Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs

Pedro B. Vasconcelos* and Kevin Hammond
{pv,kh}@dcs.st-and.ac.uk

School of Computer Science, University of St Andrews, St Andrews, KY16 9SS, UK
Telephone: +44 (0)1334 463253 Fax: +44 (0)1334 463278

Abstract This paper presents a type-based analysis for inferring size- and cost-equations for recursive, higher-order and polymorphic functional programs without requiring user annotations or unusual syntax. Our type reconstruction algorithm is capable of inferring cost equations for a subset of recursive programs whose costs can be expressed using primitive recursion. We illustrate the approach with reference to some standard examples of recursive programs.

1 Introduction

Obtaining good-quality information concerning runtime costs (whether space or time) is important to many systems engineering activities, including compiler or database optimization, parallel computing, and real-time systems. Many of these activities require predictive information, acquired automatically at compile-time. Although there has been some success in predicting costs for applicative languages in restricted settings [17,14,13,2], the problem of automatically analyzing costs of languages with recursion, higher-order functions and parametric polymorphism remains an open one. These properties are key characteristics of recent statically typed functional language designs such as Standard ML or Haskell.
This paper presents a type-based analysis to automatically infer *upper bound evaluation costs* for a simple, but representative, functional language with parametric polymorphism, higher-order functions and recursion. Our aim is to produce a practical analysis that can deal with these essential languages features without resorting to artificially restrictive syntactic forms. We use a *type and effect system* [11] approach in which a standard Hindley-Milner type system [10] and the associated Damas-Milner inference algorithm [4] are extended by "effects" describing evaluation cost. Our analysis derives cost equations with finite solutions for a non-trivial subset of recursive definitions. It is *fully automatic* in producing cost equations for recursive definitions without requiring any user intervention, even in the form of type annotations. However, obtaining closed-form solutions to those costs currently requires the use of an external solver.

* On leave from DCC-FC & LIACC, University of Porto, Portugal.

2 Language Notation and Cost Semantics

$\mathcal{L}$ 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 terms e of $\mathcal{L}$ are defined by the following grammar, where x, n, b and p are the syntactic categories for variables, natural numbers, booleans and primitive operations, respectively.

$$\begin{aligned} e ::= {} & x \mid n \mid b \mid [\,] \mid e_1 :: e_2 \mid p(e) \mid \lambda x.e \mid \mathsf{fix}\ x.e \mid e_1\ e_2 \\ & \mid \ \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \end{aligned}$$

$\mathcal{L}$ has a conventional structure. The term $\lambda x.e$ is a function abstraction, while $\mathsf{fix}\ x.e$ is a recursive function satisfying the equation $x = e$ (i.e. a least fixed point under a suitable domain). Let-bindings introduce polymorphic local variable definitions as in the standard Hindley-Milner system [10]. Constructors and primitive operations (arithmetic on naturals and lists projections) are restricted to the correct number of arguments; partial applications can be obtained for these forms, if required, using λ-abstractions. To simplify our presentation (and without loss of generality), we consider here only unary primitives. The terms '$\lambda x.e$', '$\mathsf{fix}\ x.e$' and '$\mathsf{let}\ x = e'\ \mathsf{in}\ e$' bind the variable x in the sub-term e. We follow the usual definitions of *free variables* and for *closed terms*.

2.1 A Cost Model for $\mathcal{L}$

We consider a *call-by-value reduction semantics* for $\mathcal{L}$ (a formal presentation of this semantics can be found elsewhere [16]). Rather than assigning distinct costs to primitive operations, conditionals, etc. we will define the cost of an $\mathcal{L}$-expression solely in terms of the *number of β-reduction steps*, $(\lambda x.e)\ e' \rightarrow_\beta e[e'/x]$ and assign zero cost other reduction rules for primitives, constructors, etc. This cost metric has the advantages of being both easily understood and of capturing the asymptotic costs for recursive definitions. Our effect system could easily be extended to alternative metrics if required. Runtime errors have been modelled by the *absence* of a reduction, e.g. $\mathsf{head}([\,]) \not\rightarrow$, and so have zero cost. In our model, divergent reductions thus have infinite cost, whereas both confluent *and* erroneous reduction sequences have finite cost.

3 A Type and Effect System for Sizes and Costs

This section presents the type and effect system that will form the basis for our analysis. We have already proven that this type and effect system correctly expresses upper bound cost information for recursive, higher-order and polymorphic programs [16]. This paper builds on that earlier work by developing the corresponding type reconstruction (cost inference) algorithm.

$z + \epsilon = \epsilon + z = \epsilon$	$z \times \epsilon = \epsilon \times z = \epsilon$
$n + \omega = \omega + n = \omega$	$n \times \omega = \omega \times n = \omega$, if $n > 0$
$\epsilon - n = \epsilon$	$0 \times \omega = \omega \times 0 = 0$
$\omega - n = \omega$	$\epsilon \times \omega = \omega \times \epsilon = \epsilon$
$n_1 - n_2 = \epsilon$, if $n_2 > n_1$	

Figure 1. Extending arithmetic to $\overline{\mathbb{N}}$

3.1 Cost Expressions

We represent both *sizes* of data types and *costs* for reductions uniformly using terms from a *cost algebra*. The basic values for this algebra are elements of the set $\overline{\mathbb{N}} = \mathbb{N} \cup \{\epsilon, \omega\}$. Natural numbers represent finite sizes and costs, ϵ represents the *undefined* value and ω represents the *unbounded* value. The usual ordering $\leq$ on naturals extends to $\overline{\mathbb{N}}$ by setting $x \leq \omega$ and $\epsilon \leq x$ for all $x \in \overline{\mathbb{N}}$, i.e. ϵ and ω are, respectively, the bottom and top elements of $(\overline{\mathbb{N}}, \leq)$. Let $\ell \in \mathbf{ZVar}$ be the syntactical category of *effect variables* and $\{f_0, f_1, \ldots\}$ be a countable set of *function names* (used to construct recurrence equations for recursive definitions — Section 4.4). The set **ZExp** of *cost expressions* is generated by the grammar:

$$z ::= \ell \mid n \mid \epsilon \mid \omega \mid z_1 + z_2 \mid z - n \mid z_1 \times z_2 \mid \max(z_1, z_2) \mid f_i(\vec{z}) \,.$$

Note that we only allow subtraction of constant values; this suffices for our development and ensures that cost expressions are *monotone*, i.e. costs can only increase when any variable increases. This property is desirable for obtaining an inference algorithm for our type system (Section 4.5). We write cost expressions following the usual associativity and precedence rules for $+, -$ and $\times$.

3.2 Semantics for Cost Expressions

A *valuation* ρ is a total mapping from effect variables to cost values $\rho : \mathbf{ZVar} \to \overline{\mathbb{N}}$. Given a valuation ρ, the semantics of a cost expression is defined by the *evaluation function* $[\![\cdot]\!]\rho : \mathbf{ZExp} \to \overline{\mathbb{N}}$. Evaluation is defined by extending arithmetic from $\mathbb{N}$ to $\overline{\mathbb{N}}$ (cf. Figure 1). This evaluation semantics allows us to define extensional equality on cost expressions: $z = z'$ iff $[\![z]\!]\rho = [\![z']\!]\rho$ for all valuations ρ. Similarly, we lift the ordering $\leq$ from $\overline{\mathbb{N}}$ to a (partial) order on cost expressions.

3.3 Sized Types

Our effect system uses *sized types* [6], a small extension to standard Hindley-Milner polymorphic types: each type, other than function and boolean types, carries a superscript specifying an upper bound for its *size*. For function types, a *latent cost* [13] is attached to the function arrow. This latent cost is an upper bound on the *cost* of evaluating the function body.

Let α be the syntactical category for type variables **TVar**; the *sized types* τ are defined inductively by the following grammar:

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

Sized types allow us 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.
In order to represent polymorphic types we allow universal quantification over type or effect variables, yielding a *sized type scheme* as in [13]. We will write type schemes with a single outermost quantifier and a sequence of variables, i.e. $\forall\vec{\gamma}.\tau \equiv \forall\gamma_1.\ldots\forall\gamma_n.\tau$, where $\gamma_i \in \mathbf{TVar} \cup \mathbf{ZVar}$ are type or effect variables. Polymorphism allows size dependencies to be expressed for function types; for example, the type scheme for a function to double its argument (a natural number) might be $\forall n.\mathsf{Nat}^{n} \xrightarrow{0} \mathsf{Nat}^{2\times n}$, where we assume zero cost for the operation. The variables $\gamma_1, \ldots, \gamma_n$ are *bound* in the type scheme $\forall\gamma_1 \ldots \gamma_n.\tau$. A variable that is not bound is said to be *free*.
We use a number of standard notational conventions: given a sized type τ, we denote the *sequences* of type and effect variables occurring in τ by $\mathrm{TV}(\tau)$ and $\mathrm{ZV}(\tau)$, respectively. We use $+\!\!+$ for *sequence concatenation*; when the ordering among elements is not relevant, we treat sequences as *sets* and combine them using set operations $\cup, \cap, \setminus$. Finally, we use θ for substitutions from type variables to sized types and ϕ for substitutions from effect variables to cost expressions.

3.4 Type and Effect System

Figure 2 presents type system rules for core $\mathcal{L}$ expressions. The system derives judgements of the form $\Gamma \vdash e : \sigma \ \& \ z$ which can be informally read as "under type assumptions Γ, expression e admits type scheme σ and z is an upper bound for the cost of e". A *type environment* Γ is a sequence of assumptions $[x : \sigma]$ mapping $\mathcal{L}$ variables to type schemes. An environment can be seen as a partial finite mapping by defining $\Gamma(x) = \sigma$ if the rightmost occurrence of $[x : \cdots]$ in Γ is $[x : \sigma]$. The set of all *free type and effect variables* in Γ is represented by $\mathrm{FV}(\Gamma)$. With the exception of the $[\mathsf{weak}_{st}]$ and $[\mathsf{fix}_{st}]$ rules, this represents a straightforward extension of the standard Hindley-Milner rules. Note that:

- The $[\mathsf{weak}_{st}]$ rule allows *weakening*, i.e. relaxing the upper bounds on sizes or cost. It uses a subtyping relation $\trianglelefteq$ (Figure 3), which is *structural*, i.e. if $\tau \trianglelefteq \tau'$ then τ and τ' have the same type constructor.
- In the $[\mathsf{abs}_{st}]$ rule, the latent cost for the arrow type is the cost of evaluating the body of the abstraction, while the cost for the actual abstraction is zero; this is because our reduction semantics evaluates only to *weak normal forms*.
- The $[\mathsf{app}_{st}]$ rule adds the latent cost of the function to the costs of obtaining both function and argument, plus one to count for the β-reduction (this is the only rule where a positive cost is added).
- The $[\mathsf{if}_{st}]$ rule requires that both branches admit the same type and cost, which may necessitate weakening judgements for one or both branches.

$$\frac{}{\Gamma \vdash x : \Gamma(x) \;\&\; 0}\;[\mathsf{var}_{st}] \qquad \frac{}{\Gamma \vdash n : \mathsf{Nat}^{n} \;\&\; 0}\;[\mathsf{nat}_{st}] \qquad \frac{}{\Gamma \vdash b : \mathsf{Bool} \;\&\; 0}\;[\mathsf{bool}_{st}]$$

$$\frac{\Gamma[x : \tau_1] \vdash e : \tau_2 \;\&\; z}{\Gamma \vdash \lambda x.e : \tau_1 \xrightarrow{z} \tau_2 \;\&\; 0}\;[\mathsf{abs}_{st}] \qquad \frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{z_3} \tau_2 \;\&\; z_1 \quad \Gamma \vdash e_2 : \tau_1 \;\&\; z_2}{\Gamma \vdash e_1\ e_2 : \tau_2 \;\&\; 1{+}z_1{+}z_2{+}z_3}\;[\mathsf{app}_{st}]$$

$$\frac{\Gamma \vdash e_0 : \mathsf{Bool} \;\&\; z \quad \Gamma \vdash e_1 : \tau \;\&\; z' \quad \Gamma \vdash e_2 : \tau \;\&\; z'}{\Gamma \vdash \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau \;\&\; z{+}z'}\;[\mathsf{if}_{st}]$$

$$\frac{\Gamma[x : \forall \vec{\ell}.\tau] \vdash e : \tau \;\&\; 0 \quad \{\vec{\ell}\} \cap \mathrm{FV}(\Gamma) = \emptyset}{\Gamma \vdash \mathsf{fix}\ x.e : \tau \;\&\; 0}\;[\mathsf{fix}_{st}]$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \;\&\; z_1 \quad \Gamma[x : \sigma_1] \vdash e_2 : \tau_2 \;\&\; z_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2 \;\&\; z_1{+}z_2}\;[\mathsf{let}_{st}] \qquad \frac{\Gamma \vdash e : \tau \;\&\; z \quad \tau \trianglelefteq \tau' \quad z \leq z'}{\Gamma \vdash e : \tau' \;\&\; z'}\;[\mathsf{weak}_{st}]$$

$$\frac{\Gamma \vdash e : \tau \;\&\; z \quad \{\vec{\gamma}\} \cap \mathrm{FV}(\Gamma) = \emptyset}{\Gamma \vdash e : \forall \vec{\gamma}.\tau \;\&\; z}\;[\mathsf{gen}_{st}] \qquad \frac{\Gamma \vdash e : \forall \vec{\gamma}.\tau \;\&\; z \quad \mathrm{dom}(\theta) \cup \mathrm{dom}(\phi) \subseteq \{\vec{\gamma}\}}{\Gamma \vdash e : \phi(\theta\tau) \;\&\; z}\;[\mathsf{ins}_{st}]$$

Figure 2. Typing Rules for the Core $\mathcal{L}$ Expressions

$$\frac{\tau = \tau'}{\tau \trianglelefteq \tau'}\;[\mathsf{reflex}_{\trianglelefteq}] \qquad \frac{\tau_1 \trianglelefteq \tau_2 \quad \tau_2 \trianglelefteq \tau_3}{\tau_1 \trianglelefteq \tau_3}\;[\mathsf{trans}_{\trianglelefteq}] \qquad \frac{\tau_1 \trianglelefteq \tau_1' \quad \tau_2' \trianglelefteq \tau_2 \quad z' \leq z}{\tau_1' \xrightarrow{z'} \tau_2' \trianglelefteq \tau_1 \xrightarrow{z} \tau_2}\;[\mathsf{abs}_{\trianglelefteq}]$$

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

Figure 3. Subtyping Relation

- The $[\mathsf{let}_{st}]$ rule implements polymorphism by allowing a quantified type for the locally defined variable; let is *not* costed as a β-reduction.
- The $[\mathsf{ins}_{st}]$ and $[\mathsf{gen}_{st}]$ rules are straightforward extensions of the Hindley-Milner forms to allow for polymorphism on both *type* variables and *effect* variables. Note that unlike [6], our system does not require a side-condition for ω-instantiation in the $[\mathsf{ins}_{st}]$ rule[1].
- The $[\mathsf{fix}_{st}]$ rule allows the body of the recursive function to be typed using *polymorphic recursion restricted to size and cost variables*. The idea is to allow capturing the recursive uses of the function through instantiation.

[1] Although we have not yet constructed a formal semantics for our sized types, we conjecture that this is because, unlike Hughes, Pareto and Sabry [6], our intended semantics for sized types includes *divergent values*.

Unlike elementary strong functional programming [15] and the sized type system of [6], our system does not reject divergent computations. For example, the term

$$loop \equiv \mathsf{fix}\ f.\lambda x.f\ x$$

admits the type judgement $\vdash loop : \forall\alpha\beta.\alpha \xrightarrow{\omega} \beta\ \&\ 0$. As a consequence, *all types are inhabited* (for example, by the term '*loop* true'). Note that the non-termination is still captured by the latent cost ω in our sized type for *loop*. The reciprocal, however, is not true — i.e. there exist terminating terms that admit only an ω cost:

$$M \equiv \mathsf{if\ false\ then}\ loop\ \mathsf{true\ else\ false}\ .$$

Clearly M is terminating but all type judgements $\vdash M : \mathsf{Bool}\ \&\ z$ must derive an unbounded cost $z = \omega$ because of the application of *loop* in one of the branches of the conditional.
In general, our system can only assign finite costs to recursions when the size of some component of an argument decreases strictly in each iteration (i.e. when a single argument-derived size induces a well-founded ordering). It follows that we can infer cost equations with finite solutions for many *primitive recursive* definitions (subject to the limitations of expressibility in the cost algebra), plus some more general forms as shown in Section 5.

4 Inference Algorithm

This section describes a type reconstruction algorithm for our system that is an extension of Damas-Milner algorithm W [4]. The algorithm takes an unannotated $\mathcal{L}$ expression and yields a sized type and a cost effect, together with a set of constraints and recurrence equations.

4.1 Flat Sized Types and Constraints

As is done in other analysis based on type and effect inference (e.g. [1]), we restrict annotations in the types to variables (yielding *flat sized types*) and separately collect *effect constraints*. This allows us to employ standard unification to solve type equations and deal with the more complex cost algebra only in the constraints. Our constraints express *lower bounds* for the effect variables (as in [13]) and *recurrence equations* collected from recursive definitions (discussed in Section 4.4):

$$\begin{aligned} c &::= \ell \geq z \ \mid\ f_i(\vec{\ell}) = z \\ C &::= \emptyset \mid \{c\} \cup C \end{aligned} \tag{1}$$

4.2 Flat Sized Type Schemes

In order to represent polymorphic types, our *flat sized type schemes* $\forall\vec{\gamma}.(\tau,\ C)$ quantify over both a flat type and a constraint set. The constraint set C is chosen

to capture the subtyping relation allowed by the *weakening rule*. For example, the type scheme $\forall m.\mathsf{Nat}^m \xrightarrow{1} \mathsf{Nat}^{m+1}$ can be translated to the flat form,

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

Because of the restriction to the form of constraints, we cannot represent sized types expressing *functions with partial domains*. For example, $\mathsf{Nat}^{10} \xrightarrow{z} \mathsf{Bool}$ should be translated to $(\mathsf{Nat}^n \xrightarrow{k} \mathsf{Bool},\ \{n \leq 10,\ k \geq z\})$ but the constraint $n \leq 10$ is not in the form of equation (1).
However, allowing type assumptions with partial domain such as $f : \mathsf{Nat}^{10} \xrightarrow{z} \mathsf{Bool}$ would cause us to *reject* an application like $f\ 11$ that is typeable in the underlying Hindley-Milner system. By restricting the constraints to the form of equation (1) and cost expressions to be monotone (cf. Section 3.2), we guarantee not to reject terms that admit a Hindley-Milner type.

4.3 Type Reconstruction Rules

The type reconstruction algorithm is presented in Figure 4, in the same inference-rule style used for the type system (cf. Figure 2). The reconstruction rules, however, are *structural*, i.e. exactly one rule applies for each $\mathcal{L}$ syntax form. In particular, we no longer have separate rules for *generalization* and *instantiation* of polymorphic types and *weakening* (i.e. relaxation of sizes or costs). Instead, generalization and instantiation are applied at let-bindings and at the use of identifiers, respectively (as in Damas-Milner algorithm W). Weakening is applied in two distinct situations:

1. in conditionals, to obtain a super-type of the types of both branches [6] and an upper bound on the costs of the branches; and
2. in function applications, to construct a correct sub-typing relation between the type of a concrete argument to a function and the function's domain.

Type reconstruction yields judgements of the form $\Gamma \vdash e : \langle \tau, \theta, z, C \rangle$, where the inputs are a list of well-formed assumptions Γ and an $\mathcal{L}$ expression e, and the output is the tuple $\langle \tau, \theta, z, C \rangle$ consisting of a flat sized type τ, a *unifying substitution* θ, a *cost expression* z and a *constraint set* C.
Our algorithm separates the inference of the *type structure* from the inference of the *effects*. Note that:

- The *unification algorithm* of Figure 5 is used to solve equations on flat sized types: it yields a substitution making two types equal up to annotations and uses an auxiliary *freshening* function ν to avoid unwanted capturing of variables.
- The *domain matching* function of Figure 5 yields a set of constraints imposing a sub-typing relation between two Hindley-Milner identical types (this is possible because our sub-typing relation is structural).
- The $[\mathsf{nat}_{ra}]$ rule captures the size of the natural as a constraint, illustrating that the algorithm manipulates only flat sized types.

$$\frac{}{\Gamma \vdash b : \langle \mathsf{Bool}, [\,], 0, \emptyset \rangle}\ [\mathsf{bool}_{ra}] \qquad \frac{}{\Gamma \vdash n : \langle \mathsf{Nat}^{\ell}, [\,], 0, \{\ell \geq n\} \rangle}\ [\mathsf{nat}_{ra}] \qquad \text{fresh } \ell$$

$$\frac{\theta = [\vec{\alpha'}/\vec{\alpha}] \quad \phi = [\vec{\ell'}/\vec{\ell}]}{\Gamma[x : \forall \vec{\alpha}\vec{\ell}.(\tau, C)] \vdash x : \langle \theta\phi\tau, \theta, 0, \phi C \rangle}\ [\mathsf{var}_{ra}] \quad \text{fresh } \vec{\alpha'},\ \vec{\ell'}$$

$$\frac{\Gamma[x : (\alpha, \emptyset)] \vdash e : \langle \tau, \theta, z, C \rangle}{\Gamma \vdash \lambda x.e : \langle \theta\alpha \xrightarrow{\ell} \tau, \theta, 0, \{\ell \geq z\} \cup C \rangle}\ [\mathsf{abs}_{ra}] \quad \text{fresh } \alpha, \ell$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \langle \tau_1, \theta_1, z_1, C_1 \rangle \qquad \theta_1\Gamma \vdash e_2 : \langle \tau_2, \theta_2, z_2, C_2 \rangle \\ \theta_3 = \mathcal{U}(\theta_2\tau_1, \tau_2 \xrightarrow{\ell} \alpha) \qquad C_3 = \mathcal{D}(\theta_3\theta_2\tau_1, \theta_3(\tau_2 \xrightarrow{\ell} \alpha))\end{array}}{\Gamma \vdash e_1\ e_2 : \langle \theta_3\alpha, \theta_3\theta_2\theta_1, 1{+}\ell{+}z_1{+}z_2, C_1 \cup C_2 \cup C_3 \rangle}\ [\mathsf{app}_{ra}] \qquad \text{fresh } \alpha, \ell$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \langle \tau_1, \theta_1, z_1, C_1 \rangle \quad \theta_1\Gamma \vdash e_2 : \langle \tau_2, \theta_2, z_2, C_2 \rangle \quad \theta_2\theta_1\Gamma \vdash e_3 : \langle \tau_3, \theta_3, z_3, C_3 \rangle \\ \theta_4 = \mathcal{U}(\theta_3\theta_2\tau_1, \mathsf{Bool}) \qquad \theta_5 = \mathcal{U}(\theta_4\theta_3\tau_2, \theta_4\tau_3) \\ \tau = \nu(\theta_5\theta_4\theta_2\tau_1) \qquad C = C_1 \cup C_2 \cup C_3 \cup \mathcal{D}(\theta_5\theta_4\theta_3\tau_2, \tau) \cup \mathcal{D}(\theta_5\theta_4\tau_3, \tau)\end{array}}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \langle \tau, \theta_5\theta_4\theta_3\theta_2\theta_1, z_1{+}\max(z_2, z_3), C \rangle}\ [\mathsf{if}_{ra}]$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \langle \tau_1, \theta_1, z_1, C_1 \rangle \qquad \{\vec{\gamma}\} = (\mathrm{TV}(\tau_1) \cup \mathrm{ZV}(\tau_1) \cup \mathrm{ZV}(C_1)) \setminus \mathrm{FV}(\theta_1\Gamma) \\ \theta_1\Gamma[x : \forall \vec{\gamma}.(\tau_1, C_1)] \vdash e_2 : \langle \tau_2, \theta_2, z_2, C_2 \rangle\end{array}}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \langle \tau_2, \theta_2\theta_1, z_1{+}z_2, C_1 \cup C_2 \rangle}\ [\mathsf{let}_{ra}]$$

$$\frac{\begin{array}{ll}\Gamma[x : (\alpha, \emptyset)] \vdash e : \langle \tau_1, \theta_1, \ldots, \ldots \rangle & \\ \theta'_1 = \mathcal{U}(\theta_1\alpha, \tau_1) \qquad \tau = \theta'_1\theta_1\alpha \qquad \tau' = \nu(\tau) & \\ X = \mathrm{FZV}(\theta'_1\theta_1\Gamma) \quad \{\vec{\ell}\} = \mathrm{ZV}(\tau') \setminus X & \text{fresh } \alpha \\ C_1 = \mathcal{L}(\tau, X, 1) \qquad C' = \mathcal{L}(\tau', X, 1) & \text{fresh } f_1, f_2, \ldots, f_n, \\ \theta'_1\theta_1\Gamma[x : \forall \vec{\ell}.(\tau', C')] \vdash e : \langle \tau_2, \theta_2, \ldots, C_2 \rangle & \text{where } n = |C_1| = |C'| = |\mathcal{E}| \\ \mathcal{E} = \mathcal{R}(\tau_2, X, 1) & \end{array}}{\Gamma \vdash \mathsf{fix}\ x.e : \langle \theta_2\tau, \theta_2\theta'_1\theta_1, 0, C_1 \cup C_2 \cup \mathcal{E} \rangle}\ [\mathsf{fix}_{ra}]$$

Figure 4. Type Reconstruction Rules for the Core $\mathcal{L}$ Expressions

- The $[\mathsf{app}_{ra}]$ rule uses domain matching to impose a sub-typing relation between the *argument type* and the *function's domain type*.
- The $[\mathsf{if}_{ra}]$ rule uses unification to obtain the type structure, and domain matching to constrain the result to be a super-type of the types of the branches.
- The $[\mathsf{let}_{ra}]$ rule generalizes not only over free type- and effect-variables in the *type*, but also over variables in the *constraint set* to ensure proper capture of dependencies in constraint chains.

$$
\begin{array}{ll}
\mathcal{U} \;:\; \tau \times \tau \to \theta & \\
\mathcal{U}(\alpha, \alpha') & = [\alpha'/\alpha] \\
\mathcal{U}(\alpha, \tau) = \mathcal{U}(\tau, \alpha) & = [\nu(\tau)/\alpha], \\
\quad \text{if } \alpha \text{ does not occur in } \tau & \\
\mathcal{U}(\mathsf{Bool}, \mathsf{Bool}) & = [] \\
\mathcal{U}(\mathsf{Nat}^{\ell_1}, \mathsf{Nat}^{\ell_2}) & = [] \\
\mathcal{U}(\mathsf{List}^{\ell_1}\tau_1, \mathsf{List}^{\ell_2}\tau_2) & = \mathcal{U}(\tau_1, \tau_2) \\
\mathcal{U}(\tau_1 \xrightarrow{\ell} \tau_2, \tau_1' \xrightarrow{\ell'} \tau_2') & = \mathcal{U}(\theta_1\tau_2, \theta_1\tau_2')\theta_1 \\
\quad \text{where } \theta_1 = \mathcal{U}(\tau_1, \tau_1') & \\
\text{otherwise, unification fails} &
\end{array}
$$

$$
\begin{array}{lll}
\nu \;:\; \tau \to \tau & & \\
\nu(\alpha) & = \alpha & \\
\nu(\mathsf{Bool}) & = \mathsf{Bool} & \\
\nu(\mathsf{Nat}^{\ell}) & = \mathsf{Nat}^{\ell'}, & \text{fresh } \ell' \\
\nu(\mathsf{List}^{\ell}\tau) & = \mathsf{List}^{\ell'}\nu(\tau), & \text{fresh } \ell' \\
\nu(\tau_1 \xrightarrow{\ell} \tau_2) & = \nu(\tau_1) \xrightarrow{\ell'} \nu(\tau_2), & \text{fresh } \ell'
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{D} \;:\; \tau \times \tau \to C & \\
\mathcal{D}(\alpha, \alpha) & = \emptyset \\
\mathcal{D}(\mathsf{Bool}, \mathsf{Bool}) & = \emptyset \\
\mathcal{D}(\mathsf{Nat}^{\ell_1}, \mathsf{Nat}^{\ell_2}) & = \{\ell_2 \geq \ell_1\} \\
\mathcal{D}(\mathsf{List}^{\ell_1}\tau_1, \mathsf{List}^{\ell_2}\tau_2) & = \{\ell_2 \geq \ell_1\} \cup \mathcal{D}(\tau_1, \tau_2) \\
\mathcal{D}(\tau_1 \xrightarrow{\ell} \tau_2, \tau_1' \xrightarrow{\ell'} \tau_2') & = \{\ell' \geq \ell\} \cup \mathcal{D}(\tau_1', \tau_1) \cup \mathcal{D}(\tau_2, \tau_2')
\end{array}
$$

Figure 5. Unification, Annotation Freshening & Domain Matching Functions

$$
\begin{array}{ll}
\mathcal{L} \;:\; \tau \times \vec{\ell} \times n \to C & \\
\mathcal{L}(\alpha, \vec{x}, i) & = \emptyset \\
\mathcal{L}(\mathsf{Bool}, \vec{x}, i) & = \emptyset \\
\mathcal{L}(\mathsf{Nat}^{\ell}, \vec{x}, i) & = \{\ell \geq f_i(\vec{x})\} \\
\mathcal{L}(\mathsf{List}^{\ell}\tau, \vec{x}, i) & = \\
\quad \{\ell \geq f_i(\vec{x})\} \cup \mathcal{L}(\tau, \vec{x}, i+1) & \\
\mathcal{L}(\tau \xrightarrow{\ell} \tau', \vec{x}, i) & = \\
\quad \{\ell \geq f_i(\vec{x'})\} \cup \mathcal{L}(\tau', \vec{x'}, i+1) & \\
\quad \text{where } \vec{x'} = \vec{x} +\!\!+ \mathrm{ZV}(\tau) &
\end{array}
\qquad
\begin{array}{ll}
\mathcal{R} \;:\; \tau \times \vec{\ell} \times n \to \mathcal{E} & \\
\mathcal{R}(\alpha, \vec{x}, i) & = \emptyset \\
\mathcal{R}(\mathsf{Bool}, \vec{x}, i) & = \emptyset \\
\mathcal{R}(\mathsf{Nat}^{\ell}, \vec{x}, i) & = \{f_i(\vec{x}) = \ell\} \\
\mathcal{R}(\mathsf{List}^{\ell}\tau, \vec{x}, i) & = \\
\quad \{f_i(\vec{x}) = \ell\} \cup \mathcal{R}(\tau, \vec{x}, i+1) & \\
\mathcal{R}(\tau \xrightarrow{\ell} \tau', \vec{x}, i) & = \\
\quad \{f_i(\vec{x'}) = \ell\} \cup \mathcal{R}(\tau', \vec{x'}, i+1) & \\
\quad \text{where } \vec{x'} = \vec{x} +\!\!+ \mathrm{ZV}(\tau) &
\end{array}
$$

Figure 6. Recurrence Labelling and Collection

4.4 Inference for Recursive Definitions

We describe in detail the inference rule for recursive definitions '$\mathsf{fix}\ x.e$':

- We first infer a type for the function body e under a generic assumption α for the recursive function x. By unifying the result type with the assumed variable α, we obtain *the Hindley-Milner type structure* τ for the function. The cost and constraints obtained from this step are discarded.
- Next we use an auxiliary *recurrence labelling* function $\mathcal{L}$ (Figure 6) to traverse the type and yield a *skeleton of the cost and size effects dependencies*, i.e. a set of constraints relating the type annotations to *fresh cost function symbols* $f_1, \ldots, f_n$. As we want to infer *functional dependencies* for sizes and costs, whenever we encounter a type $\tau \xrightarrow{z} \tau'$ we make the annotations in τ *parameters* of the cost functions synthesized for τ' and the latent cost z.

– Finally, we infer a type for the body e again, this time with a polymorphic assumption quantifying over all free annotations in the type. This yields a flat type τ_2 and set of constraints C_2 capturing a *single-step unfolding* of the recursive function. The auxiliary function $\mathcal{R}$ of Figure 6 collects the relations on costs and sizes for this unfolding as a set of *recurrence equations* involving the cost functions $f_1, \ldots, f_n$.

In order to make the presentation self-contained, we use the type inference algorithm *twice* for the body of the recursive function: firstly to obtain the type structure and secondly to obtain the recurrence relations. We could, however, avoid this extra work by requiring Hindley-Milner type signatures for fixpoint terms (this information might be available, for example, from compile-time type-inference prior to the analysis). Alternatively, we could employ a standard Hindley-Milner type inference (rather than our sized-type inference) and avoid unnecessary constraint bookkeeping.

4.5 Solving the Constraint Sets

We now address the issue of solving the *effect constraints* collected during type inference; the *recurrence equations* are left in open-form in the output of type inference (see Section 7 for a discussion on recurrence solving techniques). Our algorithm is presented in Figure 7 and is based on that of Reistad and Gifford [13] and on the *worklist iteration* algorithms for solving dataflow analysis constraints (e.g. [11]).

We say that an assignment ρ *validates* a constraint set C (and write $\rho \models C$) iff $\rho(\ell) \geq [\![z]\!]\rho$ for all $(\ell \geq z) \in C$. Clearly $\rho(\ell) = \omega$, $\forall \ell$ is always a solution, but we are interested in obtaining the *minimal* solution. Because our cost algebra is monotone, this solution can be computed as a *least fixpoint* of the associated equations [13]. This fixpoint could be reached by assigning ϵ to all variables and iterating through the constraints, updating variable values.[2] However, this procedure will not terminate if the least solution of a variable is ω.

To circumvent this problem, we first decompose the constraint set into *strongly connected components* according to constraint dependencies and solve each component separately. For an SCC with n constraints, a finite solution to the variables (if it exists) must be reached within n iterations (because the largest cyclic dependency will involve at most n constraints). If after n iterations we fail to obtain a solution, then the least solution must be ω.

The algorithm is complete with complexity which is quadratic on the size of the largest SCC. We believe this size will remain small and bounded with larger program sizes. The complexity could be further reduced (at the expense of losing completeness) by limiting the outer j-loop to a fixed limit. We have implementated a modified version of this algorithm that computes *symbolic solutions* by starting with an initial assignment where relevant variables are bound to *symbolic parameters*, and subsequently using symbolic evaluation for costs.

[2] The *monotonicity* of cost expressions allows variable assignment to be extended incrementally, since if $\rho \models C$ then $\rho' \models C$ for any $\rho' \geq \rho$.

Initial variable assignment: $\rho(\ell) := \epsilon$, $\forall \ell$.
Iterate over strongly connected components in topological ordering.
For each SCC $C = \{(\ell_i \geq z_i)_{i=1}^n\}$:
  For $j = 1, 2 \ldots n$ or until $\rho \models C$:
    For $i = 1, 2 \ldots n$:
      set $\rho(\ell_i) := \max(\rho(\ell_i), [\![z_i]\!]\rho)$
  If $\rho \not\models C$, then for $i = 1, 2 \ldots n$: set $\rho(\ell_i) := \omega$

Figure 7. Constraint Solving Algorithm

5 Examples from our Prototype Implementation

We have implemented our type reconstruction algorithm and successfully used it to derive good cost information for a variety of sample programs, including simple numeric recursive functions (e.g. *factorial*, *naïve Fibonacci* and *power*) and a representative subset of the Haskell standard Prelude list functions (e.g. *length*, *append*, *map*, *iterate*, *filter*, *foldl/r*, *reverse*, *drop*, *take*, *zipWith* and an insertion sort algorithm). The prototype implementation has proved to be acceptably efficient in all the examples we have tested. A web implementation of the algorithm, together with several of these examples, is available at `http://www.dcs.st-and.ac.uk/~pv/cost.html`. We present three examples chosen to illustrate the inference process for recursion in the presence of higher-order functions and polymorphism, and to be representative of the scope of our analysis.

5.1 A Worked Example: Map

Our first example is a worked type reconstruction for *map*, the standard higher-order function that applies an argument function to each element in a list:

$$map \equiv \lambda f.\mathsf{fix}\ map'.\lambda xs.\mathsf{if}\ \mathsf{null}(xs)\ \mathsf{then}\ [\,]\ \mathsf{else}\ f\ \mathsf{head}(xs) :: map'\ \mathsf{tail}(xs)$$

This example illustrates how the sized type inference captures the dependency on the argument function cost and how recurrence equations are obtained. We present only the major inference steps for *map*, omitting intermediate results.

1. *Infer function body type under generic assumption*
   $\Gamma = [f : (\alpha_1, \emptyset)]$
   $\Gamma[map' : (\alpha_2, \emptyset)] \vdash \lambda xs.\mathsf{if}\ \mathsf{null}(xs)\ \mathsf{then}\ [\,]\ \mathsf{else}\ \ldots : \langle \tau_1, \theta_1, \ldots, \ldots \rangle$,
   $\tau_1 = \mathsf{List}^{\ell_5}\alpha_3 \xrightarrow{\ell_6} \mathsf{List}^{\ell_7}\alpha_4$
   $\theta_1 = [\alpha_3 \xrightarrow{\ell_1} \alpha_4/\alpha_1,\ \mathsf{List}^{\ell_2}\alpha_3 \xrightarrow{\ell_3} \mathsf{List}^{\ell_4}\alpha_4/\alpha_2]$
2. *Unify to get the type structure*
   $\theta_1' = \mathcal{U}(\tau_1, \theta_1\alpha_2) = []$
   $\tau = \mathsf{List}^{\ell_2}\alpha_3 \xrightarrow{\ell_3} \mathsf{List}^{\ell_4}\alpha_4$
   $\tau' = \nu(\tau) = \mathsf{List}^{\ell_8}\alpha_3 \xrightarrow{\ell_9} \mathsf{List}^{\ell_{10}}\alpha_4$

3. *Collect free effect variables in environment*
   $X = \mathrm{FZV}(\theta_1'\theta_1\Gamma) = \{\ell_1\}$
4. *Recurrence labelling*
   $C_1 = \mathcal{L}(\tau, \{\ell_1\}, 1) = \{\ell_3 \geq f_1(\ell_1, \ell_2),\ \ell_4 \geq f_2(\ell_1, \ell_2)\}$
   $C' = \{\ell_9 \geq f_1(\ell_1, \ell_8),\ \ell_{10} \geq f_2(\ell_1, \ell_8)\}$
5. *Second inference under polymorphic assumption*
   $\Gamma' = [f : (\alpha_3 \xrightarrow{\ell_1} \alpha_4, \emptyset)]$
   $\Gamma'[map' : \forall \ell_8 \ell_9 \ell_{10}.(\tau', C')] \vdash \lambda xs.\mathsf{if}\ \mathsf{null}(xs)\ \mathsf{then}\ [\,]\ \mathsf{else}\ \ldots : \langle \tau_2, \ldots, \ldots, C_2 \rangle$
   $\tau_2 = \mathsf{List}^{\ell_{11}} \alpha_3 \xrightarrow{\ell_{12}} \mathsf{List}^{\ell_{13}} \alpha_4$
   $C_2 \simeq \{\ell_{12} \geq \max(2 + \ell_1 + f_1(\ell_1, \ell_{11}-1), 0),\ \ell_{13} \geq \max(1 + f_2(\ell_1, \ell_{11}-1), 0)\}$
6. *Recurrence collection*
   $\mathcal{E} = \mathcal{R}(\tau_2, \{\ell_1\}, 1) = \{f_1(\ell_1, \ell_{11}) = \max(2 + \ell_1 + f_1(\ell_1, \ell_{11}-1), 0),$
   $f_2(\ell_1, \ell_{11}) = \max(1 + f_2(\ell_1, \ell_{11}-1), 0)\}$

To facilitate the comprehension of the inference process, we presented the constraint set C_2 after symbolic simplification, and substituted the solutions in the right-hand sides of the recurrence equations in $\mathcal{E}$. Both these steps are done automatically by our implementation of the algorithm.
The result of type inference for *map* is then:

$$map : (\alpha_3 \xrightarrow{\ell_1} \alpha_4) \xrightarrow{\ell_{12}} \mathsf{List}^{\ell_2} \alpha_3 \xrightarrow{\ell_3} \mathsf{List}^{\ell_4} \alpha_4,\ \{\ell_{12} \geq 0,\ \ell_3 \geq f_1(\ell_1, \ell_2),\ \ell_4 \geq f_2(\ell_1, \ell_2)\}$$

where the recurrence functions f_1 and f_2 express the cost for the map and the size of the result list, respectively.
The upper-bound for costs of the *base* and *recursive* cases are represented by a single equation in the recurrences: for the empty list, we have $\ell_{11} = 0$ and the base cost is $f_1(\ell_1, 0) = \max(2 + \ell_1 + f_1(\ell_1, \epsilon), 0) = \max(\epsilon, 0) = 0$. Note that ϵ represents the *undefined cost* corresponding to an erroneous computation path (in this example, taking the tail of an empty list).
We can obtain closed-form solutions to the recurrences either by inspection or using computer algebra software: $f_1(\ell_1, \ell_{11}) = (2+\ell_1) \times \ell_{11}$ and $f_2(\ell_1, \ell_{11}) = \ell_{11}$, i.e. map maintains the list size and its cost is proportional to the list size and function latent cost. Note that these are the best estimates expressible in our cost algebra.

5.2 List Reverse

The next example illustrates analysis for a two-parameter recursion (list reversal) using an accumulating parameter:

$$rev \equiv \mathsf{fix}\ rev'.\lambda x.\lambda y.\mathsf{if}\ \mathsf{null}(x)\ \mathsf{then}\ y\ \mathsf{else}\ rev'\ \mathsf{tail}(x)\ (\mathsf{head}(x)::y)$$

We obtain the following sized type and constraints solution:

$$\begin{aligned}
&\tau_{rev} = \mathsf{List}^n \alpha \xrightarrow{\ell_1} \mathsf{List}^m \alpha \xrightarrow{\ell_2} \mathsf{List}^k \alpha \\
&\ell_1 = f_1(n),\ \ell_2 = f_2(n, m),\ k = f_3(n, m) \\
&f_1(n) = 0 \\
&f_2(n, m) = \max(2 + f_1(n-1) + f_2(n-1, 1+m), 0) \\
&f_3(n, m) = \max(f_3(n-1, 1+m), m)
\end{aligned}$$

Simplifying the recurrence equations yields the *exact* cost and size,

$$f_2(n,m) = 2 \times n, \quad f_3(n,m) = n + m$$

i.e. the result size is the sum of the two lists sizes and the cost is proportional to the size of the of the first argument. Note that type inference automatically handles the two-parameter recursion. There is no need for the programmer to indicate which parameter is reducing in size or to rewrite the program into an explicitly primitive recursive form.

5.3 List Union

Our final example is a function that constructs the set union of two lists. We first define a higher-order function *any* that tests a predicate for some element of a list. Using *any*, we define *union* for a generic equality function *eq* given as a higher-order parameter. This example generalizes the first-order case presented by both Wegbreit [17] and Rosendahl [14].

$\mathsf{let}\ any = \lambda p.\ \mathsf{fix}\ any'.\lambda xs.\ \mathsf{if}\ \mathsf{null}(xs)\ \mathsf{then}\ \mathsf{false}$
$\quad \mathsf{else\ if}\ p\ \mathsf{head}(xs)\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ any'\ \mathsf{tail}(xs)$
$\mathsf{let}\ union = \lambda eq.\lambda xs.\ \mathsf{fix}\ union'.\lambda ys.\ \mathsf{if}\ \mathsf{null}(ys)\ \mathsf{then}\ xs$
$\quad \mathsf{else\ if}\ any\ (eq\ \mathsf{head}(ys))\ xs\ \mathsf{then}\ union'\ \mathsf{tail}(ys)$
$\quad \mathsf{else}\ \mathsf{head}(ys) :: union'\ \mathsf{tail}(ys)$

The types inferred from the definitions above, after substitution of the constraint solutions, are:

$$\begin{aligned}
\tau_{any} &= (\alpha \xrightarrow{\ell_1} \mathsf{Bool}) \xrightarrow{0} \mathsf{List}^k \alpha \xrightarrow{\ell_2} \mathsf{Bool} \\
\tau_{union} &= (\alpha \xrightarrow{\ell_3} \alpha \xrightarrow{\ell_4} \mathsf{Bool}) \xrightarrow{0} \mathsf{List}^n \alpha \xrightarrow{0} \mathsf{List}^m \alpha \xrightarrow{\ell_5} \mathsf{List}^p \alpha \\
\text{where} \quad & \ell_2 = f_1(\ell_1, k),\ \ell_5 = f_2(\ell_3, \ell_4, n, m),\ p = f_3(\ell_3, \ell_4, n, m) \\
& f_1(\ell_1, k) = \max(1 + \ell_1 + \max(1 + f_1(\ell_1, k-1), 0), 0) \\
& f_2(\ell_3, \ell_4, n, m) = \max(4 + \ell_3 + f_1(\ell_4, n) + f_2(\ell_3, \ell_4, n, m-1), 0) \\
& f_3(\ell_3, \ell_4, n, m) = \max(1 + f_3(\ell_3, \ell_4, n, m-1), n)
\end{aligned}$$

and the recurrences admit the following solutions:

$$\begin{aligned}
f_1(\ell_1, k) &= (2 + \ell_1) \times k \\
f_2(\ell_3, \ell_4, n, m) &= (4 + \ell_3 + (2 + \ell_4) \times n) \times m \\
f_3(\ell_3, \ell_4, n, m) &= n + m
\end{aligned}$$

Observe that the costs and sizes are widened to the worst-case when there are no common elements in the two lists: *any* traverses the complete list and the size the of *union* is the sum of the sizes of the two lists.
Because of the partial application of equality, the cost inferred for *union* depends on the two latent costs of the equality function: ℓ_3 is added m times (one for each invocation of $union'$), whereas ℓ_4 is added $n \times m$ times (one for each invocation

of any'). The particular case where equality is a primitive corresponds to setting $\ell_3 = \ell_4 = 0$ and the cost for *union* is then $(4+2\times n)\times m$, which is asymptotically identical to the non-generic solution presented in [14]. *It follows that our analysis can still obtain good bounds for first-order instances even when deriving costs from a higher-order definition.*

6 Related Work

To the best of our knowledge, there is no comparable analysis capable of automatically inferring costs for recursive, higher-order and polymorphic functional programs. Previous approaches have, however, considered aspects of this problem. The approach described here extends our own earlier work on inference for sized time systems [9,12] by covering recursive as well as non-recursive language forms. Our sized type system is directly influenced by that of Hughes, Pareto and Sabry [6], who have developed a type checking algorithm for sized types in a higher-order, recursive, and non-strict functional language. While the system of Hughes at al., can be used to prove *termination* for recursion and *productivity* for streams, it does not consider execution costs and does not infer sizes. Chin and Khoo [3] have extended this work to yield an inference algorithm for such sized types. Their system does not, however, infer *costs* and deals only with *monomorphic* definitions and limited forms of higher-order functions. Finally, Chin and Khoo's use of a Presburger arithmetic solver limits the expressiveness of sizes to *affine functions* over size variables, whereas our system allows for general monotone functions, including polynomials.

Most closely related to our analysis is the system by Reistad and Gifford [13] for the cost analysis of Lisp expressions. This system handles higher-order functions through "latent costs" as we have done here, and is partially based on the "time system" by Dornic et al. [5], Rather than trying to infer costs for user-defined recursive functions, however, Reistad and Gifford require the use of fixed higher-order skeletons with known latent costs.

Pioneering work on *automatic complexity analysis* was undertaken by Wegbreit [17]. Wegbreit's METRIC system derived probabilistic complexity measures of a limited range of first-order Lisp programs by solving the difference equations that occur as an intermediate step in the complexity analysis. The analysis, however, is not guaranteed to be sound as the system assumes statistical independence of tests in conditionals. Consequently, the programmer must confirm the validity of the analysis against the semantics of the program.

Le Métayer [7] 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. However, like Reistad and Gifford's approach, recursive definitions must be given in terms of a particular set of skeletons. Moreover, the analysis is not *modular* as the transformation can only be applied to a complete programs. Rosendahl [14] also uses *program transformation*; in this case to obtain a step counting version of first-order Lisp programs. This is followed by abstract interpretation to obtain a program giv-

ing an upper bound on the cost. Again this abstract interpretation requires a complete program, limiting both its scalability and its applicability to systems with e.g. compiled libraries. Finally, Benzinger [2] obtains worst-case complexity analysis for NuPrl-synthesized programs by "*symbolic execution*" followed by recurrence solving. The system supports first-order functions and lazy lists but requires higher-order functions to be annotated with complexity information. Moreover, only a restricted primitive recursion syntax is supported. These limitations are justified by Benzinger's objective, which is to aid resource analysis for automatically synthesized programs, rather than to analyze hand-written functions, as in our case.

7 Conclusions and Further Work

The main contribution of this paper is a type reconstruction algorithm to estimate sizes and costs for a simple functional language with recursive, higher-order and polymorphic functions Our algorithm is an extension of the standard Hindley-Milner type inference and as such we achieve full *modularity* of the analysis. The results obtained for recursion by our analysis are determined solely by the *deconstruction of inductive types* (i.e. naturals or lists) and not by any conditionals in the source program. Although this might lead to over-estimation of costs in some cases, it has the advantage of placing no syntactical restrictions on the forms of recursion we can analyze.

We have found that our approach produces accurate cost equations for a representative subset of the Haskell standard Prelude functions, suggesting it should yield useful information in a more practical setting. Although we have not yet analysed the complexity of the inference algorithm, our experience with the prototype implementations suggests that its execution time is comparable to ordinary type inference.

A number of issues remain to be studied. Firstly, we need to extend our notion of sized types and inference to handle *full integer arithmetic* and a richer set of data-types including *user-defined recursive structures*. This will ultimately allow us to address real languages such as our resource-bounded language *Hume*. Secondly, since this is not the primary focus of our research, we have not addressed the problem of automatically obtaining closed forms for the recurrence equations; for some subclasses of these equations there are mechanical methods that yield closed forms [8]. All general-purpose computer algebra systems (e.g. Maple, Mathematica and MuPAD) provide some functionally to solve these equations. The new Mathematica Version 5 is also able to solve recurrence equations in *multiple variables*[3]. All recurrences obtained for the examples in Section 5 can be solved by Mathematica 5 with only slight human intervention to eliminate the max terms. We intend to automate this step in due course. Thirdly, although we conjecture that a notion of principal type should hold for our system, we have not yet addressed this issue. Since our analysis will derive *an* upper bound

[3] In practice, the authors have encountered simple recurrences for which Mathematica yields a wrong solution — a bug that has been reported to the software publisher!

sized type, but not necessarily the *least* one, this is, of course, purely a *quality* rather than *soundness* issue. Finally, we have not yet constructed *soundness* or *completeness* proofs relating our inference algorithm to the type system. We believe, however, that these should be analogous to proofs for other type and effect systems [1].

We are grateful to Álvaro J. Rebón Portillo, Roy Dyckhoff, Hans-Wolfgang Loidl, Greg Michaelson and the anonymous referees for their helpful comments on earlier drafts of this paper. This work is generously sponsored by EPSRC grant GR/R 70545/01.

References

1. T. Amtoft, F. Nielson, and H.R. Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, 1999.
2. R. Benzinger. Automated Complexity Analysis of Nuprl Extracted Programs. *Journal of Functional Programming*, 11(1):3–31, 2001.
3. W.-N. Chin and S.-C. Khoo. Calculating Sized Types. *Higher-Order and Symbolic Computing*, 14(2,3), 2001.
4. L. Damas and A.J.R.G. Milner. Principal Type-Schemes for Functional Programs. In *Proc. 1982 ACM Symp. on Principles of Prog. Langs. – POPL '82*, pages 207–212, 1982.
5. V. Dornic, P. Jouvelot, and D.K. Gifford. Polymorphic Time Systems for Estimating Program Complexity. *ACM Letters on Prog. Lang. and Systems*, 1(1):33–45, March 1992.
6. R.J.M. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems using Sized Types. In *Proc 1996 ACM Symposium on Principles of Programming Languages – POPL '96*, St Petersburg, FL, January 1996.
7. D. Le Métayer. ACE: An Automatic Complexity Evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2), April 1988.
8. H. Levy and F. Lessman. *Finite Difference Equations*. Macmillan, 1961.
9. H-W. Loidl and K. Hammond. A Sized Time System for a Parallel Functional Language. In *Glasgow Workshop on Functional Programming*, Ullapool, July 1996.
10. A.J.R.G. Milner. A Theory of Type Polymorphism in Programming. *J. Computer System Sciences*, 17(3):348–375, 1976.
11. F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
12. Á. Rebón Portillo, K. Hammond, H.-W. Loidl, and P.B. Vasconcelos. Cost Analysis using Automatic Size and Time Inference. In *Proc. IFL 2002 – Implementation of Functional Languages, Madrid, Spain*, LNCS 2670. Springer-Verlag, 2003.
13. B. Reistad and D.K. Gifford. Static Dependent Costs for Estimating Execution Time. In *Proc. 1994 ACM Conference on Lisp and Functional Programming – LFP '94*, pages 65–78, Orlando, FL, June 1994.
14. M. Rosendahl. Automatic Complexity Analysis. In *Proc. 1989 Intl. Conf. on Functional Prog. Langs. and Comp. Arch. – FPCA '89*, pages 144–156, 1989.
15. D.A. Turner. Elementary Strong Functional Programming. In *Proc. Symp. on Funct. Prog. Langs. in Education — FPLE '95*, LNCS. Springer-Verlag, Dec. 1995.
16. P.B. Vasconcelos and K. Hammond. A Type and Effect System for Costing Recursive, Higher-Order and Polymorphic Functional Programs. In preparation, 2003.
17. B. Wegbreit. Mechanical Program Analysis. *Comm. of the ACM*, 18(9), 1975.