Cost Analysis
using Automatic Size and Time Inference

Álvaro J. Rebón Portillo[1], Kevin Hammond[1], Hans-Wolfgang Loidl[2]
and Pedro Vasconcelos[1]

[1] School of Computer Science, University of St Andrews,
St Andrews, KY16 9SS, UK, {alvaro,kh,pv}@dcs.st-and.ac.uk
[2] Ludwig-Maximilians-Universität München,
Institut für Informatik, D 80538 München, Germany
hwloidl@informatik.uni-muenchen.de

Abstract Cost information can be exploited in a variety of contexts, including parallelizing compilers, autonomic GRIDs and real-time systems. In this paper, we introduce a novel type and effect system – the *sized time system* that is capable of determining upper bounds for both time and space costs, and which we initially intend to apply to determining good granularity for parallel tasks. The analysis is defined for a simple, strict, higher-order and polymorphic functional language, $\mathcal{L}$, incorporating arbitrarily-sized list data structures. The inference algorithm implementing this analysis constructs cost- and size-terms for $\mathcal{L}$-expressions, plus constraints over free size and cost variables in those terms that can be solved to produce information for higher-order functions. The paper presents both the analysis and the inference algorithm, providing examples that illustrate the primary features of the analysis.

1 Introduction

Good cost information is useful or even vital to a large number of application areas. Examples range from small-scale real-time embedded systems through databases and parallel systems to large-scale *autonomic* GRID computations. We are especially concerned with the issue of determining appropriate task granularity, which is highly important to the efficient execution of parallel programs [1]: excessively fine granularity introduces high overheads; conversely, excessively coarse granularity can lead to poor load balance and starvation. This paper introduces a novel static cost analysis for automatically determining upper bound cost information in the presence of higher-order, polymorphic but non-recursive functions.

Our static analysis is defined as a *type and effect system* [2], a modern approach that uses standard type inference mechanisms to perform static analysis. A type system defines upper bound costs and sizes for expressions in a simple, strict, higher-order and polymorphic functional language, $\mathcal{L}$. Types include size- and cost-annotations, which are related by a subtyping relation [3]. The corresponding inference algorithm yields cost and size terms for $\mathcal{L}$-expressions, plus constraints over cost and size variables. These constraints are resolved in a separate constraint solver and combined with the cost and size terms to yield closed forms of those terms. These closed forms can be used to solve the costs of function applications.

While our focus is on cost rather than size information, we also infer a restricted form of size information primarily in order to obtain cost information in common cases where cost depends on input sizes. Our analysis is defined for both scalar and compound data structures. We have illustrated the approach with reference to recursive lists, the primary data structure used in functional languages. It should be straightforward to extend the analysis to other recursive data structures such as binary trees, or to arbitrary non-recursive data structures such as vectors or tuples.

The design of our analysis is guided by the intended use of the information it can provide. Although it is desirable to produce quality cost information, precise cost information is not absolutely essential for scheduling parallel tasks: it is sufficient to be able to identify tasks that are *potentially* large enough to be worth executing in parallel. We have structured our system so as to generate strict upper bounds on size and cost information.

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

$\mathcal{L}$ is a very simple functional language, intended solely as a vehicle to explore static analysis for cost determination. $\mathcal{L}$ is strict, polymorphic, and higher-order; with lists as its only compound data type. The abstract syntax of $\mathcal{L}$ is given below. For simplicity, variables, $v \in \mathbf{Var}$, and constants, $k \in \mathbf{Const}$, are required to be disjoint and all names are unique. Boolean values, $b \in \{\mathsf{true}, \mathsf{false}\}$, and natural numbers, $n \in \mathbb{N}$, are both in **Const**.

$$e \ := \ v \ | \ k \ | \ \lambda v.e \ | \ e_1\, e_2 \ | \ \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \ | \ \mathsf{let}\ v = e_1\ \mathsf{in}\ e_2 \, .$$

Local bindings (let) are non-recursive. An $\mathcal{L}$ program is defined to be an $\mathcal{L}$ expression.

2.1 The Type Language

$\mathcal{L}$ uses *sized types* [4], a small extension to standard Hindley-Milner polymorphic types: each type, other than function and boolean types, has a superscript specifying an upper bound for its size. For function types, a *latent cost* [5] is attached to the function arrow. The latent cost of a function type is an upper bound for the cost of evaluating the function body. In the following syntax of type expressions, α represents a type variable:

$$\tau \ := \ \alpha \ | \ \mathsf{Bool} \ | \ \mathsf{Nat}^z \ | \ \mathsf{List}^z \tau \ | \ \tau_1 \xrightarrow{z} \tau_2 \, .$$

Size and latent cost expressions are specified by z-expressions:

$$z \ := \ l \ | \ n \ | \ z_1 + z_2 \ | \ z_1 - z_2 \ | \ z_1 \times z_2 \ | \ \max(z_1, z_2) \ | \ \omega \, .$$

In these z-expressions, n is a constant natural number and l is a z-variable. The ω symbol is used to express an unbounded size. For sizes less than ω, the arithmetic operators $+$, $-$, $\times$ and max behave as usual over natural numbers ($-$ is subtraction over naturals, with $a - b = 0$, for $a < b$). When one of the operands is ω the result is ω, too; with the exception of $l - \omega$ which is 0 for $l \neq \omega$, and ω otherwise. The $\leq$ relation, which will be used later, is defined over natural numbers with $l \leq \omega$ for all l.

Polymorphism is achieved in the usual way by quantifying over all free type and size variables of a let-bound expression. The general structure of type schemes is:

$$\sigma \; := \; \tau \;\mid\; \forall x.\sigma \; ,$$

with x representing either a type or a size variable.

Since sizes are attached to types, and these may be embedded within other types, it is possible to describe the sizes of the elements of a structure as well as the structure itself, e.g.: $\mathsf{List}^5\left(\mathsf{Nat}^{10}\right)$ denotes a list whose length is at most 5 with natural numbers no larger than 10 as elements, and the type of the built-in constant $\mathsf{nil} \in \mathbf{Const}$ is: $\forall\alpha.\mathsf{List}^0\alpha$.

2.2 The Type System

Figure 1 shows our extended type system. A judgment $\Gamma \vdash e : \tau \; \$ \; z$ reads: "under the type assumptions Γ the expression e has type τ and z is an upper bound for the cost of its evaluation." The assumption set Γ contains bindings of variables, of constants, and of primitive operations to type schemes (of the form $v : \sigma$). There can be at most one binding of any name in an assumption set. Assumption sets are combined using set union. The construct $\tau[\tau'/\alpha]$ is used to denote a substitution of all free occurrences of α in τ by τ'. It extends to vectors, written as $\tau[\bar{\tau}'_i/\bar{\alpha}_i]$, by performing all substitutions simultaneously. Similarly, we allow size expression substitutions of the forms $\tau[z/l]$ and $\tau[\bar{z}_i/\bar{l}_i]$. For convenience, we will often combine type substitutions or size expression substitutions in a single substitution.

The cost model expressed in the system is parameterized through constants representing costs for elementary computations: c_{nat} and c_{bool} are the costs associated with evaluating naturals and booleans; c_{var} is the cost of accessing a variable from the environment; c_{abs} and c_{app} are the cost for creating a lambda-abstraction and executing an application step, respectively; c_{if} is the cost of executing a conditional and c_{let} is the cost of creating a let-binding.

With the exception of the $[\mathit{weak}_{st}]$ rule, the system represents a straightforward extension of the standard Hindley-Milner rules. The $[\mathit{weak}_{st}]$ rule allows weakening, i.e. relaxing the upper bounds on sizes. It makes use of the subtyping relation, $\trianglelefteq$, defined in Figure 2, which produces a set of inequalities over z-expressions. Note that with this definition of $\trianglelefteq$, the relation $\tau_1 \trianglelefteq \tau_2$ does not by itself imply that $\mathsf{List}^{z_1}\tau_1 \trianglelefteq \mathsf{List}^{z_2}\tau_2$, i.e. the subtype system is not structural. This is because there needs to be no relationship between the sizes of a structure and the elements of that structure.

$$\frac{}{\Gamma \vdash n : \mathsf{Nat}^n \ \$ \ c_{\mathsf{nat}}}\,[nat_{st}] \qquad \frac{\tau' = \tau[\bar{\tau}_i/\bar{x}_i], \quad \mathrm{FV}(\bar{\tau}_i) \cap (\mathrm{FV}(\tau) \cup \mathrm{FV}(\Gamma)) = \emptyset}{\Gamma \cup \{v : \forall \bar{x}_i \tau\} \vdash v : \tau' \ \$ \ c_{\mathsf{var}}}\,[var_{st}]$$

$$\frac{\Gamma \cup \{v : \tau_1\} \vdash e : \tau_2 \ \$ \ z}{\Gamma \vdash \lambda v.e : \tau_1 \xrightarrow{z} \tau_2 \ \$ \ c_{\mathsf{abs}}}\,[abs_{st}] \qquad \frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{z} \tau_2 \ \$ \ z_1 \quad \Gamma \vdash e_2 : \tau_1 \ \$ \ z_2}{\Gamma \vdash e_1\, e_2 : \tau_2 \ \$ \ c_{\mathsf{app}} + z + z_1 + z_2}\,[app_{st}]$$

$$\frac{}{\Gamma \vdash b : \mathsf{Bool} \ \$ \ c_{\mathsf{bool}}}\,[bool_{st}] \qquad \frac{\Gamma \vdash e_1 : \mathsf{Bool} \ \$ \ z_1 \quad \Gamma \vdash e_2 : \tau \ \$ \ z \quad \Gamma \vdash e_3 : \tau \ \$ \ z}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau \ \$ \ c_{\mathsf{if}} + z_1 + z}\,[if_{st}]$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \ \$ \ z_1 \quad \Gamma \cup \{v : \forall \bar{x}_i \tau_1\} \vdash e_2 : \tau_2 \ \$ \ z_2 \quad \bar{x}_i = \mathrm{FV}(\tau_1) \backslash \mathrm{FV}(\Gamma)}{\Gamma \vdash \mathsf{let}\ v\ =\ e_1\ \mathsf{in}\ e_2 : \tau_2 \ \$ \ c_{\mathsf{let}} + z_1 + z_2}\,[let_{st}]$$

$$\frac{\Gamma \vdash e : \tau \ \$ \ z \quad \tau \trianglelefteq \tau' \quad z \leq z'}{\Gamma \vdash e : \tau' \ \$ \ z'}\,[weak_{st}]$$

Figure 1. A sized time system for $\mathcal{L}$

$$\frac{}{\tau \trianglelefteq \tau}\,[\mathrm{reflex}_{\trianglelefteq}] \qquad \frac{\tau_1 \trianglelefteq \tau_2 \quad \tau_2 \trianglelefteq \tau_3}{\tau_1 \trianglelefteq \tau_3}\,[\mathrm{trans}_{\trianglelefteq}] \qquad \frac{\tau_1 \trianglelefteq \tau_1' \quad \tau_2' \trianglelefteq \tau_2 \quad l' \leq l}{\tau_1' \xrightarrow{l'} \tau_2' \trianglelefteq \tau_1 \xrightarrow{l} \tau_2}\,[\mathrm{abs}_{\trianglelefteq}]$$

$$\frac{z_1 \leq z_2}{\mathsf{Nat}^{z_1} \trianglelefteq \mathsf{Nat}^{z_2}}\,[\mathrm{nat}_{\trianglelefteq}] \qquad \frac{z_1 \leq z_2 \quad \tau_1 \trianglelefteq \tau_2}{\mathsf{List}^{z_1}\,\tau_1 \trianglelefteq \mathsf{List}^{z_2}\,\tau_2}\,[\mathrm{list}_{\trianglelefteq}]$$

Figure 2. Subtyping relation

The $[abs_{st}]$ rule infers the cost of evaluating the body of a lambda abstraction as a latent cost. The cost for the abstraction itself is just the parameter c_{abs}.

In the $[app_{st}]$ rule we add the latent cost z to the costs of obtaining the function and argument. Note that in this rule the type of the function's domain must match exactly the type of the argument. Since types can be weakened by relaxing their size bounds, this means that the size bound of the argument must be no greater than the size given in the type of the function's domain.

As an example, consider the application of a function, f, of type $\mathsf{Nat}^5 \xrightarrow{1} \mathsf{Nat}^{10}$ to an expression, e, of type Nat^3. To type the expression $(f\,e)$ we have to use the $[weak_{st}]$ rule in either of the following two ways: 1) weaken the type of f to $\mathsf{Nat}^3 \xrightarrow{1} \mathsf{Nat}^{10}$; or 2) weaken the type of e to Nat^5. Then we can apply the $[app_{st}]$ rule to infer Nat^{10} as the type of $(f\,e)$.

A problem arises when e.g. f is of type $\mathsf{Nat}^3 \xrightarrow{1} \mathsf{Nat}^{10}$ and e is of type Nat^5. There is no way, using the subtyping relation defined in Figure 2, to weaken either of these two types (or even both) so that it matches the other one in the way described in the $[app_{st}]$ rule. The type system, therefore, rejects the expression as badly typed, although it has a well-formed Hindley-Milner type, i.e. Nat. Since, for pragmatic reasons, the analysis must not fail on any legitimate program, such behavior is not acceptable.

A variant of the application rule must therefore be introduced (see Figure 3). The new rule infers the correct Hindley-Milner type for the result of the application, but loses all the invalid size information. The auxiliary function L recursively loses all the size information in the result type, by replacing it with the unbounded value ω.

$$\frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{z} \tau_2 \ \$\ z_1 \quad \Gamma \vdash e_2 : \tau_1' \ \$\ z_2 \quad \mathrm{L}(\tau_1) = \mathrm{L}(\tau_1')}{\Gamma \vdash e_1\, e_2 : \mathrm{L}(\tau_2) \ \$\ \omega}\ [app'_{st}]$$

$$\begin{array}{lll}
\mathrm{L} \quad :: \tau \to \tau & \mathrm{L}(\mathsf{Bool}) = \mathsf{Bool} & \mathrm{L}(\mathsf{List}^z\,\tau) \ = \mathsf{List}^\omega\,\mathrm{L}(\tau) \\
\mathrm{L}(\alpha) = \alpha & \mathrm{L}(\mathsf{Nat}^z) = \mathsf{Nat}^\omega & \mathrm{L}(\tau_1 \xrightarrow{z} \tau_2) = \mathrm{L}(\tau_1) \xrightarrow{\omega} \mathrm{L}(\tau_2)
\end{array}$$

Figure 3. Extra rule for application

Since $[app'_{st}]$ discards all size information, it is preferable to use a combination of the $[weak_{st}]$ rule plus the normal $[app_{st}]$ rule wherever these are applicable, i.e. when $\tau_1' \trianglelefteq \tau_1$. The $[app'_{st}]$ should be restricted to situations when the Hindley-Milner types are identical but the subtyping relation on sized types doesn't hold.

3 The Inference Algorithm

In this section we discuss an inference algorithm that yields cost and size expressions plus an associated set of constraints for all top level function definitions in $\mathcal{L}$. The constraint set is solved in the implementation via a separate constraint solver.

In order to simplify the presentation and without loss of generality, we will describe the algorithm specialized for one specific cost model, namely the one that calculates (an upper bound on) the *number of applications* that are required for the reduction of an $\mathcal{L}$-expression. This is achieved by setting the values of the cost parameters as $c_{\mathsf{app}} = 1$ and $c_{\mathsf{nat}} = c_{\mathsf{bool}} = c_{\mathsf{var}} = c_{\mathsf{if}} = c_{\mathsf{abs}} = c_{\mathsf{let}} = 0$. The algorithm can be modified for other cost models by choosing different values for these parameters; alternatively, by leaving the parameters as free variables, we would obtain a constraint set representing a *parametric cost model*.

A key question in designing a type reconstruction algorithm for our type system is where to apply the weakening rule, as all other rules are both structural and exhaustive. Our algorithm uses weakening at precisely one location: the conditional case, in order to find a super-type of the types of both branches [4], that is in order to obtain an upper bound on the costs of the conditional branches. It also uses *domain matching* in order to construct a correct subtyping relation between the type of a concrete argument to a function and that function's domain. Both of these uses are described below.

In contrast to classical type inference algorithms, the algorithm presented here returns a constraint set as part of the output [4]. The idea behind this is to simplify the weakening process by dealing only with variables in the type, maintaining the full, complex expressions only in the set of constraints. The two types of constraints we allow are as follows:

$$\begin{aligned} C &::= c \mid cC \\ c &::= z_1 \leq z_2 \mid z_1 = z_2 \,, \end{aligned}$$

with z_i being z-expressions.

With this idea of allowing only size variables to appear in a type expression, the syntax for type schemes has to be changed to:

$$\sigma \;:=\; (\tau, C) \mid \forall x.\sigma \,,$$

with C representing the constraints for the size variables involved in τ. As an example, the type scheme $\forall m.\mathsf{Nat}^m \xrightarrow{1} \mathsf{Nat}^{m+1}$ is now written as

$$\forall m \forall n \forall k.(\mathsf{Nat}^m \xrightarrow{k} \mathsf{Nat}^n, \{n = m + 1, k = 1\}) \,.$$

Substitutions over types are denoted by θ, whereas substitutions over z-expressions are denoted by ϑ. Notationally, the application of a substitution to a type expression is denoted by juxtaposition, as is also the composition of two or more substitutions. To shorten the notation, we abbreviate the substitution composition $\theta_{i+n}\theta_{i+n-1}\ldots\theta_i$ as Θ_i^{i+n}.

Figure 4 specifies a size reconstruction algorithm in the same inference style that has been used for the type system of $\mathcal{L}$. The arguments to the algorithm are a type environment Γ and the expression to be analyzed. The result of the algorithm is a 4-tuple $\langle \tau, \theta, z, C \rangle$, where τ is the type of the expression, θ is a substitution on types, z is the cost of the expression and C is the constraint set, i.e. a set of inequalities over z-expressions. The algorithm is similar to that developed by Reistad and Gifford [5] for the cost reconstruction of FX programs.

As the $[nat_{st}]$ case indicates, the algorithm maintains the invariant that size annotations in sized types are always variables. Thus, an explicit constraint $l = n$ has to be added to the constraint set in the $[nat_{st}]$ case rather than just using Nat^n as in the sized type system.

The $[if_{st}]$ case shows how the unification and weakening algorithms are used to guarantee that both branches have the same type. Note here that the substitution

$$\frac{}{\Gamma \vdash n : \langle \mathsf{Nat}^{l}, [], 0, \{l = n\}\rangle}\,[nat_{st}] \qquad \frac{\vartheta = [\bar{l}'_i/\bar{l}_i] \quad \theta = [\bar{\alpha}'_i/\bar{\alpha}_i] \quad \text{fresh } \bar{l}'_i, \bar{\alpha}'_i}{\Gamma \cup \{v : \forall \bar{l}_i \bar{\alpha}_i(\tau, C)\} \vdash v : \langle \vartheta\theta\tau, [], 0, \vartheta C\rangle}\,[var_{st}]$$

$$\frac{}{\Gamma \vdash b : \langle \mathsf{Bool}, [], 0, \emptyset\rangle}\,[bool_{st}] \qquad \frac{\Gamma \cup \{v : (\alpha, \emptyset)\} \vdash e : \langle \tau, \theta, z, C\rangle, \quad \text{fresh } \alpha, l}{\Gamma \vdash \lambda v.e : \langle \theta\alpha \xrightarrow{l} \tau, \theta, 0, \{l = z\} \cup C\rangle}\,[abs_{st}]$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \langle \tau_1, \theta_1, z_1, C_1\rangle \quad \theta_1\Gamma \vdash e_2 : \langle \tau_2, \theta_2, z_2, C_2\rangle \\ \theta = \mathcal{U}(\theta_2\tau_1, \tau_2 \xrightarrow{l} \alpha) \quad C_3 = \mathcal{D}(\theta\theta_2\tau_1, \theta(\tau_2 \xrightarrow{l} \alpha)) \quad \text{fresh } \alpha, l\end{array}}{\Gamma \vdash e_1\, e_2 : \langle \theta\alpha, \theta\Theta_1^2, 1{+}l{+}z_1{+}z_2, \bigcup_{i=1}^{3} C_i\rangle}\,[app_{st}]$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \langle \tau_1, \theta_1, z_1, C_1\rangle \quad \theta_1\Gamma \vdash e_2 : \langle \tau_2, \theta_2, z_2, C_2\rangle \quad \Theta_1^2\Gamma \vdash e_3 : \langle \tau_3, \theta_3, z_3, C_3\rangle \\ \theta_4 = \mathcal{U}(\Theta_2^3\tau_1, \mathsf{Bool}) \quad \theta_5 = \mathcal{U}(\Theta_3^4\tau_2, \theta_4\tau_3) \quad \langle \vartheta, C_4\rangle = \mathcal{W}(\Theta_3^5\tau_2, \Theta_4^5\tau_3)\end{array}}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \langle \vartheta\Theta_4^5\tau_3, \Theta_1^5, z_1{+}\max(z_2, z_3), \bigcup_{i=1}^{4} C_i\rangle}\,[if_{st}]$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \langle \tau_1, \theta_1, z_1, C_1\rangle \\ \theta_1\Gamma \cup \{v : \forall \bar{l}_i \forall \bar{\alpha}_i(\tau_1, C_1)\} \vdash e_2 : \langle \tau_2, \theta_2, z_2, C_2\rangle \\ \bar{\alpha}_i = \mathrm{FTV}(\tau_1)\backslash\mathrm{FTV}(\theta_1\Gamma) \quad \bar{l}_i = (\mathrm{FZV}(\tau_1) \cup \mathrm{FZV}(C_1))\backslash\mathrm{FZV}(\theta_1\Gamma)\end{array}}{\Gamma \vdash \mathsf{let}\ v\ =\ e_1\ \mathsf{in}\ e_2 : \langle \tau_2, \Theta_1^2, z_1{+}z_2, \bigcup_{i=1}^{2} C_i\rangle}\,[let_{st}]$$

Figure 4. A sized time reconstruction algorithm for $\mathcal{L}$

obtained from weakening, ϑ, is not part of the second component of the output tuple. This is because applications of the weakening rule are not propagated beyond the scope of the conditional, so avoiding over-weakening of the collected size information. The example in Section 4 illustrates the use of this rule.

The $[app_{st}]$ case deals with the first variant of application described in the type system. Applications requiring the alternative $[app'_{st}]$ rule in the type system will generate an inconsistent result constraint set.

Since the weakening algorithm cannot be used directly, a new function, $\mathcal{D}$ (see Figure 5), is therefore introduced. This function constructs the set of constraints needed for the argument type to match (be a subtype of) the type of the function's domain, according to the subtyping relation previously presented in Figure 2. The types given as arguments must share the same underlying Hindley-Milner structure.

The unification algorithm presented in Figure 6 outputs a substitution over types which, when applied to each of its arguments, makes them identical under Hindley-Milner, but perhaps containing different size information. The auxiliary operator ν is used to achieve this.

A separate function, $\mathcal{W}$, is used to weaken two types, computing a pair consisting of a substitution over z-expressions and a set of constraints which

$$
\begin{array}{ll}
\mathcal{D} & :: \tau \to \tau \to C \\
\mathcal{D}(\alpha, \alpha) & = \emptyset \\
\mathcal{D}(\mathsf{Bool}, \mathsf{Bool}) & = \emptyset
\end{array}
\qquad
\begin{array}{ll}
\mathcal{D}(\mathsf{Nat}^{l_1}, \mathsf{Nat}^{l_2}) & = \{l_1 \leq l_2\} \\
\mathcal{D}(\mathsf{List}^{l_1}\,\tau_1, \mathsf{List}^{l_2}\,\tau_2) & = \{l_1 \leq l_2\} \cup \mathcal{D}(\tau_1, \tau_2) \\
\mathcal{D}(\tau_1 \xrightarrow{l} \tau_2, \tau_1' \xrightarrow{l'} \tau_2') & = \{l \leq l'\} \cup \mathcal{D}(\tau_1', \tau_1) \cup \mathcal{D}(\tau_2, \tau_2')
\end{array}
$$

Figure 5. Domain matching test

$$
\begin{array}{ll}
\mathcal{U} & :: \tau \to \tau \to \theta \\
\mathcal{U}(\alpha, \tau) & = [\nu(\tau)/\alpha] \\
\mathcal{U}(\mathsf{Bool}, \mathsf{Bool}) & = [] \\
\mathcal{U}(\mathsf{Nat}^{l_1}, \mathsf{Nat}^{l_2}) & = [] \\
\mathcal{U}(\mathsf{List}^{l_1}\,\tau_1, \mathsf{List}^{l_2}\,\tau_2) & = \mathcal{U}(\tau_1, \tau_2) \\
\mathcal{U}(\tau_1 \xrightarrow{l} \tau_2, \tau_1' \xrightarrow{l'} \tau_2') & = \mathcal{U}(\theta\tau_2, \theta\tau_2')\theta \\
\quad \text{where } \theta = \mathcal{U}(\tau_1, \tau_1') &
\end{array}
\qquad
\begin{array}{lll}
\nu & :: \tau \to \tau & \\
\nu(\alpha) & = \alpha & \\
\nu(\mathsf{Bool}) & = \mathsf{Bool} & \\
\nu(\mathsf{Nat}^{l}) & = \mathsf{Nat}^{l'}, & \text{fresh } l' \\
\nu(\mathsf{List}^{l}\,\tau) & = \mathsf{List}^{l'}\,\nu(\tau), & \text{fresh } l' \\
\nu(\tau_1 \xrightarrow{l} \tau_2) & = \nu(\tau_1) \xrightarrow{l'} \nu(\tau_2), & \text{fresh } l'
\end{array}
$$

Figure 6. Unification algorithm

$$
\begin{array}{ll}
\mathcal{W} & :: \tau \to \tau \to \langle \vartheta, C \rangle \\
\mathcal{W}(\alpha, \alpha) & = \langle [], \emptyset \rangle \\
\mathcal{W}(\mathsf{Bool}, \mathsf{Bool}) & = \langle [], \emptyset \rangle \\
\mathcal{W}(\mathsf{Nat}^{l_1}, \mathsf{Nat}^{l_2}) & = \langle [l/l_1, l/l_2], \\
 & \quad \{l_1 \leq l, l_2 \leq l\} \rangle \\
\quad l \text{ is fresh} & \\
\mathcal{W}(\mathsf{List}^{l_1}\,\tau_1, \mathsf{List}^{l_2}\,\tau_2) & = \langle [l/l_1, l/l_2]\vartheta, \\
 & \quad \{l_1 \leq l, l_2 \leq l\} \cup C \rangle \\
\quad l \text{ is fresh} & \\
\quad \langle \vartheta, C \rangle = \mathcal{W}(\tau_1, \tau_2) & \\
\mathcal{W}(\tau_1 \xrightarrow{l'} \tau_2, \tau_1' \xrightarrow{l''} \tau_2') & = \langle \vartheta, C \rangle \\
\quad l \text{ is fresh} & \\
\quad \vartheta = [l/l', l/l'']\vartheta_2\vartheta_1 & \\
\quad C = \{l' \leq l, l'' \leq l\} C_1 \cup C_2 & \\
\quad \langle \vartheta_1, C_1 \rangle = \mathcal{S}(\tau_1, \tau_1') & \\
\quad \langle \vartheta_2, C_2 \rangle = \mathcal{W}(\tau_2, \tau_2') &
\end{array}
\qquad
\begin{array}{ll}
\mathcal{S} & :: \tau \to \tau \to \langle \vartheta, C \rangle \\
\mathcal{S}(\alpha, \alpha) & = \langle [], \emptyset \rangle \\
\mathcal{S}(\mathsf{Bool}, \mathsf{Bool}) & = \langle [], \emptyset \rangle \\
\mathcal{S}(\mathsf{Nat}^{l_1}, \mathsf{Nat}^{l_2}) & = \langle [l/l_1, l/l_2], \\
 & \quad \{l \leq l_1, l \leq l_2\} \rangle \\
\quad l \text{ is fresh} & \\
\mathcal{S}(\mathsf{List}^{l_1}\,\tau_1, \mathsf{List}^{l_2}\,\tau_2) & = \langle [l/l_1, l/l_2]\vartheta, \\
 & \quad \{l \leq l_1, l \leq l_2\} \cup C \rangle \\
\quad l \text{ is fresh} & \\
\quad \langle \vartheta, C \rangle = \mathcal{S}(\tau_1, \tau_2) & \\
\mathcal{S}(\tau_1 \xrightarrow{l'} \tau_2, \tau_1' \xrightarrow{l''} \tau_2') & = \langle \vartheta, C \rangle \\
\quad l \text{ is fresh} & \\
\quad \vartheta = [l/l', l/l'']\vartheta_2\vartheta_1 & \\
\quad C = \{l \leq l', l \leq l''\} C_1 \cup C_2 & \\
\quad \langle \vartheta_1, C_1 \rangle = \mathcal{W}(\tau_1, \tau_1') & \\
\quad \langle \vartheta_2, C_2 \rangle = \mathcal{S}(\tau_2, \tau_2') &
\end{array}
$$

Figure 7. Weakening and strengthening algorithms

collectively make the two types equal. This function and its dual, strengthening, $\mathcal{S}$, which is needed because of the contravariance present in the subtyping relation for functional types, are shown in Figure 7. As with the function $\mathcal{D}$, both of these algorithms require their arguments to share the same underlying Hindley-Milner structure.

4 Examples

We now present three examples to illustrate the inference algorithm just described: one using conditionals, one using lists and one using higher-order functions.

Example 1: Conditionals

Figure 8 depicts an inference for

$$\Gamma = \{\, inc : \sigma_{inc},\ p : (\mathsf{Bool}, \emptyset) \,\} \overset{?}{\vdash} \lambda v.\mathsf{if}\ p\ \mathsf{then}\ v\ \mathsf{else}\ inc\ v$$

where $\sigma_{inc} = \forall m \forall n \forall c.(\mathsf{Nat}^m \xrightarrow{c} \mathsf{Nat}^n, \{n = m + 1, c = 0\})$. We use e to denote the sub-expression 'if p then v else $inc\ v$'.

We are interested in the final type and cost for the expression, therefore we solve the resulting set of constraints:

$$\{m' \leq a, n' \leq a, n = m + 1, c = 0, m' \leq m, n \leq n', c \leq l, d = \max(0, 1 + l)\} .$$

As the type we obtain is the abstraction $\mathsf{Nat}^{m'} \xrightarrow{d} \mathsf{Nat}^a$, we need the least upper bounds for a and d, for a given m'. These are, according to the set of constraints, $m' + 1$ and 2, respectively. Thus the type for the expression is

$$\mathsf{Nat}^{m'} \xrightarrow{1} \mathsf{Nat}^{m'+1}$$

and the cost of evaluating it is zero, since it is a lambda abstraction (Section 3).

Example 2: Lists

In this example we show how the system deals with lists, adding the types of the primitive constant *nil* and the primitive operator *cons* to the initial environment. The inference for

$$\Gamma = \{\, nil : \sigma_{nil},\ cons : \sigma_{cons} \,\} \overset{?}{\vdash} \lambda v.cons\ \ v\ nil$$

with

$$\sigma_{nil} = \forall m \forall \alpha.(\mathsf{List}^m \alpha, \{m = 0\}) ,$$
$$\sigma_{cons} = \forall m \forall n \forall c \forall d \forall \alpha.(\alpha \xrightarrow{c} \mathsf{List}^m \alpha \xrightarrow{d} \mathsf{List}^n \alpha, \{n = m + 1, c = 0, d = 0\}) ,$$

is developed in figure 9, where e has been used to denote the sub-expression '*cons v nil*'. The set of constraints obtained is

$$\{d = b + 1, a = 0, c = 0, a \leq k, e \leq b, c \leq f, d \leq g, h = 0, f \leq l, h \leq e, g \leq i\} ,$$

from which the least upper bounds for k, l and i can be inferred as 0 ($0 = a \leq k$), 0 ($0 = c \leq f \leq l$) and 1 ($0 = h \leq e \leq b$, $b + 1 = d \leq g \leq i$), respectively. We can consequently express the final type for '$\lambda v.cons\ v\ nil$' as

$$\beta \xrightarrow{2} \mathsf{List}^1 \beta .$$

$$\left.\mathcal{W}(\mathsf{Nat}^{m'},\mathsf{Nat}^{n'}) = \langle [a/m', a/n'],\, C_6 = \{m' \leq a, n' \leq a\}\rangle\right\}(6)$$
$$\left.\mathcal{U}(\mathsf{Nat}^{m'},\mathsf{Nat}^{n'}) = []\right\}(5)$$
$$\left.\mathcal{U}(\mathsf{Bool},\mathsf{Bool}) = []\right\}(4)$$

$$(3.1)\left\{\frac{}{\Gamma' \vdash inc : \langle \mathsf{Nat}^m \xrightarrow{c} \mathsf{Nat}^n, [], 0, C_{3.1} = \{n = m+1, c = 0\}\rangle}\,[var_{st}]\right.$$
$$(3.2)\left\{\frac{}{\Gamma' \vdash v : \langle \alpha, [], 0, \emptyset\rangle}\,[var_{st}]\right.$$
$$(3.3)\left\{\theta_{3.3} = \mathcal{U}(\mathsf{Nat}^m \xrightarrow{c} \mathsf{Nat}^n, \alpha \xrightarrow{l} \beta) = [\mathsf{Nat}^{m'}/\alpha, \mathsf{Nat}^{n'}/\beta]\right.$$
$$(3.4)\left\{C_{3.4} = \mathcal{D}(\mathsf{Nat}^m \xrightarrow{c} \mathsf{Nat}^n, \mathsf{Nat}^{m'} \xrightarrow{l} \mathsf{Nat}^{n'}) = \{m' \leq m, n \leq n', c \leq l\}\right.$$

$$\left.\frac{(3.1)\qquad(3.2)\qquad(3.3)\qquad(3.4)}{\Gamma' \vdash inc\ v : \langle \mathsf{Nat}^{n'}, \theta_{3.3}, 1+l, C_3 = C_{3.1} \cup C_{3.4}\rangle}\,[app_{st}]\right\}(3)$$
$$\left.\frac{}{\Gamma' \vdash v : \langle \alpha, [], 0, \emptyset\rangle}\,[var_{st}]\right\}(2)$$
$$\left.\frac{}{\Gamma' \vdash p : \langle \mathsf{Bool}, [], 0, \emptyset\rangle}\,[var_{st}]\right\}(1)$$

$$\frac{\dfrac{(1)\qquad(2)\qquad(3)\qquad(4)\qquad(5)\qquad(6)}{\Gamma' = \Gamma \cup \{v : (\alpha, \emptyset)\} \vdash e : \langle \mathsf{Nat}^a, \theta_{3.3}, \max(0, 1+l), C_3 \cup C_6\rangle}\,[if_{st}]}{\Gamma = \{\ inc : \sigma_{inc},\ p : (\mathsf{Bool}, \emptyset)\ \} \vdash \lambda v.e :\ \langle \mathsf{Nat}^{m'} \xrightarrow{d} \mathsf{Nat}^a, \theta_{3.3}, 0, C_3 \cup C_6 \cup \{d = \max(0, 1+l)\}\rangle}\,[abs_{st}]$$

Figure 8. A type reconstruction for '$\lambda v.$if p then v else $inc\ v$'

Example 3: Higher-Order Functions

Finally, in order to illustrate the application of our analysis to programs using higher-order functions and to demonstrate its usefulness even without being able to infer costs for recursive definitions, we present a cost inference for a function that sums a list of naturals using partial application of the standard left-fold function over lists:

$$sum\ =\ foldl\ (+)\ 0$$

Figure 10 shows the inference for the sized-time type of the function body,

$$\Gamma = \{+ : \sigma_{plus},\, foldl : \sigma_{foldl}\} \overset{?}{\vdash} foldl\ (+)\ 0$$

assuming suitable types for $(+)$ and $foldl$:

$$\sigma_{plus} = \forall n \forall m \forall p \forall c_1 \forall c_2.\, (\mathsf{Nat}^n \xrightarrow{c_1} \mathsf{Nat}^m \xrightarrow{c_2} \mathsf{Nat}^p,\ \{c_1 = 0, c_2 = 0, p = n+m\})$$
$$\sigma_{foldl} = \forall \ldots\ ((\alpha \xrightarrow{c_1} \beta \xrightarrow{c_2} \alpha) \xrightarrow{c_3} \alpha \xrightarrow{c_4} \mathsf{List}^k \beta \xrightarrow{c_5}, \{c_3 = 0, c_4 = 0, c_5 = k(2 + c_1 + c_2)\})$$

$$C_4 = \mathcal{D}(\mathsf{List}^e\beta \xrightarrow{f} \mathsf{List}^g\beta, \mathsf{List}^h\delta \xrightarrow{l} \mathsf{List}^i\beta) = \{f \le l, h \le e, g \le i\} \Big\}\ (4)$$

$$\theta_3 = \mathcal{U}(\mathsf{List}^e\beta \xrightarrow{f} \mathsf{List}^g\beta, \mathsf{List}^h\delta \xrightarrow{l} \epsilon) = [\beta/\delta, \mathsf{List}^i\beta/\epsilon] \Big\}\ (3)$$

$$\left.\frac{}{\Gamma' \vdash nil : \langle \mathsf{List}^h\delta, [], 0, C_2 = \{h = 0\}\rangle}\,[var_{st}]\right\}\ (2)$$

$$(1.4) \left\{ \begin{aligned} C_{1.4} &= \mathcal{D}(\beta \xrightarrow{a} \mathsf{List}^b\beta \xrightarrow{c} \mathsf{List}^d\beta, \beta \xrightarrow{k} \mathsf{List}^e\beta \xrightarrow{f} \mathsf{List}^g\beta) \\ &= \{a \le k, e \le b, c \le f, d \le g\} \end{aligned} \right.$$

$$(1.3) \left\{ \theta_{1.3} = \mathcal{U}(\beta \xrightarrow{a} \mathsf{List}^b\beta \xrightarrow{c} \mathsf{List}^d\beta, \alpha \xrightarrow{k} \gamma) = [\beta/\alpha, \mathsf{List}^e\beta \xrightarrow{f} \mathsf{List}^g\beta/\gamma] \right.$$

$$(1.2) \left\{ \frac{}{\Gamma' \vdash v : \langle \alpha, [], 0, \emptyset\rangle}\,[var_{st}] \right.$$

$$(1.1) \left\{ \frac{}{\Gamma' \vdash cons : \langle \beta \xrightarrow{a} \mathsf{List}^b\beta \xrightarrow{c} \mathsf{List}^d\beta, [], 0, C_{1.1} = \{d = b+1, a = 0, c = 0\}\rangle}\,[var_{st}] \right.$$

$$\left.\frac{(1.1) \quad (1.2) \quad (1.3) \quad (1.4)}{\Gamma' \vdash cons\ \ v : \langle \mathsf{List}^e\beta \xrightarrow{f} \mathsf{List}^g\beta, \theta_1 = \theta_{1.3}, 1 + k, C_1 = C_{1.1} \cup C_{1.4}\rangle}\,[app_{st}]\right\}\ (1)$$

$$\dfrac{\dfrac{(1) \quad (2) \quad (3) \quad (4)}{\Gamma' = \Gamma \cup \{v : (\alpha, \emptyset)\} \vdash e : \langle \mathsf{List}^i\beta, \theta_3\theta_1, 2 + k + l, C_1 \cup C_2 \cup C_4\rangle}\,[app_{st}]}{\begin{aligned}\Gamma = \{\, nil : \sigma_{nil},\ \ cons : \sigma_{cons} \,\} \vdash \lambda v.e : \langle & \beta \xrightarrow{m} \mathsf{List}^i\beta, \theta_3\theta_1, 0, \\ & C_1 \cup C_2 \cup C_4 \cup \{m = 2 + k + l\}\rangle\end{aligned}}\,[abs_{st}]$$

Figure 9. A type reconstruction for '$\lambda v.\ cons\ v\ nil$'

Our assumption σ_{foldl} captures the fact that *foldl* must apply its argument function k times (where k is the length of the list) to two separate arguments. Since we are using λ-calculus style binary function application, each use of the argument function thus requires two distinct applications in addition to the latent cost for evaluating the function body. The quantifier ranges over all free type and cost variables.

Looking at just three constraints from $C_{1.1}$ and $C_{1.4}$ we can infer that $p = n + m \wedge n_1 \le n \wedge p \le n_1 \Rightarrow n = p = \omega$ because the constraints must hold for values of $m > 0$. Simplifying the remainder of the constraint set (not shown), we obtain the following type for the function:

$$\sigma_{sum} = \mathsf{List}^k\mathsf{Nat}^m \xrightarrow{2k} \mathsf{Nat}^\omega$$

As we would expect, the number of applications for executing *sum* is precisely twice the number of elements in the list because (+) incurs no additional latent costs. Unfortunately, in this case the system is unable to infer any useful size information for the result of the *sum*. Is is not difficult to see that the best upper-bound size for *sum* would be

$$\sigma'_{sum} = \mathsf{List}^k\mathsf{Nat}^m \xrightarrow{2k} \mathsf{Nat}^{km}$$

$$
(1.1)\begin{cases}\dfrac{}{\mathit{\Gamma}\vdash \mathit{foldl} : \langle(\alpha \xrightarrow{c_1} \beta \xrightarrow{c_2} \alpha) \xrightarrow{c_3} \alpha \xrightarrow{c_4} \mathsf{List}^k \beta \xrightarrow{c_5} \alpha, [], 0, C_{1.1}\rangle}\,[\mathit{var}_{st}] \\ \qquad\text{where } C_{1.1} = \{c_3 = 0, c_4 = 0, c_5 = k(2 + c_1 + c_2)\}\end{cases}
$$

$$
(1.2)\begin{cases}\dfrac{}{\mathit{\Gamma}\vdash (+) : \langle \mathsf{Nat}^n \xrightarrow{d_1} \mathsf{Nat}^m \xrightarrow{d_2} \mathsf{Nat}^p, [], 0, C_{1.2} = \{d_1 = 0, d_2 = 0, p = n + m\}\rangle}\,[\mathit{var}_{st}]\end{cases}
$$

$$
(1.3)\begin{cases}\theta_{1.3} = \mathcal{U}((\alpha \xrightarrow{c_1} \beta \xrightarrow{c_2} \alpha) \xrightarrow{c_3} \alpha \xrightarrow{c_4} \mathsf{List}^k \beta \xrightarrow{c_5} \alpha,\ (\mathsf{Nat}^n \xrightarrow{d_1} \mathsf{Nat}^m \xrightarrow{d_2} \mathsf{Nat}^p) \xrightarrow{d_3} \gamma) = \\ \qquad = [\mathsf{Nat}^{n_1}/\alpha,\ \mathsf{Nat}^{m_1}/\beta,\ (\mathsf{Nat}^{n_2} \xrightarrow{d_4} \mathsf{List}^{k_1} \mathsf{Nat}^{m_2} \xrightarrow{d_5} \mathsf{Nat}^{p_1})/\gamma]\end{cases}
$$

$$
(1.4)\begin{cases}C_{1.4} = \mathcal{D}(\theta_{1.3}((\alpha \xrightarrow{c_1} \beta \xrightarrow{c_2} \alpha) \xrightarrow{c_3} \alpha \xrightarrow{c_4} \mathsf{List}^k \beta \xrightarrow{c_5} \alpha), \\ \qquad\qquad \theta_{1.3}((\mathsf{Nat}^n \xrightarrow{d_1} \mathsf{Nat}^m \xrightarrow{d_2} \mathsf{Nat}^p) \xrightarrow{d_3} \gamma)) = \{n_1 \leq n,\ p \leq n_1,\ n_1 \leq p_1 \ldots\}\end{cases}
$$

$$
\left.\dfrac{(1.1) \qquad (1.2) \qquad (1.3) \qquad (1.4)}{\mathit{\Gamma}\vdash \mathit{foldl}\,(+) : \langle \mathsf{Nat}^{n_2} \xrightarrow{d_4} \mathsf{List}^{k_1} \mathsf{Nat}^{m_2} \xrightarrow{d_5} \mathsf{Nat}^{n_3}, \theta_{1.3}, 1, C_{1.1} \cup C_{1.2} \cup C_{1.4}\rangle}\,[\mathit{app}_{st}]\right\}(1)
$$

$$
\left.\dfrac{}{\mathit{\Gamma}\vdash 0 : \langle \mathsf{Nat}^z, [], 0, \{z = 0\}\rangle}\,[\mathit{nat}_{st}]\right\}(2)
$$

$$
\left.\theta_3 = \mathcal{U}(\mathsf{Nat}^{n_2} \xrightarrow{d_4} \mathsf{List}^{k_1} \mathsf{Nat}^{m_2} \xrightarrow{d_5} \mathsf{Nat}^{p_1},\ \mathsf{Nat}^z \xrightarrow{d_6} \delta) = [(\mathsf{List}^{k_2} \mathsf{Nat}^{m_3} \xrightarrow{d_7} \mathsf{Nat}^{p_2})/\delta]\right\}(3)
$$

$$
\left.\begin{array}{l}C_4 = \mathcal{D}(\mathsf{Nat}^{n_2} \xrightarrow{d_4} \mathsf{List}^{k_1} \mathsf{Nat}^{m_2} \xrightarrow{d_5} \mathsf{Nat}^{p_1},\ \mathsf{Nat}^z \xrightarrow{d_6} \mathsf{List}^{k_2} \mathsf{Nat}^{m_3} \xrightarrow{d_7} \mathsf{Nat}^{p_2}) = \\ \quad = \{p_1 \leq p_2 \ldots\}\end{array}\right\}(4)
$$

$$
\dfrac{(1) \qquad (2) \qquad (3) \qquad (4)}{\mathit{\Gamma}\vdash \mathit{foldl}\,(+)\ 0 : \langle \mathsf{List}^{k_2} \mathsf{Nat}^{m_3} \xrightarrow{d_7} \mathsf{Nat}^{p_2}, \theta_3\theta_{1.3}, 2, C_{1.1} \cup C_{1.2} \cup C_{1.4} \cup C_4\rangle}\,[\mathit{app}_{st}]
$$

Figure 10. A type reconstruction for *sum*

The size we obtain is indeed a super-type of this type, and is therefore a safe approximation. Moreover, our type system will not even accept σ'_{sum} as a valid type for *sum*, and we are therefore unable to derive this using our inference algorithm. Informally, the reason for this is that the argument function to *foldl* must have the same polymorphic type α both for its first argument and for its result. The sharing that is introduced in this way implies an *aliasing on size annotations* in the argument and result types. When the $(+)$ is supplied as an argument to *foldl*, this aliasing can only be resolved by substituting ω in both places.

This loss of size-information limits the quality of the analysis we can infer because further composition of *sum* with other functions will inevitably yield ω-sizes and costs. However, this unhappy situation only occurs with certain high-order functions: fold-type functions are essentially a "worst-case scenario" and our prototype implementation has demonstrated that the inference algorithm does preserve useful result sizes with applications of standard *map*, *filter* and *compose* functions, for example.

5 Experimental Results

This section describes experimental results obtained from the prototype implementation of our cost analysis. The inference algorithm was written as a Haskell program that takes as input a $\mathcal{L}$-expression together with a type environment that includes hand-written cost annotation for builtin and library functions such as $+$, *foldl* etc.

The output of the analysis is a sized-time type together with a set of constraints over the cost and size variables that appear in that type. These constraints are then simplified using a specially constructed solver written in a version of Prolog that includes *constraint handling rules* [6]. In the past we have experimented with other constraint solvers, including the Mozart implementation of the Oz constraint programming language [7].

Our experiment was conducted as follows: we took a representative set of non-recursive function definitions involving only naturals, booleans and lists from the Haskell language standard prelude [8] and coded them as $\mathcal{L}$-expressions. We then added a type environment containing hand-written sized-time types for all necessary primitive operations (e.g. arithmetic and boolean operations) and for those library functions involving recursive definitions (e.g. *foldl/r*, *map* and $+\!\!+$). All higher-order functions that can be expressed directly in $\mathcal{L}$ (e.g. *compose* and *flip*) were defined explicitly through let-bindings and their costs were inferred automatically using our analysis. Finally, wherever necessary, Haskell's overloaded operators were specialized to $\mathcal{L}$ monotypes (e.g. arithmetic functions were specialized to naturals).

The definitions for these examples, complete with the sized-type assumptions for the library functions used in the analysis, are available for download from the following URL: `http://www-fp.dcs.st-and.ac.uk/publications.html`.

Figure 11 summarizes the results obtained: the first column gives the name(s) of the function(s) that were analysed while the second column shows the sized-time type generated by the analysis. The third and fourth columns present a qualitative appreciation of the costs and sizes that were obtained by the analysis: a '$\surd$' indicates that the approximation obtained is *accurate* (i.e. the best that can be expressed using z-expressions), and an '$\times$' indicates a safe but inaccurate answer. The table shows that our analysis computes accurate cost information in all 16 cases, and accurate size information in 11 of these cases. Four cases where size information is inaccurate represent applications of folds, where all size information is lost. The remaing case is the *min* function, where a finite size is infered but it is larger than need be.

Some of the types that are derived may seem counter-intuitive, and thus require more detailed explanations. For example, the size of the result for *subtract* (the function that subtracts its first argument from its second argument) must be identical to that of its second argument because the first argument may be zero. The *concat* function is defined as by folding $+\!\!+$ over a list: each application of $+\!\!+$ requires $2m$ function application steps, and the fold itself incurs two extra applications for each element of the outer list. The *reverse* function is defined as a fold of '*flip* (:)' costing two applications each; the fold itself costs two

Function	Inferred type	Cost?	Size?
`max`	$\mathsf{Nat}^n \xrightarrow{0} \mathsf{Nat}^m \xrightarrow{2} \mathsf{Nat}^{\max(n,m)}$	$\surd$	$\surd$
`min`	$\mathsf{Nat}^n \xrightarrow{0} \mathsf{Nat}^m \xrightarrow{2} \mathsf{Nat}^{\max(n,m)}$	$\surd$	$\times$
`even`	$\mathsf{Nat}^n \xrightarrow{4} \mathsf{Bool}$	$\surd$	$\surd$
`odd`	$\mathsf{Nat}^n \xrightarrow{6} \mathsf{Bool}$	$\surd$	$\surd$
`compose`	$(\gamma \xrightarrow{z_1} \beta) \xrightarrow{0} (\alpha \xrightarrow{z_2} \gamma) \xrightarrow{0} \alpha \xrightarrow{z_1+z_2+2} \beta$	$\surd$	$\surd$
`flip`	$(\alpha \xrightarrow{z_1} \beta \xrightarrow{z_2} \gamma) \xrightarrow{0} \beta \xrightarrow{0} \alpha \xrightarrow{z_1+z_2+2} \gamma$	$\surd$	$\surd$
`subtract`	$\mathsf{Nat}^n \xrightarrow{0} \mathsf{Nat}^m \xrightarrow{2} \mathsf{Nat}^m$	$\surd$	$\surd$
`concat`	$\mathsf{List}^n\,(\mathsf{List}^m\,\alpha) \xrightarrow{n(2m+2)} \mathsf{List}^\omega\,\alpha$	$\surd$	$\times$
`reverse`	$\mathsf{List}^k\,\alpha \xrightarrow{4k} \mathsf{List}^\omega\,\alpha$	$\surd$	$\times$
`and`, `or`	$\mathsf{List}^k\,\mathsf{Bool} \xrightarrow{2k} \mathsf{Bool}$	$\surd$	$\surd$
`any`, `all`	$(\alpha \xrightarrow{z_1} \mathsf{Bool}) \xrightarrow{3} \mathsf{List}^k\,\alpha \xrightarrow{k(z_1+5)+2} \mathsf{Bool}$	$\surd$	$\surd$
`elem`	$\alpha \xrightarrow{5} \mathsf{List}^k\,\alpha \xrightarrow{5k+2} \mathsf{Bool}$	$\surd$	$\surd$
`sum`, `product`	$\mathsf{List}^k\,\mathsf{Nat}^n \xrightarrow{2k} \mathsf{Nat}^\omega$	$\surd$	$\times$

Figure 11. Summary of prototype implementation results

applications for each list element, totaling $4k$ application. Finally, the functions *any*, *all* and *elem* have non-zero costs on the first arrow because they are partial applications of higher-order functions.

6 Related Work

The work described in this paper extends our earlier work [9,10] in the following way. The ω-relaxation operator allows sizes to be given for function applications that would have been rejected in the earlier system. In the type reconstruction algorithm, unification has been separated from weakening, and substitutions on cost/size variables have been separated from those on type variables. These changes allow weakening to be restricted in the $[\mathit{if}_{st}]$ rule, preventing incorrect weakening of cost/size variables outside the scope of conditionals. Strengthening handles the contravariance of the subtype relation for function types.

Most closely related to our granularity analysis is the system by Reistad and Gifford [5] for the cost analysis of Lisp expressions. This system introduces the notion of "latent costs", partially based on the "time system" by Dornic et al. [11], annotating the function type with a cost expression representing its computation cost, thereby enabling the system to treat higher-order functions. Rather than trying to extract closed forms for general recursive functions, however, they require the use of higher-order functions with known latent costs.

Hughes, Pareto and Sabry [4] have developed "sized types" and a type checking algorithm for a simple higher-order, non-strict functional language, which influenced our design. This type system checks upper bounds for the size of arbitrary algebraic data types, and is capable of dealing with recursive definitions.

Like our own system, Hughes et al. produce sets of linear constraints that are resolved by an external constraint solver. Unlike our system, however, sized types are restricted to type checking, which is sufficient to provide type security, as is their goal, but inadequate for a use as an analysis.

The technique used by Grobauer [12] for extracting cost recurrences out of a Dependent ML (DML) program is in several aspects similar to ours: size-annotated types are used to capture size information; the analysis is based on type inference and a cost extraction algorithm is outlined. The main differences to our work are that size inference is not attempted, DML is first-order rather than higher-order, and no attempt is made at finding closed forms for the extracted recurrences.

Chin and Khoo [13] describe a type-inference based algorithm for computing size information expressed in terms of Presburger formulae, for a higher-order, strict, functional language with lists, tuples and general non-recursive data constructors. Like the other work described here, however, this work does not cover cost analysis involving cost and size inference from unannotated source expressions.

In the context of *complexity analysis* Le Métayer [14] uses program transformation via a set of rewrite rules to derive complexity functions for FP programs. A database of known recurrences is used to produce closed forms for some recursive functions. Rosendahl [15] first uses abstract interpretation to obtain size information and then program transformation to generate a time bound program for first-order Lisp programs. Flajolet et al. [16] obtain average-case complexity functions for functional programs using the Maple computer algebra system to solve recurrences. Several systems elaborate on the *dynamic use of cost information* for parallel execution. Huelsbergen et al. [17] define an abstract interpretation of a higher-order, strict language using dynamic estimates of the size of data.

Finally, systems for granularity analysis in parallel logic programs [18,19] typically combine the compile-time derivation of a cost bound, with a run-time evaluation of the cost estimate to throttle the generation of parallelism. Such systems are restricted to first-order programs.

7 Conclusions

This paper has introduced a novel type-based analysis of the computation costs in a simple strict polymorphic higher-order functional language. Formally the analysis is presented as an extension to a conventional Hindley-Milner type system, a *sized time system.* Its implementation combines a cost reconstruction algorithm, based on subtyping, with a separate constraint solver for checking type correctness and producing upper-bound cost information that is aimed at guiding parallel computation. To the best of our knowledge our sized time system is the first attempt to construct a static analysis that exploits size information to *automatically infer* upper bound time costs of polymorphic higher-order functions. This is significant since it allows the immediate application of our analysis to a wide range of non-recursive functional programs. By using standard techniques such as cost libraries for known higher-order functions [5], we can, in principle, analyze costs for a range of programs that do not directly use recursion.

We are in the process of extending our work in a number of ways. Firstly, we have successfully prototyped a system to generate constraints that capture cost and size information for certain recursive definitions. This work has not yet been formalized, however.

Although we have not provided proofs of soundness and completeness, similar results on sized types suggest these can be constructed for our own work. A full comparison of the effectiveness of our approach is also pending.

References

1. Hammond, K., Michaelson, G.: Parallel Functional Programming. Springer (2000)
2. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer (1999)
3. Mitchell, J.C.: Subtyping and Related Concepts. In: Foundations for Programming Languages. MIT Press (1996)
4. Hughes, R., Pareto, L., Sabry, A.: Proving the Correctness of Reactive Systems using Sized Types. In: POPL'96, St Petersburg, FL (1996)
5. Reistad, B., Gifford, D.: Static Dependent Costs for Estimating Execution Time. In: LFP'94, Orlando, FL (1994) 65–78
6. Frühwirth, T.: Theory and practice of constraint handling rules. Journal of Logic Programming **37** (1998)
7. Smolka, G.: The Oz Programming Model. In: Computer Science Today. LNCS 1000. Springer (1995) 324–343
8. Jones, S.P., Hughes, J., Augustsson, L., Barton, D., Boutel, B., Burton, W., Fasel, J., Hammond, K., Hinze, R., Hudak, P., Johnsson, T., Jones, M., Launchbury, J., Meijer, E., Peterson, J., Reid, A., Runciman, C., Wadler, P.: Haskell 98: A Non-Strict, Purely Functional Language. (1999)
9. Loidl, H.W., Hammond, K.: A Sized Time System for a Parallel Functional Language. In: Glasgow Workshop on Functional Programming, Ullapool (1996)
10. Loidl, H.W.: Granularity in Large-Scale Parallel Functional Programming. PhD thesis, Department of Computing Science, University of Glasgow (1998)
11. Dornic, V., Jouvelot, P., Gifford, D.: Polymorphic Time Systems for Estimating Program Complexity. ACM Letters on Prog. Lang. and Systems **1** (1992) 33–45
12. Grobauer, B.: Cost Recurrences for DML Programs. In: ICFP'01, Florence, Italy, ACM Press (2001)
13. Chin, W.N., Khoo, S.C.: Calculating sized types. Higher-Order and Symbolic Computing **14** (2001)
14. Le Métayer, D.: ACE: An Automatic Complexity Evaluator. TOPLAS **10** (1988)
15. Rosendahl, M.: Automatic Complexity Analysis. In: FPCA'89. (1989) 144–156
16. Flajolet, P., Salvy, B., Zimmermann, P.: Automatic Average-Case Analysis of Algorithms. Theoretical Computer Science **79** (1991) 37–109
17. Huelsbergen, L., Larus, J., Aiken, A.: Using Run-Time List Sizes to Guide Parallel Thread Creation. In: LFP'94, Orlando, FL (1994) 79–90
18. Debray, S., Lin, N.W., Hermenegildo, M.: Task Granularity Analysis in Logic Programs. In: PLDI'90. SIGPLAN Notices 25(6), ACM Press (1990) 174–188
19. Tick, E., Zhong, X.: A Compile-Time Granularity Analysis Algorithm and its Performance Evaluation. New Generation Computing **11** (1993) 271–295