A Sized Time System for a Parallel Functional Language

Hans-Wolfgang Loidl
Department of Computing Science, University of Glasgow
Glasgow, Scotland, U.K.
E-mail: hwloidl@dcs.gla.ac.uk

Kevin Hammond*
Division of Computer Science, University of St. Andrews
St. Andrews, Scotland, U.K.
E-mail: kh@dcs.st-and.ac.uk

Abstract

This paper describes an inference system, whose purpose is to determine the cost of evaluating expressions in a strict purely functional language. Upper bounds can be derived for both computation cost and the size of data structures. We outline a static analysis based on this inference system for inferring size and cost information. The analysis is a synthesis of the *sized types* of Hughes et al., and the polymorphic *time system* of Dornic et al., which was extended to static dependent costs by Reistad and Gifford.

Our main interest in cost information is for scheduling tasks in the parallel execution of functional languages. Using the GranSim parallel simulator, we show that the information provided by our analysis is sufficient to characterise relative task granularities for a simple functional program. This information can be used in the runtime-system of the Glasgow Parallel Haskell compiler to improve dynamic program performance.

1 Introduction

There is a sad paradox that often arises in implicit parallel programming: having striven hard to locate every expression that could possibly be executed in parallel, it is common to find that most of these are too small to be worth the overheads associated with executing them as parallel tasks. Fear not, however, that all that effort will be wasted. Under the assumption that good strictness analysis or language design can determine what *could* be executed in parallel, this paper focuses on the important issue of determining what *should* be executed in parallel.

The *sized time system* which we introduce in Section 3 can be used to give an estimate of the cost of evaluating an expression together with the size of the resulting data structure. By using source annotations, the cost information, which represents information about the *granularity* of a computation, can be exploited by the dynamic parallel scheduler of the runtime-system in order to decide which tasks should be evaluated at any given time.

In order to make the analysis tractable, this paper makes some important simplifying assumptions. In particular, we sidestep the thorny though interesting issues of strictness and sharing analyses by analysing a strict functional language, $\mathcal{L}$ (defined in Section 2). It is our ultimate intention, however, that the analysis described here should be applicable to full parallel Haskell. We also deal only with lists and flat data structures. Using recent results on sized types it should be straightforward to extend the work to arbitrary data structures, however.

*Supported by the EPSRC Parade grant.

At the current stage of our work we are mainly interested in the feasibility of checking size and cost bounds and in the possible runtime improvements when using these bounds. We do not address questions of the soundness and completeness of the inference system in this paper. However, the close relationship to sized types and static dependent costs presents a very good starting point for further investigations into that direction.

We demonstrate the feasibility of our approach in Section 4 by giving a worked example. The output of the analysis (performed by hand) has been passed to our parallel simulator, GranSim [3]. For this example, there is a demonstrable performance improvement when using our previously developed granularity control mechanisms. This confirms previous results, where a hand-annotated program was measured. In [9] we showed that granularity information, if present, can be used to achieve a significant performance improvement.

The system presented here is closely related to the sized types of Hughes, Pareto and Sabry [6] and to the time system developed by Dornic, Jouvelot and Gifford [2], which has been extended to a static dependent cost system by Reistad and Gifford [13]. However, unlike the latter system we can also derive cost information for some recursive functions. This is possible by extending the standard subtype inference mechanism with a "database" of known recurrences and (an approximation of) their closed forms. Section 5 outlines the structure of the inference algorithm.

2 The Language $\mathcal{L}$

The language $\mathcal{L}$ is a very simple purely functional language, intended solely as a vehicle to explore static analysis for parallelism. $\mathcal{L}$ is a strict polymorphic higher-order language with lists as its only compound data type.

The *abstract syntax* of $\mathcal{L}$ is given below. We assume that variables (v), constants (k) and primitive operations (p) on basic types are all disjoint.

$$\begin{array}{rcl} e & ::= & \mathtt{v} \mid k \mid p\; e_1\; \ldots\; e_n \mid \lambda \mathtt{v}\;.\; e \mid e_1\; e_2 \mid \mathtt{cons}\; e_1\; e_2 \mid \mathtt{nil} \mid \mathtt{null}\; e \mid \mathtt{hd}\; e \mid \mathtt{tl}\; e \mid \\ & & \mathtt{letrec}\; v \;=\; e_1\; \mathtt{in}\; e_2 \mid \mathtt{if}\; e_1\; \mathtt{then}\; e_2\; \mathtt{else}\; e_3 \end{array}$$

Overall, the structure of $\mathcal{L}$ expressions is similar to that of Lisp expressions, focusing on lists as the only compound datatype. Let expressions are recursive (i.e. v may occur in e_1). Since the entire $\mathcal{L}$ program is an $\mathcal{L}$ expression, nested let expressions have to be used to define auxiliary functions.

To simplify the presentation we assume that variable names in the program are unique. This avoids complications in the treatment of the envionments in the sized time system.

$\mathcal{L}$ uses sized types: each type, except the function type, has a component specifying an upper bound for its size. The type `Int` contains only positive integer numbers. The function type maps a sized type to another sized type and attaches a computation cost to the function. In the following syntax of *type expressions*, α represents a sized type variable.

$$\tau \quad ::= \quad \alpha \mid \mathtt{Int}^c \mid \mathtt{Bool}^0 \mid \mathtt{List}^c\; \tau \mid \tau_1 \;\xrightarrow{c}\; \tau_2$$

Both cost and size expressions are specified by *c-expressions*. Therefore, cost expressions can contain variables representing the size of a data structure. It is important to note that c-expressions are *linear* (i.e. there can be no expressions of the form $v_1 * v_2$ where v_1, v_2 are variables). This property plays an important role in the implementation proposed in Section 5.

$$c \quad ::= \quad l \mid \infty \mid n \mid c_1 + c_2 \mid c_1 - c_2 \mid n * c \mid \max\; c_1\; c_2 \mid f\; c_1\; \ldots\; c_n$$

In these c-expressions n is a positive integer constant and l is a c-variable. The ∞ symbol is used to express an unbounded cost/size. For sizes less than ∞ the operators $+, -, *$ and $\max$ behave as usual. When one of the operands is ∞ the result is ∞, too (with the exception of $x - \infty$ which is ∞ for $x \neq \infty$). In order to handle recursive programs we have to introduce symbolic cost functions f. The arguments $c_1 \ldots c_n$ represent the sizes of the argument expressions in the program.
Polymorphism is achieved in the usual way by quantifying over free variables of a let-bound expression. Because we are using sized types we have to quantify over type as well as size variables. In the following we will use x to represent either a type or size variable. The general structure of *type schemes* is therefore:

$$\sigma \quad ::= \quad \forall x.\sigma \mid \tau$$

The dynamic semantics of this language is standard. For example the semantics of the recursive let construct is defined by using the fixed point of the functional denoted by the local definition in the body of the let construct. We omit the details here. For the cost model used in the next section it is important to note that $\mathcal{L}$ is a strict language.

An important difference to the semantic model of sized types used by Hughes et al. in [6] is the fact our semantic domain for lists has to contain a bottom element representing non-termination. Again this difference is due to the strict semantics of $\mathcal{L}$. In contrast to our application of size information as input for cost information Hughes et al. aim at proving properties on streams (infinite lists).

3 A Static Cost Semantics for $\mathcal{L}$

This section develops a static cost semantics for $\mathcal{L}$. In order to statically estimate an upper bound for the cost of evaluating an expression, we need information about the size of values in the program. Therefore, we will develop a size as well as a cost analysis. Before presenting the analyses, we first introduce the domains we use to maintain cost and size information. Both analyses are interwoven with a standard polymorphic type system to give a *sized time system*[1] for $\mathcal{L}$.

3.1 Cost and Size Domains

In order to provide a high-level abstract model of execution cost, we use the conventional approach of step counting. The cost of evaluating an expression is given by the number of function-, operator- and constructor-applications, plus the number of conditionals.

The obvious domain for a step-counting analysis is the set of positive integer values, augmented by ∞. We also introduce an upper bound on cost, m_c, and represent all costs greater than m_c by ∞. So, the cost domain $\mathcal{D}_\mathcal{C}$ is:

$$\mathcal{D}_\mathcal{C} = \{0, 1, \ldots, m_c, \infty, \top\} \text{ with } 0 < 1 < \cdots < m_c < \infty < \top$$

for some integer m_c.

Note the difference between ∞ and $\top$ in this domain: ∞ represents a computation with costs greater than m_c whereas $\top$ represents a computation whose cost is unknown. From a pragmatic perspective, ∞ denotes a computation that is large enough to be executed in parallel, whereas $\top$ denotes a computation whose cost is unknown. For our applications it is useful to map all costs beyond a certain bound to the same value. This usage of cost information naturally gives a finite domain (it is not a restriction to make an analysis simpler).

The size of an integer expression is the value itself (where it is known). The size of booleans has been fixed at 0. The size of a list is its length plus one, and in general, the size of a data structure will be the number of function applications needed to generate that constructor: hence size is modelled by the recursion depth of the data structure.

The size domain $\mathcal{D}_\mathcal{Z}$ is defined as

$$\mathcal{D}_\mathcal{Z} = \{0, 1, \ldots, \infty\} \text{ with } 0 < 1 < \cdots < \infty$$

Note that sizes constitute parts of types in $\mathcal{L}$. This gives a convenient way to describe the size of sub-components of a data structure as well as the size of the structure itself, e.g.

$$\mathit{List}^5\ \mathit{Int}^{10}$$

denotes a list whose length is smaller than 5 with integer numbers smaller than 10 as elements.

3.2 A Sized Time System for $\mathcal{L}$

The inference rules of the sized time system in this section represent an extension to the standard type inference rules for $\mathcal{L}$. These extensions capture the two aspects of a program in a slightly different way. The size information, which represents an aspect of the value of an $\mathcal{L}$ expression, is attached to its type. The cost information, which represents an

[1] by analogy with *sized types* [6] and *time systems* [2, 13].

aspect of the evaluation of an $\mathcal{L}$ expression, is inferred together with the size information, but it is not attached to the type. The cost inference has to use size information but not vice versa.

The costs of higher-order functions are modelled by attaching *latent costs* [13] to function types. These latent costs usually contain free variables representing the size of the arguments. Section 3.3 gives the type of the function length as an example.

$$(Int)\ \frac{}{\Gamma \vdash \mathtt{n} : Int^{n}\ \$\ 0} \qquad (Bool)\ \frac{}{\Gamma \vdash \mathtt{b} : Bool^{0}\ \$\ 0} \qquad (Nil)\ \frac{}{\Gamma \vdash \mathtt{nil} : List^{1}\ \tau\ \$\ 0}$$

$$(Var)\ \frac{}{\Gamma \cup \{\mathtt{v} : \forall x_1, \ldots, x_n.\tau\} \vdash \mathtt{v} : \tau'\ \$\ 0} \quad \begin{array}{c} \tau' = \tau[x'_1/x_1, \ldots, x'_n/x_n] \\ x'_1, \ldots, x'_n \notin FV(\tau) \cup FV(\Gamma) \end{array}$$

$$(Weak)\ \frac{\Gamma \vdash \mathtt{e} : \tau'\ \$\ c' \quad \tau' \trianglelefteq \tau \quad c' \le c}{\Gamma \vdash \mathtt{e} : \tau\ \$\ c} \qquad (Prim)\ \frac{\Gamma \vdash \mathtt{e_1} : \tau^{z_1}\ \$\ c_1 \quad \cdots \quad \Gamma \vdash \mathtt{e_n} : \tau^{z_n}\ \$\ c_n}{\Gamma \vdash \mathtt{p}\,\mathtt{e_1} \ldots \mathtt{e_n} : \tau^{p'\ z_1 \ldots z_n}\ \$\ 1 + c_1 + \cdots + c_n}$$

$$(Abstr)\ \frac{\Gamma \cup \{\mathtt{v} : \tau_1\} \vdash \mathtt{e} : \tau_2\ \$\ c}{\Gamma \vdash \lambda \mathtt{v} \,.\, \mathtt{e} : \tau_1 \xrightarrow{c} \tau_2\ \$\ 0} \qquad (App)\ \frac{\Gamma \vdash \mathtt{e_1} : \tau_1 \xrightarrow{c} \tau_2\ \$\ c_1 \quad \Gamma \vdash \mathtt{e_2} : \tau_1\ \$\ c_2}{\Gamma \vdash \mathtt{e_1}\,\mathtt{e_2} : \tau_2\ \$\ 1 + c_1 + c_2 + c}$$

$$(Cons)\ \frac{\Gamma \vdash \mathtt{e_1} : \tau\ \$\ c_1 \quad \Gamma \vdash \mathtt{e_2} : List^{c'}\ \tau\ \$\ c_2}{\Gamma \vdash \mathtt{cons}\,\mathtt{e_1}\,\mathtt{e_2} : List^{c'+1}\ \tau\ \$\ 1 + c_1 + c_2} \qquad (Null)\ \frac{\Gamma \vdash \mathtt{e} : List^{c'}\ \tau\ \$\ c}{\Gamma \vdash \mathtt{null}\,\mathtt{e} : Bool^{0}\ \$\ 1 + c}$$

$$(Hd)\ \frac{\Gamma \vdash \mathtt{e} : List^{c'}\ \tau\ \$\ c}{\Gamma \vdash \mathtt{hd}\,\mathtt{e} : \tau\ \$\ 1 + c}\ c' > 1 \qquad (Tl)\ \frac{\Gamma \vdash \mathtt{e} : List^{c'}\ \tau\ \$\ c}{\Gamma \vdash \mathtt{tl}\,\mathtt{e} : List^{c'-1}\ \tau\ \$\ 1 + c}\ c' > 1$$

$$(Cond)\ \frac{\Gamma \vdash \mathtt{e_1} : Bool^{0}\ \$\ c_1 \quad \Gamma \vdash \mathtt{e_2} : \tau\ \$\ c_2 \quad \Gamma \vdash \mathtt{e_3} : \tau\ \$\ c_3}{\Gamma \vdash \mathtt{if}\ \mathtt{e_1}\ \mathtt{then}\ \mathtt{e_2}\ \mathtt{else}\ \mathtt{e_3} : \tau\ \$\ 1 + c_1 + \max\ c_2\ c_3}$$

$$(Let)\ \frac{\Gamma \vdash \mathtt{e_1} : \tau_1\ \$\ c_1 \quad \Gamma \cup \{\mathtt{v} : \forall x_1, \ldots, x_n.\tau_1\} \vdash \mathtt{e_2} : \tau_2\ \$\ c_2}{\Gamma \vdash \mathtt{letrec}\ \mathtt{v} = \mathtt{e_1}\ \mathtt{in}\ \mathtt{e_2} : \tau_2\ \$\ c_1 + c_2}\ x_1, \ldots, x_n \in FV(\tau_1) \backslash FV(\Gamma)$$

Figure 1: A Sized Time System for $\mathcal{L}$

Figure 1 shows the extended type system. The c-expression in the superscript of a type is an upper bound for the size of the object and therefore denotes an element of the domain $\mathcal{D}_{\mathcal{Z}}$. The expression after \$ in a judgement is a c-expression that represents the cost for performing the corresponding computation and therefore denotes an element of the domain $\mathcal{D}_{\mathcal{C}}$. The assumption set Γ contains bindings of variables to type schemes (of the form $x : \sigma$). Since we have assumed that the names of all variables are unique, assumption sets can be combined by using set union. We use $\tau[x'/x]$ to denote a substitution of all free occurrences of x in τ by x'.

The *(Var)* rule performs an instantiation of the abstracted size and type variables x_i by substituting all free occurrences with fresh variables x'_i in the body of the type τ. The FV function computes the set of free variables in a type expression or an assumption set.

The *(Weak)* rule allows to weaken upper bounds for size and cost. It makes use of the subtyping relation $\trianglelefteq$ defined in Figure 2.

In the *(Prim)* rule we use for each primitive operator p a corresponding abstract operation p' over the size domain in order to compute the size bounds for integer values. Most operators are analogous to their integer counterparts, but in order to maintain the linearity of size expressions, the multiplication operator $*'$ is defined to have a size bound of ∞ unless one of its operands is constant.

The *(Abstr)* rule infers the cost of evaluating the body of a lambda-abstraction, and attaches this to the type of the lambda-abstraction as a *latent cost*. The latent cost usually contains a free variable for the size of the argument x. When the function is applied using *(App)* the size bound of the corresponding argument must be no greater than the size given in the type of the function's domain.

The rules for *(Cons), (Null), (Hd), (Tl)* show how size bounds are derived for list constructors and selectors. Every application of a constructor to all of its arguments counts as one step.

In general both branches in a conditional will have different sizes. Two examples below illustrate how the *(Weak)* rule is used to ensure that the types of both branches match, which is required by the *(Cond)* rule. In order to give a cost bound for this expression we have to choose the maximum of the costs of both branches. In Section 5 we suggest some practical techniques for improving this cost bound.

The *(Let)* rule realises polymorphism over sized types: When inferring a type for e_2 the variable v is bound to a type scheme, which abstracts over type and size variables. An instantiation of type schemes is performed as part of the *(Var)* rule. It is worth noting that the *(Let)* rule in [6] is significantly more complicated because it has to propagate size information for algebraic data types from one recursion level to the next. In $\mathcal{L}$ this size propagation is encoded in the rules operating on lists.

$$\frac{c_1 \leq c_2}{\mathit{Int}^{c_1} \trianglelefteq \mathit{Int}^{c_2}} \qquad \frac{c_1 \leq c_2 \qquad \tau_1 \trianglelefteq \tau_2}{\mathit{List}^{c_1}\ \tau_1 \trianglelefteq \mathit{List}^{c_2}\ \tau_2} \qquad \frac{\tau_1 \trianglelefteq \tau_1' \qquad \tau_2' \trianglelefteq \tau_2 \qquad c' \leq c}{\tau_1' \xrightarrow{c'} \tau_2' \trianglelefteq \tau_1 \xrightarrow{c} \tau_2}$$

Figure 2: Subtyping Relation for $\mathcal{L}$

The subtyping relation in Figure 2 formalises the idea that the size component in a sized type specifies an upper bound. Therefore, it should always be possible to weaken this size bound. Similarly, the latent cost in a function type is an upper bound for the cost of evaluating the function. The need for such a subtyping relation can be motivated by an analysis of the following expression.

```
if (null xs) then 1 else 2
```

In this expression the `then` branch has a sized type of Int^1 but the `else` branch has the type Int^2. Only because of the subtyping relationship between these types $\mathit{Int}^1 \trianglelefteq \mathit{Int}^2$ is the above expression type correct.

A similar example shows how the cost information in a function type can be weakened:

```
foo xs f g = if (null xs)
                then λ x .(f x) + 1
                else λ x .(f x) + (g (hd xs))
```

When inferring the result type of this function we have to match function types with different latent costs. The latent cost of the result function must be an upper bound for the latent costs of both argument functions. Therefore, a type inference has to use the subtyping relation on function types, which weakens the upper bound for the latent cost.

3.3 From Cost-expressions to Cost-functions

The sized time system in Figure 1 is a high-level description of how to infer costs and sizes of an expression. When deriving the cost of a function application the cost expression representing the latent cost for the function has to be used. However, if the function is recursive this approach will fail to yield a cost expressions in closed form (i.e. without references to symbolic cost functions). Therefore, we need explicit cost functions. In general the result of performing cost inference will be a set of recurrences that has to be solved separately.

The *(Let)* rule in the sized time system shows that the type schemes for let-bound functions in general contain universally quantified size variables. These variables can be regarded as arguments to the cost function described by the inferred cost expression. In general we can give for every function definition

```
f x1 ... xn = e
```

a cost function

$$f_{\mathcal{C}}\, l_1\ \ldots l_n\ = c$$

and a size function

$$f_{\mathcal{Z}}\, l_1\ \ldots l_n\ = z$$

where c is the cost and z is the size expression derived from the body of the function `e`. The variables $l_1 \ldots l_n$ represent the sizes of the arguments to `f`.

For example, the sized time system can assign the following type to the polymorphic *length* function:

$$length : \forall\alpha.\, \forall l.List^l\ \alpha\ \stackrel{4*l+2}{\to}\ Int^l$$

In the following sections we use as a special notation $length_{\mathcal{Z}}$ for describing the corresponding size function ($length_{\mathcal{Z}}\ l\ =\ l$ in this case) and $length_{\mathcal{C}}$ for describing the corresponding cost function ($length_{\mathcal{C}}\ l\ =\ 4*l+2$ in this case).

The example of how to perform size and cost inference in the following section and the outline of an inference algorithm after that describe how to derive closed forms of cost expressions from user defined recursive functions.

4 Example

In this section we study a simple non-trivial $\mathcal{L}$ program, `coins`, which shows an interesting parallel behaviour. We demonstrate how to derive size and cost information by using our sized time system. Since our goal is to improve the parallel performance of the program, we then measure the runtime of the program annotated with cost information when using a priority scheduling mechanism in the runtime-system.

The `coins` program takes a price and a list representing a set of coins, and determines how many different combinations of coins could be used to pay for an object at the given price. Previously we have studied the behaviour of a hand-annotated version of this program on various different parallel architectures using GranSim. We managed to improve the overall performance by using priority scheduling as a granularity control mechanism. Priority scheduling uses granularity information as priorities for the generated threads and therefore ensures that larger threads are preferred to smaller ones.

The `coins` program consists of two mutually recursive functions `pay_price` and `choose`:

```
pay_price = \ price coins ->
  if (price==0) then 1
  else  let  coin_values = nub (dropWhile (\ x -> x>price) coins)
        in
          sum (map (choose price coins) coin_values)

choose = \ price coins c ->
  let  new_coins' = dropWhile (\ x -> x>c) coins
       new_coins  = del new_coins' c
  in
    pay_price (price-c) new_coins
```

The parallel version of this program has three main sources of parallelism: `coin_values`, `new_coins` and the `map` expression can be evaluated in parallel with the rest of the computation.

The definition of `coins` contains five auxiliary functions that must be analysed: `nub` eliminates multiple entries in a list, `del` deletes an element from a list, `dropWhile` drops an initial segment of a list, `map` applies a function to every element of a list and `sum` computes the sum of all elements of a list. As an example, we derive the size and cost functions of `del`.

Here is the definition of `del` in $\mathcal{L}$:

```
del =  \ xs x -> if (null xs) then error
                 else let  z = hd xs
                           zs = tl xs
                      in
                        if (z==x) then zs else cons z (del zs x)
```

The `del` function deletes the first instance of `x` from `xs`. The special value `error` (of the polymorphic type τ) is used to indicate that `x` did not occur in `xs` (an error).

4.1 Cost and Size Analysis

The structure of both the cost and the size analysis is based on the observation that recursive functions will in general yield a set of recurrences that has to be solved separately. It is worth noting that no symbolic cost functions will appear in the cost expressions if no user defined recursive functions are used. The main steps in the cost and size analysis are:

1. Inference of the body of the function by traversing the proof tree and collecting constraints (inequalities) over c-expressions. This step returns a type and a constraint set.
2. Simplifying the constraint set. This amounts to reducing the c-expressions in the constraint set to normal form (we use a sum-of-products normal form).
3. Resolving recurrences in the constraint set. This can be done by using a symbolic computation system for solving recurrences (where this is possible) or by using a "database" of recurrences and their closed forms.
4. Solving the constraint set. For this step we use the Omega Library [12].

Inference and Simplification: In this example we perform simplification of the constraints on-the-fly. We only describe the main steps in the inference of the body of `del`:

1. Using the *(App)* rule twice adds bindings of fresh type variables to `xs` and `x` to the assumption set.
2. In the head of the outer conditional the *(Null)* rule assigns the type $List^{l}\ \alpha$ to `xs`, where α and l are fresh type and size variables. The cost for this operation is 1 step.
3. The type of `error` in the outer then branch is τ (0 cost).
4. In the analysis of the local definitions of the let-expression the types α and $List^{l-1}\ \alpha$ are inferred for the variables `z` and `zs`, respectively. The cost for computing the expressions are 1 step each.
5. The head of the inner conditional requires `z` and `x` to be of the same type. The cost for this primitive operation is 1 step.
6. The inner then branch returns `zs`, an object of type $List^{l-1}\ \alpha$ (0 cost).
7. The most interesting part is the inner else branch.

   The *(Cons)* rule infers the type $List^{l'+1}\ \alpha$ for the whole expression, with l' being a fresh size variable. Unifying the type $List^{l'+1}\ \alpha$ of the else branch with the type $List^{l-1}\ \alpha$ of the then branch assigns the value $l-2$ to l'. Thus, we can infer the following recurrence for the size function of `del`:

   $$del_{\mathcal{Z}}\ l\ \hat{l}\ =\ 1 + del_{\mathcal{Z}}\ (l-2)\ \hat{l}$$

   Based on this size information we can infer the following cost for performing the recursive call to `del`: $2 + del_{\mathcal{C}}(l-1)\ \hat{l}$ (two applications of the *(App)* rule). The $l-1$ expression is the inferred size of the variable `zs`.

Combining this cost expression with the costs for the other parts of the function yields the following recurrence for the cost function of `del`:

$$del_{\mathcal{C}}\ l\ \hat{l}\ =\ 1 + 1 + \max\ 0\ (1 + 1 + 1 + 1 + \max\ 0\ (1 + 0 + (1 + 0 + 0 + del_{\mathcal{C}}\ (l-1)\ \hat{l})))$$

In this cost expression the occurrences of $\max$ reflect the two nested conditionals in the code.

Note that the standard unification mechanism is used to obtain the substitution $[l - 1/l' + 1]$, which yields the above recurrence. Applying the substitution to a cost expression with free variables represents an application of the corresponding cost function to a size expression.

Resolving Recurrences: The goal of this step is to bring all symbolic cost functions (like $del_{\mathcal{C}}$) into closed form in order to substitute the functions with the expressions in the constraint set. This will eliminate all symbolic cost functions. As a result of traversing the proof tree we have obtained the following system of recurrences for `del` (the base cases use the fact that 0 is the smallest possible size):

$$\begin{array}{lcl} del_{\mathcal{Z}}\ 0\ \hat{l} & = & 0 \\ del_{\mathcal{Z}}\ l\ \hat{l} & = & 1 + del_{\mathcal{Z}}\ (l-2)\ \hat{l} \end{array}$$

$$\begin{array}{lcl} del_{\mathcal{C}}\ 0\ \hat{l} & = & 2 \\ del_{\mathcal{C}}\ l\ \hat{l} & = & 1 + 1 + \max\ 0\ (1 + 1 + 1 + 1 + \max\ 0\ (1 + 0 + (1 + 0 + 0 + del_{\mathcal{C}}\ (l-1)\ \hat{l}))) \\ & = & 8 + del_{\mathcal{C}}\ (l-1)\ \hat{l} \end{array}$$

Our approach for resolving recurrences over cost functions is to use a "database" of recurrences and their closed forms. If no matching recurrence is found the constant ∞ function has to be used as the weakest approximation. Solving the above recurrences using the rules in Figure 3 gives the following closed forms for $del_{\mathcal{Z}}$ and $del_{\mathcal{C}}$:

$$\begin{array}{lcl} del_{\mathcal{Z}}\ l\ \hat{l} & = & l - 1 \\ del_{\mathcal{C}}\ l\ \hat{l} & = & 8 * l + 2 \end{array}$$

Performing a similar analysis for the other auxiliary functions yields the size functions shown below. Note that the higher-order size functions use size functions as arguments. In this case size functions represent the 'size' of a function type.

$$\begin{array}{lcllcl} sum_{\mathcal{Z}}\ l\ \hat{l} & = & l & \quad dropWhile_{\mathcal{Z}}\ f_{\mathcal{Z}}\ l\ \hat{l} & = & l \\ nub_{\mathcal{Z}}\ l & = & l & \quad map_{\mathcal{Z}}\ f_{\mathcal{Z}}\ l\ \hat{l} & = & l \end{array}$$

These size bounds have been confirmed by the existing implementation of sized types described by Hughes et al. [6]. However, because of the lack of an error value of size 0 in their language the type of the `del` function has a weaker size bound:

$$del : \forall k.\forall \alpha.List^k\ \alpha\ \dot{\rightarrow}\ \alpha\ \dot{\rightarrow}\ List^k\ \alpha$$

The size analysis of `sum` shows that we sometimes have to weaken bounds in order to derive cost expressions in our constraint language. The recurrence describing the size function of `sum` for an input of type $List^l\ Int^{\hat{l}}$ is

$$sum_{\mathcal{Z}}\ l\ \hat{l}\ =\ \hat{l} + sum_{\mathcal{Z}}\ (l-1)\ \hat{l}$$

which can be solved with the second rule of Figure 3 yielding $l\ *\ \hat{l}$. Unfortunately, this is not linear and therefore we have to use ∞ as the bound.

The cost functions for the other auxiliary functions can be derived in a similar way as $del_{\mathcal{C}}$, using cost functions as arguments in the case of the higher-order functions:

$$\begin{array}{lcllcl} sum_{\mathcal{C}}\ l\ \hat{l} & = & 6 * l + 2 & \quad dropWhile_{\mathcal{C}}\ f_{\mathcal{C}}\ l\ \hat{l} & = & (6 + f_{\mathcal{C}}\ \hat{l}) * l + 2 \\ nub_{\mathcal{C}}\ l & = & 9 * l^2 + 2 * l + 2 & \quad map_{\mathcal{C}}\ f_{\mathcal{C}}\ l\ \hat{l} & = & (5 + f_{\mathcal{C}}\ \hat{l}) * l + 2 \end{array}$$

It is important to note that all recurrences in the analysis of these functions are *linear, first-order recurrences* since the functions iterate over lists. Figure 3 shows the entire "database" of closed forms for recurrences that we used in this example.

$$\begin{array}{lcl} f\ 0 & = & a \\ f\ n & = & b + f(n-1) \end{array} \quad \Rightarrow \quad f\ n\ = a + b * n$$

$$\begin{array}{lcl} f\ 0 & = & a \\ f\ n & = & b + c * n + f(n-1) \end{array} \quad \Rightarrow \quad f\ n\ =\ a + b * n + \frac{c*n*(n+1)}{2}$$

Figure 3: Recurrences and their Closed Forms

Solving the Constraint Set: The final step has to check whether a solution for the the constraint set exists. In this case the program is well typed and for each function a corresponding size and cost function has been inferred. Since the constraint set does not contain symbolic cost functions any more at this stage we can use the Omega Library for performing this check. After that we can use the cost functions in order to derive the costs of those constructs that should be evaluated in parallel. In our example we are interested in the costs for `coin_values` and `new_coins`. The former is the result of applying `nub`. The latter is the result of applying `del`. The sizes of the argument lists are determined by calls to `dropWhile`. Using the above size functions for these auxiliary functions we obtain the following cost expressions (n is the size of the `coins` list):

$$\begin{array}{lcl} \textit{coin\_values}_{\mathcal{C}} & = & 9 * n^2 + 14 * n + 5 \\ \textit{new\_coins'}_{\mathcal{C}} & = & 16 * n + 4 \end{array}$$

These expressions represent the information we need to improve the performance of the parallel program.

4.2 Annotations

We can now use the cost information derived in the previous section to transform the parallel program by adding cost information to the spark sites:

- For each argument add an extra argument representing its size.
- Use the derived size functions to propagate size information.
- Add the derived cost expressions to the `parGlobal` annotations.

This transformation gives the following annotated parallel program (`m` represents the size of `price` and `n` represents the size of `coins`):

```
pay_price = \ m n price coins ->
  if (price==0) then 1
  else let  coin_values = nub (dropWhile (\ x -> x>price) coins)
       in
         parGlobal (9*n^2+14*n+5) coin_values
                   (sum (parMap infty (choose m n price coins) coin_values))

choose = \ m n price coins c ->
  let   new_coins' = dropWhile (\ x -> x>c) coins
        new_coins  = del new_coins' c
  in
    parGlobal (16*n+4) new_coins
              (pay_price m (n-1) (price-c) new_coins)
```

The `parGlobal` pseudo-function is used to spark new parallel tasks. It takes three arguments: the first argument is a cost or granularity measure; the second is a closure that should be sparked as a parallel task; and the final argument is the sequential continuation that should be executed once the spark has been created. The `parMap` function is a parallel implementation of `map` that takes granularity information as its first argument. The special value `infty` represents ∞ as a bound on computation cost.

4.3 Measurements

Figure 4 compares the absolute speedup obtained by a program without granularity information against the speedup of a program that uses the cost estimates derived in the previous section. A priority scheduling mechanism [9] that exploits the available granularity information normally performs better than the default scheduling algorithm that has no granularity information available to it, even though the latter is known to be a good schedule for a divide-and-conquer program such as this.

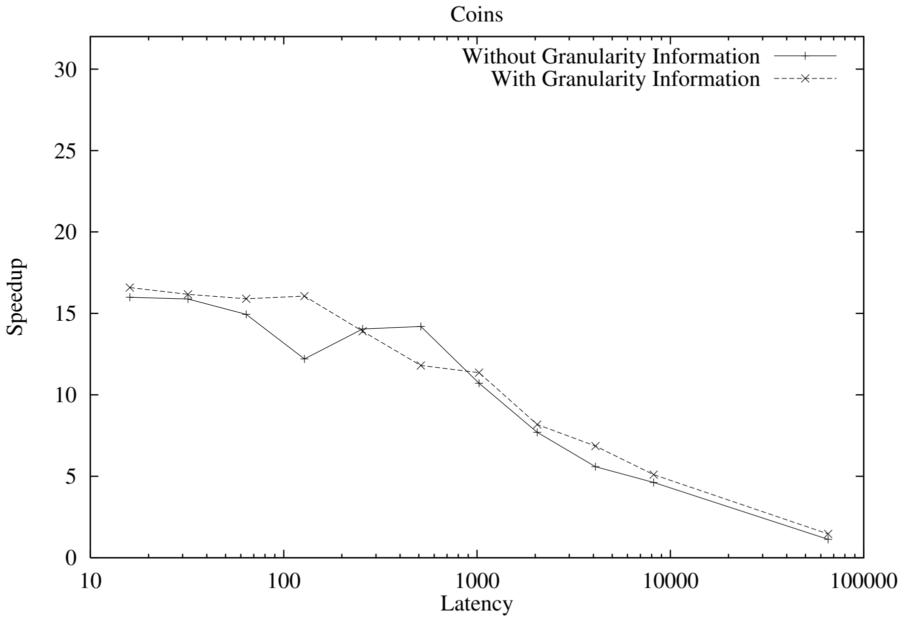

Figure 4: Speedup with and without granularity information

The runtime improvement for rather small latencies is due to the creation of many tiny tasks before the runtime system automatically discards these sparks as being worthless [11]. Therefore, there is more to gain by making the "right" decision when sparking. Unexpectedly, however, at medium latencies, priority scheduling yields worse performance than using no granularity information. This is probably due to an increased amount of blocking caused by unlucky scheduling at some point during the execution, but this requires further investigation (we should note in passing that it is only because GranSim allows us to study variations in latency, that we are able to observe this worsening in performance – using idealised simulation or a fixed machine would probably not reveal this discrepancy!). For very high latencies the cost of exporting a task and retrieving its result outweighs the cost of evaluating it locally, and we therefore do not expect much improvement in speedup.

It has to be emphasised that this program contains only two main spark sites, which limits the amount of runtime improvement we can expect by adding granularity information. Granularity control mechanisms mainly aim at improving programs with a large number of spark sites generating tasks whose granularities vary significantly. This is, for example, the case for naïve methods of generating implicit parallelism in a functional program. Another important result of these measurements is the observation that we can achieve runtime improvements for a wide range of latencies representing different kinds of parallel architectures. Therefore, we believe that a granularity analysis based on our sized time system would be an important component of an implicit parallelisation system.

5 Implementation Issues

We plan to combine an implementation of this cost analysis with the existing implementation of sized types. This amounts to adding a pass that generates constraints on cost variables and solves the resulting system of inequalities over c-expressions. These inequalities are introduced via the *(Weak)* rule. The structure of c-expressions has been designed such that an efficient solution of the constraint set is possible by using the Omega Library [12].

All rules of the sized time system except for the *(Weak)* rule are structural. Therefore, the only open question for developing a *proof strategy* is where to apply the *(Weak)* rule. The natural place for weakening the derived cost of a function is the *(App)* rule: the latent cost recorded in the type of the function represents an upper bound for the total cost. Furthermore, the *(Cond)* requires the same size bound for the then- and the else-branch. Therefore, a weakening of the size bounds may have to be performed with that rule.

A *cost checking* algorithm for the sized time system is no more complicated than the existing size checking algorithm for sized types. It uses the mandatory type declarations for all let-bound variables to compare the declared costs with the costs that are inferred from the body of the definition. This yields a set of inequalities over cost expressions in closed form. The Omega Library checks whether a solution exists. If this is not the case the expression is ill-typed. Such a checking algorithm could be used to confirm that a cost expression provided by the user is indeed an upper bound for the cost of the function.

When the checking algorithm is extended to a *cost inference* algorithm we need to deal with unknown (symbolic) functions in the constraint system. These functions are introduced during the analysis of recursive functions (for a non-recursive program the derived cost expression is already in closed form). In general, a cost function may involve arbitrarily complex recurrences, but with the current state-of-the-art it is only possible to find closed forms for the following classes of recurrences [10]:

- linear recurrences with constant coefficients;
- homogeneous linear recurrences with polynomial coefficients;
- certain divide-and-conquer recurrences;
- certain non-linear first order recurrences.

In particular, it is not possible to solve many non-linear cases that occur in c-expressions, and it is quite time consuming to solve even linear recurrences with polynomial coefficients.

Because of this complexity we do not plan to extend the implementation of the constraint solver with a general recurrence solver. Rather, we prefer to add an additional phase immediately before the constraint solver whose purpose is to replace simple and common recurrences (e.g., linear ones) in the constraint set with known closed forms. At this stage it may be necessary to replace non-linear c-expressions with ∞ in order to use the Omega Library. However, this only has to be done for cost functions that are used by other cost functions, which are not in closed form. Based on that observation, the set of cost functions can be split in two classes. The usual constraint solving is then performed only on the class of cost functions that is not in closed form.

A further advantage of such an approach is that the accuracy of the cost analysis is to some degree tunable by varying the accuracy of the closed forms in this *"database"*. Based on measurements performed with weak cost bounds we can then decide whether the accuracy of the cost analysis has to be improved.

An open problem with an inference algorithm of this kind is how to find a minimal solution of the constraints that are derived. When using a "database" of recurrences with approximate closed forms the solution won't be minimal. However, as should be clear from the discussion in the previous section a minimal solution is not absolutely necessary in order to extract useful information out of the cost analysis. This agrees with observations in [13] on the quality of statically determined cost estimates.

Refining the structure given in Section 4 we can give the following structure for a type and cost inference system:

1. Hindley-Milner type inference;
2. sized type inference;
3. cost inference.

Both sized type and cost inference have the same internal structure:

1. collect constraints while traversing the proof tree;
2. simplify the set of inequalities (containing symbolic functions) by reducing c-expressions to a normal form;
3. spot common patterns of recurrences and replace them with closed forms. If no matching recurrence is found the symbolic function is defined to yield ∞ for every input;
4. replace non-linear c-expressions with ∞;
5. eliminate trivial constraints containing ∞;
6. solve the resulting constraint system (using the Omega Library);
7. simplify the result further.

The main source of inaccuracy for the derived cost bounds in our sized time system is the *(Cond)* rule. This might prove to be a problem if a costly branch is rarely executed, for example if the base case of a recursive function is much more expensive than the normal recursive case. Although this seems unlikely to be a major issue, one way to alleviate this particular problem would be to add special cases to the "database" of recurrences to avoid counting the base case several times. A variant of the max operator could then be used to indicate that the conditional is on the critical path of a recursion. A more pragmatic approach would be to allow probabilities to be specified for the branches of the conditional, perhaps automatically using the profiles generated by the simulator for some sample data ("hybrid" approaches of this kind are often encountered in the literature). We have no plans at present to implement such an automated scheme.

6 Related Work

Pioneering work on automatic complexity analysis was done by Wegbreit [17]. His METRIC system can derive the average case complexity of a wide range of programs by solving the difference equations that occur as an intermediate step in the complexity analysis. However, this general approach is very expensive and therefore only possible in an off-line algorithm (and not in a static analysis that can be be performed by a compiler). LeMetayer [8] takes a similar approach based on program transformation: he uses a set of rewrite rules to derive complexity functions, simplify them and to finally eliminate recursion. His ACE system works on FP programs.

Work on granularity analysis has also been undertaken in both the Lisp and SML communities: for example Huelsbergen, Larus and Aiken have defined an abstract interpretation ("dynamic granularity estimation") of a higher-order, strict language for determining computation costs, which uses dynamic estimates of the size of data structures [5]. Their analysis uses the well-known trick of iteration in the abstract interpretation stops as soon as a certain bound for the computation costs of an expression is surpassed. This prevents non-termination in the analysis.

An alternative approach combines cost information with type information in a "time system" for a higher-order, strict language ("static dependent costs") [2, 13]. The authors also show how to use this information for certain parallel machines. However, they do not extend the results to multiple classes of parallel machines as we have done with GranSim. Their system is also limited in that recursive functions can't be analysed (they rely instead on a predefined set of data-parallel operators).

Rosendahl [14] presents a program transformation that yields a time bound program for a given first-order Lisp program (his system deals with recursive functions by providing a set of transformation rules that eliminate recursion). For solving finite difference equations he uses a small set of eight transformation rules to find a closed form. This works for simple programs but fails for even moderately complex programs such as bubble-sort.

Most recently, Hughes, Pareto and Sabry [6] have developed a sized type system for a simple higher-order, lazy functional language. This type system checks upper bounds for the size of algebraic data types. Hughes et al. use this information to prove termination and liveness properties for reactive systems. However, as we have shown in this paper, sized types can also be used to analyse the costs of user-defined recursive functions.

Only a few authors have attempted to derive cost information from a lazy language in order to use it in a parallel system. Hudak and Goldberg [4] developed heuristics for improving the granularity of parallel threads based on a cost model for a lazy language. These heuristics work only in certain cases. A cost analysis for a lazy language has to take the context of an expression into account to make it compositional [1]. This can be done by using projections, modelling how much of a data structure is needed in a certain context [16]. The closest to a cost analysis for a lazy language is the cost calculus developed by Sands [15].

7 Conclusions

The sized time system introduced in this paper is the basis for performing a static cost analysis of expressions in a simple strict, polymorphic, higher-order language $\mathcal{L}$. The basic structure is that of a type inference system. Augmenting the types with size information and, in the higher-order case, with latent cost information yields a subtyping system. In order to infer the size information we use recent results on sized types [6]. The cost analysis we outlined in Section 5 can infer the cost of non-recursive functions (similar to [13]) and of recursive functions that yield linear first-order recurrences. The costs of recursive functions are produced by using a "database" of known recurrences and their closed forms. This approach is applicable for both analyses and makes it possible to tune the accuracy of the result to some degree.

Although our results are given purely for lists, Hughes et al. have already shown how to define sized types on arbitrary data structures [6]. Based on these results, it should be straightforward to extend the cost analysis in this way. When extending this analysis to lazy languages information about the demand on an expression is required. This can be modelled by projections as described in [15]. For the implicit parallelisation of purely functional languages, for which such a cost analysis would be mainly used, precise strictness information is required, to detect expressions that may be evaluated in parallel. The cost analysis could then use this strictness information in order to decide which expressions to evaluate in parallel. Therefore, a combination of a projection based strictness analysis [7] with a size and cost analysis would be the most promising approach for integrating our sized time system into an implicitly parallel lazy functional language.

In this paper we have focused on the feasibility of a static cost analysis and on the quality of the resulting cost expressions. For our application area of using cost information in the scheduling of parallel tasks it is sufficient to have relative cost information available. However, we have not yet studied the soundness and completeness of our sized time system. In future we plan to provide a step counting semantics for $\mathcal{L}$ and to study the correctness of our sized time system. One important difference from the semantic model used in [6] will be the existence of a bottom element in the semantic domain we intend to use for lists, since the strict semantics of $\mathcal{L}$ prohibits the use of infinite data structures.

Preliminary measurements of a non-trivial example program, annotated with the results of a size analysis (confirmed by type checking) and of a cost analysis (done by hand) show that the annotated version has a better runtime for a wide range of latencies. This extends results on the usability of static cost information, which has so far only been studied for specific parallel machines.

References

[1] B. Bjerner and S. Holmström. A Compositional Approach to Time Analysis of First Order Lazy Functional Programs. In *FPCA'89 — Intl. Conf. on Functional Programming Languages and Computer Architecture*, pp. 157–165. ACM Press, 1989.

[2] V. Dornic, P. Jouvelot, and D.K. Gifford. Polymorphic time systems for estimating program complexity. *ACM Letters on Programming Languages and Systems*, 1(1):33–45, March 1992.

[3] K. Hammond, H-W. Loidl, and A. Partridge. Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell. In *HPFC'95 — High Performance Functional Computing*, pp. 208–221, Denver, Colorado, April 10–12, 1995.

[4] P. Hudak and B. Goldberg. Distributed Execution of Functional Programs Using Serial Combinators. *IEEE Computer*, 34(10):881–891, October 1985.

[5] L. Huelsbergen, J.R. Larus, and A. Aiken. Using Run-Time List Sizes to Guide Parallel Thread Creation. In *LFP'94 — Conf. on Lisp and Functional Programming*, pp. 79–90, Orlando, Florida, June 27–29, 1994. ACM Press.

[6] R.J.M. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems using Sized Types. In *POPL'96 — Symp. on Principles of Programming Languages*, St Petersburg, Florida, January 1996. ACM Press.

[7] R. Kubiak, R.J.M. Hughes, and J. Launchbury. A Projection-Based Strictness Analyser for a Haskell Compiler. Technical report, Dept. of Comp. Sci., Univ. of Glasgow, May 1992.

[8] D. Le Métayer. ACE: An Automatic Complexity Evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2), April 1988.

[9] H-W. Loidl and K. Hammond. On the Granularity of Divide-and-Conquer Parallelism. In *Glasgow Workshop on Functional Programming*, Ullapool, Scotland, July 8–10, 1995. Springer-Verlag.

[10] M. Petkovsek. *Finding Closed-Form Solutions of Difference Equations by Symbolic Methods*. PhD thesis, School of Computer Science, Carnegie Mellon Univ. , September 1990.

[11] S.L. Peyton Jones, C. Clack, and J. Salkild. High-Performance Parallel Graph Reduction. In *PARLE'89 — Conf. on Parallel Architectures and Languages Europe*, LNCS 365, pp. 193–206, 1989. Springer-Verlag.

[12] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, 1992.

[13] B. Reistad and D.K. Gifford. Static Dependent Costs for Estimating Execution Time. In *LFP'94 — Conf. on Lisp and Functional Programming*, pp. 65–78, Orlando, Florida, June 27–29, June 1994. ACM Press.

[14] M. Rosendahl. Automatic Complexity Analysis. In *FPCA'89 — Intl. Conf. on Functional Programming Languages and Computer Architecture*, pp. 144–156. ACM Press, 1989.

[15] D. Sands. Complexity Analysis for a Lazy Higher-Order Language. In *ESOP'90 — European Symposium on Programming*, LNCS 432. Springer-Verlag, May 1990.

[16] P. Wadler. Strictness analysis aids time analysis. In *POPL'88 — Symp. on Principles of Programming Languages*, January 1988.

[17] B. Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9):528–539, 1975.