Cost-Driven Autonomous Mobility

Xiao Yan Deng and Greg Michaelson and Phil Trinder
School of Mathematical and Computer Sciences
Heriot-Watt University

May 30, 2007

Abstract

Developments in distributed system technology facilitate the sharing of computational resources, even at a global level. To share resource in open distributed systems we have developed *autonomous mobile programs*, which are aware of their resource needs and sensitive to the environment in which they execute. Autonomous mobile programs periodically use a cost model to decide where to execute in a network. Unusually this form of autonomous mobility affects only where the program executes and not what it does.

In an autonomous mobile program, the program must contain explicitly control of self-aware mobile coordination. To encapsulate self-aware mobile coordination for common patterns of computation over collections, *autonomous mobility skeletons* are built. Autonomous mobility skeletons are akin to algorithmic skeletons in being polymorphic higher-order functions, but where algorithmic skeletons abstract over parallel coordination, autonomous mobility skeletons abstract over autonomous mobile coordination.

An autonomous mobility skeleton considers only the cost of the current collective computation. It is more useful to know the cost of the entire program. A cost calculus to calculate the costs for the rest of a computation at arbitrary points has been built. We extend our autonomous mobility skeleton cost models to be parameterised on the cost of the remainder of the program, and build *costed autonomous mobility skeletons* based on the extended cost models. A automatic cost analyser which implements the calculus has also been built in Jocaml. The analyser generalises cost equations parameterised on program variables in context, and outputs Jocaml programs with higher-order functions replaced by costed autonomous mobility skeletons. Performance improvements are assessed by comparing costed autonomous mobility skeleton to autonomous mobility skeleton programs and show that the former often execute faster than the later, because they have more accurate cost information.

Contents

Chapter 1

Introduction

Developments in networks have made it possible to exploit the computational power and resources available in global networks [2]. Load management systems have been built for resources sharing using mobile computation, agents, and autonomic techniques [7].

One way of using the resources on both local and global networks is through mobile computations especially using *mobile languages* e.g. Jocaml [10] and Java Voyager [30] where executing computations can move in the network in order to better use the resources available. Mobile computation with mobile languages give programmers control over the placement of code or active computations across a network for sharing computational resources [12]. Basically a mobile program can transport its state and code to another location in the network, where it resumes execution [13].

The agents community has focused on autonomous problem solving e.g. building self-managing, or autonomous, systems which can act flexibly in uncertain and dynamic environments. Mobile languages potentially allow agents to move more flexibly in a large scale network. That is, a mobile agent can migrate across a network to locate the resources it requires.

In distributed resource sharing systems, the task of placing work on location is termed load management, and is a branch of global scheduling [3]. In load management systems, programs are transported from heavily loaded locations to lightly loaded locations to use the computational power efficiently. Using mobile computation, more flexible and efficient applications can be developed where a program can move between locations to better utilise computational resources.

One of the most important issues in load management systems, is to identify effective techniques for the distribution of the work over the locations, to achieve performance goals such as balancing the load of each location, or minimising execution time. One technique is to use *cost models* to estimate the execution time of a program [20], and the accuracy of the cost model is crucial.

We have investigates load management which is driven by cost models in collections of autonomous mobile computations. Programs in this system can

in principle control when and where to move by consulting cost models, thus if a program determines that the current location is busy it will find a faster processor and migrate there. To build cost-driven autonomous load management system, two approaches have been exbited:

1. To manage load on large and dynamic networks we have developed *autonomous mobile programs* (AMPs)[6] that periodically make a decision about where to execute in a network. The decisions are informed by cost models that measure current performance, the relative speeds of alternative network locations, and communication costs.

2. A disadvantage of directly programming AMPs is that the cost model, mobility decision function, and network interrogation are all explicit in the program. To encapsulate self-aware mobile control for common patterns of computation over collections, we have explored *autonomous mobility skeleton* (AMSs) in [5]. AMSs are akin to algorithmic skeletons in being polymorphic higher-order functions, but where algorithmic skeletons abstract over parallel coordination, AMSs abstract over autonomous mobile coordination. AMSs make mobility decisions by combining generic and task specific cost models. We have presented the `automap`, `autofold` and `AutoIterator` AMSs, together with performance measurements of Jocaml, Java Voyager, and JavaGo implementations on modest LANs.

A limitation of AMS is that they assume that a single higher-order function is the dominating computation for the program. For example, if there are multiple higher-order functions in the program, e.g. in program `(automap f1 l1); (automap f2 l2)`, `automap f2 l2` is the remainder of `automap f1 l1` in the program. When `automap f1 l1` make the decision to move or not, it might not move if it only considers the cost of itself, but it might move if it also considers the cost of `automap f2 l1`. In general, to deploy autonomous mobility effectively, it is necessary to know the cost of the remainder of the program, not just of a single iteration.

Thus, this paper explores a calculus to manipulate, and ultimately automatically statically extract costs for the remainder of a computation (costafter), at arbitrary points at compile time. The advantage of the cost calculus is that it is not necessary to provide a closed form solution as environmental information for a computation is always available implicitly at run-time. Thus, branches are not necessarily a source of loss of accuracy as concrete data values are available at the point where the cost is calculated.

An automatic cost analyser, which implements the cost calculus, is built to produce cost equations parameterised on program variables in context, and may be used to find both cost in higher order functions and the costafter of the higher order functions. As *costafter* is the cost of the continuations, the analyser is also called a *automatic continuation cost analyser*.

We extends our AMS cost model to be parameterised on the cost of the remainder of the program. Costed autonomous mobility skeletons (CAMSs) are built based on the extended cost model. The CAMSs not only encapsulate the

common patten of autonomous mobility but take additional cost parameters representing the costs of the remainder of the program. We have constructed an automatic cost analyser which implements the cost calculus for a Jocaml subset, produces cost equations parameterised on program variables in context, and may be used to find both cost in higher-order functions and the cost for the remainder of the program.

The structure of the paper is as follows. Chapter 2 discusses related work. Chapter 3 represents AMPs and AMSs which are published in [6] and [5]. Chapter 4 defines a small strict higher-order language $\mathcal{J}'$ (Section 4.1) and builds the cost calculus for $\mathcal{J}'$, which include *indexing* the program (Section 4.2), *calculating the costs* of every expressions (Section 4.3), and *calculating the costafter* of the given expression (Section 4.4). Chapter 5 builds the cost model for costed autonomous mobility skeletons, and implements costed autonomous mobility skeletons. Chapter 6 compares the performance of the programs using costed autonomous mobility skeletons with these using autonomous mobility skeletons. Chapter 7 describes the implementation of an automatic continuation cost analyser which implements the cost calculus. This chapter also compares the performances of costed autonomous mobility skeletons produced from the analyser directly, with those are built by hand in Chapter 6. Chapter 8 summaries.

Note in this paper higher-order functions are autonomous mobility skeletons e.g. `automap`, and `autofold` or costed autonomous mobility skeletons e.g. `camap` or `cafold`, according to the context.

Chapter 2

Related Work

2.1 Mobility

Mobile computation especially mobile languages e.g. Jocaml [10], Java Voyager [19], and JavaGo [23] give programmers control over the placement of code or active computations across the network for sharing computational resources e.g. CPU speed [12]. Basically a mobile program can transport its state and code to another location in a network, where it resumes execution [13].

Fuggetta et. al. distinguish two forms of mobility supported by mobile languages [9]. *Weak mobility* is the ability to move only code from one machine to another. *Strong mobility* is the ability to move both code and its current execution state[5].

AMPs[6] and AMSs[5] have been developed in Jocaml, Java Voyager, and JavaGo. Jocaml are functional programming language which supports strong mobility. Java Voyager and JavaGo extend Java: Voyager supports weak mobility, while JavaGo supports strong mobility.

2.2 Agents & 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" [31, 25]. Mobility is the ability to transport itself from one machine to another and retaining its current state. Agents with mobility is called *mobile agents* [15]. 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" [11, 16]. The essence of autonomic computing systems is self-management, the intent of which is to free system administrators from the details of system operation 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-optimization*, *Self-healing*, and *Self-protection*. Different autonomic systems may have some or all these four aspects. AMPs are self-optimization 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 processing elements.

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 processing elements they share. The agents community has focused on autonomous problem solving, which can act flexibly in uncertain and dynamic environments. Mobile languages provides 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 [31].

AMPs have strong connections with both Agents and Autonomous Systems, 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. AMPs also differ from tradition autonomous systems [11, 1, 26], which use schedulers to decide whether to move, AMPs themselves can make the decision when and where to move according to the cost model [6].

2.3 Cost Analysis

In the systems for sharing computational power, *cost models* are used to estimate the cost of a program in terms of time and to predict the behaviour of the program[20]. There are two levels of cost models in general. *Computation cost models* estimate the sequential computation time for programs. *Coordination cost models* predict the coordination and communication behaviours of parallel, distributed and mobile programing. Usually, coordination cost models take costs which have been got from the computation cost models into account to make more efficient coordination decisions.

Coordination cost models are developed for parallel programs. Algorithmic skeletons encapsulate the expression of parallelism, communication, synchronisation and embedding, and having an associated cost complexity. The skeleton's cost complexity is calculated according to the particular cost model e.g.

Rangaswami has developed a parallel programming model called **HOPP** for skeleton oriented programming in [18], and Skillicorn and Cai have developed a cost calculus for the Bird Meertens Formalism (BMF) in [24] etc.. A generic AMP coordination cost model and problem specified cost models for matrix multiplication and AMSs have been built in [6, 5].

This paper focuses on the *computation cost models* for AMPs. This kind of cost analysis usually happens at compile time, so is also called static cost analysis. Different static cost models have been built for different systems. Early work on these problems includes that of Cohen and Zuckerman, who consider cost analysis of Algol-60 programs [4]; Wegbreit, whose pioneering work on cost analysis of Lisp programs addressed the treatment of recursion [28]; and Ramshaw [17] and Wegbreit [29], who discuss the formal verification of cost specifications. Many of the cost analysis use semantics-based methods e.g Rosendahl [21] uses abstract interpretation for cost analysis, and Wadler [27] uses projection analysis. In [14], Loidl has built static cost semantics for for language $\mathcal{L}$, and in [20] Reistad and Gifford built static cost for data-dependent expressions.

We built a static computation cost semantics for the Jocaml subset $\mathcal{J}'$. It calculates the continuation cost of each expression in the program. The cost semantics is based on these cost semantics in [14] and simplifies the work. The cost semantics for $\mathcal{J}'$ does not consider the type system and size system of the language.

Chapter 3

Previous Work

3.1 Autonomous Mobile Programs

To manage load on large and dynamic networks we have developed what we term *autonomous mobile programs* (AMPs) in [6], which are aware of their processing resource needs and sensitive to the environment in which they execute. 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 processing time in the presence of varying external loads on shared processing elements. 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 cost of movement may be kept to a very small proportion of the overall execution time.

3.2 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 explores *autonomous mobility skeletons* (AMS) in [5], that encapsulate mobility control for common patterns of computation over collections. AMSs are polymorphic higher order functions, such as `automap` or `autofold` 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 [22].

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 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. `automap` and `autofold` AMSs are built in both functional language e.g. Jocaml and Object-Oriented language e.g. Java Voyager.

An iterator is a class that implements the Java `Iterator` interface, which specifies a generic mechanism to enumerate the elements of a collection. The methods in the `Iterator` interface are `hasNext, next` and `remove`[22]. The `AutoIterator` class implements all three methods, and extends it with `autonext`, which has the same functionality as `next` but can make autonomous mobility decisions.

`automap` and `autofold` can use weak mobility as the computation to be moved is encapsulated in the function mapped or folded. In `AutoIterator`, however, there is no encapsulated computation to be weakly moved and strong mobility is required to move the entire collection. Hence Voyager, with only weak mobility, cannot be used. JavaGo [23] supports strong mobility. `automap` and `autofold` can also be built using strong mobility such as Jocaml.

3.3 AMP Performance

Figure 3.1 compares the execution times of static and mobile matrix multiplication programs. Our test environment is based on three locations with CPU speed 534MHZ, 933MHZ and 1894MHZ. The loads on these three computers are almost zero. We started both the static and the mobile programs on the first CPU. We can see that the bigger the size of the matrix the faster the mobile version is compared with the static version.

3.4 Single AMP Movements

Figure 3.2 show the movement of the AMP matrix multiplication during successive execution time periods with CPU speeds normalised by the local loads. We start the AMP in time period 0 on $Loc1$. In time period 1 it moved to the fastest processor $Loc3$. When $Loc3$ became more heavily loaded the program moved to $Loc5$, the fastest processor in period 2. In time period 3, $Loc4$ became less loaded, and was the fastest location at that moment, so the program moved to it. In time period 7 $Loc5$ was a little faster than $Loc2$. So the program moved to $Loc5$ rather than staying on $Loc2$.

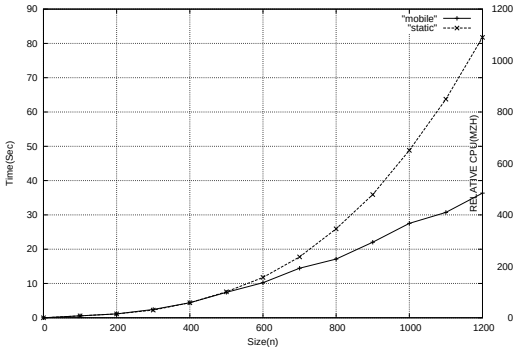

Figure 3.1: AMP and Static Matrix Multiplication Execution Time

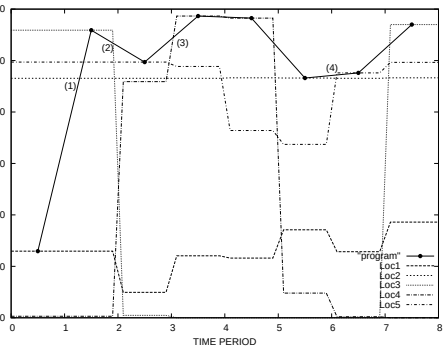

Figure 3.2: Single AMP Matrix Multiplication Movement

We draw the following conclusions from Figure 3.2:

- The program may move repeatedly to adapt to changing loads and always find the fastest location in one 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.

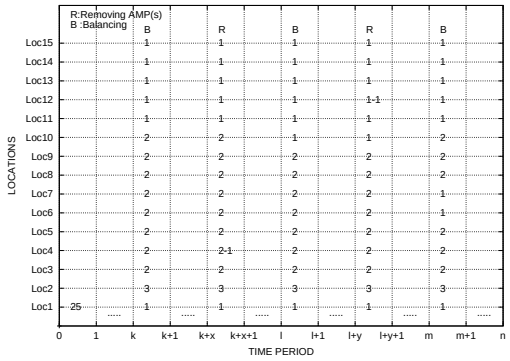

Figure 3.3: 25 AMPs on Heterogeneous Network (15 Locations)

3.5 Collections of AMPs

Experiments on a homogeneous and heterogeneous network show that collections of AMPs quickly obtain and maintain balance.

We have measured the behaviour of multiple AMPs on a heterogeneous network of fifteen locations, with CPU speeds 3193MHz (Loc1-Loc5), 2168MHZ (Loc6-Loc10), and 1793MHz (Loc11-Loc15). For illustration the movement of 25 AMPs between the 15 locations is shown in Figure 3.3. "B" is the *balanced status*, where every AMP has similar *relative CPU speed*. In this state, AMPs will stay in their current locations and not move any more until the balance is broken.

Note that the results in this chapter are published in [6] and [5].

Chapter 4

Cost Calculus for $\mathcal{J}'$

Autonomous mobility skeletons only consider the cost in higher-order functions. To deploy autonomous mobility effectively, it is necessary to know the cost of the remainder of the program (costafter), not just of a single iteration. To calculate the costafter of an arbitrary point in a program, the cost of every expression must be calculated. To illustrate the concept this chapter gives a small language, $e ::= n \mid e{+}e$, where n is integer. Using expression `2+3` as an example, the costafter of `2` can be calculated as $\frac{\mathrm{E} \vdash_c 3 \,\$\, c_3 \qquad \mathrm{E} \vdash_c + \,\$\, c_+}{\mathrm{E} \vdash_a 2 \trianglelefteq (2{+}3) \,\pounds\, c_3{+}c_+}$, where c_3 is the cost of `3`, c_+ is the cost of "`+`", and the costafter of `2` is $c_3 + c_+$. The semantic functions $\vdash_c$ and $\vdash_a$ produce the cost and costafter of expressions in the environment E, and will be introduced in Section 4.

The problem with this calculation is if there are two similar expression or one expression in two or more place in the program, it is difficult to identify the expression whose costafter needs to be calculated. For example, in expression `10+10`, there are two `10`s. To calculate the costafter of `10` in `10+10`, but it is difficult to decide which `10` is required, and the costafters of these two `10`s are different. To solve this problem, every expression in a program is given a unique number, which is called its *index*. The process of giving these indices is called indexing. After indexing, the expression `10+10` becomes $< 3, < 1, 10 > + < 2, 10 >>$. The two `10`s become $< 1, 10 >$ and $< 2, 10 >$.

The general three stages to calculate the costafter are:

- Index the program.
- Calculate the cost of expressions in the program.
- Calculate the costafter of the point to be required.

Calculating the cost of expression is standard and use techniques similar to e.g. cost models in [14, 21, 27]. The third stage, to calculate the costafter, which is to predict a continuation cost in a program, is novel.

The cost calculus includes three parts. The first part converts the abstract syntax tree (AST) to *indexed abstract syntax tree* (IAST), i.e. gives every expression a unique integer as an index. The second part calculates the cost of

each expression (Section 4.3). The third part calculates the costafter of each expression (Section 4.4).

$\vdash_i : n \rightarrow e \rightarrow e * n$	index
$\vdash_c : env \rightarrow e \rightarrow e * cost$	expression cost
$\equiv : e \rightarrow e$	expression equality
$\in : e \rightarrow e \rightarrow boolean$	contains
$\vdash_a : env \rightarrow e \rightarrow e \rightarrow cost$	costafter

Figure 4.1: Semantic Functions

Figure 4.1 shows the semantic functions which will be used in the calculus to present the semantics of index, cost, and costafter, etc. The meaning of these functions will be given in the sections where the functions are used. In these semantic functions, *env* is a semantic domain, which is the environment used in the calculus, and is presented as E. The type of *env* is, $env : (v * cost)^*$. *Cost* is integer.

4.1 Syntax of Language $\mathcal{J}'$

A cost semantics has been built for language $\mathcal{J}$, which is a subset of Jocaml. $\mathcal{J}$ is a core functional language and readily able to describe non-trivial programs like matrix multiplication and ray tracing. To explain the principles, this section introduces a simple language $\mathcal{J}'$, a subset of $\mathcal{J}$, and Section 4 uses $\mathcal{J}'$ to introduce the cost calculus. The full syntax and cost calculus of $\mathcal{J}$ is presented in Appendix A.1. Figure 4.2 shows the abstract syntax of $\mathcal{J}'$. To simplify the

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
|	$< n, e >$	index
op ::=		operator
	+ | - | * | /	arithmatical
|	> | < | >= | <= | = | !=	logical
|	::	cons
|	;	sequential composition

Figure 4.2: Syntax of $\mathcal{J}'$

$$\frac{}{i \vdash_i c \Rightarrow_i (< i, k >, i+1)} \tag{4.1}$$

$$\frac{}{i \vdash_i v \Rightarrow_i (< i, x >, i+1)} \tag{4.2}$$

$$\frac{i \vdash_i e \Rightarrow_i (e', i')}{i \vdash_i \texttt{fun}\ x\ \rightarrow\ e \Rightarrow_i (< i', \texttt{fun}\ x\ \rightarrow\ e' >, i'+1)} \tag{4.3}$$

$$\frac{i \vdash_i e_1 \Rightarrow_i (e_1', i') \qquad i' \vdash_i e_2 \Rightarrow_i (e_2', i'')}{i \vdash_i (e_1\ e_2) \Rightarrow_i (< i'', (e_1'\ e_2') >, i''+1)} \tag{4.4}$$

$$\frac{i \vdash_i e_1 \Rightarrow_i (e_1', i') \qquad i' \vdash_i e_2 \Rightarrow_i (e_2', i'')}{i \vdash_i e_1\ op\ e_2 \Rightarrow_i (< i'', e_1'\ op\ e_2' >, i''+1)} \tag{4.5}$$

$$\frac{i \vdash_i e \Rightarrow_i (e', i')}{i \vdash_i e\ (*\ c\ *) \Rightarrow_i (< i', e'\ (*\ c\ *) >, i'+1)} \tag{4.6}$$

$$\frac{i \vdash_i e_1 \Rightarrow_i (e_1', i') \qquad i' \vdash_i e_2 \Rightarrow_i (e_2', i'')}{i \vdash_i \texttt{map}\ e_1\ e_2 \Rightarrow_i (< i'', \texttt{map}\ e_1'\ e_2' >, i''+1)} \tag{4.7}$$

$$\frac{}{i \vdash_i < i', e > \Rightarrow_i (< i', e >, i)} \tag{4.8}$$

Figure 4.3: Index Semantics for $\mathcal{J}'$

presentation it is assumed that variable (v) names in the program are unique. Note that $\mathcal{J}$ also contains `fold` higher-order function.

$\mathcal{J}'$ is a core lambda calculus with two unusual expression, the *index* expression and *user cost* expression. The *index* expression is presented because the whole program has been indexed. Costing recursive functions is undecidable. Thus to deal with the cost of recursive functions, *user costs* are introduced into $\mathcal{J}'$, which will be explained in Section 4.3.

4.2 Index Semantics for Language $\mathcal{J}'$

Figure 4.3 shows the semantics of index. $\vdash_i : n \rightarrow e \rightarrow e * n$, is a semantic function, which takes two parameters, one is an expression and the other is an integer(i), which is the current index and i is the index of the expression, and returns a tuple of index expression and another integer which is i increased by 1. In this semantic function n is integer, e is expression.

- Equation (4.1) shows if the current index is i the expression is a constant k, after indexing the new expression is $< i, k >$ and the current index becomes $i + 1$.
- Equation (4.2) is similar to Equation (4.1), but the expression is a variable (v) instead of constant.
- Equation (4.3) decorates lambda expressions. In this equation the dummy does not need to be decorated, only the body of the lambda expression

does. For indexing `fun` $x \rightarrow e$, if the current index is i, and after indexing e the current index is i', and e becomes e', the indexed lambda expression is $< i', \texttt{fun}\ x \rightarrow e' >$ and the current index is $i' + 1$.

- Equation (4.4) decorates application expressions ($e_1\ e_2$). If the current index is i, and after indexing e_1 the current index becomes i' and e_1 becomes e_1', and then after indexing e_2 the current index is i'' and e_2 becomes e_2', then the index for the application expression is i'' and the current index becomes $i'' + 1$.
- Equation (4.5) decorates operation expressions using the same rules as Equation (4.4).
- In Equation (4.6), user cost expressions are indexed. In this expression only e, the expression part, has been indexed but not the cost part $(* \ c \ *)$.
- In Equation (4.7), `map` expressions are indexed, which is similar to Equation (4.4).
- If the expression is an index expression, it should not be indexed again. So in Equation (4.8), the return expression is the index expression, and the current index number is still i.

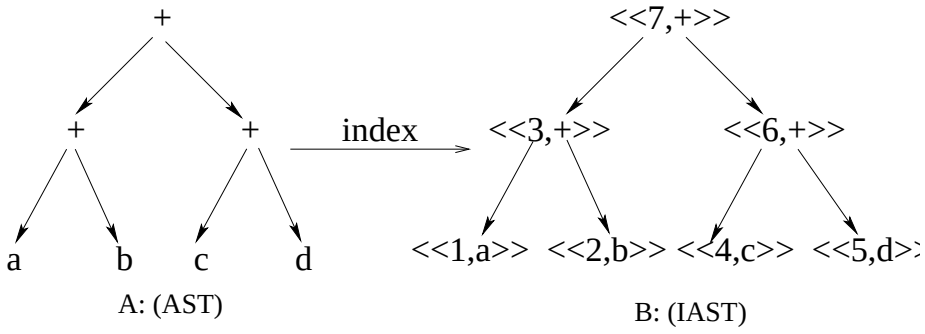

Figure 4.4: An Example of Index in $\mathcal{J}'$

Figure 4.4 shows an example of indexing an AST. In the figure, tree A is the original abstract syntax tree for expression e, `((a+b)+(c+d))`, and tree B is the indexed abstract syntax tree. The indexed expression for e is: `<7,(<3,(<1,a>+<2,b>)>+ <6,(<4,c>+<5,d>)>)>`.

4.3 Cost Semantics for Language $\mathcal{J}'$

Figure 4.5 shows the cost semantics of Language $\mathcal{J}'$. Semantic function $\vdash_c$: $env \rightarrow e \rightarrow e\ *\ cost$ takes the environment (env) and an expression (e), and returns a tuple of the expression and the cost ($cost$) of the expression under the environment.

$$\frac{}{\mathrm{E} \vdash_c k \,\$\, 0} \tag{4.9}$$

$$\frac{}{\{v, c\} + \mathrm{E} \vdash_c v \,\$\, c + 1} \tag{4.10}$$

$$\frac{\mathrm{E} \vdash_c e_1 \,\$\, c_1 \qquad \mathrm{E} \vdash_c e_2 \,\$\, c_2}{\mathrm{E} \vdash_c e_1 \; op \; e_2 \,\$\, 1 + c_1 + c_2} \tag{4.11}$$

$$\frac{\mathrm{E} \vdash_c e \,\$\, c}{\mathrm{E} \vdash_c \texttt{fun}\; x \;\rightarrow\; e \,\$\, c} \tag{4.12}$$

$$\frac{\mathrm{E} \vdash_c e_1 \,\$\, c_1 \qquad \mathrm{E} \vdash_c e_2 \,\$\, c_2}{\mathrm{E} \vdash_c (e_1 \; e_2) \,\$\, c_1 + c_2} \tag{4.13}$$

$$\frac{}{\mathrm{E} \vdash_c e \; (* \; c \; *) \,\$\, c} \tag{4.14}$$

$$\frac{\mathrm{E} \vdash_c e_1 \,\$\, c_1 \qquad \mathrm{E} \vdash_c e_2 \,\$\, c_2}{\mathrm{E} \vdash_c \texttt{map} \; e_1 \; e_2 \,\$\, c_1 * (length \; e_2) + c_2} \tag{4.15}$$

$$\frac{\mathrm{E} \vdash_c e \,\$\, c}{\mathrm{E} \vdash_c < i, e > \,\$\, c} \tag{4.16}$$

Figure 4.5: Cost Semantics for $\mathcal{J}'$

- Equation (4.9) infers the cost of an constant as 0 in environment E.
- Equation (4.10) shows that the cost of the value of variable (v), here c, has been stored in the environment, so the total cost of variable (v) is the cost to access the variable and the cost of the value of the variable, giving $c + 1$.
- Equation (4.11) performs the cost of operation expressions (e_1 *op* e_2). If the cost of e_1 is c_1, and the cost of e_2 is c_2 then the cost of e_1 *op* e_2 is $1 + c_1 + c_2$, where 1 is the cost for getting the operator.
- Equation (4.12) performs the cost of lambda expressions `fun` $x \;\rightarrow\; e$, which is the the cost of the body e.
- Equation (4.13) shows that the cost of application expression ($e_1 \;\; e_2$) is the cost of e_1 (c_1) plus the cost of e_2 (c_2).
- Equation (4.14) enables the user cost to provide in particular for a recursive functions. It is difficult to calculate the static cost of recursive functions. So in e $(* \; c \; *)$, the cost of e is c which is calculated by hand rather by the cost analyser automatically.
- Equation (4.15) infers the cost of the `map` expression. Here we assume that only regular cases of map are used. So function e_1 applied to every element of list e_2 has the same cost. So the total cost of `map` expression

is $c_1 * (length\ e_2) + c_2$, where c_1 is the cost of function e_1 applied to the first element of list e_2.

- Equation (4.16) shows that the cost of index expression $< i, e >$ is the cost of expression e (c).

Figure 4.6 shows the progress of costing the expression in Figure 4.4.

$$
\begin{array}{lll}
 & \text{cost of } < 7, (< 3, (< 1, a > + < 2, b >) > + < 6, (< 4, c > + < 5, d >) >) > & \\
\Rightarrow & \text{cost of } (< 3, (< 1, a > + < 2, b >) > + < 6, (< 4, c > + < 5, d >) >) & (4.16) \\
\Rightarrow & 1+\text{cost of } < 3, (< 1, a > + < 2, b >) > + \text{cost of } < 6, (< 4, c > + < 5, d >) > & (4.11) \\
\Rightarrow & 1+\text{cost of } (< 1, a > + < 2, b >) + \text{cost of } < 6, (< 4, c > + < 5, d >) > & (4.16) \\
\Rightarrow & 1+(1+\text{cost of } < 1, a > + \text{cost of } < 2, b >) + \text{cost of } < 6, (< 4, c > + < 5, d >) > & (4.11) \\
\Rightarrow & 1+(1+\text{cost of } a + \text{cost of } < 2, b >) + \text{cost of} < 6, (< 4, c > + < 5, d >) > & (4.16) \\
\Rightarrow & 1+(1+(1+c_a) + \text{cost of } < 2, b >) + \text{cost of } < 6, (< 4, c > + < 5, d >) > & (4.10) \\
\Rightarrow & 1+(1+(1+c_a) + \text{cost of } b) + \text{cost of } < 6, (< 4, c > + < 5, d >) > & (4.16) \\
\Rightarrow & 1+(1+(1+c_a)+(1+c_b)) + \text{cost of } < 6, (< 4, c > + < 5, d >) > & (4.10) \\
\Rightarrow & 1+(1+(1+c_a)+(1+c_b)) + \text{cost of } (< 4, c > + < 5, d >) & (4.16) \\
\Rightarrow & 1+(1+(1+c_a)+(1+c_b)) + (1+\text{cost of } < 4, c > + \text{cost of } < 5, d >) & (4.11) \\
\Rightarrow & 1+(1+(1+c_a)+(1+c_b)) + (1+\text{cost of } c + \text{cost of } < 5, d >) & (4.16) \\
\Rightarrow & 1+(1+(1+c_a)+(1+c_b)) + (1+(1+c_c) + \text{cost of } < 5, d >) & (4.10) \\
\Rightarrow & 1+(1+(1+c_a)+(1+c_b)) + (1+(1+c_c) + \text{cost of } d) & (4.16) \\
\Rightarrow & 1+(1+(1+c_a)+(1+c_b)) + (1+(1+c_c)+(1+c_d)) & (4.10)
\end{array}
$$

Figure 4.6: An Example of Costing in $\mathcal{J}'$

4.4 Costafter Semantics for Language $\mathcal{J}'$

There are two approaches to calculate the cost of the continuations (*costafter*) in a program. One is translating the direct program into continuation passing style (CPS) [8] to obtain the continuation of the current expression, then calculating the cost of the continuation. Another approach is to pass the cost of the remainder of the program directly rather than pass the continuation back to the current expression. There are two advantages of this approach. First, we do not need to translate the program into CPS. Second, it is easier to pass an integer (cost) back to the current expression than the execution state of the program e.g. the call stack or values of variables.

This section introduces the costafter semantics of $\mathcal{J}'$. In the costafter definitions of expression equity and contains will be used.

4.4.1 Expression Equality

Figure 4.7 shows the definition of expression equality. Semantic function $\equiv : e \rightarrow e$ applies expression equality, which compares if two expressions are the same. $\equiv : e \rightarrow e$.

$$\frac{e = k}{e \equiv k} \tag{4.17}$$

$$\frac{e = v}{e \equiv v} \tag{4.18}$$

$$\frac{e_1 \equiv e_3 \qquad e_2 \equiv e_4}{(e_1\ op\ e_2) \equiv (e_3\ op\ e_4)} \tag{4.19}$$

$$\frac{x_1 \equiv x_2 \qquad e_1 \equiv e_2}{(\texttt{fun}\ x_1 \rightarrow e_1) \equiv (\texttt{fun}\ x_2 \rightarrow e_2)} \tag{4.20}$$

$$\frac{e_1 \equiv e_3 \qquad e_2 \equiv e_4}{(e_1\ e_2) \equiv (e_3\ e_4)} \tag{4.21}$$

$$\frac{e_1 \equiv e_3 \qquad e_2 \equiv e_4}{(\texttt{map}\ e_1\ e_2) \equiv (\texttt{map}\ e_3\ e_4)} \tag{4.22}$$

$$\frac{e_1 \equiv e_2 \qquad c_1 \equiv c_2}{(e_1\ (*\ c_1\ *)) \equiv (e_2\ (*\ c_2\ *))} \tag{4.23}$$

$$\frac{}{< i, e_1 > \equiv < i, e_2 >} \tag{4.24}$$

Figure 4.7: Definition of Syntactic Expression Equality in $\mathcal{J}'$

- Equation (4.17) defines constant expression equality. If expression e is an constant and its value is equal to k, so expression e is equal to expression k.
- Equation (4.18) performs variable equality. In $\mathcal{J}'$, it is assumed that variables (v) names in the program are unique, so if two variables expressions have the same name, the two expressions are equal.
- Equation (4.19) performs the equality of two operation expressions. Expression (e_1 *op* e_2) equals to expression (e_3 *op* e_4), if e_1 equals to e_3 and e_2 equals to e_4.
- Equation (4.20) performs the equality of two lambda expression. Expression `fun` ($x_1 \rightarrow e_1$) equals to expression `fun` ($x_2 \rightarrow e_2$), if variable x_1 equals to variable x_2 and expression e_1 equals to e_2.
- Equations (4.21) and (4.22) define application expression and `map` expression equality, which are all similar to Equation (4.20).
- Equation (4.23) shows that two user cost expression are equal if both the expression parts and cost parts are equal.
- Equation (4.24) defines the index expression equality. In the calculus, the whole program has been indexed, so every expression in a program has a

unique integer as an index, so in $\mathcal{J}'$, two index expressions are equal to each other only if their indices are equal.

4.4.2 Expression Contains

Figure 4.8 gives the definition of contains. Semantic function $\in$ takes two expression, and returns true if the second expression contains the first expression.

$$\frac{e_1 \equiv e_2}{e_1 \in e_2} \tag{4.25}$$

$$\frac{e_1 \in e_2}{e_1 \in \texttt{fun}\ x\ \rightarrow e_2} \tag{4.26}$$

$$\frac{e_1 \in e_2}{e_1 \in (e_2\ e_3)} \tag{4.27a}$$

$$\frac{e_1 \in e_3}{e_1 \in (e_2\ e_3)} \tag{4.27b}$$

$$\frac{e_1 \in e_2}{e_1 \in e_2\ op\ e_3} \tag{4.28a}$$

$$\frac{e_1 \in e_3}{e_1 \in e_2\ op\ e_3} \tag{4.28b}$$

$$\frac{e_1 \in e_2}{e_1 \in e_2\ (*\ c\ *)} \tag{4.29}$$

$$\frac{e_1 \in e_2}{e_1 \in \texttt{map}\ e_2\ e_3} \tag{4.30a}$$

$$\frac{e_1 \in e_3}{e_1 \in \texttt{map}\ e_2\ e_3} \tag{4.30b}$$

$$\frac{e_1 \in e_2}{e_1 \in < i, e_2 >} \tag{4.31}$$

Figure 4.8: Definition of Contains in $\mathcal{J}'$

- Equation (4.25) identifies that if expression e_1 is equal to expression e_2 then the two expressions contain each other.
- Equation (4.26) shows that if e_1 is contained in e_2, then e_1 is contained in `fun` $x \rightarrow e2$.
- Equation (4.27a) to (4.31) gives the definition of contains for different expressions, which are all similar to Equation (4.26).

An example to deduce if expression `(a+,b)+(c+d)` contains expression `b` is given in Figure 4.9, where expression `<7,(<3,(<1,a>+<2,b>)>+<6,(<4,c>+<5,d>)>)>` is indexed expression `(a+b)+(c+d)`.

$$
\begin{array}{lll}
 & b = b & \\
 & b \equiv b & (4.18) \\
\Rightarrow & b \in b & (4.25) \\
\Rightarrow & b \in <2, b> & (4.31) \\
\Rightarrow & b \in (<1, a> + <2, b>) & (4.28b) \\
\Rightarrow & b \in <3, (<1, a> + <2, b>)> & (4.31) \\
\Rightarrow & b \in <3, (<1, a> + <2, b>)> + <6, (<4, c> + <5, d>)> & (4.28a) \\
\Rightarrow & b \in <7, (<3, (<1, a> + <2, b>)> + <6, (<4, c> + <5, d>)>)> & (4.31)
\end{array}
$$

Figure 4.9: An Example of Contains in $\mathcal{J}'$

4.4.3 Costafter Semantics for $\mathcal{J}'$

Figure 4.10 show the semantics of costafter of expression e in different expressions. Semantic function $\vdash_a : \mathit{env} \rightarrow e \rightarrow e \rightarrow \mathit{cost}$ takes the environment (*env*) and two expressions and returns a cost, which is the costafter of the first expression in the second expression.

- Equation (4.32) states that if expression e is equal to e' then the costafter of e in e' is 0.
- Equation (4.33a) and (4.33b) define the costafter of e in lambda expressions. If the costafter of e in e_1 is c, then the costafter of e in lambda expression `fun` $x \rightarrow e_1$ is c too. If e_1 does not contains e then the costafter of e in `fun` $x \rightarrow e_1$ is 0.
- Equation (4.34a), (4.34b), and (4.34c) define the costafter of e in application expressions (e_1 e_2). If e_1 contains e, then the costafter of e in (e_1 e_2) is the costafter of e in e_1, here c_1, plus the cost of e_2, here c_2. If e_2 contains e then the costafter of e in (e_1 e_2) is the cost after e in e_2. If e is not contained in e_1 or e_2, then the costafter of e in (e_1 e_2) 0.
- Equation (4.35a), (4.35b), and (4.35c) define the costafter of e in operation expressions e.g. (e_1 *op* e_2). If e_1 contains e, then the costafter of e in (e_1 e_2) is the costafter of e in e_1 (c_1), plus the cost of e_2 (c_2) plus 1, which is the cost for getting the operator.
- In language $\mathcal{J}'$ the cost of recursive function can not be calculated automatically, so the costafters in recursive functions are not considered. The user cost expression gives the cost of a recursive function, so the costafter of any expression in a user cost expressions is 0, see Equation (4.36).
- Equation (4.37a), (4.37b), and (4.37c) define the costafter of e in `map` expression `map` e_1 e_2. Equation (4.37a) shows that if e_1 contains e, then the costafter of e in `map` e_1 e_2 is the costafter of e in e_1, plus the cost of e_2, and plus the cost of `map` expression, because after e_1 and e_2 `map` expression will be executed. A similar situation of e_2 containing e is handled in Equation (4.37b). Equation (4.37c) shows that if e is not contained in e_1 or e_2 then the costafter of e in `map` e_1 e_2 is 0.

$$\frac{e \;\equiv\; e'}{\mathrm{E} \vdash_a e \trianglelefteq e' \pounds 0} \tag{4.32}$$

$$\frac{\mathrm{E} \vdash_a e \trianglelefteq e_1 \pounds c}{\mathrm{E} \vdash_a e \trianglelefteq \mathtt{fun}\; x \;\to e_1 \pounds c} \tag{4.33a}$$

$$\frac{}{\mathrm{E} \vdash_a e \trianglelefteq \mathtt{fun}\; x \;\to e_1 \pounds 0} \tag{4.33b}$$

$$\frac{e \;\in\; e_1 \qquad \mathrm{E} \vdash_a e \trianglelefteq e_1 \pounds c_1 \qquad \mathrm{E} \vdash_c e_2 \$ c_2}{\mathrm{E} \vdash_a e \trianglelefteq (e_1\; e_2) \pounds c_1 + c_2} \tag{4.34a}$$

$$\frac{e \;\in\; e_2 \qquad \mathrm{E} \vdash_a e \trianglelefteq e_2 \pounds c_2}{\mathrm{E} \vdash_a e \trianglelefteq (e_1\; e_2) \pounds c_2} \tag{4.34b}$$

$$\frac{}{\mathrm{E} \vdash_a e \trianglelefteq (e_1\; e_2) \pounds 0} \tag{4.34c}$$

$$\frac{e \;\in\; e_1 \qquad \mathrm{E} \vdash_a e \trianglelefteq e_1 \pounds c_1 \qquad \mathrm{E} \vdash_c e_2 \$ c_2}{\mathrm{E} \vdash_a e \trianglelefteq (e_1\; op\; e_2) \pounds 1 + c_1 + c_2} \tag{4.35a}$$

$$\frac{e \;\in\; e_2 \qquad \mathrm{E} \vdash_a e \trianglelefteq e_2 \pounds c_2}{\mathrm{E} \vdash_a e \trianglelefteq (e_1\; op\; e_2) \pounds c_2 + 1} \tag{4.35b}$$

$$\frac{}{\mathrm{E} \vdash_a e \trianglelefteq (e_1\; op\; e_2) \pounds 0} \tag{4.35c}$$

$$\frac{}{\mathrm{E} \vdash_a e \trianglelefteq e_1\; (*\; c\; *) \pounds 0} \tag{4.36}$$

$$\frac{e \;\in\; e_1 \qquad \mathrm{E} \vdash_c \mathtt{map}\; e_1\; e_2 \$ c \qquad \mathrm{E} \vdash_a e \trianglelefteq e_1 \pounds c_1 \qquad \mathrm{E} \vdash_c e_2 \$ c_2}{\mathrm{E} \vdash_a e \trianglelefteq \mathtt{map}\; e_1\; e_2 \pounds c + c_1 + c_2} \tag{4.37a}$$

$$\frac{e \;\in\; e_2 \qquad \mathrm{E} \vdash_c \mathtt{map}\; e_1\; e_2 \$ c \qquad \mathrm{E} \vdash_a e \trianglelefteq e_2 \pounds c_2}{\mathrm{E} \vdash_a e \trianglelefteq \mathtt{map}\; e_1\; e_2 \pounds c + c_2} \tag{4.37b}$$

$$\frac{}{\mathrm{E} \vdash_a e \trianglelefteq \mathtt{map}\; e_1\; e_2 \pounds 0} \tag{4.37c}$$

$$\frac{\mathrm{E} \vdash_a e \trianglelefteq e_1 \pounds c}{\mathrm{E} \vdash_a e \trianglelefteq <i, e_1> \pounds c} \tag{4.38}$$

Figure 4.10: Costafter Semantics for $\mathcal{J}'$

costafter of b in
$< 7, (< 3, (< 1, a > + < 2, b >) > + < 6, (< 4, c > + < 5, d >) >) >$

$\Rightarrow$ costafter of b in
$(< 3, (< 1, a > + < 2, b >) > + < 6, (< 4, c > + < 5, d >) >)$ (4.38)

$\Rightarrow$ 1+costafter of b in
$< 3, (< 1, a > + < 2, b >) >$+cost of $< 6, (< 4, c > + < 5, d >) >$ (4.35a)

$\Rightarrow$ 1+costafter of b in
$(< 1, a > + < 2, b >)$+cost of $< 6, (< 4, c > + < 5, d >) >$ (4.38)

$\Rightarrow$ 1+(1+costafter of b in
$< 2, b >)$+cost of $< 6, (< 4, c > + < 5, d >) >$ (4.35b)

$\Rightarrow$ 1+(1+costafter of b in b)+cost of $< 6, (< 4, c > + < 5, d >) >$ (4.38)

$\Rightarrow$ 1+(1+0)+cost of $< 6, (< 4, c > + < 5, d >) >$ (4.32)

$\Rightarrow$ 1+(1+0)+(1+ (1+c_c) + (1+c_d)) (From Figure 4.6)

Figure 4.11: An Example of Costafter in $\mathcal{J}'$

- Equation (4.38) shows that the costafter of e in index expression $< i, e_1 >$ is the same as the costafter of e in expression e_1.

The expression in Figure 4.4 is used as an example to demonstrate how to get the costafter of expression in another expression. In Figure 4.6, the cost of expression `((a+b)+(c+d))` have been calculated. The question is how to get the costafter of each sub-expression in the expression. From the deduction in Figure 4.9, expression `b` contains in expression `b` so it contains in expression e_i in Section 4.2, the indexed expression of `((a+b)+(c+d))`. Figure 4.11 gives the example to get the costafter of expression `b` in expression `((a+b)+(c+d))`. The result from the program, which implements the costafter semantics, is `((0+1)+((1+((1+0)+(1+0)))+1))`, where the first `0` is the costafter of `b` in `b` and the second and the third `0`s are the cost of `c`, and `d`, because in the program `a`, `b`, `c`, and `d` are integers and the cost of integer is 0. So the result is the same as which is calculated by hand in Figure 4.11.

Chapter 5

Costed Autonomous Mobility Skeletons

5.1 AMP Generic Cost Model

For AMPs a cost model is used to inform the decision whether to move to a new location. A generic AMP cost model has been built in [6] and is shown in Figure 5.1.

- Equation (5.2) gives the condition under which the program will move, i.e. if the time to complete in the current location is more than the time to complete in the remote location.
- Equation (5.3) states that if there are m communications in a program's lifetime and it will take T_{comm} time for each communication then the total time for communication is T_{comm} by m.
- Equation (5.4) states that if there are p processors and n times checking the status of the processors in a program's lifetime and it will take T_{coord} time for checking one processor once then the total time for coordination is T_{coord} by p by n.
- Equation (5.5) gives the condition under which the program will do the coordination work. This seekers to guarantee that the autonomous mobile program will never worse than $100 + O$ percent of the static version. This guarantee is only valid providing that the loads on the current and target location does not change dramatically immediately after the move. For example it is easy to construct a pernicious example where each time an AMP moves to a location the load on that location becomes very high.
- Substituting Equation (5.4) in (5.5) gives Equation (5.6), where n specifies how many times the AMP will consider moving.

$$T_{total} = T_{Comp} + T_{Comm} + T_{Coord} \tag{5.1}$$

$$T_h > T_{comm} + T_n \tag{5.2}$$

$$T_{Comm} = mT_{comm} \tag{5.3}$$

$$T_{Coord} = npT_{coord} \tag{5.4}$$

$$T_{Coord} < OT_{static} \tag{5.5}$$

$$n < \frac{OT_{static}}{pT_{coord}} \tag{5.6}$$

$$T_e = W_d/S_h \tag{5.7}$$

$$T_h = W_l/S_h \tag{5.8}$$

$$T_n = W_l/S_n \tag{5.9}$$

$$W_a = \sum W_d \tag{5.10}$$

$$W_d = W_{a(thistime)} - W_{a(lasttime)} \tag{5.11}$$

$$W_l = W_{all} - W_a \tag{5.12}$$

O	:	Overhead e.g. 5%
T_{total}	:	total 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 processor(location)
T_{Comp}	:	time for computation
T_e	:	time has elapsed at current location
T_h	:	time will take here
T_n	:	time will take in the next location
W_{all}	:	all work
W_a	:	the total work which has been done
W_d	:	the work has been done at current location
W_l	:	the work left
S_h	:	the current CPU speed
S_n	:	the next location CPU speed
m	:	number of communication
n	:	number of coordination
p	:	number of processor

Figure 5.1: Generic Cost Model for AMPs

$$W_{all} = costf * (length\ of\ the\ list) + costafter\ =\ a \tag{5.13}$$

$$W_a = r \tag{5.14}$$

$$W_l = W_{all} - W_a = a - r \tag{5.15}$$

$$W_d = costf \tag{5.16}$$

$$T_e = \frac{W_d}{S_h} Sec = \frac{costf}{S_h} Sec \tag{5.17}$$

$$T_h = \frac{W_l}{S_h} Sec = \frac{(a-r)Sec}{S_h} \tag{5.18}$$

$$T_h = \frac{((a-r)Sec)T_e}{Sec} = \frac{(a-r)}{costf} T_e \tag{5.19}$$

$$T_n = \frac{W_l}{S_n} = \frac{(a-r)Sec}{S_n} = \frac{S_h T_h}{S_n} \tag{5.20}$$

Figure 5.2: Cost Model for CAMS

- Equations (5.7), (5.8) and (5.9) relate time, work and CPU speed. The time equals the work measured by CPU speed.
- In Equation (5.10), W_d is the work that has been done at one location, so the total work is the sum of all the W_d. In other words, (5.11) shows that the work done at the current location equals the total work done before the program moved to the current location ($W_{d(lasttime)}$) minus all the work that has been done ($W_{d(thistime)}$ or W_d).
- Equation (5.12) gives the remaining work, that is the total work minus all the work that has been done.

5.2 CAMS Cost Model

The CAMS cost model instantiates the generic AMP cost model and is parameterised on the costafter of the skeletons, and is shown in Figure 5.2. In this cost model:

- In Equation (5.13) the total work is the cost in the CAMS and the costafter of the CAMS.
- Equation (5.14) gives the work that has been done, `r`.
- Equation (5.15) shows that the remaining work is the total work minus the work has been done.
- Equation (5.16) shows that the work that has been done at the current location is the cost of `f`, which is the cost for processing one element in the list.

- Substituting Equation (5.16) in (5.7), the time that has elapsed at current location can be calculated giving Equation (5.17).
- Substituting Equation (5.15) in (5.8) the time it will take at the current location can be calculated giving Equation (5.18).
- Substituting Equation (5.17) in (5.18) Equation (5.19) can be got, which shows the time it will take at the current location is a function of T_e.
- Substituting Equation (5.18) in (5.8) the time that will be taken in the next location can be predicted as Equation (5.20).

5.3 An Implementation of Costed Autonomous Mobility Skeletons

A `camap` is implemented in Jocaml as shown in Figure 5.3. The auxiliary functions `check_move` `checkInfo` are the same as those in AMSs in [5]. The movement check is encoded in `check_move` function. The `checkInfo` function is to evaluate the benefits of a move and to recalculate when should check again.

```
let rec camap' f l costf costaftermap 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 costaftermap fhtime' (workleft-costf)
   )
   else
   (let fh = f h in
    t_current := Unix.gettimeofday();
    fh::camap' f t costf costaftermap fhtime (workleft-costf)
   )

let camap f l costf costaftermap =
   let (h::t) = l in
   (let localwork = costf * (length l) in
    let work = localwork + costaftermap in
    let (fh,fhtime) = timedapply f h in
    t_current := Unix.gettimeofday();
    getInfo costf fhtime;
    fh::camap' f t costf costaftermap fhtime (work-costf)
   )
```

Figure 5.3: Implementation of `camap` in Jocaml

Chapter 6

Evaluating Costed Autonomous Mobility Skeletons

To evaluate the performance of `camap` against `automap`, six AMPs have been built using different numbers of higher-order functions. To construct the AMPs using `camap f l costf costafter`, the `costf` and `costafter` are calculated by hand according to the cost calculus for $\mathcal{J}'$. Section 7 will introduce an automatic continuation cost analyser, which can automatically generate higher-order functions into costed autonomous mobility skeleton.

6.1 Single Higher Order Function Examples

The initial hypothesis is if there is only one higher-order functions in the AMP, the performances of the CAMS programs should be the the same as the corresponding AMS programs. That is the AMPs should exhibit the same movement behaviour at the same moment and hence have the same execution times. Two single higher-order function AMPs have been tested: matrix multiplication and ray tracing.

Different size matrix multiplication have been executed to see if the CAMS programs have the same execution time as the corresponding AMS programs. The test environment has three locations with CPU speeds 534MHZ(ncc1710), 933MHZ(jove) and 1894MHZ(lxtrinder). The loads on these three locations are almost zero, and both the CAMS and AMS programs are started on the first location.

Figure 6.1 shows that the CAMS programs behave the same as the corresponding AMS programs. Both the CAMS and AMS programs start to move when the matrix size increased to 330*330, and hence the execution time of the CAMS and AMS programs are almost identical. Figure 6.2 shows similar

results for ray tracing AMPs.

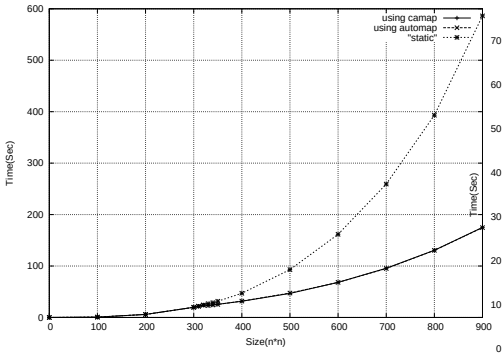

Figure 6.1: CAMS and AMS Matrix Multiplication Execution Time Comparison

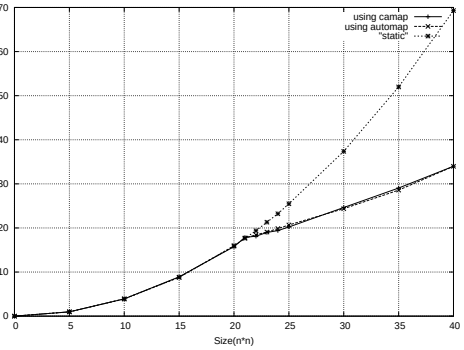

Figure 6.2: CAMS and AMS Ray Tracing Execution Time Comparison

From the results of matrix multiplication and ray tracing, it can be concluded when there is only one higher-order functions in the AMP, the performances of the `camap` AMP are as good as the `automap` AMP.

6.2 Sequential Composed Higher Order Function Examples

This section investigates the performance of the skeletons when an AMP contains the sequential composition of several higher-order functions. Four experiments have been performed with sequential `camap`. The AMPs are double matrix multiplication, invertible matrix, double ray tracing, and five matrix multiplications. The test environments in this section are the same as in Section 6.1.

6.2.1 Invertible Matrix Comparison

The invertible matrix program takes two matrix and checks if them are invertible to each. 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;;
```

In this program there are two matrix multiplications, so there are two higher-order functions as in double matrix multiplications. The `checkEqual` function checks if the matrix is identical matrix, which takes a little execution time, so

the invertible matrix program is very similar to double matrix multiplication program.

Figure 6.3 compares the `camap` and `automap` invertible matrix, where two matrix multiplication are performed sequentially. The `camap` AMP starts moving at matrix size of 230*230, but the `automap` AMP starts moving still at 330*330, because the `camap` considers the total remaining costs of the whole program but `automap` only consider the remaining costs in the function.

The figure shows that in some cases the CAMS programs are slower than the corresponding AMS programs, for the following reasons.

- When the matrix is large enough for both the CAMS and AMS programs to move to a faster location with the same size of matrix e.g. 330*330, the AMS program move at an *earlier element* (not size) than the CAMS program. For example in the AMP of size 330*330 invertible matrix, the first `automap` in the program only calculates the 330 elements in itself, and decides to move once, so the program moves at the 165th element. However, the first `camap` in the CAMS program, calculates 330*2=660 elements, and considers moving twice, so the program moves at the 220th element. Hence, the CAMS program moves to a faster location later than the AMS program and it is a little slower.

- Another reason is that the CAMS programs may perform more checks than the corresponding AMS programs. Still using the program with the 330*330 invertible matrix, after the AMS program moves to a faster location the first `automap` considers the remaining 165 elements only, so it might not check information again. However the first `camap` in the CAMS program considers 110+330=440 elements, so it might check the locations information again.

In these two cases the CAMS programs may be a little slower than the corresponding AMS programs, but are both faster than the static programs. Similar

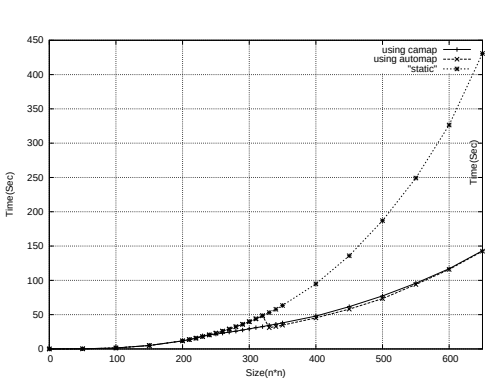

Figure 6.3: CAMS and AMS Invertible Matrix Multiplications Execution Time Comparison

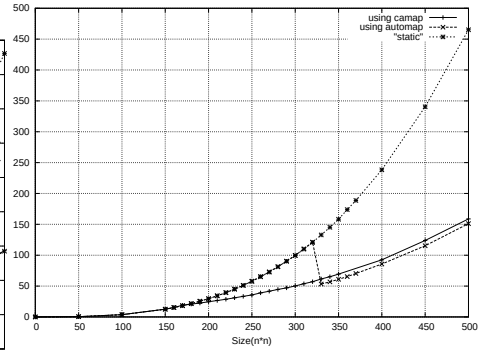

Figure 6.4: CAMS and AMS Five Matrix Execution Time Comparison

results have been gained for double ray tracing and double matrix multiplication.

6.2.2 Five Matrix Multiplication Comparison

Figure 6.4 compares movements of CAMS and AMS five matrix multiplications AMPs, where matrix multiplication are done five times. Hence, there are five higher-order functions in the program. The results are similar to the results for invertibla matrix.

Comparing Figure 6.4 with Figure 6.3 shows that the CAMS five matrix multiplication program starts to move at size 170*170, which is smaller than the CAMS invertible matrix program at size 230*230. However, the AMS five matrix multiplication program still starts to move at size 330*330 matrix, which is the same as the AMS invertible matrix program. The reason is that in CAMS five matrix multiplication, the first `camap` considers the costs of 5 higher-order functions, but the first `automap` in the corresponding AMS program still only considers the cost of the current collection computation.

6.2.3 Discussion

From the results for AMPs with different number of higher-order functions, the following properties of CAMS and AMS can be noted:

- In some cases the CAMS programs are faster than the corresponding AMS programs because CAMS programs move to a faster location with smaller size data than the corresponding AMS programs, and the more higher-order functions in a AMP, the earlier the CAMS program moves compared with the corresponding AMS program.

- The CAMS programs may do more checks than the corresponding AMS programs. This is a disadvantage for execution time i.e. in this case the CAMS programs may be slower than the corresponding AMS programs, but is an advantage for reacting to the change of environment, which will be considered in Section 6.2.4.

- For the same size of matrix, if both the CAMS and corresponding AMS programs move to a faster location, then the AMS programs move at an *earlier element* (not size) than the corresponding CAMS programs. In this case, the AMS programs are faster than the corresponding CAMS programs.

- Even if the AMS programs may move at an earlier element than the corresponding CAMS programs, the ratio of CAMS program execution compared to the corresponding AMS program becomes smaller when the data becomes bigger, as the number of the element at which the CAMS program starts moving compared to the AMS program becomes smaller when the data becomes bigger. For example, in an AMP l is the length of list in the

higher-order functions, n is number of higher-order function, and m is the checks for one higher-order function. The *gran*, at which the AMP moves, for `automap` is $gran_{automap} = \frac{l}{m+1}$, and for `camap` is $gran_{camap} = \frac{l*n}{m*n+1}$. $gran_{camap}/gran_{automap} = \frac{l*n}{m*n+1}/\frac{l}{m+1}$ can be simplified to $1 + \frac{n-1}{m*n+1}$. So when n is a constant, the bigger the m the smaller the ratio. As bigger AMPs can do more checks, which is the bigger m, the bigger the AMP, the smaller the ratio. The assumption here is that the n higher-order functions are working on the same function and the same size list.

6.2.4 Performance Comparison with Changing Load

When there is more than one higher-order function in the AMP, CAMS programs may do more checks than the corresponding AMS programs (see Section 6.2). This is an advantage for reacting to the change of environment. The experiments in this section test the behaviour of CAMS and AMS programs when the loads of locations in the network are changed. Tests for two AMPs have been done: invertible matrix and five matrix multiplications. The tests are based on four locations, (1)ncc1710, (2)jove, (3)lxtrinder, and (4)linux81, with CPU speed of 534MHz, 933MHz, 1894MHz, and 2800MHz. At the beginning, the first three location are idle and the fourth location is heavily loaded with relative CPU speed of 56MHz.

Figure 6.5 compares the execution time of CAMS to AMS invertible matrix programs. Both CAMS and AMS invertible matrix programs are tested one by one. The AMPs are started on ncc1710 and move to lxtrinder as expected. At the same time linux81 finishes the work and becomes idle.

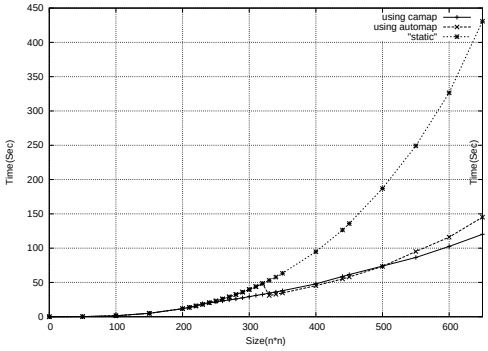

Figure 6.5: CAMS and AMS Invertible Matrix Execution Time Comparison with Changing Loads

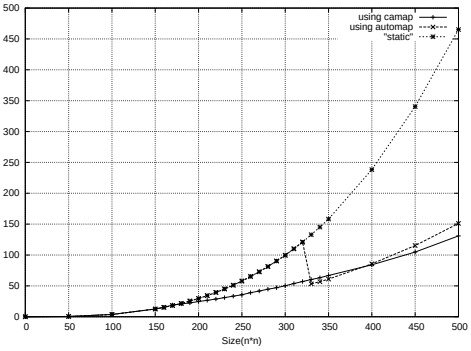

Figure 6.6: CAMS and AMS Five Matrix Multiplications Execution Time Comparison with Changing Loads

When the sizes of matrices are larger than 450*450, the CAMS programs move to the faster location linux81 again, but the corresponding AMS programs do not, as the `camap` in the CAMS programs do more checks than `automap` in the corresponding AMS programs. In this case, the CAMS programs may finish

more quickly than the corresponding AMS programs. When the size of the matrix is larger than 450*450 but smaller than 500*500, the CAMS programs is slower than the corresponding AMS programs, as the additional coordination time is bigger than the reduced execution time in a faster location. When the size of the matrix is larger than 500*500, the CAMS programs is faster than the corresponding AMS programs.

When the size of matrix is smaller than 450*450, the CAMS programs do not move, because the cost of moving (T_{comm}) to another location is too big compared to the reduced execution time in a faster location. In this case, the results are similar to the result in Figure 6.3. Similar results are also obtained for five matrix multiplication AMPs, shown in Figure 6.6.

Note that the more higher-order functions there are in AMPs, the smaller size data with which the CAMS programs start to move again. In the invertible matrix, there are two higher-order functions and the CAMP programs move again when the size of the matrix is larger than 450*450. In five matrix multiplication, there are five higher-order functions and the CAMS programs move again when the size of the matrix is larger than 330*330.

6.2.5 Summary

From the results in Sections 6.1, 6.2, and 6.2.4, the following conclusion can be drawn:

- If there is only one high-order function dominating the computation, CAMS programs reproduce the movement of the corresponding AMS programs (Section 6.1).
- If there is more than one higher-order functions, CAMS programs move with smaller size data than the corresponding AMS programs. The more higher-order functions in AMPs, the CAMS programs move with the smaller data than the corresponding AMS programs and hence the greater the potential for gains (Section 6.2).
- When both the CAMS and AMS programs move, the CAMS programs may be slower than the corresponding AMS programs, but the larger the data in the AMPs the earlier the CAMS programs move relative to the corresponding AMS programs (Section 6.2).
- The CAMS programs react to the change of environment more sensitively than the corresponding AMS programs (Section 6.2.4).

Chapter 7

Automatic Continuation Cost Analyser

7.1 Structure of the Automatic Continuation Cost Analyser

The cost calculus has been implemented as an automatic cost analyser in Jocaml. The analyser produces cost equations parameterised on program variables in context, and is used to find both cost in higher order functions and the costafter of the higher order functions. As *costafter* is the cost of the continuations, the analyser is also called an *automatic continuation cost analyser*. The analyser takes programs in a subset of Jocaml with higher-order functions as input and outputs Jocaml AMPs with CAMSs. Figure 7.1 shows the structure of the automatic continuation cost analyser. The analyser has four parts.

1. Parser takes Jocaml programs with higher-order functions e.g. `map` and `fold` and outputs the abstract syntax tree (AST).
2. Indexer is an implementation of the index semantics in Section 4.2. Indexer takes the AST and decorates every nodes, i.e. gives each expression a unique integer as an index, so the output is *indexed abstract syntax tree (IAST)*.
3. Coster takes the IAST and outputs the costafter for each node. The coster has two parts, the first is to calculate the cost of each expression, which implements the cost semantics in Section 4.3, the second part is to calculate the costafter of each expression, which implements the costafter semantics in Section 4.4. The second part will use the the costs which have been got from the first part.
4. The Generator generates a Jocaml program to a Jocaml AMP, which has the same functionality as the Jocaml program but using CAMSs instead of higher-order functions.

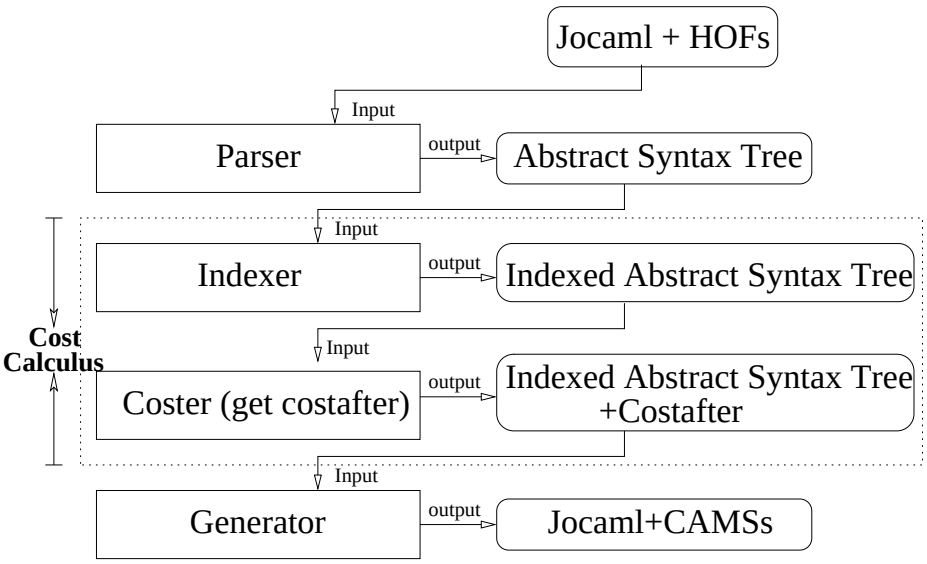

Figure 7.1: Structure of Automatic Continuation Cost Analyser

7.2 An Implementation of the Cost Calculus

In the automatic continuation cost analyser, `index`, `cost`, and `costafter` functions have been implemented.

The following code shows part of the implementation of the `index` function, where `index` takes the current index number `i` and the expression to be indexed `e` and returns the indexed expression and the next index number `i+1`. The type of `index` is: `int -> expression -> (expression * int)`.

```
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 implements the cost semantics introduced in Section 4.3. The type of `cost` is: `env -> expression -> int`. The following is the part of the code of `cost`.

```
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 `costafter` function implements the costafter semantics introduced in Section 4.4. The following is part of the implementation of `costafter`. The type of `cost` function is: `env -> expression -> expression -> int`.

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

7.3 Generator: Generating Autonomous Mobility Skeletons

The functionality of the generator is to find the higher-order functions in a program and translate them to CAMSs with the costafters. For example, if the original program is `map f l`, the object program after the generator is `camap f l costf costafter`, where `costf` is the cost of `f` applied to the first element of `l`, which can be calculated using the cost semantics, and `costafter` is the cost after the `map` expression in the program, which can be calculated using costafter semantics.

This section uses expression e, `(map (fun x -> x+1) [1;2]);(map (fun y -> y-1) [3;4])`, as an example to explain how to convert higher-order functions to CAMS. 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])`, so e can be presented as $e_1; e_2$. To construct CAMSs, four issues should be considered:

(1) the cost of `(fun x -> x+1)`,

(2) the costafter of e_1 in e,

(3) the cost of `(fun y -> y-1)`,

(4) and the costafter of e_2 in e.

From the cost equations in Section 4.3, the cost of `(fun x -> x+1) (hd [1;2])` can be calculated, which is `(fun x -> (1+((1+0)+0))) (hd [1;2])` (Equation 4.13, 4.12, 4.10). The simplified result is 2.

The costafter of e_1 in e is the costafter of e_1 in e_1, which is 0 (Equation 4.32), plus the cost of e_2, plus 1 (Equation 4.35a). The cost of e_2 is `((fun y -> (1+((1+0)+0))) (hd [3;4]))*(length [3;4])`, which is simplified as 4, so the total costafter of e_1 in e is 5. The generator converts e_1 to `(camap (fun x -> (x+1)) [1;2] (2) (5))`. Similarly, e_2 is converted to `(camap (fun y -> (y-1)) [3;4] (2) (1))`.

The result from the program, which implements the generator is shown as follows. The result is the same as that which is calculated by hand. The generator takes the original program e, and generates it to autonomous mobile program with CAMSs. The AMP is as follows.

```
(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)
                                           (* costafter of first map *)
);
(camap (fun y -> (y-1)) [3;4]
  (((fun y -> (1+((1+0)+0))) (hd [3;4])))  (* cost of (fun x -> x+1) *)
  (0)                                      (* costafter of second map *)
)
```

The AMP can be simplified to:

```
(camap (fun x -> (x+1)) [1;2] (2) (5) );
(camap (fun y -> (y-1)) [3;4] (2) (1) )
```

7.4 Matrix Multiplication Example

The following is the source code of matrix multiplication in Jocaml.

```
let rec dist = (fun vec1 -> fun vec2 ->
  match (vec1,vec2) with
    ([],vec) -> [] |
    ((h1::t1),(h2::t2)) -> (h1::h2)::(dist t1 t2) |
    ((h1::t1), []) -> [h1]::(dist t1 []))
  (* fun mv11-> fun mv22 ->  3*(length mv11)*(length mv22) *)
  in
  let rec transpose = (fun e -> fold_right dist e [])
  (* fun le -> 2*(length le)*(length le) *)
  in
  let rec dotprod = (fun mat1 -> fun mat2 ->
    match (mat1,mat2) with
      ((h1::t1),(h2::t2)) -> h1*h2+(dotprod t1 t2) |
      (m1,m2) -> 0 )
  (* fun l1 -> fun l2 -> (4*(length l2)) *)
  in
  let rec rowmult  = (fun cls -> fun row -> map (dotprod row) cls )
  (* fun rowc -> fun lsc -> 4*(length rowc)*(length rowc) *)
  in
  let rec rowsmult = (fun rows -> fun cols ->
```

```
    map ( rowmult cols) rows )
  in
  let tm2 = transpose mat2(*0*) in
  rowsmult mat1(*0*) tm2(*0*)
```

The analyser takes the source code as input and outputs the target code of matrix multiplication where top level `map` has been converted to `camap`:

```
 let rec dist =
((fun vec1 -> (fun vec2 -> match (vec1,vec2) with
([],vec) -> [] |
((h1::t1),(h2::t2)) -> ((h1::h2)::((dist t1) t2)) |
((h1::t1),[]) -> ([h1]::((dist t1) [])))));;
 let rec transpose =
((fun e -> (fold_right dist e [])));;
 let rec dotprod =
((fun mat1 -> (fun mat2 -> match (mat1,mat2) with
((h1::t1),(h2::t2)) -> ((h1*h2)+((dotprod t1) t2)) |
(m1,m2) -> 0)));;
 let rec rowmult =
((fun cls -> (fun row -> (map (dotprod row) cls))));;
 let rec rowsmult =
(fun rows -> (fun cols -> (map (rowmult cols) rows)));;
 let tm2 = (transpose mat2)
 in
 (camap (rowmult tm2) (mat1)
  (((4*(length tm2))*(length tm2))) (0) )
```

Note this example is in language $\mathcal{J}$, which is an extension of $\mathcal{J}'$. The cost calculus for $\mathcal{J}$ is presented in Appendix A.2.

7.5 Comparing Automatic and Hand Analysis

This section compares the performances CAMSs generated by the analyser with the CAMSs which have been produced by hand in Section 6. Comparison of double matrix multiplication, invertible matrix, and double ray tracing AMPs have been made.

Figure 7.2 compares the execution times of different sizes of double matrix multiplication, which show that the execution time of CAMS programs producted by the analyser automatically are very similar to the CAMS programs producted by hand. Similar results are gained invertible matrix AMPs and double ray tracing AMPs.

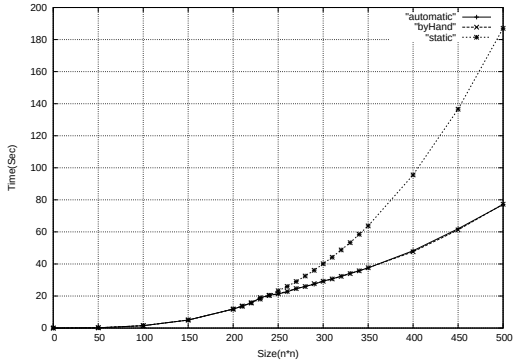

Figure 7.2: Automatic and Byhand Cost Double Matrix Multiplications Execution Time Comparison

Chapter 8

Conclusion & Future Work

8.1 Conclusion

One of the biggest issues for distributed systems is how to share the resources. This research demonstrates autonomous mobility for *self-optimization* of computational resource. We have proposed autonomous mobile programs (AMPs). AMPs can periodically use a cost model to decide mobility affects. The advantages of an AMP model are as follows. AMPs making decentralised decisions about where to execute. The model manages potentially dynamic networks very easily with each AMP selecting where to execute from the current set of locations. The AMP model can obtain a better balance than a classical distributed load balancer as, unlike the latter, each AMP has a cost model giving accurate information about the time to complete and to communicate the program. Moreover it is possible to parameterise the AMP cost model with a maximum overhead, e.g. 5%, and guarantee under a reasonable assumption that autonomous mobility overheads will not exceed it.

The AMP architecture introduces coordination costs as every AMPs must obtain load information about locations. The coordination cost is the additional cost of AMPs against the sequential program, so the smaller the coordination cost the better performance of AMPs will be gained. Minimising the effect of the coordination cost must be take into account when building AMPs. In the AMP load server architecture, every location has a load server, which collects the local information and get information from other load server. The AMPs get information about locations from the local load server rather than collect by itself.

Collections of AMPs behave like a decentralised load balancing system. This has been explored under the load server structure on both homogeneous and heterogeneous networks. Collections of AMPs quickly obtain and maintain optimal or near-optimal balance until the balance is broken.

The disadvantages of AMPs are that the programmer must explicitly control when the program moves and AMPs also require an accurate model of compu-

tation. To encapsulate self-aware mobile coordination for common patterns of computation over collections *autonomous mobility skeleton* (AMSs) have been developed. Autonomous mobility skeletons are akin to algorithmic skeletons in being polymorphic higher order functions, but where algorithmic skeletons abstract over parallel coordination, AMSs abstract over autonomous mobile coordination. By analogy with other skeleton species, they hide low level mobile coordination details from users and provide higher level loci for designing load-aware mobile systems. Our experiments suggest that, for our set of test programs, autonomous mobility skeletons can offer considerable savings in execution times, which scale well as overall execution times increase. Cost models for AMS are dynamic and substantially implicit. During the traversal of a collection, the skeleton implementation periodically measures the time to compute a single collection element, and uses the value to parameterise an implicit cost for the remainder of the traversal.

An AMS considers only the cost of the current collective computation, but it is more useful to know the cost of the entire program. We have extended our AMS cost models to be parameterised on the cost of the remainder of the program. A cost calculus to estimate the costs for the remainder of a computation at arbitrary points has been built. An automatic Jocaml cost analyser based on the calculus produces cost equations parameterised on program variables in context, and may be used to find both cost in higher-order functions and the cost for the remainder of the program. Costed autonomous mobility skeleton (CAMSs) have been built, which not only encapsulate common patterns of autonomous mobility but take additional cost parameters to provide costs for the remainder of the program. Performance improvements are assessed by comparing CAMS to AMS programs. The results show that CAMS programs perform more effectively than AMS programs, because they have more accurate cost information. Hence a CAMS program may move to a faster location when the corresponding AMS program does not.

8.2 Further Work

AMPs on Large Scale Network

The current AMP experiments have been done on small local area networks. We would like to generalise the AMPs architecture to large scale network e.g. WAN, Grid, etc., where the communication time and system architecture are different from our current system. To generalise AMPs on large scale network, three activities should be considered.

- To propose a new architecture with super load servers.
- To improve the cost model.
- To evaluate the result.

Autonomous Mobility for Irregular Computation

Autonomous mobility skeletons have a limitation because the skeletons dynamically parameterise the cost model with measurements of performance on the preceding collection segment. If the program is reasonably regular, i.e. computing each segment of the collection represents a similar amount of work, then the cost model will be valid, and hence the movement decisions reasonable. However, as the computations become increasingly irregular, the cost model will be less valid, and hence the movement decisions may not optimise performance. We would like to generalise autonomous mobility skeleton to irregular problems with cost models and strategies to adapt to their behaviour.

Resource Driven Mobility

For further development of AMPs, we would like to build automatic resource driven mobility. We propose to investigate the application of a generic cost-based ethology to autonomous mobile multi-agent system. Specifically, we would like to use evolved biological foraging strategies to better engineer scalable self-organising resource-location systems in large-scale dynamic networks.

Cost Calculus for Java

We have developed an experimental automatic cost analyser, from a structural operational semantic execution time model, for a substantial Jocaml subset. Java is more mainstream and widely used than Jocaml, and it is imperative/object-oriented. We would like to build a calculus for Java to predict the costafter of Java programs at arbitrary points. For building a cost calculus for Java, there is significant challenge in making models, and hence analyses, of patterns and of object-oriented constructs, in particular in the presence of arbitrary inheritance.

Appendix A

Cost Calculus for $\mathcal{J}$

A cost calculus has been built for a very small language $\mathcal{J}'$ which is a subset of Jocaml. Language $\mathcal{J}$ extends $\mathcal{J}'$ with *null, boolean, list, tuple, pattern, compose, if, match*, and *fold*, and facilitates the construction of non-trival programs like matrix multiplication, and ray tracing in Section 6. In this appendix the cost sematics of $\mathcal{J}$ is given.

A.1 Syntax of $\mathcal{J}$

Figure A.1 shows the abstract syntax of $\mathcal{J}$. To simplify the presentation it is assumed that variables (id) names in the program are unique. In let, fun, and $match$ expressions, p is pattern, which is also expression. The syntax of pattern is shown in Figure A.1.

A.2 Cost Calculus for $\mathcal{J}$

A.2.1 Index Semantics

Figure A.2 shows the semantics of index for $\mathcal{J}$, where $\vdash_i : n \rightarrow e \rightarrow e * n$ is a function, which takes two parameters, an expression and an integer(i), and returns a tuple of index expression and another integer which is i increased by 1.

- Equation (A.1) shows that if the current index is i the expression is a constant c, after indexing the new expression is $< i, c >$ and the current index becomes $i + 1$.
- Equation (A.2) is similar to Equation (A.1), but the expression is a variable (id) instead of integer.
- Equation (A.3) decorates lambda expressions. In this equation the pattern does not need to be decorated. For indexing `fun` $p \rightarrow e$, if the current

$e ::=$		expression
	`null`	`null` expression
	c	constant
	id	variable
	`(`e...e`)`	tuple
	`[ ]`	empty list
	`[`e...e`]`	list
	`fun` $p \rightarrow e$	lambda (abstract)
	$e\ e$	application
	$e\ op\ e$	operation
	`let` p `=` e `in` e	`let`
	`if` e `then` e `else` e	condition
	`match` e `with` $p \rightarrow e\ \|\|...\|\|\ p \rightarrow e$	`match`
	`map` $e\ e$	`map` (higher order function)
	`fold` $e\ e\ e$	`fold` (higher order function)
	e `(*` e `*)`	user cost
	<n,e>	index i.e. expression e has index n, where n is an integer

p (pattern) ::= id | `(`p...p`)` | `[`p...p`]` | p`::`p

op ::= + | - | * | / |
> | < | >= | <= | = | ! = |
:: (cons) |
; (sequence)

Figure A.1: Syntax of $\mathcal{J}$

index is i, and after indexing e the current index is i', and e becomes e', the index lambda expression is $< i', \texttt{fun}\ p\ \rightarrow\ e' >$ and the current index is $i' + 1$.

- Equation (A.4) decorates application expressions ($e_1\ e_2$). If the current index is i, after indexing e_1 the current index becomes i' and e_1 becomes e_1', and then after indexing e_2, the current index is i'' and e_2 becomes e_2', then the index for the application expression is i'' and the current index becomes $i'' + 1$.

- Equation (A.5) decorates `let` expressions using the same rules as in Equation (A.4), Equation (A.6) decorates operation expressions, and Equation (A.7) decorates lists.

- Equation (A.8) indexes the empty list, which is similar to Equation (A.2).

- In Equation (A.9), user cost expressions e ($*\ c\ *$) are indexed, where only e has been indexed.

$$\frac{}{i \vdash_i c \Rightarrow_i (< i, c >, i+1)} \tag{A.1}$$

$$\frac{}{i \vdash_i id \Rightarrow_i (< i, id >, i+1)} \tag{A.2}$$

$$\frac{i \vdash_i e \Rightarrow_i (e', i')}{i \vdash_i \texttt{fun}\ p\ \rightarrow\ e \Rightarrow_i (< i', \texttt{fun}\ p\ \rightarrow\ e' >, i'+1)} \tag{A.3}$$

$$\frac{i \vdash_i e_1 \Rightarrow_i (e_1', i') \qquad i' \vdash_i e_2 \Rightarrow_i (e_2', i'')}{i \vdash_i (e_1\ e_2) \Rightarrow_i (< i'', (e_1'\ e_2') >, i''+1)} \tag{A.4}$$

$$\frac{i \vdash_i e_1 \Rightarrow_i (e_1', i') \qquad i' \vdash_i e_2 \Rightarrow_i (e_2', i'')}{i \vdash_i \texttt{let}\ p\ =\ e_1\ \texttt{in}\ e_2 \Rightarrow_i (< i'', \texttt{let}\ p\ =\ e_1'\ \texttt{in}\ e_2' >, i''+1)} \tag{A.5}$$

$$\frac{i \vdash_i e_1 \Rightarrow_i (e_1', i') \qquad i' \vdash_i e_2 \Rightarrow_i (e_2', i'')}{i \vdash_i e_1\ op\ e_2 \Rightarrow_i (< i'', e_1'\ op\ e_2' >, i''+1)} \tag{A.6}$$

$$\frac{i \vdash_i e_1 \Rightarrow_i (e_1', i') \qquad i' \vdash_i e_n \Rightarrow_i (e_n', i'')}{i \vdash_i [e_1\ ...\ e_n] \Rightarrow_i (< i'', [e_1'\ ...\ e_n'] >, i''+1)} \tag{A.7}$$

$$\frac{}{i \vdash_i [\ ] \Rightarrow_i (< i, [\ ] >, i+1)} \tag{A.8}$$

$$\frac{i \vdash_i e \Rightarrow_i (e', i')}{i \vdash_i e\ (*\ c\ *) \Rightarrow_i (< i', e'\ (*\ c\ *) >, i'+1)} \tag{A.9}$$

$$\frac{i \vdash_i e_1 \Rightarrow_i (e_1', i') \qquad i' \vdash_i e_2 \Rightarrow_i (e_2', i'')}{i \vdash_i \texttt{map}\ e_1\ e_2 \Rightarrow_i (< i'', \texttt{map}\ e_1'\ e_2' >, i''+1)} \tag{A.10}$$

$$\frac{i \vdash_i e_1 \Rightarrow_i (e_1', i') \qquad i' \vdash_i e_n \Rightarrow_i (e_n', i'')}{i \vdash_i (e_1\ ...\ e_n) \Rightarrow_i (< i'', (e_1'\ ...\ e_n') >, i''+1)} \tag{A.11}$$

$$\frac{i \vdash_i e_1 \Rightarrow_i (e_1', i') \qquad i' \vdash_i e_2 \Rightarrow_i (e_2', i'') \qquad i'' \vdash_i e_3 \Rightarrow_i (e_3', i''')}{i \vdash_i \texttt{fold}\ e_1\ e_2\ e_3 \Rightarrow_i (< i''', \texttt{fold}\ e_1'\ e_2'\ e_3' >, i'''+1)} \tag{A.12}$$

$$\frac{i \vdash_i e_1 \Rightarrow_i (e_1', i') \qquad i' \vdash_i e_2 \Rightarrow_i (e_2', i'') \qquad i'' \vdash_i e_3 \Rightarrow_i (e_3', i''')}{i \vdash_i \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow_i (< i''', \texttt{if}\ e_1'\ \texttt{then}\ e_2'\ \texttt{else}\ e_3' >, i'''+1)} \tag{A.13}$$

$$\frac{i \vdash_i e \Rightarrow_i (e', i') \qquad i' \vdash_i e_1 \Rightarrow_i (e_1', i'') \qquad i'' \vdash_i e_n \Rightarrow_i (e_n', i''')}{i \vdash_i \texttt{match}\ e\ \texttt{with}\ p_1 \rightarrow e_1 \| ... \| p_n \rightarrow e_n \Rightarrow_i (< i''', \texttt{match}\ e'\ \texttt{with}\ p_1 \rightarrow e_1' \| ... \| p_n \rightarrow e_n' >, i'''+1)} \tag{A.14}$$

$$\frac{}{i \vdash_i < i', e > \Rightarrow_i (< i', e >, i)} \tag{A.15}$$

Figure A.2: Index Semantics of $\mathcal{J}$

- In Equation (A.10), `map` expressions are indexed, which is similar to Equation (A.4).
- Equation (A.11) indexes tuples, which is similar to Equation (A.7).
- Equation (A.12), and Equation (A.13) index `fold` and `if` expressions, which are similar to Equation (A.4).
- Equation (A.14) indexes `match` expressions, which is similar to Equation (A.5) and Equation (A.3), where patterns are not indexed.
- If the expression is an index expression, it should not be indexed again. So in Equation (A.15), the return expression is the index expression, and the current index number is still i.

A.2.2 Cost Semantics

To calculate the cost of `let`, `fun`, and `match` expressions, the cost of the pattern should be calculated first, so this section first presents the pattern cost semantics, and then presents the cost semantics.

Pattern Cost Semantics of $\mathcal{J}$

Figure A.3 shows the semantics of pattern cost, where $\vdash_p : \mathit{env} \rightarrow p \rightarrow \mathit{env} * \mathit{cost}$, takes the environment (env) and an pattern expression, and returns a new environment which is the old environment extended with the pattern and its cost. (env) is a semantic domain, which stores the names of variable and the cost of the variable.

$$\frac{}{\mathrm{E} \vdash_p id \Rightarrow_p (\{id, 1\} + \mathrm{E}, 1)} \tag{A.16}$$

$$\frac{\mathrm{E} \vdash_p p_1 \Rightarrow_p (\mathrm{E}_1, c_1) \quad ... \quad \mathrm{E}_{n-1} \vdash_p p_n \Rightarrow_p (\mathrm{E}_n, c_n)}{\mathrm{E}_{n-1} \vdash_p (p_1 ... p_n) \Rightarrow_p (\mathrm{E}_n, c_1 + ... + c_n)} \tag{A.17}$$

$$\frac{}{\mathrm{E} \vdash_p [\ ] \Rightarrow_p (\mathrm{E}, 0)} \tag{A.18}$$

$$\frac{\mathrm{E} \vdash_p p_1 \Rightarrow_p (\mathrm{E}_1, c_1) \quad ... \quad \mathrm{E}_{n-1} \vdash_p p_n \Rightarrow_p (\mathrm{E}_n, c_n)}{\mathrm{E}_{n-1} \vdash_p [p_1 ... p_n] \Rightarrow_p (\mathrm{E}_n, n + c_1 + ... + c_n)} \tag{A.19}$$

$$\frac{\mathrm{E} \vdash_p p_1 \Rightarrow_p (\mathrm{E}_1, c_1) \quad \mathrm{E}_1 \vdash_p p_2 \Rightarrow_p (\mathrm{E}_2, c_2)}{\mathrm{E}_{n-1} \vdash_p (p_1 :: p_2) \Rightarrow_p (\mathrm{E}_2, 1 + c_1 + c_2)} \tag{A.20}$$

Figure A.3: Pattern Cost Semantics of $\mathcal{J}$

- Equation (A.16) shows that the pattern cost of id is 1. The semantic function $\vdash_p$ returns this cost, and at the same time updates the environment E with $id, 1$.

- Equation (A.17) shows the pattern cost of tuples. If the cost of the first element of tuple $(p_1...p_n)$ (p_1) is c_1, and the environment E becomes E_1, and the cost of the last element of the tuple (p_n) is c_n, and the environment is updated to E_n, then the pattern cost of the tuple is $c_1 + ... + c_n$, and the environment becomes E_n.
- Equation (A.18) shows that the pattern cost of the empty list is 0.
- Equation (A.19) shows the pattern cost of lists, which is similar to Equation (A.17).
- Equation (A.20) shows if two patterns p_1 and p_2 are combined together, the total cost is the pattern cost of p_1 (c_1) plus the pattern cost of p_2 (c_1) and plus 1, which is the cost for combining. The environment is updated as well.

Cost Semantics of $\mathcal{J}$

Figure A.4 shows the cost semantics of $\mathcal{J}$, where $\vdash_c : \ env \rightarrow e \rightarrow e * cost$ takes the environment (*env*) and an expression (e), and returns a tuple of the expression and the cost (*cost*) of the expression under the environment.

- Equation (A.21) infers that the cost of an constant is 0 in environment E.
- Equation (A.22) shows that the cost of the value of variable, here c, has been stored in the environment, if the cost of a variable is needed, we need to look up the environment $\{id, c\} + E$.
- Equation (A.23) performs the cost of operation expressions (e_1 *op* e_2). So if the cost of e_1 is c_1, and the cost of e_2 is c_2 then the cost of e_1 *op* e_2 is $1 + c_1 + c_2$.
- Equation (A.24) performs the cost of lambda expressions `fun` $p \ \rightarrow \ e$, which have two part, the cost of pattern p (c_p), and the cost of the body e. The cost of pattern p can be get using the pattern cost semantics. Under the environment E if the cost of e is c, then the cost of `fun` $p \ \rightarrow \ e$ is $c_p + c$.
- Equation (A.25) shows that under environment E the cost of `let` expression (`let` $p \ = \ e_1$ `in` e_2) has three parts, the cost of e_1 (c_1), the cost of e_2 (c_2), and the cost of p (c_p).
- Equation (A.26) shows that the cost of application expression ($e_1 \ \ e_2$) is the cost of e_1 (c_1) and the cost of e_2 (c_2).
- In any environment E the cost of empty list [ ] is 0, as shown in Equation (A.27).
- Equation (A.28) shows that the cost of list $[e_1...e_n]$ is the sum of the costs of every elements in the list.

$$\frac{}{\mathrm{E} \vdash_c \ c \,\$\, 0} \tag{A.21}$$

$$\frac{}{\{id, c\} + \mathrm{E} \vdash_c \ id \,\$\, c} \tag{A.22}$$

$$\frac{\mathrm{E} \vdash_c \ e_1 \,\$\, c_1 \qquad \mathrm{E} \vdash_c \ e_2 \,\$\, c_2}{\mathrm{E} \vdash_c \ e_1 \ op \ e_2 \,\$\, 1 + c_1 + c_2} \tag{A.23}$$

$$\frac{\mathrm{E} \vdash_p \ p \Rightarrow_p \ (\mathrm{E}_1, c_p) \qquad \mathrm{E}_1 \vdash_c \ e \,\$\, c}{\mathrm{E} \vdash_c \ \texttt{fun} \ p \ \rightarrow \ e \,\$\, c_p + c} \tag{A.24}$$

$$\frac{\mathrm{E} \vdash_p \ p \Rightarrow_p \ (\mathrm{E}_1, c_p) \qquad \mathrm{E}_1 \vdash_c \ e_1 \,\$\, c_1 \qquad \mathrm{E} \vdash_c \ e_2 \,\$\, c_2}{\mathrm{E} \vdash_c \ \texttt{let} \ p \ = \ e_1 \ \texttt{in} \ e_2 \,\$\, c_p + c_1 + c_2} \tag{A.25}$$

$$\frac{\mathrm{E} \vdash_c \ e_1 \,\$\, c_1 \qquad \mathrm{E} \vdash_c \ e_2 \,\$\, c_2}{\mathrm{E} \vdash_c \ (e_1 \ e_2) \,\$\, c_1 + c_2} \tag{A.26}$$

$$\frac{}{\mathrm{E} \vdash_c \ [\ ] \,\$\, 0} \tag{A.27}$$

$$\frac{\mathrm{E} \vdash_c \ e_1 \,\$\, c_1 \qquad ... \qquad \mathrm{E} \vdash_c \ e_n \,\$\, c_n}{\mathrm{E} \vdash_c \ [e_1 ... e_n] \,\$\, c_1 + ... + c_n} \tag{A.28}$$

$$\frac{}{\mathrm{E} \vdash_c \ e \ (* \ c \ *) \,\$\, c} \tag{A.29}$$

$$\frac{\mathrm{E} \vdash_c \ e_1 \,\$\, c_1 \qquad \mathrm{E} \vdash_c \ e_2 \,\$\, c_2}{\mathrm{E} \vdash_c \ \texttt{map} \ e_1 \ e_2 \,\$\, c_1 * (length \ e_2) + c_2} \tag{A.30}$$

$$\frac{\mathrm{E} \vdash_c \ e_1 \,\$\, c_1 \qquad ... \qquad \mathrm{E} \vdash_c \ e_n \,\$\, c_n}{\mathrm{E} \vdash_c \ (e_1 ... e_n) \,\$\, c_1 + ... + c_n} \tag{A.31}$$

$$\frac{\mathrm{E} \vdash_c \ e_1 \,\$\, c_1 \qquad \mathrm{E} \vdash_c \ e_2 \,\$\, c_2 \qquad \mathrm{E} \vdash_c \ e_3 \,\$\, c_3}{\mathrm{E} \vdash_c \ \texttt{fold} \ e_1 \ e_2 \ e_3 \,\$\, c_1 * (length \ e_2) + c_2 + c_3} \tag{A.32}$$

$$\frac{\mathrm{E} \vdash_c \ e_1 \,\$\, c_1 \qquad \mathrm{E} \vdash_c \ e_2 \,\$\, c_2 \qquad \mathrm{E} \vdash_c \ e_3 \,\$\, c_3}{\mathrm{E} \vdash_c \ \texttt{if} \ e_1 \ \texttt{then} \ e_2 \ \texttt{else} \ e_3 \,\$\, c_1 + max(c_2 + c_3)} \tag{A.33}$$

$$\frac{\mathrm{E} \vdash_c \ e \,\$\, c \ \ \mathrm{E} \vdash_p \ p_1 \Rightarrow_p \ (\mathrm{E}_1, c_1) \ \ \mathrm{E}_1 \vdash_c \ e_1 \,\$\, c_1' \ \ ... \ \ \mathrm{E}_n \vdash_p \ p_n \Rightarrow_p \ (\mathrm{E}_n, c_n) \ \ \mathrm{E}_n \vdash_c \ e_n \,\$\, c_n'}{\mathrm{E} \vdash_c \ \texttt{match} \ e \ \texttt{with} \ p_1 \rightarrow e_1 \| ... \| p_n \rightarrow e_n \,\$\, c + c_1 + ... + c_n + max(c_1' ... c_n')} \tag{A.34}$$

$$\frac{\mathrm{E} \vdash_c \ e \,\$\, c}{\mathrm{E} \vdash_c < i, e > \ \$\, c} \tag{A.35}$$

Figure A.4: Cost Semantics of $\mathcal{J}$

- Equation (A.29) enables user specified costs. In particular, it is difficult to get the static cost of recursive functions. So in e $(*\ c\ *)$, the cost of e is c which is calculated by hand rather by the cost analyser automatically.

- Equation (A.30) infers the cost of a `map` expression. Here only the regular cases of `map` are considered. Under this condition function e_1 applied to every elements of list e_2 has the same cost. So the total cost of a `map` expression is $c_1 * (length\ e_2) + c_2$, where c_1 is the cost of function e_1 applied to the first element of list e_2.

- Equation (A.31) shows the cost of tuples, which is similar to the cost of lists in Equation (A.28).

- Equation (A.32) shows the cost of `fold` expressions, which is similar to the cost of `map` expressions in Equation (A.30).

- Equation (A.33) finds the cost of `if` expression `if` e_1 `then` e_2 `else` e_3. If the cost of e_1 is c_1, the cost of e_2 is c_2, and the cost of e_3 is c_3, then the cost of the if expression is c_1 plus the maximum of c_2 and c_3.

- Equation (A.34) finds the cost of expression `match` e `with` $p_1 \rightarrow e_1 \parallel ... \parallel p_n \rightarrow e_n$. Firstly the cost of patterns from p_1 to p_n are calculated as c_1 to c_n, and at the same time the environment has been updated according to the pattern semantics in Figure A.3. Then the cost of each expression from e_1 to e_n should be got as c'_1 to c'_n. The cost of the `match` expression is the sum of c_1 to c_n plus the maximum of c'_1 to c'_n.

- Equation (A.35) shows that under environment E the cost of an index expression $< i, e >$ is the cost of expression e (c).

A.2.3 Costafter Semantics

This section introduces the costafter semantics of $\mathcal{J}$. In costafter, definitions of expression equality and expression contains will be used.

Expression Equality

Figure A.5 shows the definition of expression equality, where $\equiv : e \rightarrow e \rightarrow$ *boolean* applies expression equality, which checks if two expressions are the same.

- Equation (A.36) defines constant expression equality. If expression e is an constant and its value is equal to c, so expression e is equal to expression c.

- Equation (A.37) defines variable equality. In $\mathcal{J}$, it is assumed that variables (id) names in the program are unique, so if two variables expressions have the same name, the two expression are equal.

$$\frac{e = c}{e \equiv c} \tag{A.36}$$

$$\frac{e = id}{e \equiv id} \tag{A.37}$$

$$\frac{e_1 \equiv e_3 \qquad e_2 \equiv e_4}{(e_1\ op\ e_2) \equiv (e_3\ op\ e_4)} \tag{A.38}$$

$$\frac{p_1 \equiv p_2 \qquad e_1 \equiv e_2}{(\texttt{fun}\ p_1 \rightarrow e_1) \equiv (\texttt{fun}\ p_2 \rightarrow e_2)} \tag{A.39}$$

$$\frac{e_1 \equiv e_3 \qquad e_2 \equiv e_4}{(e_1\ e_2) \equiv (e_3\ e_4)} \tag{A.40}$$

$$\frac{p_1 \equiv p_2 \qquad e_1 \equiv e_3 \qquad e_2 \equiv e_4}{(\texttt{let}\ p_1\ =\ e_1\ \texttt{in}\ e_2) \equiv (\texttt{let}\ p_2\ =\ e_3\ \texttt{in}\ e_4)} \tag{A.41}$$

$$\frac{e_1 \equiv e_3 \qquad e_2 \equiv e_4}{(\texttt{map}\ e_1\ e_2) \equiv (\texttt{map}\ e_3\ e_4)} \tag{A.42}$$

$$\frac{e_i \equiv e'_i \qquad i = 1...n}{[e_1...e_n] \equiv [e'_1...e'_n]} \tag{A.43}$$

$$\frac{}{[\ ] \equiv [\ ]} \tag{A.44}$$

$$\frac{e_i \equiv e'_i \qquad i = 1...n}{(e_1...e_n) \equiv (e'_1...e'_n)} \tag{A.45}$$

$$\frac{e_1 \equiv e_4 \qquad e_2 \equiv e_5 \qquad e_3 \equiv e_6}{(\texttt{fold}\ e_1\ e_2\ e_3) \equiv (\texttt{fold}\ e_4\ e_5\ e_6)} \tag{A.46}$$

$$\frac{e_1 \equiv e_4 \qquad e_2 \equiv e_5 \qquad e_3 \equiv e_6}{(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) \equiv (\texttt{if}\ e_4\ \texttt{then}\ e_5\ \texttt{else}\ e_6)} \tag{A.47}$$

$$\frac{e \equiv e' \qquad p_i \equiv p'_i \qquad e_i \equiv e'_i \qquad i = 1..n}{(\texttt{match}\ e\ \texttt{with}\ p_1 \rightarrow e_1 \| ... \| p_n \rightarrow e_n) \equiv (\texttt{match}\ e'\ \texttt{with}\ p'_1 \rightarrow e'_1 \| ... \| p'_n \rightarrow e'_n)} \tag{A.48}$$

$$\frac{e_1 \equiv e_2 \qquad c_1 \equiv c_2}{(e_1\ (*\ c_1\ *)) \equiv (e_2\ (*\ c_2\ *))} \tag{A.49}$$

$$\frac{}{< i, e_1 > \equiv < i, e_2 >} \tag{A.50}$$

Figure A.5: Definition of Expression Equality in $\mathcal{J}$

- Equation (A.38) defines the equality of two operation expressions. Expression (e_1 *op* e_2) equals expression (e_3 *op* e_4), if e_1 equals e_3 and e_2 equals e_4.
- Equation (A.39) defines the equality of two lambda expression. Expression `fun` ($p_1 \rightarrow e_1$) equals expression `fun` ($p_2 \rightarrow e_2$), if pattern p_1 equals pattern p_2 and expression e_1 equals e_2. The patterns are also expressions, so the expression equality can also be used for patterns.
- Equation (A.40), Equation (A.41), and Equation (A.42) define application expressions, let expressions and `map` expressions equality, which are all similar to Equation (A.39).
- Equation (A.43) shows that two list expressions are equal, if the elements in one list are equal to the elements in the other list.
- Equation (A.44) shows that empty lists are always equal.
- Equation (A.45) defines expression equality for tuples, which is similar to list equality in Equation (A.43).
- Equation (A.46) defines expression equality for `fold` expressions, which is similar to Equation (A.39).
- Equation (A.47) defines expression equality for if expressions,
- and Equation (A.48) defines expression equality for `match` expressions, which are all similar to Equation (A.39).
- In Equation (A.49), two user cost expression are equal, if both the expression parts and cost parts of the expressions are equal.
- In the calculus, the whole program has been indexed, so every expression in a program has a unique integer as an index. Thus two index expressions are equal to each other only if their indices are equal.

Expression Contains

Figure A.6 and A.7 give the definition of contains, where $\in$ takes two expression. If the second expression contains the first expression it returns true, or it returns false if not.

- Equation (A.51) identifies that if expression e_1 is equal to expression e_2 then the two expressions contain each other.
- Equation (A.52) shows that if e_1 is contained in e_2, then e_1 is contained in `fun` $p \rightarrow e2$.
- Equation (A.53a) to (A.63c) give the definition of contains for different expressions, which are all similar to Equation (A.52).

$$\frac{e_1 \equiv e_2}{e_1 \in e_2} \tag{A.51}$$

$$\frac{e_1 \in e_2}{e_1 \in \texttt{fun}\ p\ \rightarrow e_2} \tag{A.52}$$

$$\frac{e_1 \in e_2}{e_1 \in (e_2\ e_3)} \tag{A.53a}$$

$$\frac{e_1 \in e_3}{e_1 \in (e_2\ e_3)} \tag{A.53b}$$

$$\frac{e_1 \in e_2}{e_1 \in \texttt{let}\ p\ =\ e_2\ \texttt{in}\ e_3} \tag{A.54a}$$

$$\frac{e_1 \in e_3}{e_1 \in \texttt{let}\ p\ =\ e_2\ \texttt{in}\ e_3} \tag{A.54b}$$

$$\frac{e_1 \in e_2}{e_1 \in e_2\ op\ e_3} \tag{A.55a}$$

$$\frac{e_1 \in e_3}{e_1 \in e_2\ op\ e_3} \tag{A.55b}$$

$$\frac{e_1 \in e_i \qquad i = 2..n}{e_1 \in [e_2\ ...\ e_n]} \tag{A.56}$$

$$\frac{e_1 \in e_2}{e_1 \in e_2\ (*\ c\ *)} \tag{A.57}$$

$$\frac{e_1 \in e_2}{e_1 \in \texttt{map}\ e_2\ e_3} \tag{A.58a}$$

$$\frac{e_1 \in e_3}{e_1 \in \texttt{map}\ e_2\ e_3} \tag{A.58b}$$

$$\frac{e_1 \in e_2}{e_1 \in < i, e_2 >} \tag{A.59}$$

$$\frac{e_1 \in e_i \qquad i = 2..n}{e_1 \in (e_2\ ...\ e_n)} \tag{A.60}$$

$$\frac{e_1 \in e_2}{e_1 \in \texttt{fold}\ e_2\ e_3\ e_4} \tag{A.61a}$$

$$\frac{e_1 \in e_3}{e_1 \in \texttt{fold}\ e_2\ e_3\ e_4} \tag{A.61b}$$

$$\frac{e_1 \in e_4}{e_1 \in \texttt{fold}\ e_2\ e_3\ e_4} \tag{A.61c}$$

Figure A.6: Definition of Contains in $\mathcal{J}$

$$\frac{e_1 \in e_2}{e_1 \in \texttt{if } e_2 \texttt{ then } e_3 \texttt{ else } e_4} \tag{A.62a}$$

$$\frac{e_1 \in e_3}{e_1 \in \texttt{if } e_2 \texttt{ then } e_3 \texttt{ else } e_4} \tag{A.62b}$$

$$\frac{e_1 \in e_4}{e_1 \in \texttt{if } e_2 \texttt{ then } e_3 \texttt{ else } e_4} \tag{A.62c}$$

$$\frac{e_1 \in e_2}{e_1 \in \texttt{match } e_2 \texttt{ with } p_3 \rightarrow e_3 \| ... \| p_n \rightarrow e_n} \tag{A.63a}$$

$$\frac{e_1 \in p_i \qquad i = 3...n}{e_1 \in \texttt{match } e_2 \texttt{ with } p_3 \rightarrow e_3 \| ... \| p_n \rightarrow e_n} \tag{A.63b}$$

$$\frac{e_1 \in e_i \qquad i = 3...n}{e_1 \in \texttt{match } e_2 \texttt{ with } p_3 \rightarrow e_3 \| ... \| p_n \rightarrow e_n} \tag{A.63c}$$

Figure A.7: Definition of Contains in $\mathcal{J}$ Cont.

Semantics of Costafter

Figure A.8 and A.9 show the semantics of costafter of one expression e.g e in different expressions, where $\vdash_a : \mathit{env} \rightarrow e \rightarrow e \rightarrow \mathit{cost}$ takes the environment (env) and two expressions and returns a cost, which is the costafter of the first expression in the second expression.

(1) Equation (A.64) states that if expression e is equal to e' then the costafter of e in e' is 0.

(2) Equations (A.65a) and (A.65b) define the costafter of e in lambda expressions. If the costafter of e in e_1 is c, then the costafter of e in lambada expression $\texttt{fun } p \rightarrow e_1$ is c too. If e_1 does not contains e then the costafter of e in $\texttt{fun } p \rightarrow e_1$ is c is 0.

(3) Equations (A.66a), (A.66b), and (A.66c) define the costafter of e in application expressions ($(e_1\ e_2)$). If e_1 contains e, then the costafter of e in $(e_1\ e_2)$ is the costafter of e in e_1, here c_1, plus the cost of e_2, here c_2. If e_2 contains e then the costafter of e in $(e_1\ e_2)$ is the cost after e in e_2. If e is not contained in e_1 or e_2, then the costafter of e in $(e_1\ e_2)$ is 0.

(4) Similar to the equations in rule (3), Equation (A.67a), (A.67b), (A.67c), and (A.67d) define the costafter of e in `let` expression $\mathit{let}\ p\ = e_1\ \mathit{in}\ e_2$.

(5) Equations (A.68a), (A.68b), and (A.68c) define the costafter of e in operation expressions e.g. $(e_1\ \mathit{op}\ e_2)$. If e_1 contains e, then the costafter of e in $(e_1\ e_2)$ is the costafter of e in e_1 (c_1), plus the cost of e_2 (c_2) plus 1, which is the cost for getting the operator, see Equation (A.68b).

(6) Equation (A.69a) defines the costafter of e in list $[e_1...e_n]$, if e is contained in e_i (i=1..n), then the costafter of e in the list is the costafter of e in e_i, plus the costs of all elements in the list after e_i $(e_{i+1}...e_n)$. If e is not

$$\frac{e \ \equiv \ e'}{\mathrm{E} \vdash_a \ e \trianglelefteq e' \pounds 0} \tag{A.64}$$

$$\frac{\mathrm{E} \vdash_a \ e \trianglelefteq e_1 \pounds c}{\mathrm{E} \vdash_a \ e \trianglelefteq \texttt{fun}\ p \ \rightarrow \ e_1 \pounds c} \tag{A.65a}$$

$$\frac{}{\mathrm{E} \vdash_a \ e \trianglelefteq \texttt{fun}\ p \ \rightarrow \ e_1 \pounds 0} \tag{A.65b}$$

$$\frac{e \ \in \ e_1 \qquad \mathrm{E} \vdash_a \ e \trianglelefteq e_1 \pounds c_1 \qquad \mathrm{E} \vdash_c \ e_2 \$ c_2}{\mathrm{E} \vdash_a \ e \trianglelefteq (e_1 \ e_2) \pounds c_1 + c_2} \tag{A.66a}$$

$$\frac{e \ \in \ e_2 \qquad \mathrm{E} \vdash_a \ e \trianglelefteq e_2 \pounds c_2}{\mathrm{E} \vdash_a \ e \trianglelefteq (e_1 \ e_2) \pounds c_2} \tag{A.66b}$$

$$\frac{}{\mathrm{E} \vdash_a \ e \trianglelefteq (e_1 \ e_2) \pounds 0} \tag{A.66c}$$

$$\frac{\mathit{notFun}\ e_1 \qquad e \ \in \ e_1 \qquad \mathrm{E} \vdash_a \ e \trianglelefteq e_1 \pounds c_1 \qquad \mathrm{E} \vdash_c \ e_2 \$ c_2}{\mathrm{E} \vdash_a \ e \trianglelefteq \texttt{let}\ p \ = e_1 \ \texttt{in}\ e_2 \pounds c_1 + c_2} \tag{A.67a}$$

$$\frac{\mathit{notFun}\ e_1 \qquad e \ \in \ e_2 \qquad \mathrm{E} \vdash_a \ e \trianglelefteq e_2 \pounds c_2}{\mathrm{E} \vdash_a \ e \trianglelefteq \texttt{let}\ p \ = e_1 \ \texttt{in}\ e_2 \pounds c_2} \tag{A.67b}$$

$$\frac{e \ \in \ e_2 \qquad \mathrm{E} \vdash_a \ e \trianglelefteq e_2 \pounds c_2}{\mathrm{E} \vdash_a \ e \trianglelefteq \texttt{let}\ p \ = e_1 \ \texttt{in}\ e_2 \pounds c_2} \tag{A.67c}$$

$$\frac{}{\mathrm{E} \vdash_a \ e \trianglelefteq \texttt{let}\ p \ = e_1 \ \texttt{in}\ e_2 \pounds 0} \tag{A.67d}$$

$$\frac{e \ \in \ e_1 \qquad \mathrm{E} \vdash_a \ e \trianglelefteq e_1 \pounds c_1 \qquad \mathrm{E} \vdash_c \ e_2 \$ c_2}{\mathrm{E} \vdash_a \ e \trianglelefteq (e_1 \ \mathit{op}\ e_2) \pounds 1 + c_1 + c_2} \tag{A.68a}$$

$$\frac{e \ \in \ e_2 \qquad \mathrm{E} \vdash_a \ e \trianglelefteq e_2 \pounds c_2}{\mathrm{E} \vdash_a \ e \trianglelefteq (e_1 \ \mathit{op}\ e_2) \pounds c_2 + 1} \tag{A.68b}$$

$$\frac{}{\mathrm{E} \vdash_a \ e \trianglelefteq (e_1 \ \mathit{op}\ e_2) \pounds 0} \tag{A.68c}$$

$$\frac{e \ \in \ e_i \qquad \mathrm{E} \vdash_a \ e \trianglelefteq e_i \pounds c_i \qquad \mathrm{E} \vdash_c \ e_{i+1} \$ c_{i+1} \qquad ... \qquad \mathrm{E} \vdash_c \ e_n \$ c_n}{\mathrm{E} \vdash_a \ e \trianglelefteq [e_1 \ ... \ e_n] \pounds c_i + \ ... \ + c_n} \tag{A.69a}$$

$$\frac{}{\mathrm{E} \vdash_a \ e \trianglelefteq [e_1 \ ... \ e_2] \pounds 0} \tag{A.69b}$$

$$\frac{}{\mathrm{E} \vdash_a \ e \trianglelefteq [\ ] \pounds 0} \tag{A.70}$$

$$\frac{}{\mathrm{E} \vdash_a \ e \trianglelefteq e_1 \ (* \ c \ *) \pounds 0} \tag{A.71}$$

Figure A.8: Costafter Semantics of $\mathcal{J}$

$$\frac{e \in e_1 \qquad \mathrm{E} \vdash_c \texttt{map}\ e_1\ e_2 \,\$\, c \qquad \mathrm{E} \vdash_a e \trianglelefteq e_1 \,\pounds\, c_1 \qquad \mathrm{E} \vdash_c e_2 \,\$\, c_2}{\mathrm{E} \vdash_a e \trianglelefteq \texttt{map}\ e_1\ e_2 \,\pounds\, c + c_1 + c_2} \tag{A.72a}$$

$$\frac{e \in e_2 \qquad \mathrm{E} \vdash_c \texttt{map}\ e_1\ e_2 \,\$\, c \qquad \mathrm{E} \vdash_a e \trianglelefteq e_2 \,\pounds\, c_2}{\mathrm{E} \vdash_a e \trianglelefteq \texttt{map}\ e_1\ e_2 \,\pounds\, c + c_2} \tag{A.72b}$$

$$\frac{}{\mathrm{E} \vdash_a e \trianglelefteq \texttt{map}\ e_1\ e_2 \,\pounds\, 0} \tag{A.72c}$$

$$\frac{\mathrm{E} \vdash_a e \trianglelefteq e_1 \,\pounds\, c}{\mathrm{E} \vdash_a e \trianglelefteq <i, e_1> \,\pounds\, c} \tag{A.73}$$

$$\frac{e \in e_i \qquad \mathrm{E} \vdash_a e \trianglelefteq e_i \,\pounds\, c_i \qquad \mathrm{E} \vdash_c e_{i+1} \,\$\, c_{i+1} \qquad ... \qquad \mathrm{E} \vdash_c e_n \,\$\, c_n}{\mathrm{E} \vdash_a e \trianglelefteq (e_1\ ...\ e_n) \,\pounds\, c_i + \ ...\ + c_n} \tag{A.74a}$$

$$\frac{}{\mathrm{E} \vdash_a e \trianglelefteq (e_1\ ...\ e_n) \,\pounds\, 0} \tag{A.74b}$$

$$\frac{e \in e_1 \qquad \mathrm{E} \vdash_a e \trianglelefteq e_1 \,\pounds\, c_1 \qquad \mathrm{E} \vdash_c e_2 \,\$\, c_2 \qquad \mathrm{E} \vdash_c e_3 \,\$\, c_3}{\mathrm{E} \vdash_a e \trianglelefteq \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \,\pounds\, c_1 + max(c_2, c_3)} \tag{A.75a}$$

$$\frac{e \in e_2 \qquad \mathrm{E} \vdash_a e \trianglelefteq e_2 \,\pounds\, c_2}{\mathrm{E} \vdash_a e \trianglelefteq \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \,\pounds\, c_2} \tag{A.75b}$$

$$\frac{e \in e_3 \qquad \mathrm{E} \vdash_a e \trianglelefteq e_3 \,\pounds\, c_3}{\mathrm{E} \vdash_a e \trianglelefteq \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \,\pounds\, c_3} \tag{A.75c}$$

$$\frac{}{\mathrm{E} \vdash_a e \trianglelefteq \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \,\pounds\, 0} \tag{A.75d}$$

$$\frac{e \in e_1 \qquad \mathrm{E} \vdash_c \texttt{fold}\ e_1\ e_2\ e_3 \,\$\, c \qquad \mathrm{E} \vdash_a e \trianglelefteq e_1 \,\pounds\, c_1 \qquad \mathrm{E} \vdash_c e_2 \,\$\, c_2 \qquad \mathrm{E} \vdash_c e_3 \,\$\, c_3}{\mathrm{E} \vdash_a e \trianglelefteq \texttt{fold}\ e_1\ e_2\ e_3 \,\pounds\, c + c_1 + c_2 + c_3} \tag{A.76a}$$

$$\frac{e \in e_2 \qquad \mathrm{E} \vdash_c \texttt{fold}\ e_1\ e_2\ e_3 \,\$\, c \qquad \mathrm{E} \vdash_a e \trianglelefteq e_2 \,\pounds\, c_2 \qquad \mathrm{E} \vdash_c e_3 \,\$\, c_3}{\mathrm{E} \vdash_a e \trianglelefteq \texttt{fold}\ e_1\ e_2\ e_3 \,\pounds\, c + c_2 + c_3} \tag{A.76b}$$

$$\frac{e \in e_3 \qquad \mathrm{E} \vdash_c \texttt{fold}\ e_1\ e_2\ e_3 \,\$\, c \qquad \mathrm{E} \vdash_a e \trianglelefteq e_3 \,\pounds\, c_3}{\mathrm{E} \vdash_a e \trianglelefteq \texttt{fold}\ e_1\ e_2\ e_3 \,\pounds\, c + c_3} \tag{A.76c}$$

$$\frac{}{\mathrm{E} \vdash_a e \trianglelefteq \texttt{fold}\ e_1\ e_2\ e_3 \,\pounds\, 0} \tag{A.76d}$$

$$\frac{e \in e' \qquad \mathrm{E} \vdash_a e \trianglelefteq e' \,\pounds\, c' \qquad \mathrm{E} \vdash_p p_i \Rightarrow_p (\mathrm{E}_i, c_i) \qquad \mathrm{E} \vdash_c e_i \,\$\, c'_i \qquad i = 1..n}{\mathrm{E} \vdash_a e \trianglelefteq \texttt{match}\ e'\ \texttt{with}\ p_1 \rightarrow e_1\ \|...\|\ p_n \rightarrow e_n \,\pounds\, c' + c_1 + ... + c_n + max(c'_1 ... c'_n)} \tag{A.77a}$$

$$\frac{e \in e_i \qquad \mathrm{E} \vdash_a e \trianglelefteq e_i \,\pounds\, c_i}{\mathrm{E} \vdash_a e \trianglelefteq \texttt{match}\ e'\ \texttt{with}\ p_1 \rightarrow e_1\ \|...\|\ p_n \rightarrow e_n \,\pounds\, c_i} \tag{A.77b}$$

$$\frac{}{\mathrm{E} \vdash_a e \trianglelefteq \texttt{match}\ e'\ \texttt{with}\ p_1 \rightarrow e_1\ \|...\|\ p_n \rightarrow e_n \,\pounds\, 0} \tag{A.77c}$$

Figure A.9: Costafter Semantics of $\mathcal{J}$ Cont.

contained in any element in the list, the costafter of e in the list is 0. See Equation (A.69b).

(7) According to Equation (A.70), the costafter of any expression e in empty lists is 0.

(8) The costafter of any expression in a user cost expressions is 0, see Equation (A.71).

(9) Equations (A.72a), (A.72b), and (A.72c) define the costafter of e in `map` expression (`map` e_1 e_2). Equation (A.72a) shows that if e_1 contains e, then the costafter of e in `map` e_1 e_2 is the costafter of e in e_1, plus the cost of e_2, plus the cost of the `map` expression. The similar situation of e_2 containing e is defined in Equation (A.72b). Equation (A.72c) shows if e is not contained in e_1 or e_2 then the costafter of e in `map` e_1 e_2 is 0.

(10) Equation (A.73) shows that the costafter of e in index expression $< i, e_1 >$ is the same as the costafter of e in e_1.

(11) Equation (A.74a) and Equation (A.74b) define the costafters in tuples, which are similar to the equations for lists in Equation (A.69a) and (A.69b).

(12) Equation (A.75a) to Equation (A.75d) define the costafter of e in `if` expressions. If e is in e_1 and the the costafter of e in e_1 is c_1, then the costafter of e is c plus the maximum cost of e_2 (c_2) and e_3 (c_3). See Equation (A.75a). If e is in e_2 and the the costafter of e in e_2 is c_2, then the costafter of e is c_2. See Equation (A.75b). Similarly, if e is in e_3 and the the costafter of e in e_3 is c_3, then the costafter of e is c_3. See Equation (A.75c).

(13) Equation (A.76a) to Equation (A.76d) gives the costafter e in `fold` expressions, which is similar to the costafter in *map* expression in Equation (A.72a), (A.72b), and (A.72c).

(14) Equation (A.77a) to Equation (A.77c) define the costafters of e in `match` expressions, which are similar to the costafter in `if` expression in Equation (A.75a), (A.75b), (A.75c), and (A.75d).

Bibliography

[1] J.H. Abawajy. Autonomic Job Scheduling Policy for Grid Computing. In *Lecture Notes in Computer Science, LNCS 3516*, pages 213–220, Germany, May 2005. Internation Conference on Computationl Science - ICCS 2005, part 3, Springer.

[2] L. Cardelli. Abstrations for Mobile Computation. *Secure Internet Programming*, pages 51–94, 1999.

[3] T.L. Casavant and J.G. Kuhl. A Taxonomy of Scheduling in General-Pupose distributed Computing Systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1988.

[4] Jacques Cohen and Carl Zuckerman. Two Languages for Estimating Program Efficiency. *Commun. ACM*, 17(6):301–308, 1974.

[5] Xiao Yan Deng, Greg Michaelson, and Phil Trinder. Autonomous Mobility Skeletons. *Journal of Parallel Computing*, Volume 32, Issues 7-8:Pages 463–478 Algorithmic Skeletons, September 2006.

[6] Xiao Yan Deng, Phil Trinder, and Greg Michaelson. 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)*, pages 177–186, Hong Kong, December 2006. IEEE Computer Society, Washington, DC, USA.

[7] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organization . *International Journal of High Performance Computing Applications*, 15:200–222, 2001.

[8] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press, 1992.

[9] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[10] Institut National de Recherche en Informatique et en Automatique. *The JoCaml language beta release: Documentation and user's manual*, January 2001.

[11] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[12] Z.D. Kirli. *Mobile Computation with Functions*. PhD thesis, University of Edinburgh, Laboratory for Foundations of Computer Science:Division of Informatics, 2001.

[13] Danny B. Lange and Mitsuru Oshima. Seven good reasons for mobile agents. *Commun. ACM*, 42(3):88–89, 1999.

[14] H-W Loidl. *Granularity in Large-Scale Parallel Functional Programming*. PhD thesis, University of Glasgow, April 1998. Department of Computing Science.

[15] Dejan Milojicic, Frederick Douglis, and Richard Wheeler. *Mobility: processes, computers, and agents*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.

[16] Richard Murch. *Autonomic Computing*. Published by IBM Press, 1st edition, March 2004.

[17] Lyle Harold Ramshaw. *Formalizing the analysis of algorithms.* PhD thesis, 1979.

[18] R. Rangaswami. *A Cost Analysis for a Higher-order Parallel Programming Model*. PhD thesis, Department of Computer Science, Edinburgh University, 1996.

[19] Recursion Software, Inc, 2591 North Dallas Parkway, Suite 200, Frisco, TX 75034. *Voyager User Guide*, May 2005. http://www.recursionsw.com/Voyager/Voyager_User_Guide.pdf.

[20] Brian Reistad and David K. Gifford. Static dependent costs for estimating execution time. In *LFP '94: Proceedings of the 1994 ACM conference on LISP and Functional Programming*, pages 65–78, Orlando, Florida, United States, 1994. ACM Press New York, NY, USA.

[21] Mads Rosendahl. Automatic Complexity Analysis. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 144–156, Imperial College, London, United Kingdom, 1989. ACM Press New York, NY, USA.

[22] Sartaj Sahni. *Data Structures, Algorithms, and Applications in Java*. Mc Graw Hill, University of Florida, 2000.

[23] Tatsurou Sekiguchi. JavaGo, May 2006. http://homepage.mac.com/t.sekiguchi/ javago/index.html.

[24] David. B. Skillicorn. *Parallelism and the Bird-Meertens Formalism*. Department of Computing and Information Science, Queen's University, Kingston, Ontario, 1992.

[25] Predrag T. Tosic and Gul A. Agha. Towards a Hierarchical Taxonomy of Autonomous Agents. In *IEEE SMC'2004: International Conference on Systems, Man and Cybernetics*, pages 3421–3426, Hague, The Netherlands, October 2004. IEEE Xplore.

[26] Carlos Varela Travis Desell, Kaoutar El Maghraoui. Load Balancing of Autonomous Actors over Dynamic Networks. page 90268.1, 2004.

[27] P. Wadler. Strictness Analysis Aids Time Analysis. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–132, San Diego, California, United States, 1988. ACM Press, New York, USA.

[28] Ben Wegbreit. Mechanical Program Analysis. *Commun. ACM*, 18(9):528–539, 1975.

[29] Ben Wegbreit. Verifying program performance. *J. ACM*, 23(4):691–699, 1976.

[30] Thomas Wheeler. Voyager Architecture Best Practices. Technical report, Recursion Software, March 2005. http://www.recursionsw.com/Voyager/2005-03-31-Voyager_Architecture_Best_Practices.pdf.

[31] M. Wooldridge. Agent-Based Software Engineering. *IEE Proceedings Software Engineering*, 144(1):26–37, 1997.