term it the *continuation cost*.

To the best of our knowledge the continuation cost model presented here is the first such model ever described. To calculate the continuation cost at an arbitrary program point, the cost of every expression must first be calculated. To illustrate the concept we use a small language, $e ::= n|e + e$, where n is integer. Cost judgements have the form $E\vdash_c e\$c$ indicating that the cost e is c in cost environment E. Given this cost function, the continuation cost judgements have the form $E\vdash_a e\trianglelefteq e'£c$ indicating that the continuation cost after e in e' is c in cost environment E. For example in $2 + 3$, the continuation cost of `2` can be calculated as $(E\vdash_c 3\$c_3 E\vdash_c +\$c_+)/(E\vdash_a 2\trianglelefteq(2+3)£c_3+c_+)$, where c_3 is the cost of `3`, c_+ is the cost of "+", and the continuation cost of `2` is $c_3 + c_+$.

A key issue with continuation costs is to distinguish the intended expression if it occurs more than once in a program. For example, in expression $10 + 10$, there are two `10`s and their continuation costs are different. To solve this problem the program is indexed, i.e. every expression in a program is assigned a unique number, its *index*. After indexing, the expression $10 + 10$ becomes $<3, <1, 10> + <2, 10>>$, and the two `10`s can be distinguished as $<1, 10>$ and $<2, 10>$.

$\vdash_i : n \rightarrow e \rightarrow e * n$	index
$\vdash_c : env \rightarrow e \rightarrow n$	expression cost
$\equiv : e \rightarrow e$	syntactic equality
$\in : e \rightarrow e \rightarrow boolean$	syntactic containment
$\vdash_a : env \rightarrow e \rightarrow e \rightarrow cost$	continuation cost

Fig. 4. Semantic functions.

$e ::=$		expression
	k	constant
|	v	variable
|	`fun` $v \rightarrow e$	lambda
|	$e\ e$	application
|	$e\ op\ e$	operation
|	`map` $e\ e$	`map`
|	e `(*` e `*)`	user cost pragma
|	$< n, e >$	index
$op ::=$		operator
	+ | - | * | /	arithmetical
|	> | < | >= | <= | = | !=	logical
|	::	cons
|	;	sequential composition

Fig. 5. Syntax of $\mathscr{J}'$.

Continuation costs are calculated in the following three stage process

- Index the program (Appendix A).
- Calculate the cost of all expressions in the program (Section 4.2).
- Calculate the continuation cost of a specified program point (Section 4.3).

For reference Fig. 4 lists the semantic functions used in the calculus, and defined in the following sections. In these semantic functions, *cost* is integer and the environment *env* incorporates costs, i.e. $env : (v * cost)^*$.

Calculating the cost of expression uses standard static cost semantics techniques as discussed in Section 2.3. However, calculating the continuation costs is entirely novel as most cost models cost entire terms. Both the cost semantics and continuation cost semantics have been implemented as components of an automatic continuation cost analyser, as described in Section 7.1.

4.1. Syntax of language $\mathscr{J}'$

Continuation cost semantics have been defined for $\mathscr{J}$, a substantial subset of the Jocaml mobile programming language. $\mathscr{J}$ is a core functional language including specific higher-order functions like `map` and `fold`, and is readily able to describe non-trivial programs like matrix multiplication and ray tracing. The syntax and cost calculus for $\mathscr{J}$ is presented in [41]. As a vehicle for explaining the principles of the semantics in this paper, this section introduces a simpler language $\mathscr{J}'$, a subset of $\mathscr{J}$. A subset of a strict functional language is chosen because it is significantly easier to define cost analyses for them than for their lazy counterparts.

Fig. 5 shows the abstract syntax of $\mathscr{J}'$. To simplify the presentation it is assumed that all identifiers (v) are unique. $\mathscr{J}'$ is a core lambda calculus with two unusual expression, the *index* expression and*user cost* pragmas. The *index* expression is required as every subexpression in the program is indexed with a unique integer *n*. As costing arbitrary recursive functions is undecidable, *user cost pragmas* are introduced, and their application is elaborated in Section 4.2, with examples in Section 7.4.

4.2. Cost semantics

Fig. 6 defines the cost semantics for $\mathscr{J}'$, E$\vdash_c e\$n$ The $\vdash_c$ function takes a cost environment (E) and an expression (e), and returns the predicted cost (n) in that environment. This cost semantics is standard, and is similar to [38] and others.

The cost environment records the cost of accessing variables, and in a language like $\mathscr{J}'$ that omits data structures, e.g. tuples, this is always 0. The cost environments are included in the cost semantics in Figs. 6 and 8 to illustrate how they are used in richer languages like $\mathscr{J}$ that have structured data. The cost semantics reflects the strict, or applicative, semantics of Jocaml, e.g. the cost of evaluating function arguments is reflected in Eqs. (4) and (7).

$$\frac{}{\mathrm{E} \vdash_c k \,\$\, 0} \tag{1}$$

$$\frac{}{\{v, c\} \oplus \mathrm{E} \vdash_c v \,\$\, c+1} \tag{2}$$

$$\frac{\{v, 0\} \oplus \mathrm{E} \vdash_c e \,\$\, c}{\mathrm{E} \vdash_c \texttt{fun}\ v\ \rightarrow\ e \,\$\, c} \tag{3}$$

$$\frac{\mathrm{E} \vdash_c e_1 \,\$\, c_1 \qquad \mathrm{E} \vdash_c e_2 \,\$\, c_2}{\mathrm{E} \vdash_c (e_1\ e_2) \,\$\, c_1 + c_2} \tag{4}$$

$$\frac{\mathrm{E} \vdash_c e_1 \,\$\, c_1 \qquad \mathrm{E} \vdash_c e_2 \,\$\, c_2}{\mathrm{E} \vdash_c e_1\ op\ e_2 \,\$\, 1 + c_1 + c_2} \tag{5}$$

$$\frac{}{\mathrm{E} \vdash_c e\ (*\ c\ *) \,\$\, c} \tag{6}$$

$$\frac{\mathrm{E} \vdash_c e_1 \,\$\, c_1 \qquad \mathrm{E} \vdash_c e_2 \,\$\, c_2}{\mathrm{E} \vdash_c \texttt{map}\ e_1\ e_2 \,\$\, c_1 * (length\ e_2) + c_2} \tag{7}$$

$$\frac{\mathrm{E} \vdash_c e \,\$\, c}{\mathrm{E} \vdash_c < i, e > \,\$\, c} \tag{8}$$

Fig. 6. Cost semantics for $\mathscr{J}'$.

$$\dfrac{\dfrac{\mathrm{E}' \vdash_c x \,\$\, 0+1 \qquad \mathrm{E}' \vdash_c 10 \,\$\, 0 \quad (2,1)}{\mathrm{E}' \vdash_c x+10 \,\$\, 1+((0+1)+0) \quad (5)} \qquad \dfrac{\dfrac{\mathrm{E} \vdash_c 20 \,\$\, 0 \qquad \mathrm{E} \vdash_c [\ ] \,\$\, 0 \quad (1,1)}{\mathrm{E} \vdash_c 20::[\ ] \,\$\, 1+(0+0) \quad (5)}}{\mathrm{E} \vdash_c fun\ x \rightarrow (x+10) \,\$\, 2 \quad (3)}}{\mathrm{E} \vdash_c map\ (fun\ x \rightarrow (x+10))\ ([20]) \,\$\, 2 * (length\ [20]) + 1 \quad (7)}$$

Fig. 7. $\mathscr{J}'$ cost example.

Eq. (1) calculates the cost of an constant as 0.

Eq. (2) shows that the cost of the value of a variable v has been stored in the environment as c, so the total cost of v is calculated as c plus an access cost of 1.

Eq. (3) calculates the cost of a lambda abstraction as the cost of the body in an environment where the parameter has zero cost.

Eq. (4) calculates the cost of a function application as the cost of the function body c_1 plus the cost of the argument c_2. Eq. (5) is very similar.

Eq. (6) enables the user to specify a cost and is intended to be used for arbitrary recursive functions.

Eq. (7) calculates the cost of a `map` as the sum of the cost of computing the list c_2 and the product of the cost of the function body c_1 and the length of the list. Note that this equation assumes that the cost of applying the mapped function to every list elements is uniform.

Eq. (8) calculates the cost of index expression as the cost of the expression.

Fig. 7 shows the costing of (`map(fun x → x+10)` [20]) in a cost environment E that is initially empty. During the evaluation the environment is extended with x bound to zero, i.e. $\mathrm{E}' = \{x, 0\} \oplus E$. Indexing this expression produces $<8, (\texttt{map} < 4, (\texttt{fun x} \rightarrow < 3, (< 1, \texttt{x} > + < 2, 10 >) >) > < 7, < 5, 20 > :: < 6, [\,] > >) >$, as depicted in Fig. 29. However, for clarity in the example costing we elide the indices. The predicted cost of 3 is the sum of the cost of evaluating the list (1) and the product of the cost of the function body (2) and the length of the list (1). The automated cost analyser presented in Section 7 produces the following unsimplified cost term for this expression: $(1 + ((1 + 0) + 0)) * (\mathit{length}[20]) + (1 + (0 + 0))$.

4.3. Continuation cost semantics

This section introduces the continuation cost semantics of $\mathscr{J}'$, and we believe the first ever continuation cost semantics for any language. Continuation costs could be calculated by translating the direct program into continuation passing style (CPS) [42], and then costing the continuations now explicit in the program. We prefer the direct programming style and hence to pass the continuation costs rather than the continuations. An additional advantage is that the integer cost parameters are rather simpler than continuations to pass and manipulate.

To define continuation costs we must be able to determine both the syntactic *equality* ($e \equiv e'$) of expressions and to determine when one expression syntactically *contains* another ($e \in e'$). These definitions are standard and are discussed in Appendix B.

$$\frac{e \equiv e'}{\mathrm{E} \vdash_a e \trianglelefteq e' \pounds 0} \quad (9)$$

$$\frac{e \in e_1 \qquad \mathrm{E} \vdash_a e \trianglelefteq e_1 \pounds c}{\mathrm{E} \vdash_a e \trianglelefteq \mathtt{fun}\ v \rightarrow e_1 \pounds c} \quad (10a)$$

$$\frac{e \notin e_1}{\mathrm{E} \vdash_a e \trianglelefteq \mathtt{fun}\ v \rightarrow e_1 \pounds 0} \quad (10b)$$

$$\frac{e \in e_1 \qquad \mathrm{E} \vdash_a e \trianglelefteq e_1 \pounds c_1 \qquad \mathrm{E} \vdash_c e_2 \$ c_2}{\mathrm{E} \vdash_a e \trianglelefteq (e_1\ e_2) \pounds c_1 + c_2} \quad (11a)$$

$$\frac{e \in e_2 \qquad \mathrm{E} \vdash_a e \trianglelefteq e_2 \pounds c_2}{\mathrm{E} \vdash_a e \trianglelefteq (e_1\ e_2) \pounds c_2} \quad (11b)$$

$$\frac{e \notin e_1 \qquad e \notin e_2}{\mathrm{E} \vdash_a e \trianglelefteq (e_1\ e_2) \pounds 0} \quad (11c)$$

$$\frac{e \in e_1 \qquad \mathrm{E} \vdash_a e \trianglelefteq e_1 \pounds c_1 \qquad \mathrm{E} \vdash_c e_2 \$ c_2}{\mathrm{E} \vdash_a e \trianglelefteq (e_1\ op\ e_2) \pounds 1 + c_1 + c_2} \quad (12a)$$

$$\frac{e \in e_2 \qquad \mathrm{E} \vdash_a e \trianglelefteq e_2 \pounds c_2}{\mathrm{E} \vdash_a e \trianglelefteq (e_1\ op\ e_2) \pounds c_2 + 1} \quad (12b)$$

$$\frac{e \notin e_1 \qquad e \notin e_2}{\mathrm{E} \vdash_a e \trianglelefteq (e_1\ op\ e_2) \pounds 0} \quad (12c)$$

$$\frac{}{\mathrm{E} \vdash_a e \trianglelefteq e_1\ (*\ c\ *) \pounds 0} \quad (13)$$

$$\frac{e \in e_1 \qquad \mathrm{E} \vdash_c \mathtt{map}\ e_1\ e_2 \$ c \qquad \mathrm{E} \vdash_a e \trianglelefteq e_1 \pounds c_1}{\mathrm{E} \vdash_a e \trianglelefteq \mathtt{map}\ e_1\ e_2 \pounds c + c_1 + c_2} \quad (14a)$$

$$\frac{e \in e_2 \qquad \mathrm{E} \vdash_c \mathtt{map}\ e_1\ e_2 \$ c \qquad \mathrm{E} \vdash_a e \trianglelefteq e_2 \pounds c_2}{\mathrm{E} \vdash_a e \trianglelefteq \mathtt{map}\ e_1\ e_2 \pounds c + c_2} \quad (14b)$$

$$\frac{e \notin e_1 \qquad e \notin e_2}{\mathrm{E} \vdash_a e \trianglelefteq \mathtt{map}\ e_1\ e_2 \pounds 0} \quad (14c)$$

$$\frac{\mathrm{E} \vdash_a e \trianglelefteq e_1 \pounds c}{\mathrm{E} \vdash_a e \trianglelefteq <i, e_1> \pounds c} \quad (15)$$

Fig. 8. Continuation cost semantics for $\mathscr{J}'$.

$$\frac{\dfrac{\mathrm{E} \vdash_a 15 \trianglelefteq 15 \pounds 0 \quad (9) \qquad \mathrm{E} \vdash_c map\ (fun\ x \rightarrow (x + 10))\ (20 :: [\ ])\ \$\ 3 \quad (\text{Figure 7})}{\mathrm{E} \vdash_a 15 \trianglelefteq 15 :: map\ (fun\ x \rightarrow (x + 10))\ (20 :: [\ ])\ \pounds\ 1 + 0 + 3} \quad (12a)}{\mathrm{E} \vdash_a 15 \trianglelefteq 8 :: 15 :: map\ (fun\ x \rightarrow (x + 10))\ (20 :: [\ ])\ \pounds\ 1 + 0 + 3} \quad (12b)$$

Fig. 9. $\mathscr{J}'$ continuation cost example.

Fig. 8 defines the continuation cost semantics for $\mathscr{J}'$, $\mathrm{E} \vdash_a e \trianglelefteq e' \pounds n$. The $\vdash_a$ function takes a cost environment (E) and two expressions (e and e') and returns the predicted continuation cost of the first expression in the second: that is, the work that remains to be done in e' after evaluating e. Like the cost semantics, the continuation costs reflect the strict semantics of Jocaml.

Eq. (9) specifies that the continuation cost of an expression in itself is 0.

Eq. (10a) specifies that the continuation cost of an expression contained in a lambda abstraction the continuation cost in the body. Conversely Eq. (10b) specifies that is expressions not contained in the abstraction have continuation cost 0.

Eqs. (11a), (11b), and (11c) specify the continuation cost of an expression in a function application. Eq. (11c) specifies that expressions not contained in the application have continuation cost 0. Eq. (11b) specifies that expressions contained in the argument have the continuation cost of the argument. Eq. (11a) specifies that expressions contained in the function body have continuation cost comprising the cost of the argument plus the continuation cost of the expression in the function body. Note that as every expressions has a unique index, only one of Eq. (11a) or (11b) will ever apply.

Equation sets (20) and (22) are similar to equation set (19).

Eq. (13) specifies that the continuation cost of an expression in a user cost expression is 0.

Eq. (15) specifies that the continuation cost of an expression e in an indexed expression $e1$ is the continuation cost of e in $e1$.

Fig. 9 illustrates continuation costing by determining the continuation cost of `15` in `8 :: 15 :: (map(fun x -> x + 10) [20])`. The continuation cost of $1 + 0 + 3$ comprises three parts: a unit cost for the second cons (`::`) operation, a zero cost for the 15,

and a 3 unit cost for the `map` expression, as determined in Fig. 7. This latter cost is a cost and not a continuation cost. Note also that the cost of performing the first cons operation (`8::`) is not included as it is incurred before the 15 in the term.

5. Costed autonomous mobility skeletons

To produce skeletons capable of modelling not only the cost of the current iteration, but also the cost of the remainder of the program, the continuation cost equations from the previous section must be incorporated into the skeleton cost model. The new skeletons are termed CAMS and are parameterised with both costs and continuation costs. As we shall see, the CAMS cost model is a specialisation of the generic AMP cost model. This section concludes by outlining the implementation of the `camap` and `cafold` CAMS in Jocaml.

5.1. AMP generic cost model

A cost model is used by an AMP to inform the decision whether to move to a new location. The cost model is generated statically, and is parameterised dynamically to determine movement behaviour. The generic AMP cost model in Fig. 10 is described in detail in [1].

Eq. (16) states that the total execution time of an AMP is the sum of the computation, communication and coordination times. All times are measured in seconds.

Eq. (17) gives the condition under which the program will move, i.e. if the time to complete in the current location T_h is more than the time to complete in the best available remote location T_n plus the time to send the computation to the new location T_{comm}.

Eq. (18) states that total communication costs is the product of the number of moves and the communication cost of each move.

Eq. (19) states that total coordination cost is the product of the number of locations, the number of movement checks and the time for a single movement check.

Eq. (20) limits the coordination cost of AMPs by selecting some overhead value O, say 5% and seeking to guarantee that the AMP execution time will never exceed than $100 + O\%$ a static version of the program. This guarantee is only valid providing that the loads on the locations, primarily current and target locations, do not change dramatically immediately after the move. A more complete discussion of this issue can be found in [1].

Substituting Eq. (19) in (20) gives Eq. (21) which specifies how many coordination actions will occur during the AMP execution.

Eqs. (22), (23) and (24) relate time, work and CPU speed. Implicit in these equations is the assumption that the computation is regular in the sense that the work left to be done W_l can be predicted from the work already completed W_a. While this is true for many programs including the example programs in this paper, many other programs do not possess this property. A more complete discussion of this issue can be found in [1], including ideas for adapting the model for less regular computations.

Eq. (25) states that the total work completed is the sum of the work done W_d at each location.

Eq. (26) states that the work done at the current location is work done W_a less the work done at the point the AMP arrived at the location $W_{a'}$.

Eq. (27) states that the remaining work is the total work minus work done.

5.2. CAMS cost model

The cost model for CAMS in Fig. 11 instantiates the generic AMP cost model and is parameterised on the continuation cost of the skeletons. That is CAMS determine whether to move or not by predicting not only the cost of the current iteration, but also the cost of the remainder of the program. In this cost model:

Eq. (28) states that the total work is the cost of the current iteration plus the continuation cost. The cost of the current iteration is, as for AMS, the product of the cost of evaluating a single element c_f and the *size* of the collection. For `map` and `fold`, *size* is the length of the list. The abstract costs produced by the continuation cost semantics are converted into cycles by a mapping function: $(|.|)$. The calibration of this mapping is covered next.

Eq. (29) reflects the CAMS design where the speed of the current location is calculated just once by computing a single element of the collection, and hence the work done at the current location is simply $(|c_f|)$. Comparing the elapsed time for this computation $(|c_f|)$ with the predicted abstract time c_f, calibrates the mapping between the two on this location.

Substituting Eq. (29) in (22) derives Eq. (30) that predicts the elapsed time at the current location.

The speed of the current location can be derived from Eq. (30) as $S_h = (|c_f|)/T_e$, and substituting in Eq. (23) derives Eq. (31) that predicts the time to complete the program at the current location as a function of T_e.

The work remaining can be derived from the first equality in Eq. (31) as $W_l = T_h S_h$, and substituting in Eq. (24) derives Eq. (32) that predicts the time to complete the program at the best available alternative location as the product of the time to complete here and the ratio of the current and best available location speeds.

As an instantiation of the generic AMP cost model, the CAMS cost model inherits its limitations. That is the model is valid only for regular computations, and only guarantees minimal overheads if location loads remain stable, as discussed for Eqs. (20) and (22).

$$T_{total} = T_{Comp} + T_{Comm} + T_{Coord} \tag{16}$$

$$T_h > T_{comm} + T_n \tag{17}$$

$$T_{Comm} = mT_{comm} \tag{18}$$

$$T_{Coord} = npT_{coord} \tag{19}$$

$$T_{Coord} < OT_{static} \tag{20}$$

$$n < \frac{OT_{static}}{pT_{coord}} \tag{21}$$

$$T_e = W_d/S_h \tag{22}$$

$$T_h = W_l/S_h \tag{23}$$

$$T_n = W_l/S_n \tag{24}$$

$$W_a = \sum W_d \tag{25}$$

$$W_d = W_a - W_{a'} \tag{26}$$

$$W_l = W_{all} - W_a \tag{27}$$

O	:	Overhead e.g. 5%
T_{total}	:	total execution time
T_{static}	:	time for static program running on the current location
T_{Comm}	:	total time for communication
T_{comm}	:	time for a single communication
T_{Coord}	:	total time for coordination
T_{coord}	:	time for coordination with a single location
T_{Comp}	:	time for computation
T_e	:	time elapsed at current location
T_h	:	predicted time at current location (here)
T_n	:	predicted time at best available (next) alternative location
W_{all}	:	all work (cycles)
W_a	:	total work that has been completed (cycles)
W_d	:	work completed at current location (cycles)
W_l	:	the work left (cycles)
S_h	:	CPU speed of current (here) location (cycles/s)
S_n	:	CPU speed of best available (next) alternative location (cycles/s)
m	:	number of AMP movements
n	:	number of movement checks
p	:	number of locations

Fig. 10. Generic cost model for AMPs.

$$W_{all} = (\!|c_f * size(collection) + continuationCost|\!) \tag{28}$$

$$W_d = (\!|c_f|\!) \tag{29}$$

$$T_e = \frac{W_d}{S_h} = \frac{(\!|c_f|\!)}{S_h} \tag{30}$$

$$T_h = \frac{W_l}{S_h} = \frac{W_l T_e}{(\!|c_f|\!)} \tag{31}$$

$$T_n = \frac{W_l}{S_n} = \frac{S_h T_h}{S_n} \tag{32}$$

Fig. 11. Cost model for CAMS.

```
let rec camap' f l costf continuationCost fhtime workleft=
   let (h::t) = l in
   if (((!t_current)-.(!t_last)) >= (!whencheck))
   then
   (check_move costf workleft fhtime;
    let (fh,fhtime') = timedapply f h in
    t_current := Unix.gettimeofday();
    getInfo costf fhtime';
    fh::camap' f t costf continuationCost fhtime' (workleft-costf)
   )
   else
   (let fh = f h in
    t_current := Unix.gettimeofday();
    fh::camap' f t costf continuationCost fhtime (workleft-costf)
   )
let camap f l costf continuationCost =
   let (h::t) = l in
   (let localwork = costf * (length l) in
    let work = localwork + continuationCost in
    let (fh,fhtime) = timedapply f h in
    t_current := Unix.gettimeofday();
    getInfo costf fhtime;
    fh::camap' f t costf continuationCost fhtime (work-costf)
   )
```

Fig. 12. Implementation of `camap` in Jocaml.

5.3. Implementing CAMS

Fig. 12 shows a Jocaml implementation of a `camap` costed autonomous mobility skeleton. `f` is the function to be mapped over the list `l`. `costf` is the cost for one application of `f` and `continuationCost` is the continuation cost.

`camap` is the top level CAMS which in turn calls the auxiliary `cmap'` after calculating the overall work, and timing `f` applied to the first element of `l` as a base measure. The movement check is encoded in `check_move` function that applies the cost model from Fig. 10, and specifically Eq. (17). The `checkInfo` function accumulates the dynamic information required by the cost model, and recalculates when the AMP should consider moving again. Both functions are the same as used in AMSs, and the implementation of `cafold` is similar [41].

6. Evaluating CAMS

To demonstrate the utility of the continuation costs we compare the performance of CAMS with AMSs. More specifically we compare the performance of six pairs of programs where one program is constructed with an AMS and the other with a CAMS. All of the results reported in this section are for programs using either `automap` or `camap`. Measurements using the `cafold` CAMS are reported in [3,41]. The evaluation is broadly classified into single-iteration, sequences of iterations, and behaviour under varying loads.

The predicted costs required to parameterise the CAMS programs are calculated by hand using the cost calculus for $\mathcal{J}$. Section 7 will introduce an automatic continuation cost analyser that can automatically translate iterations into CAMS.

6.1. Single iteration examples

To show the consistency of the cost and continuation cost semantics we consider programs dominated by a single iteration. In these programs the continuation cost approximates zero and hence does not usefully contribute to movement decisions. In this case we hypothesise that CAMS programs will reproduce the movement of the corresponding AMS program. That is both AMPs should exhibit the same movement behaviours at the same points during execution and hence have similar execution times.

Moreover, there is the risk that the runtime cost of maintaining and utilising continuation costs in CAMS could have a deleterious effect on AMP performance. This effect will be most pronounced for programs dominated by a single iteration, i.e. where the continuation cost so carefully managed provides no benefit. If, however, the cost of maintaining and utilising continuation costs is small we further hypothesise that a CAMS encoding of single-iteration AMP should have very similar performance to an AMS encoding.

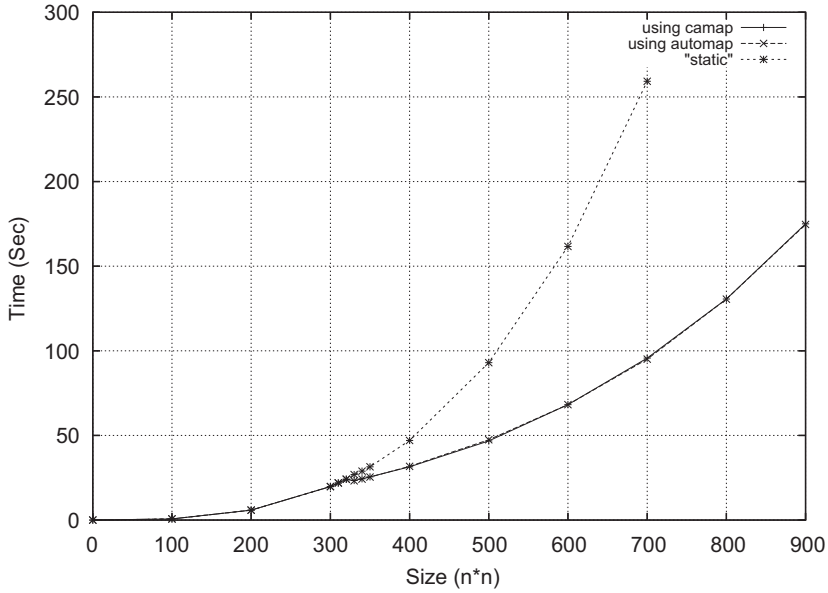

Fig. 13. CAMS and AMS single iteration (matrix multiplication) execution time.

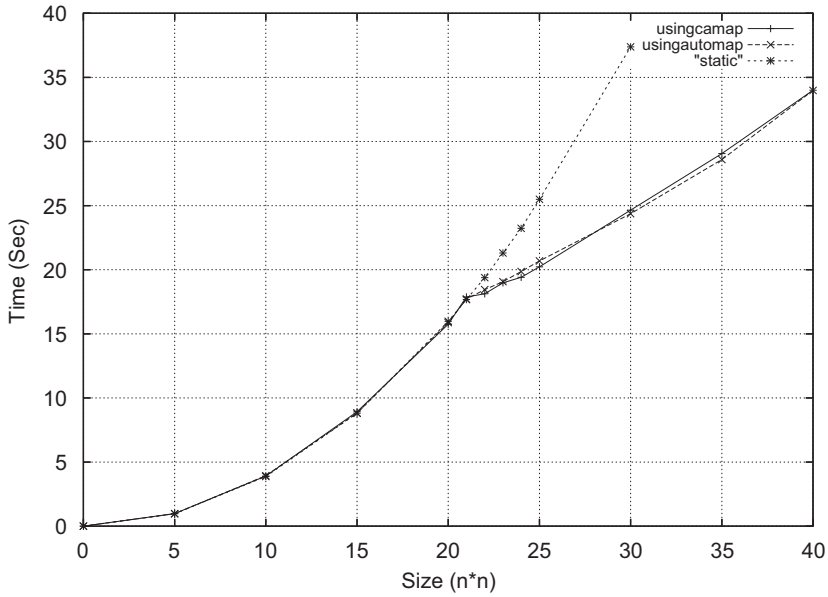

Fig. 14. CAMS and AMS single iteration (ray tracing) execution time.

Two single iteration AMPs have been tested both using the map higher-order function: matrix multiplication and ray tracing. Ray tracing is a well-known graphics algorithm that models the path taken by light rays as they interact with optical surfaces. Different size matrices are multiplied, and scenes traced, to compare the coordination behaviour of the AMS and CAMS programs. The test environment has three locations with CPU speeds 534 MHz(ncc1710), 933 MHz(jove) and 1894 MHz(lxtrinder). The loads on these three locations are almost zero, and both the CAMS and AMS programs are launched on the first location.

Figs. 13 and 14 show that both hypotheses are substantiated. That is the cost and continuation cost models are consistent as the CAMS and AMS programs exhibit very similar movement behaviours. Moreover the overheads of collecting and utilising the continuation costs are relatively small as the CAMS and AMS programs have very similar performance. For example both the CAMS and AMS matrix multiplications compute that it is beneficial to move when the matrix size reaches $330 * 330$, and hence the execution time curves in Fig. 13 have very similar shapes. Moreover the execution time of the programs at all data sizes are

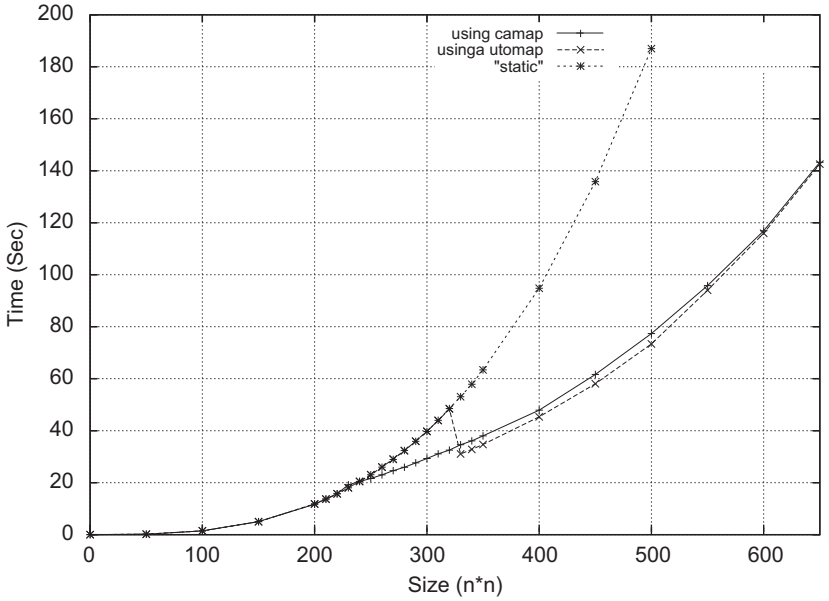

Fig. 15. CAMS and AMS multiple iteration (invertible matrix) execution time.

very similar and we conclude that the overheads of collecting and utilising the continuation costs are relatively small. Fig. 14 shows that the ray tracing CAMS and AMS programs repeat the pattern, as do the graphs in the following sections.

6.2. Sequences of iterations

A CAMS encoding should deliver better performance than an AMS encoding for AMPs comprising sequences of iterations because the cost model used in the earlier iterations includes the predicted costs of the remainder of the program, i.e. the subsequent iterations. This section investigates the performance of two pairs of AMS and CAMS encodings of programs comprising sequences of iterations, specifically sequential compositions of maps. The AMPs embody a test to see if two matrices are invertible and a sequence of five matrix multiplications. The test environment is the same as in Section 6.1. The results for a further two pairs of programs, including ray tracing a sequence of scenes, are reported in [41].

The invertible matrix program takes two matrices `m1` and `m2` and checks if they are invertible by multiplying them in both orders and checking that the result is the identity matrix in each case. The essence of the program is as follows.

```
let m12 = mmult m1 m2;;
let isId12 = checkEqual m12 idMat;;
let m21 = mmult m2 m1;;
let isId21 = checkEqual m21 idMat;;
```

Fig. 15 compares the performance of an CAMS encoding of invertible matrix using `camap`, and an AMS encoding using `automap`. The `camap` AMP computes that it is beneficial to move when the matrix size reaches $230 * 230$, but the `automap` AMP only moves when the matrix size reaches $330 * 330$. The `camap` AMP moves sooner because its cost model incorporates the continuation cost, i.e. the cost of the second matrix multiplication. For matrices between $230 * 230$ and $330 * 330$ the CAMS program has a significant performance advantage, up to a maximum of 33%.

For any multi-iteration program, a CAMS encoding will gain a performance advantage over the corresponding AMS program by moving at a smaller data size. For illustration, Figs. 16 and 17 show that the CAMS encoding of a double raytracer and of a fivefold matrix multiplication move at smaller data size than the corresponding AMS encodings. The analysis of other CAMS/AMS program pairs in [41] further supports this claim.

Figs. 15–17 alsoshow that in some cases the CAMS programs are marginally slower, i.e. no more than 11% slower, than the corresponding AMS programs due to some additional movement checks and some rather subtle quantum effects on the frequency of movement checks, as explained below. In all cases the CAMS programs remain faster than the static programs.

- The AMS program may move at an *earlier element* (not size) than the CAMS program if the matrix is large enough for both programs to move, e.g. at size $330 * 330$. This arises as follows, the first `automap` cost model is parameterised only with the current 330 element iteration, and hence determines to consider moving just once at element 165th. In contrast the first `camap` cost model is parameterised with both the current 330 element iteration and the continuation cost of the second iteration,

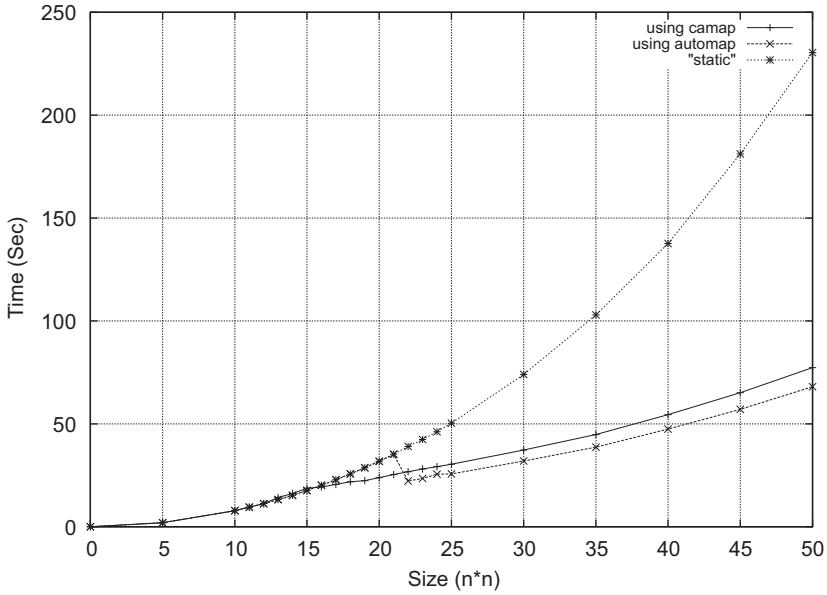

Fig. 16. CAMS and AMS multiple iteration (double ray tracing) execution time.

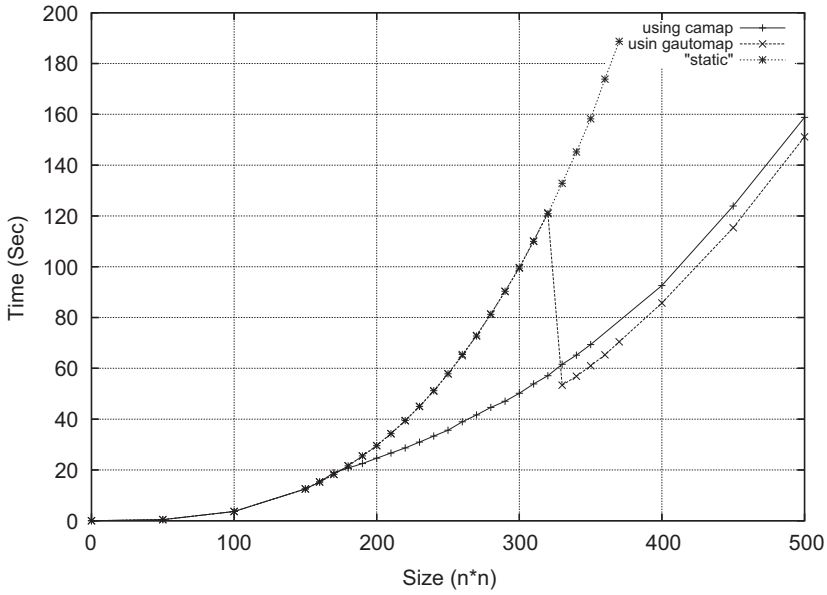

Fig. 17. CAMS and AMS multiple iteration (five matrix multiplication) execution time.

again 330 elements, and hence determines to consider moving twice i.e. at element $220 = (330 + 330)/3$. As the CAMS program moves to the faster location later than the AMS program it is a little slower.

- A CAMS program may perform more movement checks than the corresponding AMS program. For example, after the $330 * 330$ invertible matrix AMS program moves to a faster location, the first `automap` cost model is parameterised with only 165 elements and may not consider moving again. However, the first `camap` in the CAMS program is parameterised with $110 + 330 = 440$ elements, and so may consider moving again. The overhead of the additional movement checks will not exceed the overhead specified in Eq. (20) of the cost model under reasonable assumptions.

Fig. 17 compares the performance of CAMS, AMS and static versions of a program performing a sequence of five matrix multiplications. The results are similar to those for the invertible matrix, but the composition of more computations gives CAMS

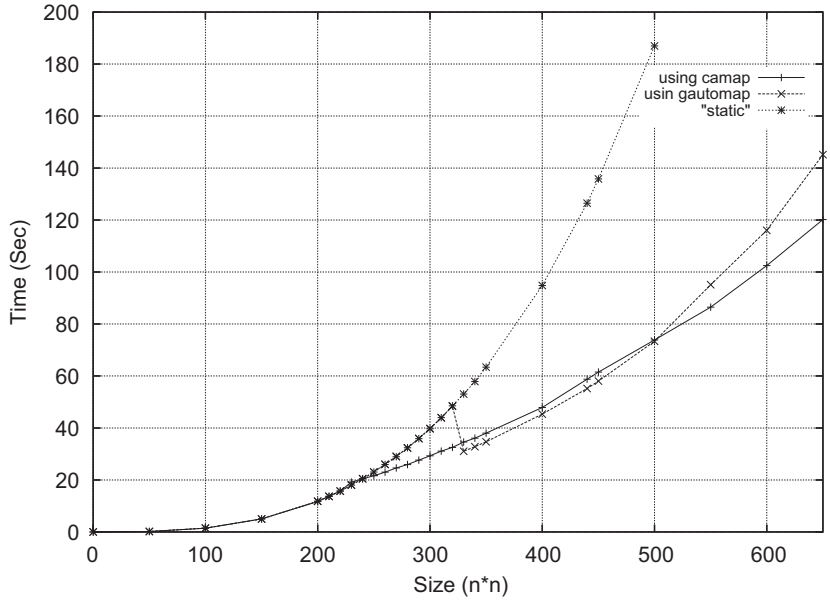

Fig. 18. CAMS and AMS on dynamic networks (invertible matrix) execution time.

both a greater performance advantage over AMS, i.e. up to a maximum of 53%, for a larger set of problem sizes: i.e. between matrix sizes of $170 * 170$ and $330 * 330$. The CAMS program computes that it is beneficial to move when the matrix size reaches $170 * 170$ based on the predicted cost of all five matrix multiplications. When the matrix size reaches $330 * 330$ the CAMS program is marginally slower, i.e. no more 15% slower, than the AMS program due to some additional movement checks and movement check quantisation, as for the invertible matrix program.

6.3. Performance of CAMS in dynamic networks

In the previous section we saw that a CAMS program with multiple iterations may more frequently determine whether to move (i.e. do more movement checks) than the corresponding AMS program. The additional movement checks may make the CAMS program marginally slower than the static or AMS versions, as illustrated for matrices larger than $330 * 330$ in Fig. 15, and similarly in Figs. 16 and 17.

The additional movement checks enable a CAMS program to react better to changes in its environment than the corresponding AMS program. Fig. 18 compares the execution times of CAMS and AMS invertible matrix programs on a network where a very fast machine becomes available only late in the computation. The experiment is described in detail in [41], but the essence is as follows. The AMS program performs only a single movement check and moves to a fast location early in the computation. In contrast the CAMS program performs two movement checks, moving first to the fast location, and then to the very fast location. In consequence the CAMS program outperforms the AMS program for matrices larger than $500 * 500$. Fig. 19 shows similar results for five matrix multiplication AMPs in the same scenario.

6.4. Evaluation summary

The conclusions drawn from the evaluation of CAMS in Sections 6.1–6.3 are:

- The cost and continuation cost models are consistent as, for programs dominated by a single iteration and hence where the continuation cost is not useful, CAMS programs reproduce the movement of the corresponding AMS program (Section 6.1).
- The overheads of collecting and utilising the continuation costs are relatively small as CAMS and AMS programs have very similar performance (Section 6.1).
- For programs dominated by sequences of iterations a CAMS encoding models the cost of all iterations and hence may move when an AMS program does not, thereby gaining a performance advantage. The more iterations in the program the greater the potential for gains (Section 6.2).
- When both the CAMS and AMS programs move, the CAMS programs may be marginally slower, e.g. at most 11% in invertible matrix and 15% in five matrix multiplications, than the corresponding AMS program due to some additional movement checks and quanta effects on the frequency of movement checks.

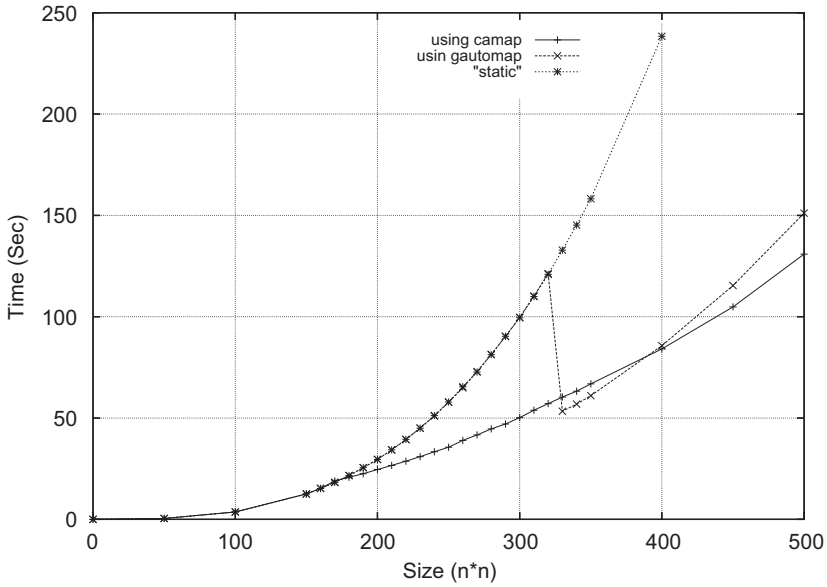

Fig. 19. CAMS and AMS on dynamic networks (five matrix multiplication) execution time.

- The additional movement checks enable CAMS programs to react to network changes more sensitively than the corresponding AMS programs (Section 6.3).

7. Automatic continuation cost analyser

Requiring the programmer to insert CAMS and their cost models into a program places a burden on them. This might be avoided by automatically converting a sequential program into an AMP that will move to exploit computational resources on a network. In theory the automation should be straightforward: the continuation cost semantics can be implemented as a static analysis and the cost equations generated can be used by a translator that replaces iterating higher-order functions with the corresponding CAMS, e.g. a `map f l` is replaced by `camap f l costf continuationCost`. The technique of replacing higher-order functions with algorithmic skeletons is common in parallelising compilers, e.g. [43].

So much for the theory; in reality the challenge is to produce an *effective* automatic analysis. That is, automated analyses typically produce unsimplified cost terms containing redundant arithmetic or other functions that incur runtime overheads. The key issue is whether the AMPs with automatically generated CAMS have acceptable performance compared with hand-costed CAMS.

The cost calculus has been implemented as an *automatic continuation cost analyser* that generates cost equations parameterised on program variables in context. The analyser generates both the *cost* of expressions, and the *continuation cost* of iterations. The analyser takes a $\mathcal{J}$ program as input and outputs $\mathcal{J}$ AMPs with CAMSs.[1]

7.1. Continuation cost analyser structure

Fig. 20 shows the structure of the continuation cost analyser which has the following four primary components.

1. The *Parser* takes a $\mathcal{J}$ and outputs the abstract syntax tree (AST).
2. The *Indexer* is an implementation of the index semantics in Appendix A and adds a unique index to each AST node to produce an *indexed abstract syntax tree* (*IAST*).
3. The *Coster* takes the IAST and adds the continuation cost to every node. The coster has two parts: the first calculates the cost of each expression using the cost semantics from Section 4.2; the second calculates the continuation cost of each expression using the continuation cost semantics from Section 4.3 which incorporates the costs previously generated.
4. The *Generator* converts a $\mathcal{J}$ program into an AMP by replacing specific higher-order functions with the corresponding CAMS, e.g. `map` is replaced by `camap`, parameterised with the continuation costs previously calculated.

[1] As discussed in Section 4.1, $\mathcal{J}$ is a subset of the Jocaml mobile programming language, and a superset of the $\mathcal{J}'$ language illustrated in Section 4.

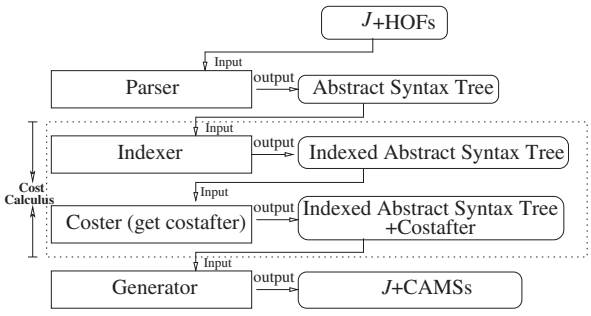

Fig. 20. Structure of automatic continuation cost analyser.

7.2. Implementing the cost calculus

The implementations of the `index`, `cost`, and `continuation cost` functions are as direct translations from their definitions in Figs. 28, 6 and 7, respectively. The semantic functions use syntactic equality (=) and syntactic containment `contains` functions directly translated from their definitions in Appendix B.

The `index` has type `int -> expression -> (expression * int)`, and takes the current index `i` and the expression to be indexed `e` and returns the indexed expression and the next index, as outlined below.

```
let rec index i e =
    match e with
        (VAR s) -> (INDEX (i,VAR s),i + 1)|
        (INT i1) -> (INDEX (i,INT i1),i + 1)|
        .......
```

The `cost` function has type `env -> expression -> int` and computes the cost of expression `e` in cost environment `env`, as outlined below.

```
let rec cost env e=
    match e with
        (VAR i) -> (*cost env*) (lookup env i)|
        (INT _) -> INT 0|
        (OP(_,e1,e2)) -> OP(LADD,INT 1,OP(LADD,cost env e1,cost env e2))|
        .......
```

The `continuationCost` has type `env -> expression -> expression -> int` and computes the continuation cost of expression `e1` in expression `e2` in cost environment `env`, as outlined below.

```
let rec continuationCost env e1 e2=
   if e1 = e2
   then INT 0
   else continuationCost' env e1 e2
and continuationCost' env e e'=
   match e' with
      (VAR i) -> INT 0|
      (INT i) -> INT 0|
      (OP(_,e1,e2)) ->
         if contains e e1
         then OP(LADD,continuationCost env e e1,OP(LADD,cost env e2,INT 1))
         else
            if contains e e2
            then OP(LADD,continuationCost env e e2,INT 1)
            else INT 0|
         .......
```

7.3. Generating AMSs

The generator replaces specific higher-order functions with the corresponding CAMS parameterised with the continuation costs previously calculated. For example, if the original program is `map f l`, the object program after the generator is

```
fun dist [] _ = [] |
    dist (h1::t1) (h2::t2) = (h1::h2)::dist t1 t2 |
    dist (h1::t1) [] = [h1]::dist t1 [];
fun transpose [] = [] |
    transpose (row::rows) = dist row (transpose rows);
fun dotprod (h1::t1) (h2::t2) = h1*h2+dotprod t1 t2 |
    dotprod _ _ = 0;
fun rowmult  _ [] = [] |
    rowmult row (col::cols) = dotprod row col::rowmult row cols;
fun rowsmult [] _ = [] |
    rowsmult (row::rows) cols = rowmult row cols::rowsmult rows cols;
fun mmult m1 m2 = rowsmult m1 (transpose m2);
```

Fig. 21. Recursive matrix multiplication.

`camap f l costf continuationCost`, where `costf` is the cost of `f` applied to the first element of `l` calculated using the cost semantics, and `continuationCost` is the cost after the `map` expression in the program, calculated using continuation cost semantics.

As a simple example, let us consider generating CAMS corresponding to expression e, (`map(fun x -> x + 1)[1;2]`); (`map(fun y -> y − 1)[3;4]`). Expression e has two sub-expressions e_1, (`map(fun x -> x + 1)[1;2]`), and e_2, (`map(fun y -> y − 1)[3;4]`). From Section 4, the following four costs are used in the generation.

(1) The cost of the first mapped function (`fun x -> x + 1`); which reduces to (`fun x -> (1 + ((1 + 0) + 0))`)(`hd[1;2]`) after applying Eqs. (4), (3), and (2), and simplifies to 2.
(2) The continuation cost of e_1 in e; which reduces to the continuation cost of e_1 in e_1, which is 0 (Eq. (9)), plus the cost of e_2, plus 1 (Eq. (12a)). The cost of e_2 is ((`fun y -> (1 + ((1 + 0) + 0))`)(`hd[3;4]`)) ∗ (`length[3;4]`) + ((0 + 1) + 1) (Eq. (7)), which simplifies to 6. Hence the total continuation cost of e_1 in e is 7.
(3) The cost of the second mapped function (`fun y -> y − 1`), which reduces and simplifies as above to 2.
(4) The continuation cost of e_2 in$e$, which reduces and simplifies as above to 1.

The generator produces the following CAMS AMP.

```
(camap(fun x -> (x + 1))[1;2]
(((fun x -> (1 + ((1 + 0) + 0)))(hd[1;2])))(* cost of (fun x -> x + 1)*)
((((0 + (((fun y -> (1 + ((1 + 0) + 0)))(hd[3;4])) * (length[3;4]))) + (1 + (0 + 1))) + 1)
                                        (* cont. cost of 1st map *)
);
(camap(fun y -> (y − 1))[3;4]
(((fun y -> (1 + ((1 + 0) + 0)))(hd[3;4])))(* cost of (fun y -> y − 1)*)
 (1)                                     (* cont. cost of 2nd map *)
)
```

The AMP can be simplified to:

```
(camap(fun x -> (x + 1))[1;2](2)(7));
(camap(fun y -> (y − 1))[3;4](2)(1))
```

The generated CAMS AMP is very similar to what might have been written after a manual analysis.

7.4. Automatic analysis example: matrix multiplication

The automatic continuation cost analyser is further illustrated by analysing a $\mathscr{J}$ matrix multiplication program to generate a CAMS AMP.

Given matrices $A[R,N]$ and $B[N,C]$, we wish to form matrix $C[N,N]$ such that $C[i,j] = sumA[i,k] * B[k,j] : 1 <= k <= N$. We are working with a pure functional language without mutable arrays and so we reformulate the problem to represent matrices as lists of lists. Fig. 21 shows a simple recursive matrix multiplication in SML.

The second matrix is `transpose`d to form a list of list of columns. `rowsmult` uses tt rowmult to construct successive rows of the final matrix. `rowmult` then uses `dotprod` to form each element of a final matrix row from the dot product of one row of the first matrix and one transposed column of the second matrix.

In order to enable our automatic cost analysis, we next reformulate the program to expose occurrences of *map* in `rowmult` and `rowsmult` as shown in Fig. 22.

```
fun map _ [] = [] |
    map f (h::t) = f h::map f t;
fun inner row col = (dotprod row) col;
fun rowmult row cols = map (inner row) cols;
fun outer cols x = rowmult x cols
fun rowsmult rows cols = map (outer cols) rows
```

Fig. 22. Matrix multiplication with higher order functions.

```
1  let rec dist = (fun vec1 -> fun vec2 ->
2    match (vec1,vec2) with
3      ([],vec) -> [] |
4      ((h1::t1),(h2::t2)) -> (h1::h2)::(dist t1 t2) |
5      ((h1::t1), []) -> [h1]::(dist t1 []))
6    (* fun mv11-> fun mv22 ->  3*(length mv11)*(length mv22) *)
7    in
8    let rec transpose = (fun e -> fold_right dist e [])
9    (* fun le -> 2*(length le)*(length le) *)
10  in
11   let rec dotprod = (fun mat1 -> fun mat2 ->
12    match (mat1,mat2) with
13      ((h1::t1),(h2::t2)) -> h1*h2+(dotprod t1 t2) |
14      (m1,m2) -> 0 )
15   (* fun l1 -> fun l2 -> (4*(length l2)) *)
16   in
17   let rec rowmult  = (fun cls -> fun row -> map (dotprod row) cls )
18   (* fun rowc -> fun lsc -> 4*(length rowc)*(length rowc) *)
19   in
20   let rec rowsmult = (fun rows -> fun cols ->
21     map ( rowmult cols) rows )
22   in
23   let tm2 = transpose mat2(*0*) in
24   rowsmult mat1(*0*) tm2(*0*)
```

Fig. 23. Annotated matrix multiplication for automatic cost analysis.

Finally, we rewrite the program in a form suitable for input to our analyser as shown in Fig. 23. Firstly, we now use the 𝒥 Jocaml subset rather than the somewhat more elegant SML above. Secondly, as we cannot automatically cost arbitrary recursive functions we attach hand generated cost functions to the `dist`, `transpose`, `dotprod` and `rowmult` functions on lines 6, 9, 15 and 18, respectively. Moreover the pragmas on lines 23 and 24 specify that the cost of constructing the matrices are ignored.

We provide an imputed cost for `rowmult` despite the `map`. We do so because our experimental system cannot yet adequately account for continuation costs in nested higher order functions.

The analyser takes the code in Fig. 23 as input and outputs the code in Fig. 24. Ignoring the reformatting the key change is on lines 18 and 19 where the top level `map` has been converted to the `camap` parameterised with the matrix multiplication cost `(4*(length tm2))*(length tm2)` and a continuation cost of `0`.

7.5. Performance comparison of automatic and hand analyses

Compared with hand-written cost predictions, the costs generated by the automatic continuation cost analyser (e.g. Fig. 24) contain some additional cost calculations. However, performance comparisons between automatic and hand-costed CAMS programs show that the additional cost calculations do not significantly effect performance. The performance of four pairs of AMPs have been compared: double matrix multiplication, invertible matrix, and double ray tracing. Fig. 25 is a typical graph showing almost identical execution times for automatic and for hand-costed CAMS AMPs at all data sizes, in this case for double matrix multiplication. Figs. 26 and 27 show similar patterns for invertible matrix and a double ray tracing programs.

```
1  let rec dist =
2    ((fun vec1 -> (fun vec2 -> match (vec1,vec2) with
3    ([],vec) -> [] |
4    ((h1::t1),(h2::t2)) -> ((h1::h2)::((dist t1) t2)) |
5    ((h1::t1),[]) -> ([h1]::((dist t1) []))));;
6  let rec transpose =
7    ((fun e -> (fold_right dist e [])));;
8  let rec dotprod =
9    ((fun mat1 -> (fun mat2 -> match (mat1,mat2) with
10   ((h1::t1),(h2::t2)) -> ((h1*h2)+((dotprod t1) t2)) |
11   (m1,m2) -> 0)));;
12 let rec rowmult =
13   ((fun cls -> (fun row -> (map (dotprod row) cls))));;
14 let rec rowsmult =
15   (fun rows -> (fun cols -> (map (rowmult cols) rows)));;
16 let tm2 = (transpose mat2)
17 in
18  (camap (rowmult tm2) (mat1)
19    (((4*(length tm2))*(length tm2))) (0) )
```

Fig. 24. Example cost analyser output: matrix multiplication AMP.

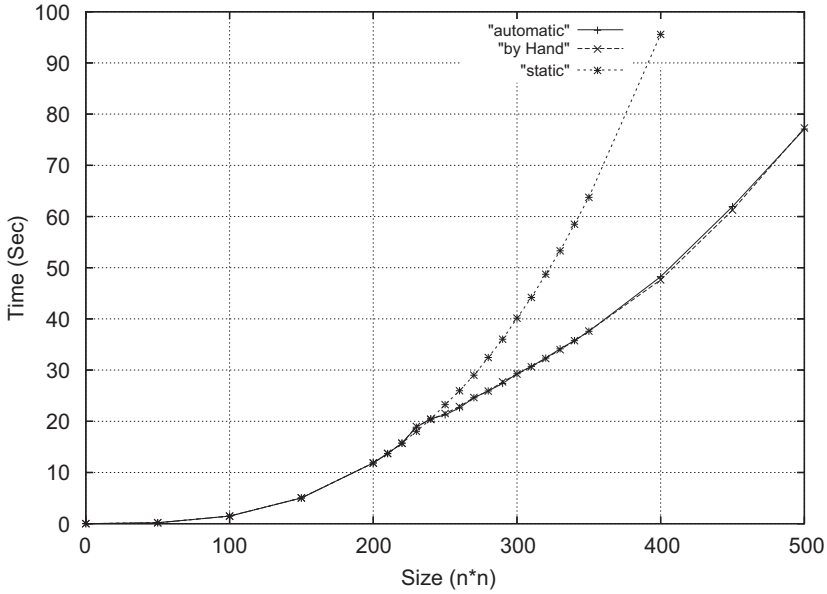

Fig. 25. Comparing automatic and hand-costed execution times (double matrix multiplication).

8. Conclusion and future work

8.1. Summary

A central issue in the burgeoning area of distributed systems is how dynamic collections of programs locate and share resources efficiently. Rather than relying on external load management we have, in earlier work, developed autonomous mobile programs (AMPs) that periodically use a cost model to decide where to execute in a network [1]. The key contribution of this paper is to show how sequential programs can be converted into AMPs in a substantially automatic process that applies a novel continuation cost semantics.

The AMP cost model is generated statically, and is parameterised dynamically to determine movement behaviour. A limitation of the cost model is that the parameterisation assumes that the computation is regular in the sense that the computational cost of the following iterations is similar to those of the preceding iterations. AMPs may dramatically reduce execution time, while guaranteeing to never make it worse by more than a small specified overhead under realistic assumptions. Collections of AMPs perform decentralised load balancing on both homogeneous and heterogeneous networks. Directly programming AMPs makes the cost model, mobility decision function, and network interrogation explicit in the program. To provide a more transparent

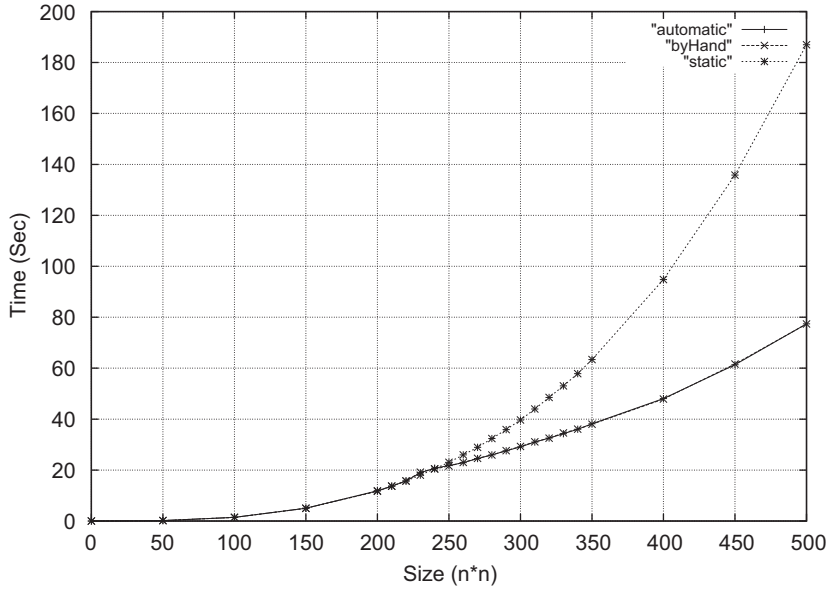

Fig. 26. Comparing automatic and hand-costed execution times (invertible matrix).

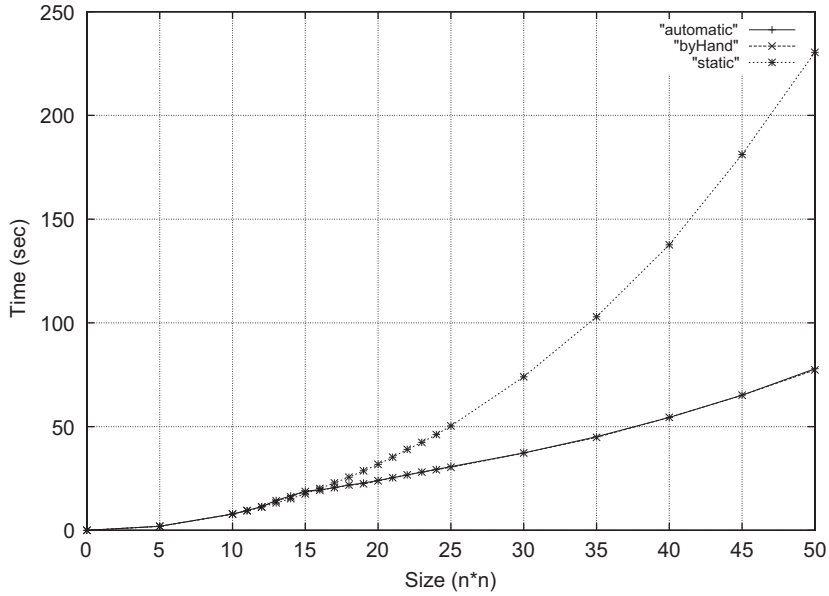

Fig. 27. Comparing automatic and hand-costed execution times (double ray tracing).

interface, we have defined and evaluated *autonomous mobility skeletons* (AMS) that encapsulate autonomous mobility for common collection iterations [2] (Section 3).

AMSs only consider the costs of a single collection iteration. This is adequate only if a single collection iteration dominates the computational cost of the program. To deploy autonomous mobility more generally it is necessary to know, in addition to the cost of the current iteration, the cost of the remainder of the program, or *continuation cost* [3].

We have developed a novel cost semantics that predicts the continuation costs at arbitrary points in a $\mathscr{J}'$ program, i.e. for a core subset of the Jocaml mobile programming language including iterating higher-order functions like `map`. The continuation cost semantics requires that the program is first indexed to distinguish common subexpressions, and that the cost of all expressions have been calculated. The continuation cost equations are generated statically but are designed to be parameterised dynamically to more accurately predict execution time (Section 4).

To produce skeletons capable of making autonomous movement decisions not only the cost of the current iteration, but also the cost of the remainder of the program, the continuation costs are incorporated into a cost model for CAMS. The CAMS cost model is a specialisation of the generic AMP cost model, and the implementation of two CAMS (`camap` and `cafold`) in Jocaml is also outlined. The CAMS cost model inherits the AMP cost model restriction to regular computations, and to only guarantee minimal overheads if location loads remain stable (Section 5).

To demonstrate the utility of the continuation costs we compare the performance of CAMS with both AMS and static versions of six programs. The evaluation shows the following. The cost and continuation cost models are consistent for single-iteration programs. The overheads of collecting and utilising the continuation costs are relatively small as CAMS and AMS programs have very similar performance. For programs dominated by sequences of iterations a CAMS program has a performance advantage as the continuation cost model encourages it to move when an AMS does not, and the more iterations in the program the greater the potential for performance gains, e.g. up to a maximum of 53% for five matrix multiplications. When both the CAMS and AMS programs move, the CAMS programs may be marginally slower, e.g. at most 15% for five matrix multiplications, than the corresponding AMS program due to some additional movement checks and quanta effects on the frequency of movement checks. However, the additional movement checks enable CAMS programs to react to network changes more sensitively than the corresponding AMS programs (Section 6).

We have shown how sequential programs can be automatically converted into AMPs that move to better exploit computational resources on a network. We do so by describing an *automatic continuation cost analyser* that implements the continuation cost semantics to generate cost equations parameterised on program variables in context. The analyser generates both the *cost* of expressions, and the*continuation cost* of iterations, i.e. specific higher-order functions. The analyser translates a $\mathcal{J}$ program into a $\mathcal{J}$ AMP with CAMSs. We show example AMPs generated by the analyser and demonstrate that they have very similar performance to hand-costed CAMS programs (Section 7).

8.2. Discussion

The significance of our work is to demonstrate that, in principle, many programs can be automatically converted to become autonomously mobile, and hence gain substantial performance advantages on networks. The enabling technology is a prediction of the costs and continuation costs of expressions, and specifically the costs of iterations over collections.

It is perhaps surprising that such simple cost models are so effective, even though they are relatively crudely parameterised on environmental changes. However, we are not trying to make accurate worst case execution time (WCET) predictions, where very precise model coefficients accounting for low level time and space characteristics of the underlying hardware are needed. Rather, we wish to know how long the rest of a program will take to execute relative to how long some of it has taken to execute already. Here, a simple model that accounts for the proportionate times taken by program components suffices, under simplifying assumptions about the relative independence of such comparisons from particular implementations. Furthermore movement decisions are based on ratios of predicted costs, for example Eq. (32) predicts a completion time as the ratio between the predicted speeds of two locations, and hence inaccuracies are uniform and smoothed.

Our models differ from others in several important respects. First of all, while we might seek to *simplify* our automatically generated models, we do not seek to *solve* them. This is because our models are for run-time rather than compile time use. Environments change dynamically and so models must be able to capture such change. Thus, our models are *open form*, that is they have free variables which are bound in the context of use to appropriate program variables.

There are substantial limitations of the current work. It is well known that the costs of arbitrary recursive functions cannot be predicted, indeed to do so would solve the halting problem. In our current work we rely on the programmer to provide cost estimates using pragmas for arbitrary recursive functions. We also cannot yet account for continuation costs in nested higher order functions. However, there is further good evidence from the algorithmic skeleton community that even simple nested higher order function cost models are very effective, e.g. [36], and we could incorporate more elaborate models e.g. [37] or solving recurrence relations as in [28].

Another limitation is that the AMS and CAMS cost models assume that the iterations are regular in the sense that the time to compute one element is a good predictor of the time to compute the remaining elements. There are many computations where this is not the case, e.g. Mandlebrot sets. Potentially the cost models could be adapted to cope better with irregularity, e.g. using the computation of several elements as a basis for prediction. Finally the performance of AMPs including direct, AMS and CAMS encodings, have only been demonstrated on relatively small LANs. We hypothesise that they would function well on WANs, and have a design for scalable WAN deployment [41], but have yet to validate it.

8.3. Further work

The current work could be developed in a number of ways.

8.3.1. Resource driven mobility

The cost and continuation cost models presented here model execution time, and AMPs effectively forage in a network for computational resource. We speculate that similar cost models could be developed for other network resources, e.g. bandwidth or repository capacity. Indeed there are well-developed models of foraging behaviours in biology and we propose to investigate the

application of a generic cost-based ethology to autonomous mobile multi-agent systems. Potentially evolved biological foraging strategies enable the better engineering of scalable self-organising resource-location systems in large-scale dynamic networks.

8.3.2. Costed autonomously mobile java programs

Our current continuation cost semantics and CAMS are defined for a subset of Jocaml. Java is more mainstream and widely used than Jocaml, and has several mobile variants, e.g. [13,44]. We would like to build a continuation cost semantics for a substantial Java subset. The immediate challenge would be to integrate continuation costs into our `AutoIter` autonomously mobile iterator interface. In the longer term we would also require to analyse patterns and object-oriented constructs in the presence of inheritance.

Appendix A. Indexing semantic functions

Fig. 28 defines the indexing semantic function: $n \vdash_i : e \Rightarrow (e', n')$. Index takes an expression (e) and an integer representing the current index (n) and returns a tuple comprising an indexed expression (e') and an updated index value (n').

Eqs. (33) and (34) show that indexing a constant or a variable increments the index by one.

Eq. (35) shows that the body of a lambda abstraction is indexed before indexing the abstraction. Eq. (38) is very similar.

Eq. (36) shows that in function applications the functions are indexed first, then the argument, and finally the application. Eqs. (37) and (39) are very similar.

Finally, indexing is idempotent so indexing an indexed expression leaves it unchanged: Eq. (40).

Fig. 29 shows an example of indexing the expression ($\texttt{map}(\texttt{fun x} \rightarrow \texttt{x} + 10)$ [20]). In the figure tree A is the original AST, and tree B is the IAST representing the indexed expression $<8, (\texttt{map} < 4, (\texttt{fun x} \rightarrow < 3, (< 1, \texttt{x} > + < 2, 10 >) >) > < 7, < 5, 20 > :: < 6, [] > >) >$.

$$\frac{}{n \vdash_i k \Rightarrow_i (< n, k >, n+1)} \tag{33}$$

$$\frac{}{n \vdash_i v \Rightarrow_i (< n, v >, n+1)} \tag{34}$$

$$\frac{n \vdash_i e \Rightarrow_i (e', n')}{n \vdash_i \texttt{fun}\ v \rightarrow e \Rightarrow_i (< n', \texttt{fun}\ v \rightarrow e' >, n'+1)} \tag{35}$$

$$\frac{n \vdash_i e_1 \Rightarrow_i (e_1', n') \qquad n' \vdash_i e_2 \Rightarrow_i (e_2', n'')}{n \vdash_i (e_1\ e_2) \Rightarrow_i (< n'', (e_1'\ e_2') >, n''+1)} \tag{36}$$

$$\frac{n \vdash_i e_1 \Rightarrow_i (e_1', n') \qquad n' \vdash_i e_2 \Rightarrow_i (e_2', n'')}{n \vdash_i e_1\ op\ e_2 \Rightarrow_i (< n'', e_1'\ op\ e_2' >, n''+1)} \tag{37}$$

$$\frac{n \vdash_i e \Rightarrow_i (e', n')}{n \vdash_i e\ (*\ c\ *) \Rightarrow_i (< n', e'\ (*\ c\ *) >, n'+1)} \tag{38}$$

$$\frac{n \vdash_i e_1 \Rightarrow_i (e_1', n') \qquad n' \vdash_i e_2 \Rightarrow_i (e_2', n'')}{n \vdash_i \texttt{map}\ e_1\ e_2 \Rightarrow_i (< n'', \texttt{map}\ e_1'\ e_2' >, n''+1)} \tag{39}$$

$$\frac{}{n \vdash_i < n', e > \Rightarrow_i (< n', e >, n)} \tag{40}$$

Fig. 28. Index semantics for $\mathscr{J}'$.

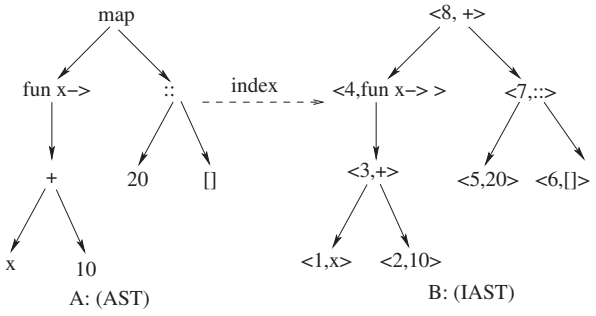

Fig. 29. $\mathscr{J}'$ indexing example.

$$\frac{e_1 \equiv e_2}{e_1 \in e_2} \quad (41)$$

$$\frac{e_1 \in e_2}{e_1 \in \texttt{fun}\ v\ \rightarrow e_2} \quad (42)$$

$$\frac{e_1 \in e_2}{e_1 \in (e_2\ e_3)} \quad (43a)$$

$$\frac{e_1 \in e_3}{e_1 \in (e_2\ e_3)} \quad (43b)$$

$$\frac{e_1 \in e_2}{e_1 \in e_2\ op\ e_3} \quad (44a)$$

$$\frac{e_1 \in e_3}{e_1 \in e_2\ op\ e_3} \quad (44b)$$

$$\frac{e_1 \in e_2}{e_1 \in e_2\ (*\ c\ *)} \quad (45)$$

$$\frac{e_1 \in e_2}{e_1 \in \texttt{map}\ e_2\ e_3} \quad (46a)$$

$$\frac{e_1 \in e_3}{e_1 \in \texttt{map}\ e_2\ e_3} \quad (46b)$$

$$\frac{e_1 \in e_2}{e_1 \in < i, e_2 >} \quad (47)$$

Fig. 30. $\mathscr{J}'$ syntactic containment.

$$\begin{array}{lll} & 15 = 15 & \\ & 15 \equiv 15 & (\text{Equiv}) \\ \Rightarrow & 15 \in 15 & (41) \\ \Rightarrow & 15 \in 5 :: map\ (fun\ x \rightarrow (x+10))\ (20 :: [\ ]) & (44a) \\ \Rightarrow & 15 \in 8 :: (5 :: map\ (fun\ x \rightarrow (x+10))\ (20 :: [\ ])) & (44b) \end{array}$$

Fig. 31. $\mathscr{J}'$ contains example.

Appendix B. Auxiliary semantic functions

This appendix presents the relatively standard syntactic containment function, used in the continuation cost semantics in Section 4.3. The continuation cost semantics also uses a standard syntactic, or structural, equality without alpha conversion which is defined in [41] but not reproduced here.

Fig. 30 shows the definition of syntactic containment, $\in$ takes two expressions and returns true if the second expression contains the first expression.

Eq. (41) specifies that if two expressions are syntactically equal then the first contains the other.

Eq. (42) specifies that an expression is contained in a lambda abstraction if it is contained in the body.

Eqs. (43a) and (43b) specify that an expression is contained in a functional application if it is contained in either the function or the argument.

The remaining equations follow the pattern of the previous equations. As an example Fig. 31 shows the deductions to determine whether expression from Fig. 9, i.e. `8 :: 15 :: (map(fun x -> x + 10) [20])`, contains the expression `15`.

References

[1] Deng XY, Trinder P, Michaelson G. Autonomous mobile programs. In: IAT '06: Proceedings of the IEEE/WIC/ACM international conference on intelligent agent technology (IAT 2006 main conference proceedings) (IAT'06). Washington, DC, USA, Hong Kong: IEEE Computer Society; 2006. p. 177–86 ⟨http://dx.doi.org/10.1109/IAT.2006.42⟩.

[2] Deng XY, Michaelson G, Trinder P. Autonomous mobility skeletons. Journal of Parallel Computing 2006;32(7–8):463–78 (Algorithmic Skeletons. ⟨http://dx.doi.org/10.1016/j.parco.2006.04.002⟩).

[3] Deng XY, Trinder P, Michaelson G. Automatically costed autonomous mobility. In: IAT '07: proceedings of the IEEE/WIC/ACM international conference on intelligent agent technology (IAT 2007 main conference proceedings). Washington, DC, USA, Silicon Valley: IEEE Computer Society; 2007. p. 95–101.

[4] Milojicić D, Douglis F, Weeler R. Mobility: processes, computers, and agents. Reading, MA, USA: Addison-Wesley; 1999.

[5] Baron R, Rashid R, Siegel E, Tevanian A, Young M. Mach-1: an operating environment for large-scale multiprocessor applications. IEEE Software 1985;2(4): 65–7.

[6] Barak A, La'adan O. The MOSIX multicomputer operating system for high-performance cluster computing. Future Generation Computer Systems 1998;13(4–5):361–72.

[7] Kale LV, Krishnan S. Charm++: a portable concurrent object oriented system based on c. In: Proceedings of the conference on object oriented programming systems, languages and applications. New York: ACM Press; 1993. p. 91–108.

[8] Yu J, Buyya R. A taxonomy of workflow management systems for grid computing. Journal of Grid Computing 2005;3:171–200.
[9] Herrera J, Huedo E, Montero R, Llorente I. Loosely-coupled loop scheduling in computational grids. In: 20th international parallel and distributed processing symposium, 2006. 6pp, doi:10.1109/IPDPS.2006.1639657.
[10] Cole M. Algorithmic skeletons: structured management of parallel computation. Cambridge: MIT Press; 1989.
[11] Kirli Z. Mobile computation with functions. PhD Thesis, University of Edinburgh, Laboratory for Foundations of Computer Science, Division of Informatics; 2001.
[12] Institut National de Recherche en Informatique et en Automatique, The JoCaml language beta release: Documentation and user's manual; January 2001.
[13] Recursion Software, Inc, 2591 North Dallas Parkway, Suite 200, Frisco, TX 75034, Voyager User Guide ⟨http://www.recursionsw.com/Voyager/Voyager_User_Guide.pdf⟩, May 2005.
[14] Lange DB, Oshima M. Seven good reasons for mobile agents. Communications of the ACM 1999;42(3):88–9 ⟨http://doi.acm.org/10.1145/295685.298136⟩.
[15] Fuggetta A, Picco GP, Vigna G. Understanding code mobility. IEEE Transactions on Software Engineering 1998;24(5):342–61 ⟨citeseer.ist.psu.edu/fuggetta98understanding.html⟩.
[16] Wooldridge M. Agent-based software engineering. IEE Proceedings Software Engineering 1997;144(1):26–37 ⟨citeseer.ist.psu.edu/wooldridge94agentbased.html⟩.
[17] Tosic PT, Agha GA. Towards a hierarchical taxonomy of autonomous agents. In: IEEE SMC'2004: international conference on systems, man and cybernetics. Hague, The Netherlands: IEEE Xplore; 2004. p. 3421–6.
[18] Milojicic D, Douglis F, Wheeler R. Mobility: processes, computers, and agents. New York, NY, USA: ACM Press, Addison-Wesley Publishing Co.; 1999.
[19] Kephart JO, Chess DM. The vision of autonomic computing. Computer 2003;36(1):41–50 ⟨http://dx.doi.org/10.1109/MC.2003.1160055⟩.
[20] Murch R. Autonomic computing. 1st ed., IBM Press; 2004.
[21] Abawajy J. Autonomic job scheduling policy for grid computing. In: Lecture notes in computer science, vol. 3516, international conference on computational science—ICCS 2005, part 3. Germany: Springer; 2005. p. 213–20.
[22] Travis Desell CV, El Maghraoui K. Load balancing of autonomous actors over dynamic networks, 2004. p. 90268.1.
[23] Reistad B, Gifford DK. Static dependent costs for estimating execution time. In: LFP '94: proceedings of the 1994 ACM conference on LISP and functional programming. New York, NY, USA, Orlando, Florida, USA: ACM Press; 1994. p. 65–78 ⟨http://doi.acm.org/10.1145/182409.182439⟩.
[24] Cohen J, Zuckerman C. Two languages for estimating program efficiency. Communications of the ACM 1974;17(6):301–8 ⟨http://doi.acm.org/10.1145/355616.361015⟩.
[25] Wegbreit B. Mechanical program analysis. Communications of the ACM 1975;18(9):528–39 ⟨http://doi.acm.org/10.1145/361002.361016⟩.
[26] Ramshaw LH. Formalizing the analysis of algorithms. PhD Thesis, Department of Computer Science, Stanford University; 1979.
[27] Wegbreit B. Verifying program performance. Journal of the ACM 1976;23(4):691–9 ⟨http://doi.acm.org/10.1145/321978.321987⟩.
[28] Rosendahl M. Automatic complexity analysis. In: FPCA '89: proceedings of the fourth international conference on functional programming languages and computer architecture. New York, NY, USA, Imperial College, London, UK: ACM Press; 1989. p. 144–56 ⟨http://doi.acm.org/10.1145/99370.99381⟩.
[29] Wadler P. Strictness analysis aids time analysis. In: POPL '88: proceedings of the 15th ACM SIGPLAN-SIGACT symposium on principles of programming languages. New York, USA, San Diego, California, United States: ACM Press; 1988. p. 119–32 ⟨http://doi.acm.org/10.1145/73560.73571⟩.
[30] Hughes J, Pareto L. Recursion and dynamic data-structures in bounded space: towards embedded ML programming. In: International conference on functional programming. 1999. p. 70–81.
[31] Portillo AJR, Hammond K, Loidl H-W, Vasconcelos PB. Cost analysis using automatic size and time inference. In: Pena R, Arts T, editors. Implementation of functional languages, 14th international workshop, IFL 2002, Madrid, Spain, September 16–18, 2002. Lecture notes in computer science, 2670. Berlin: Springer; 2002. p. 232–48 [Revised selected papers].
[32] Hofmann M, Jost S. Static prediction of heap space usage for first-order functional programs. Proceedings of the 30th ACM symposium on principles of programming languages, vol. 38. New York: ACM Press; 2003. p. 185–97.
[33] Hofmann M, Jost S. Type-based amortised heap-space analysis. In: ESOP 2006. Lecture notes in computer science, vol. 3924. Berlin: Springer; 2006. p. 22–37.
[34] Herrmann CA, Bonenfant A, Hammond K, Jost S, Loidl H-W, Pointon R. Automatic amortised worst-case execution time analysis. In: Rochange C, editor. 7th international workshop on worst-case execution time (WCET) analysis, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI). Germany: Schloss Dagstuhl; 2007.
[35] Brady E, Hammond K. A dependently typed framework for static analysis of program execution costs. In: Butterfield A, Grelck C, Huch F, editors. Implementation and application of functional languages, 17th international workshop, IFL 2005, Dublin, Ireland, September 19–21, 2005. Lecture notes in computer science, vol. 4015. Springer; 2006. p. 74–90 [Revised Selected Papers].
[36] Skillicorn DB. Parallelism and the Bird–Meertens Formalism. Queen's University, Kingston, Ontario: Department of Computing and Information Science; 1992 URL ⟨citeseer.ist.psu.edu/skillicorn92parallelism.html⟩.
[37] Rangaswami R. A cost analysis for a higher-order parallel programming model. PhD thesis, Department of Computer Science, Edinburgh University; 1996 URL ⟨citeseer.ist.psu.edu/rangaswami96cost.html⟩.
[38] Loidl H-W. Granularity in large-scale parallel functional programming, PhD thesis, University of Glasgow, Department of Computing Science; April 1998.
[39] Sahni S. Data structures, algorithms, and applications in java. University of Florida: Mc-Graw Hill; 2000.
[40] Strachey C, Wadsworth CP. Continuations: a mathematical semantics for handling full jumps. Technical Report, Oxford University Computing Laboratory (reprinted in Higher Order and Symbolic Computation, January 1974; 13(1/2):135–52).
[41] Deng XY. Cost-driven autonomous mobility. PhD thesis, School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, UK; June 2007.
[42] Friedman DP, Wand M, Haynes CT. Essentials of programming languages. Cambridge: MIT Press; 1992.
[43] Michaelson G, Scaife N, Bristow P, King P. Nested algorithmic skeletons from higher order functions. Parallel Algorithms and Applications: Special Issue on High Level Models and Languages for Parallel Processing 2001;16:181–206.
[44] Sekiguchi T. JavaGo ⟨http://homepage.mac.com/t.sekiguchi/javago/index.html⟩ (May 2006).