Extending Sized Type with Collection Analysis

Wei-Ngan Chin* and Siau-Cheng Khoo and Dana N. Xu
School of Computing
National University of Singapore
{chinwn,khoosc,xun}@comp.nus.edu.sg

ABSTRACT

Many program optimizations and analyses, such as array-bounds checking, termination analysis, depend on knowing the size of a function's input and output. However, size information can be difficult to compute. Firstly, accurate size computation requires detecting a size relation between different inputs of a function. Secondly, size information may also be contained inside a collection (data structure with multiple elements). In this paper, we introduce some techniques to derive universal and existential size properties over collections of elements of recursive data structures. We shall show how a mixed constraint system could support the enhanced size type, and highlight examples where collection analysis are useful.

Categories and Subject Descriptors

D.3.2 [**Language Classifications**]: Applicative (functional) languages; D.3.4 [**Processors**]: Optimization; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Invariants; F.3.2 [**Semantics of Programming Languages**]: Program analysis

General Terms

Languages, Theory, Reliability

Keywords

Sized Type, Polymorphism, Mixed Constraints, Collection Analysis, Fix-Point

1. INTRODUCTION

Programming languages play a crucial role in helping to develop correct and efficient programs within reasonable efforts. While strongly-typed languages, such as Haskell, ML and Java, have come a long way in helping to eliminate type-based errors, many simple logical errors, such as bound checks or pattern-matching failure, remain outside the reach of classical type-systems. While the complete elimination of such logical errors is undecidable, it is still beneficial to detect as many scenarios as possible where checks can be determined to be safe and hence be made redundant. It is also useful to detect those scenarios in which checks are determined as possibly unsafe, since these are the places where closer scrutiny for bugs may be needed.

A number of proposals for more powerful type systems, such as usage types [17], binding-time types [4] for partial evaluation, and flow types[5], have been proposed. These annotated type systems are capable of capturing extra static properties that could be propagated through type inference. In an earlier work[2], we have proposed a sized type system for capturing and propagating size information. These size properties are relational in nature, where relationships between inputs, as well as relationships between input/output may be captured by linear arithmetic formulae. To illustrate the basic concepts behind sized types, consider the following three simple functions (in Haskell syntax):

$$
\begin{array}{l}
f(a,b,c) \;=\; (a+b, a*a, c) \\
\mathit{tail}\ \mathit{xs} \;=\; \textbf{case}\ \mathit{xs}\ \textbf{of}\ (y:\mathit{ys}) \;\rightarrow\; \mathit{ys} \\
\mathit{append}(\mathit{xs},\mathit{ys}) \;=\; \textbf{case}\ \mathit{xs}\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathit{Nil} \qquad\; \rightarrow \mathit{ys} \\
\qquad\qquad\qquad\qquad (x:\mathit{xs}') \;\rightarrow x \;:\; \mathit{append}(\mathit{xs}',\mathit{ys})
\end{array}
$$

Based on the type rules given in [2], we can denote the sized types for each expression e using $(e :: \tau \text{ s.t. } \phi)$ where τ is an annotated size type with size variables, while ϕ is a set of constraints that holds for the size variables present in τ. For example, if e is of *Int* type, its annotated type would be Int^v where v is the size variable of the annotated type.

In case e is a function, we denote its sized type using $e :: (\tau_1, .., \tau_n) \rightarrow \tau'$ s.t. **size** ϕ_1; **inv** ϕ_2, where ϕ_1 is a set of constraint that holds between the size variables present in the annotated type, while ϕ_2 represents a size invariant that holds for the size variables of parameters of the recursive function. For example, the above functions can be given the following sized types.

$$
\begin{array}{l}
f :: (Int^a,\ Int^b, \alpha^c) \;\rightarrow\; (Int^{r1}, Int^{r2}, \alpha^{r3}) \\
\quad \text{s.t. } \textbf{size}\ \ r1 = a+b \;\wedge\; r3 =_t c \\
\mathit{tail} \;::\; [\alpha^d]^m \;\rightarrow\; [\alpha^r]^n \text{ s.t. } \textbf{size}\ \ n = m-1 \;\wedge\; m > 0 \\
\mathit{append} \;::\; ([\alpha^d]^m,\ [\alpha^e]^n) \;\rightarrow\; [\alpha^r]^p \\
\quad \text{s.t. } \textbf{size}\ p = m+n \;;\ \textbf{inv}\ m^+ < m \;\wedge\; n^+ = n
\end{array}
$$

Note the use of integer size variables, e.g a, b, m, n, to capture the sizes of the corresponding *Int* and *List* types, while a polymorphic variable for f has to be captured by

*Fellow, Singapore-MIT Alliance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PEPM'03, June 7, 2003, San Diego, California, USA.
Copyright 2003 ACM 1-58113-667-6/03/0006 ...$5.00.

polymorphic size variable, c, where *term equality*, $=_t$, is supported, while some size variables, (e.g. d, e), of *tail* and *append* are unconstrained. For practical reasons, formulae for size predicates have been restricted to Presburger arithmetic for which efficient constraint-solving technology (e.g. Pugh's Omega Calculator[16]) exists. Sized types are fairly expressive, as they are able to capture a variety of static properties, including:

- *Input-output size relationships* that are in Presburger form. For example, the relationship between input a, b and the first output, $r1$, of function f is captured by $r1 = a + b$. The second output, $r2 = a * b$ of f, is not captured since it is outside of Presburger form.
- *Pre-conditions* that should be satisfied to avoid run-time errors. For example, the *tail* function requires that input list be non-empty. This is specified by the constraint $m > 0$.
- *Invariant properties* that describe relationships between parameters (denoted by m^+, n^+) of an arbitary recursive call, and the original arguments (denoted by m, n) of the first recursive call. For example, the invariant properties $m^+ < m$ and $n^+ = n$ captures the fact that the first argument is decreasing (in size) across each recursive step, while the second parameter is unchanged (in size) for the recursive *append* function.

While a major contribution in our previous work was an inference procedure for calculating a reasonably accurate sized-type, our earlier proposal suffered from three major deficiencies, namely:

- Type system is size- but not type-polymorphic.
- Size information of elements kept inside recursive data structures are usually not captured.
- Higher-order programs are not properly handled since function-type parameters were size monomorphic.

In this paper, we propose to address the first two deficiencies by enhancing sized type rules with extra constraints. These constraints are beyond arithmetic, but can be used to propagate arithmetic properties. We shall propose the use of polymorphic size variables to capture *size structures*, and see how this simple mechanism is able to handle polymorphic programs. Size structures are size terms containing size variables. For example, the size structure of $[Int^a]^n$ is denoted by $[a]n$. We shall also show how the flow of elements inside recursive data structures (also known as *collection*) can be modelled with universal/existential arithmetic relations over bags. These bag relations allow properties of collections to be safely propagated in an interprocedural fashion.

For example, the *tail* and *append* functions have polymorphic collection from the elements of input and output list. These elements can be related more precisely by bag relations, as shown in the following sized types (omitting the invariant constraint for *append*).

$$\begin{array}{l} tail :: [\alpha^a]^m \rightarrow [\alpha^r]^n \\ \quad \text{s.t. } \mathbf{size}\ n = m - 1 \wedge m > 0 \wedge r.bag \sqsubseteq a.bag \\ append :: ([\alpha^a]^m, [\alpha^b]^n) \rightarrow [\alpha^r]^p \\ \quad \text{s.t. } \mathbf{size}\ p = m + n \wedge r.bag = a.bag \sqcup b.bag \end{array}$$

Note that the notation $b.bag$ is used to denote a collection of elements that are associated with size variable b. As a shorthand we shall also use $b\$$ to denote $b.bag$. Relations involving such bag relations are more expressive than (instance) relations on their individual elements. A more detailed comparison will be given later. (The corresponding instance relation for elements of *append* function would be $r = a \vee r = b$.) Bag relations also facilitate the propagation of properties from one collection to another. As an example, we can propagate universal property over bags through the following propagation rule:

$$A\$ \sqsubseteq B\$ \wedge (\forall x \in B\$.\ P[x]) \Rightarrow (\forall x \in A\$.\ P[x])$$

Note that $P[x]$ is a predicate formula over the x instance. As another shorthand, we shall use $\forall B\$.\ P[B]$ to denote $\forall x \in B\$.\ P[x]$. A corresponding propagation rule for existential property can be expressed as follows.

$$A\$ \sqsubseteq B\$ \wedge (\exists A\$.\ P[A]) \Rightarrow (\exists B\$.\ P[B])$$

The main contributions of this paper are:

- We make use of a *mixed constraint* system. This mixed constraint system uses arithmetic constraint for size analysis, bag constraint for collection analysis, and term constraint to support type polymorphism. We show how information from one class of constraints may be utilized by another class of constraints. (Section 3)
- We enhance a previous sized type system to support *type polymorphism* in a relatively straightforward way. (Section 4)
- We introduce the concept of *collection analysis* for recursive data structures with multiple elements, such as lists and arrays. This analysis allows us to determine how size properties of elements may be propagated. (Section 5)
- We highlight how fixed-point computation may be done for collections of elements, and provide techniques that could discover universal/existential properties over these collections. (Section 5.2)

2. LANGUAGE AND SIZED TYPE

We apply our technique to a first-order typed functional language with strict semantics. The language is defined in Figure 1. Recursive functions in the language are confined to self-recursion. No loss of generality results since mutual recursion can be transformed to self-recursion through suitable tagging of its input/output values.

We only consider *well-typed* programs. We enhance the type system with a mixed constraint syetem with two main pieces of safety information, namely : (a) size relation and (b) collection/bag relation. Together with the type system, they are jointly called *sized types*.

A polymorphic function will have some type variables in its type. We do not explicitly quantify these type variables, as we do not need to handle conventional generalization of type to type schemes. Nevertheless, it is understood that all type variables occurring in a program are universally quantified.

The syntax of sized types is depicted in Fig. 2. It is a pair containing an annotated type and a formula. An annotated

$f \in \mathbf{Fun}$ ⟨Functions⟩
$x \in \mathbf{Var}$ ⟨Variables⟩
$c \in \mathbf{Con}$ ⟨Constructor Names⟩
$n \in \mathbf{Int}$ ⟨Integer Constants⟩
$e \in \mathbf{Exp}$ ⟨Expressions⟩

$$
\begin{array}{rcl}
e & ::= & x \mid n \mid c(e_1, \ldots, e_n) \\
& & \mid \textbf{let } x = e_1 \textbf{ in } e2 \mid f(e_1, \ldots, e_n) \\
& & \mid \textbf{letrec } f(x_1, \ldots, x_n) = e_1 \textbf{ in } e_2 \\
& & \mid \textbf{case } e_0 \textbf{ of } \{c_i(x_{1_i}, \ldots, x_{m_i}) \rightarrow e_i\}_{i=1}^{n}
\end{array}
$$

$p \in \mathbf{Pat}$ ⟨Patterns⟩

$$p ::= x \mid c(x_1, \ldots, x_n)$$

Figure 1: The Language Syntax

Sized Type = (**AnnType**, **FS**)

Annotated Type Expressions:

$t \in \mathbf{TVar}$ ⟨Type Variables⟩
$v \in \mathbf{V}$ ⟨Size Variables⟩
$\alpha \in \mathbf{A}$ ⟨Basic Size Annotation⟩
$\sigma \in \mathbf{AnnType}$ ⟨Annotated Types⟩

$$
\begin{array}{rcl}
\sigma & ::= & \tau \mid (\tau_1, \ldots, \tau_n) \rightarrow \tau \\
\tau & ::= & t^{\alpha} \mid \texttt{Int}^{\alpha} \mid \texttt{Bool}^{\alpha} \mid [\tau]^{\alpha} \mid (\tau_1, \ldots, \tau_n) \\
\alpha & ::= & v \mid \top
\end{array}
$$

UnAnnotated Type Expressions:

$$\overline{\tau} ::= t \mid \texttt{Bool} \mid \texttt{Int} \mid [\overline{\tau}] \mid (\overline{\tau_1}, \ldots, \overline{\tau_n})$$

Size Term Extraction :

$s \in \mathbf{S}$ ⟨Size Term Annotation⟩

$$
\begin{array}{rcl}
\tau & \equiv & \overline{\tau}^{s} \\
s & ::= & \alpha \mid [s]\alpha \mid (s_1, \ldots, s_n)
\end{array}
$$

Constraint Abstraction Store:

$\Sigma \in \mathbf{CAStore}$

$$\Sigma ::= \{ (f :: (\overline{\tau_1}^{\,s_1}, \ldots, \overline{\tau_n}^{\,s_n}) \rightarrow \overline{\tau_{n+1}}^{\,s_{n+1}}, \; Q(s_1, \ldots, s_{n+1}) = \Phi)^* \}$$

Formulae:

$\Phi \in \mathbf{FS}$ ⟨Formulae⟩

$$
\begin{array}{rcl}
\Phi & ::= & \phi \mid \gamma \mid Q(s_1, \ldots, s_{n+1}) \mid \exists\, v \,.\, \Phi \\
& & \mid \Phi_1 \vee \Phi_2 \mid \Phi_1 \wedge \Phi_2 \mid s_1 =_t s_2 \\
& & \mid \exists\, v\$ \,.\, \Phi \mid \forall\, v\$ \,.\, \Phi \\
& & \mid \mathit{case}\, \{\Phi_i \rightarrow \Phi_i'\}_{i=1}^{n}
\end{array}
$$

Size Formulae:

$\phi \in \mathbf{F}$ ⟨Boolean Expressions⟩

$$
\begin{array}{rcl}
\phi & ::= & \mathit{True} \mid \mathit{False} \mid \neg\, b \mid a_1 = a_2 \\
& & \mid a_1 \neq a_2 \mid a_1 < a_2 \mid a_1 > a_2 \\
& & \mid a_1 \leq a_2 \mid a_1 \geq a_2
\end{array}
$$

$a \in \mathbf{AExp}$ ⟨Arithmetic Expressions⟩

$$a ::= n \mid v \mid n * a \mid a_1 + a_2 \mid -a$$

$n \in \mathcal{Z}$ ⟨Integer constants⟩

Collection Formulae:

$\gamma \in \mathbf{C}$ ⟨Collection Formulae⟩

$$
\begin{array}{rcl}
\gamma & ::= & v\$ = se \mid v_1\$ \sqsubseteq v_2\$ \\
se & ::= & \{\,\} \mid \{v\} \mid se_1 \sqcup se_2 \mid v\$
\end{array}
$$

Figure 2: Syntax of Sized Types

type expression augments an ordinary type expression with size variables; the relationship among these variables are expressed in the associated formula.

In this paper, we consider only three basic types: lists of the form $(e_1 : e_2)$ and $[\,]$, integers, and booleans of values *False* and *True*. While the array data type is omitted in this paper, its inclusion is straightforward. (Note that values of *Bool*, $[\tau]$ and $(\tau_1, .., \tau_n)$ are special cases of constructors of the form $c(e_1, .., e_n)$.) The annotated type for lists is $[\,\tau\,]^v$, where v describes the length of the list; for integers, it is $\texttt{Int}^v$, where v captures the integer value; for booleans, it is $\texttt{Bool}^v$, where v can be either 0 or 1, representing the values *False* and *True* respectively. Occasionally, we omit writing size variables in the annotated type when these variables are unconstrained.

Given an annotated type τ, we denote its corresponding unannotated type by the notation $\overline{\tau}$; while its corresponding size-term extraction can be captured using $\overline{\tau}^{\,s}$. The purpose of this extraction is to gather all size information associated with an expression at a place. Using size term extraction, we separate the unannotated type from the associated size information. For instance, the annotated sized type $[[Int^v]^u]^w$ can be re-expressed as $\overline{\tau}^s$ where $\overline{\tau} = [[Int]]$ and $s = [[v]u]w$.

We associate a *constraint abstraction* Q to each function. $Q(s_1, \ldots, s_{n+1}) = \Phi$ defines the constraint between the size of the function's input and output. Note the parameters associated with Q: $s_1, \ldots, s_n$ denotes the size terms (or we may call it size patterns) of the associated function's inputs, and s_{n+1} denotes that of the output. Φ is a formulae expressed in terms of only the free variables occurring in Q's parameters.

We maintain a constraint abstraction store (denoted by Σ) to keep size information about functions: the annotated size type of a function and its corresponding constraint abstraction.

Size formulae in constraint abstraction are basically *Presburger* formulae. We shall use the term "formulae" and "constraint" interchangeably. Note that arithmetic expressions used in formulae are restricted to affine functions of the variables. For example, $2 * v$ is legal, but $v * v$ is not. When $X = \{v_1, \ldots, v_n\} \subseteq \mathbf{V}$, we write $\exists\, X.\, \phi$ or $\exists\, v_1 \ldots v_n \,.\, \phi$ as a short hand for $\exists\, v_1 \,.\, \ldots \,.\, \exists\, v_n \,.\, \phi$.

We view the part of our constraint system that handles Presburger formulae as a *cylindric algebra* [10], where constraints are identified modulo logical equivalence. Cylindric algebra provides an algebraic-style reasoning about the formulae. Our cylindric algebra, $\mathbf{P}$, is a boolean algebra where existential quantification ($\exists\, X$) is the (idempotent) projection operation, $\vee$ and $\wedge$ are the meet (denoted by $\sqcup_{\mathbf{P}}$) and the join (denoted by $\sqcap_{\mathbf{P}}$) operator respectively, $\neg$ is the complement, and *False* and *True* are bottom and top elements, respectively. The algebra has the following cylindrification properties[14, 15].

P1. $\phi \vdash_{\mathbf{P}} \exists\, v.\phi$
P2. $\phi \vdash_{\mathbf{P}} \psi$ implies $\exists\, v \,.\, \phi \vdash_{\mathbf{P}} \exists\, v.\psi$
P3. $\exists\, v.\, (\phi \wedge \exists\, v.\, \psi) =_{\mathbf{P}} (\exists\, v.\phi) \wedge (\exists\, v.\psi)$
P4. $\exists\, v.\, \exists\, w.\, \phi =_{\mathbf{P}} \exists\, w.\, \exists\, v.\phi$.

In the above, $\vdash_{\mathbf{P}}$ denotes logical entailment. We say that ϕ *entails* ψ, written $\phi \vdash_{\mathbf{P}} \psi$, if ψ is a logical consequence of ϕ. Also, $\phi =_{\mathbf{P}} \psi$ if and only if $\phi \vdash_{\mathbf{P}} \psi$ and $\psi \vdash_{\mathbf{P}} \phi$; in this case, we say ϕ and ψ are logically equivalent.

The other part of the constraint system involves size term. Basically, it identifies two size terms: $s_1 =_t s_2$.

Collection formulae capture the containment of information flown from input to output. A collection/bag can be empty ($\{\,\}$); or it can contain a single size information captured by a size variable; or it can be composed by merging size variables from two sources. $v\$$ is a bag variable, and it can be assigned a collection information, as in $v\$ = se$. The relation $v_1\$ \sqsubseteq v_2\$$ expresses that elements available in $v_1\$$ are drawn from those in $v_2\$$.

Free variables in a bag formula can be quantified both universally and existentially. We shall discuss these properties in detail later.

3. TYPE SYSTEMS

$$\frac{v\$ = \{v\} \ \wedge \ \Phi(v)}{\forall\, v\$ \,.\, \Phi(v)} \qquad [\forall\text{-I}]$$

$$\frac{v\$ = \{v\} \ \wedge \ \Phi(v)}{\exists\, v\$ \,.\, \Phi(v)} \qquad [\exists\text{-I}]$$

$$\frac{\begin{array}{c} v\$ = v'\$ \sqcup v''\$ \ \wedge \ \forall\, v'\$ \,.\, \Phi'(v') \\ \wedge \forall\, v''\$ \,.\, \Phi''(v'') \end{array}}{\forall\, v\$.\ \Phi'(v) \ \vee \ \Phi''(v)} \qquad [\text{UP-1}]$$

$$\frac{v_1\$ \sqsubseteq v\$ \ \wedge \ \forall\, v\$ \,.\, \Phi(v)}{\forall\, v_1\$ \,.\, \Phi(v_1)} \qquad [\text{UP-2}]$$

$$\frac{v\$ = \{\,\}}{\forall\, v\$ \,.\, \top} \qquad [\text{UP-3}]$$

$$\frac{v_1\$ \sqsubseteq v\$ \ \wedge \ \exists\, v_1\$ \,.\, \Phi(v_1)}{\exists\, v\$ \,.\, \Phi(v)} \qquad [\text{UP-4}]$$

Figure 3: Uniform Property Rules

The type rules are formulated under the theory of Presburger arithmetic and uniform property. The former is well understood from the literature; the latter is the construction of a theory for uniform properties, which are either universally or existentially quantified. Figure 3 depicts the set of rules for introducing and propagating these uniform properties about bags. The set of type rules are shown in Figures 4 and 5. Γ is a type environment associating program variables to their annotated types, while Σ maintains the association of function names with its annotated types and its corresponding constraint abstraction. The type judgment $\Gamma, \Sigma \ \vdash_{\mathsf{UP}} \ e :: (\tau^s, \Phi)$ means that an expression e has the annotated type (τ^s, Φ) under the theories of Presburger arithmetic and uniform property if all its free variables have their annotated types given by the annotated type environment Γ, and all the constraint abstractions occurring in Φ have their formulae defined in the store Σ.

A variable has constraint *True* when it is within the scope of its definition. The size of an integer is the integer itself. The size of booleans are defined as 0 and 1, for the values *False* and *True* respectively.

With list, its elements constitute the collection information. The size of an empty list is its length – zero; its collection is an empty set. The size of the (:) operation is one more than its second operand, and its collection is formed by merging the collection information of both operands. Note that its first operand contains a singleton bag.

The annotated type of a **let**-expression contains the formula resulting from the conjunction of the corresponding formulae for local abstraction and the **let**-body. Note that the constraint associated with the local variable is not used during the computation of the **let**-body; this avoids duplication of constraint at each occurrence of the variable.

In definition of a recursive function f (rule [*Rec*]), we compute the constraint associated with the function. The computed constraint for e_1, Φ, differs from the eventual constraint for f, Φ_f, only in the set of free size variables. Indeed, all intermediate size variables in Φ will be eliminated via existentially quantification. It is easy to verify that $\Phi \Rightarrow \Phi_f$, and thus the type rule does compute a fixed point for the constraint of the recursive function. Notation wise, we note that the size-term extractions of the parameters and result of f are represented by s_i. Lastly, function ι takes in either an annotated type or a formula, and returns the set of free size variables occurred in its argument. It is extended naturally to operate on type environment.

The formula for function application expresses the application of constraint abstraction Q to the size variables associated with the arguments. The type variables of the function's type are first instantiated with respect to the arguments, and the function's annotated type is then properly instantiated to match the application arguments. These are performed by operations *eqP* and *subs*, the detail of which are described in the next section. Then, the constraint abstraction Q is invoked with the set of size terms, which represent the instantiated arguments and return result. Lastly, the formulae associated with the arguments are also captured in the resulting formula.

The formula of **case**-expression maintains the relation between the test and consequence of each branch. This is captured by the *case*-formula. Expressed in the usual logic formulation, we have

$$case\ \{\Phi_{p_i} \ \rightarrow \ \Phi_i\}_{i=1}^{n} \ \stackrel{def}{=} \ \bigvee_{i=1}^{n} \{\Phi_{p_i} \ \wedge \ \Phi_i\}.$$

In the type rule, function *eqC* takes in two annotated types (with identical underlying type) and equates the corresponding size variables occurring in them. Function *sizeR* replaces all size variables occurred in an annotated type by fresh names.

To formulate the soundness of the type system, we first specify the notion the *satisfiability* with respect to the semantics of the program. Given a denotational value d with type τ, let $\mathcal{S}$ be the formula expressing the size and container information of d. Then,

> **Satisfiability** Given a denotational value d of an annotated type τ, we say d *satisfies* a constraint Φ under τ if $\mathcal{S}(d :: \tau) \Rightarrow \exists V.\Phi$, where $V = \iota(\Phi) - \iota(\tau)$.

Soundness of type system can thus be expressed as follows:

$$\frac{\Gamma(x) \;=\; \tau}{\Gamma, \Sigma \;\vdash_{\texttt{UP}}\; x \;::\; (\tau,\, True)} \qquad \text{[Var]}$$

$$\Gamma, \Sigma \;\vdash_{\texttt{UP}}\; n \;::\; (Int^{v},\, v \;=\; n) \qquad \text{[Con]}$$

$$\Gamma, \Sigma \;\vdash_{\texttt{UP}}\; [\,] \;::\; ([\overline{\tau}^{s}]^{v},\, v \;=\; 0 \;\wedge\; v_1 \;=_t\; s \;\wedge\; v_1\$ = \{\,\}) \qquad \text{[List-Nil]}$$
$$\text{where } v_1 \text{ is a fresh variable}$$

$$\frac{\Gamma,\, \Sigma \;\vdash_{\texttt{UP}}\; e_1 \;::\; (\overline{\tau}^{s1}, \Phi_1) \quad \Gamma,\, \Sigma \;\vdash_{\texttt{UP}}\; e_2 \;::\; ([\overline{\tau}^{s2}]^{n2},\, \Phi_2)}{\Gamma,\, \Sigma \;\vdash_{\texttt{UP}}\; (e_1 : e_2) \;::\; ([\overline{\tau}^{s3}]^{n3}, \Phi_3)} \qquad \text{[List-Cons]}$$

$$\begin{array}{rcl} \text{where} \quad \Phi_3 & = & \Phi_1 \;\wedge\; \Phi_2 \;\wedge\; n3 \;=\; n2 + 1 \;\wedge \\ & & v_1 \;=_t\; s1 \;\wedge\; v_2 \;=_t\; s2 \;\wedge\; v_3 \;=_t\; s3 \;\wedge \\ & & v_3\$ \;=\; (v_1\$ \sqcup v_2\$) \;\wedge\; v_1\$ \;=\; \{\, v_1 \,\} \end{array}$$
$$v_1,\; v_2,\; v_3 \text{ are fresh variables}$$

$$\Gamma, \Sigma \;\vdash_{\texttt{UP}}\; False \;::\; (Bool^{v},\, v \;=\; 0) \qquad \text{[Bool-False]}$$

$$\Gamma, \Sigma \;\vdash_{\texttt{UP}}\; True \;::\; (Bool^{v},\, v \;=\; 1) \qquad \text{[Bool-True]}$$

Figure 4: Type Rules – Part 1

Type Soundness Theorem Given an expression e of type $\overline{\tau}$, for some τ. If $[], \Sigma \vdash_{\texttt{UP}} e :: (\tau, \Phi)$, then for any monomorphic instance of e with denotation d, d satisfies Φ under τ.

The proof is an extension of our previous proof about sized type [1]. We are currently finalizing the proof detail.

4. TYPE POLYMORPHISM

One of the most important aspects of program analyses is that they be *property polymorphic*, with the ability to infer different properties for a definition f at separate uses of f. While related, this aspect is in fact orthogonal to the issue of polymorphism in the underlying type. Recent work [18, 12, 6] have suggested that extension to type polymorphism is not necessarily straightforward.

In this section, we highlight how type polymorphism is handled by our enhanced sized type system. Specifically, our approach allows the instantiation of type variables at the point of function application. Here, the type of an argument is likely to be more specific than that of the parameter for the polymorphic function. Under this scenario, we must collect all type instantiation that has taken place. We use $eqP\ \tau_a\ \tau_p$, where τ_a represents the argument type, and τ_p is a possibly polymorphic parameter type, so as to capture a set of type scheme instantiations.

$$\begin{array}{ll} eqP\ \tau\ t^{\alpha} & \Rightarrow \{t = \overline{\tau}\} \\ eqP\ \overline{\tau}^{a}\ \overline{\tau}^{b} & \Rightarrow \{\,\} \text{ if } \overline{\tau} \;=\; Int \;\vee\; \overline{\tau} \;=\; Bool \\ eqP\ ([\tau]^{a})\ ([\tau']^{b}) & \Rightarrow (eqP\ \tau\ \tau') \\ eqP\ (\tau_1, .., \tau_n)\ (\tau'_1, .., \tau'_n) & \Rightarrow \bigcup_{i=1}^{n} (eqP\ \tau_i\ \tau'_i) \end{array}$$

Given that $D = eqP\ \tau_a\ \tau_p$, we may now apply a substitution to convert the polymorphic type of the function τ to its instantiated equivalent τ' using $subs\ D\ \tau \Rightarrow (\tau', \phi)$. In this operator, new size variables are introduced for each type instantiation (via the *new* operation.) The constraint ϕ relates the new size variables with existing ones.

$$\begin{array}{ll} subs\ D\ t^{\alpha} & \Rightarrow \\ \quad if (t = \overline{\tau}) \in D\ then\ let\ \overline{\tau}^{s} \;=\; new\ \overline{\tau}\ in\ (\overline{\tau}^{s}, \alpha =_t s) & \\ \quad else\ (t^{\alpha}, True) & \\ subs\ D\ Int^{a} & \Rightarrow (Int^{a}, True) \\ subs\ D\ Bool^{b} & \Rightarrow (Bool^{b}, True) \\ subs\ D\ (\tau_1, .., \tau_n) & \Rightarrow ((\tau'_1, .., \tau'_n),\ \wedge_{i=1}^{n} \phi_i) \\ \quad \texttt{where}\ \forall\, i.\, (\tau'_i, \phi_i) \;=\; subs\ D\ \tau_i & \\ subs\ D\ ([\tau]^{a}) & \Rightarrow ([\tau']^{a},\ \phi) \\ \quad \texttt{where}\ (\tau', \phi) \;=\; subs\ D\ \tau & \\ subs\ D\ ((\tau_1, .., \tau_n) \;\rightarrow\; \tau_{n+1}) & \Rightarrow \\ \quad ((\tau'_1, .., \tau'_n) \rightarrow \tau'_{n+1}, \wedge_{i=1}^{n+1} \phi_i) & \\ \quad \texttt{where} \forall i \in \{1, .., n+1\}.(\tau'_i, \phi_i) = subs\ D\ \tau_i & \end{array}$$

As a simple example of how type polymorphism is handled, consider:

$$\begin{array}{l} fst \;::\; ((A^{a}, B^{b}) \;\rightarrow\; A^{m},\; m = a) \\ fst\ (a, b) \;=\; a \end{array}$$

When *fst* is applied to the expression: $e :: ((Int^{n}, E^{c}), F^{d}), n = 2)$ we can obtain its type instantiation by relating the size of the argument to the size of its corresponding parameter, namely: $eqP\ ((Int^{n}, E^{c}),\ F^{d})\ (A^{a}, B^{b})$. This yields

$$\frac{\Gamma,\Sigma \vdash_{\mathtt{UP}} e_1 :: (\tau_1,\ \Phi_1) \quad \Gamma \cup \{x :: \tau_1\},\Sigma \vdash_{\mathtt{UP}} e_2 :: (\tau_2,\ \Phi_2)}{\Gamma,\Sigma \vdash_{\mathtt{UP}} (\mathbf{let}\ x\ =\ e_1\ \mathbf{in}\ e_2) :: (\tau_2,\ \Phi_2 \wedge \Phi_1)} \quad [\text{Let}]$$

$$\frac{\begin{array}{c}\Gamma \cup \Gamma',\ \Sigma \cup \Sigma_f \vdash_{\mathtt{UP}} e_1 :: (\tau_{n+1},\ \Phi) \\ \Gamma,\ \Sigma \cup \Sigma_f \vdash_{\mathtt{UP}} e_2 :: (\tau',\ \Phi')\end{array}}{\begin{array}{c}\Gamma,\ \Sigma \vdash_{\mathtt{UP}} (\ \mathtt{letrec}\ f(x_1,\ldots,x_n)\ =\ e_1\ \mathrm{in}\ e_2)\ :\ (\tau',\ \Phi') \\ \text{where}\quad \begin{array}{rcl}\Sigma_f & = & \{\,(f :: (\tau_1,\ldots,\tau_n) \rightarrow \tau_{n+1},\ Q(s_1,\ldots,s_{n+1}) = \Phi_f)\,\} \\ \Gamma' & = & \bigcup_{i=1}^{n}\{x_i :: \tau_i\} \\ \Phi_f & = & \exists V\,.\,\Phi \\ V & = & \iota(\Phi) - (\bigcup_{i=1}^{n+1}\iota(\tau_i) \cup \iota(\Gamma))\end{array}\end{array}} \quad [\text{Rec}]$$

$$\frac{\begin{array}{c}(f :: (\tau_{p1},\ldots,\tau_{pn}) \rightarrow \tau_{p(n+1)},\ Q(s_{p1},\ldots,s_{p(n+1)}) = \Phi_f\ ) \in \Sigma \\ \Gamma,\Sigma \vdash_{\mathtt{UP}} e_i :: (\overline{\tau_{ai}}^{\,s_{ai}},\ \Phi_i)\ \forall\, i \in \{1,\ldots,n\}\end{array}}{\begin{array}{c}\Gamma,\Sigma \vdash_{\mathtt{UP}} f\ (e_1,\ldots,e_n) :: (\tau',\ (Q(s'_{p1},\ldots,s'_{pn},s') \wedge \Phi')) \\ \text{where}\quad \begin{array}{rcl}((\overline{\tau'_{p1}}^{\,s'_{p1}},\ldots,\overline{\tau'_{pn}}^{\,s'_{pn}}) \rightarrow \overline{\tau'}^{\,s'}) & = & subs\ D\ ((\tau_{p1},\ldots,\tau_{pn}) \rightarrow \tau_{p(n+1)}) \\ D & = & \bigcup_{i=1}^{n}\{eqP\ \overline{\tau_{ai}}^{\,s_{ai}}\ \tau_{pi}\} \\ \Phi' & = & \wedge_{i=1}^{n}\{s'_{pi} =_t s_{ai} \wedge \Phi_i\}\end{array}\end{array}} \quad [\text{App}]$$

$$\frac{\begin{array}{c}\Gamma,\ \Sigma \vdash_{\mathtt{UP}} e_0 :: (\tau_0,\Phi_0) \\ \Gamma_i,\ [\,] \vdash_{\mathtt{UP}} p_i :: (\tau_0,\ \Phi_{p_i}) \quad \forall\, i \in \{1,\ldots,n\} \\ \Gamma \cup \Gamma_i,\ \Sigma \vdash_{\mathtt{UP}} e_i :: (\tau_i,\Phi_i) \quad \forall\, i \in \{1,\ldots,n\}\end{array}}{\begin{array}{c}\Gamma,\ \Sigma \vdash_{\mathtt{UP}} \mathbf{case}\ \ e_0\ \mathbf{of}\ (p_i \rightarrow e_i)_{i=1}^{n} :: (\tau',\ case\,\{(\Phi_0 \wedge \Phi_{p_i}) \rightarrow \Phi'_i\}_{i=1}^{n}) \\ \text{where}\quad \begin{array}{rcl}\Phi'_i & = & (eqC\ \tau'\ \tau_i) \wedge \Phi_i \\ \tau' & = & sizeR\,\tau_1\end{array}\end{array}} \quad [\text{Case}]$$

Figure 5: Type Rules – Part 2

$D = \{A = (Int, E), B = F\}$. Using this type substitution, we can now obtain an instantiated type using

$$subs\ D\ ((A^a, B^b) \rightarrow A^m)$$

This gives a renamed sized type

$$((Int^p, E^q), F^b) \rightarrow (Int^r, E^s)$$

and a size constraint $a =_t (p, q) \wedge m =_t (r, s)$. Together with the original size constraint $m = a$, we can now obtain the size constraint $p = r \wedge q = s$, showing how the two components of the polymorphic type have flowed from input to output.

Polymorphic typing introduces equality constraints between polymorphic size variables and the instantiated size structures. Their resolution can be handled by term unification which allows the size relation to be transformed to Presburger formulae, where possible.

5. COLLECTION ANALYSIS

Collection refers to a bag of elements that occur inside a recursive data structure[1], such as a list or an array. Properties of collections are crucial in a number of scenarios. For example, indexes are stored as elements of lists inside sparse matrices. The ability to propagate size properties for collections is essential for safety checking should the indexes be retrieved for array accesses. Collection analysis helps us determine the kind of size properties that can be propagated.

Consider a list of size-annotated type $[t^a]^n$, where the elements have underlying type $\overline{t}$. The size variable n denotes the length of the list, while a captures the size property of elements of type $\overline{t}$. To represent the collection of elements of type $\overline{t}$, we use notation $a.bag$ (or $a\$$ in short). This bag notation is superior to alternative notation based on instance relation. For example, given $[1, 2, 3] :: [Int^a]^n$ we can either capture the size relation of elements using either bag relation $a\$ = \{1, 2, 3\}$ or instance relation $(a = 1 \vee a = 2 \vee a = 3)$. While the instance relation only specifies the possible values of the elements, the bag relation allows us to capture both *possible* as well as *definite* elements of each bag. In general, for any given relation of the form $minbag \sqsubseteq a\$ \sqsubseteq maxbag$, $minbag$ denotes the definite elements of $a\$$, while $maxbag$ captures the possible elements. For information to be as precise as possible, $minbag$ should be as large as possible, while $maxbag$ should be kept as small as possible.

As another example, consider the expression:

$$(\mathbf{case}\ e\ \mathbf{of}\ True \rightarrow [1, 2]; False \rightarrow [2, 3]) :: [Int^a]^n$$

The instance relation, $(a = 1 \vee a = 2 \vee a = 3)$ helps to capture the possible size values of the above. However, bag relation $a\$ = \{1, 2\} \vee a\$ = \{2, 3\}$ is more informative with pos-

[1]a data structure with repeated elements

sible and definite values captured. Possible values are useful for universal properties, while definite values are related to existential properties. In particular, the above bag relation can be simplified to $\{2\} \sqsubseteq a\$ \sqsubseteq \{1,2,3\}$, from which the following size properties are derivable,

$$(\exists a\$.a = 2) \wedge (\forall a\$.1 \leq a \leq 3).$$

5.1 Examples

To highlight the expressive power of collection analysis, let us consider two examples below.

$$\begin{array}{l} part(a, xs) \ = \textbf{case}\ xs\ \textbf{of} \\ \qquad\qquad Nil \quad\quad \rightarrow (Nil, Nil) \\ \qquad\qquad (x : xs') \ \rightarrow \textbf{let}\ (c, d) = part(a, xs')\ in \\ \qquad\qquad\qquad\qquad \textbf{case}\ (x < a)\ \textbf{of} \\ \qquad\qquad\qquad\qquad True \ \rightarrow (x : c, d) \\ \qquad\qquad\qquad\qquad False \ \rightarrow (c, x : d) \\ exist(a, xs) \ = \textbf{case}\ xs\ \textbf{of} \\ \qquad\qquad Nil \quad\quad \rightarrow False \\ \qquad\qquad (x : xs') \ \rightarrow \textbf{case}\ (a == x)\ \textbf{of} \\ \qquad\qquad\qquad\qquad True \ \rightarrow True \\ \qquad\qquad\qquad\qquad False \ \rightarrow exist(a, xs') \end{array}$$

The function *part* is the partition operation where an input list, xs is being split into two portions based on whether each element is smaller-than or larger-or-equal-to a given pivot element, a. The sized type for this function is as follows:

$$\begin{array}{l} part \ :: \ (Int^a, [Int^x]^m) \ \rightarrow \ ([Int^y]^n, [Int^z]^p) \\ \ \textbf{size} \ \ m = n + p \ \wedge \ x\$ = y\$ \sqcup z\$ \\ \qquad \wedge \ (\forall\, y\$.\ y < a) \ \wedge \ (\forall\, z\$.\ a \ \leq \ z) \end{array}$$

If we look at the collection formula, $x\$ = y\$ \sqcup z\$$, we can see that the output bags of $y\$$ and $z\$$ are obtained from the input bag, $x\$$. This relation can be used to propagate universal size properties of $x\$$ to the output bags, $y\$$ and $z\$$. Furthermore, our sized type system is able to deduce two universal properties, $(\forall\, y\$.\ y < a)$ and $(\forall\, z\$.\ a \leq z)$, over the output collections of $y\$$ and $z\$$ via size information propagated by the case selector.

The function *exist* is used to determine if a value a exists in the collection of list xs. Its output is a boolean result which indicates the existence of the given value. In this case, the property obtained is directly related to boolean output, r, as shown by the following sized type:

$$\begin{array}{l} exist \ :: \ (Int^a, [Int^b]^m) \ \rightarrow \ Bool^r \\ \ \textbf{size} \ \ (r = 0 \ \wedge \ (\forall\, b\$.b \ \neq \ a)) \\ \qquad \vee \ (r = 1 \ \wedge \ (\exists\, b\$.b = a)) \end{array}$$

If $r = 0$, we can deduce that all elements of input bag $b\$$ is not equal to a. If $r = 1$, we can deduce that at least one element of $b\$$ is equal to a. Again, these uniform properties are propagated by the case selector with the help of fix-point analysis, that is described next.

5.2 Fix-Point Computation

Sized type inference is decidable and precise in the absence of recursion. The presence of recursive function typically requires least fix-point to be computed for its sized type. Due to infinite lattice domain, such fix-point computation are often approximated in a conservative way using both hulling and widening operations. Fix-point approximation typically yields an upper-bound to the least fix-point. Our procedure for *fix-point analysis* involves the following main steps.

1. Build a constraint abstraction for the recursive function. This can be achieved via the sized type rules, gathering the constraint relations of input/output size variables.

2. Extract both the base and recursive cases from the constraint abstraction.

3. Obtain the first approximation of fix-point (from base case).

4. Derive the next approximation with the help of the recursive step, combined with the previous approximation. Apply hulling and widening, as appropriate.

5. Repeat Step 4, until a fix-point is detected.

The algorithm starts with an approximate fix-point based on the base case. It then repeatedly applies the recursive step until a fix-point is detected. To illustrate this fix-point computation, let us use the following running example.

$$\begin{array}{l} repl(xs, c) \ = \textbf{case}\ xs\ \textbf{of}\ Nil \quad\quad \rightarrow Nil \\ \qquad\qquad\qquad\qquad\qquad\qquad (x : xs') \ \rightarrow c : repl(xs', c + 1) \end{array}$$

Note that an output list is generated from the accumulative parameter (named c). We expect the following sized type from the above example:

$$\begin{array}{l} repl \ :: \ ([Int^a]^n, Int^c) \ \rightarrow \ [Int^b]^m \\ \ \textbf{size} \ \ m = n \ \wedge \ (\forall\, b\$.c \ \leq \ b \ \leq \ c + n) \\ \ \textbf{inv} \ \ (c < c^+ \leq c + n) \ \wedge \ (0 \leq n^+ < n) \ \wedge \ (a^+\$ \ \sqsubseteq \ a\$) \end{array}$$

There are two size relations, as can be seen from the above sized type. The first size information is meant to relate the size variables of both the input and the output. This is achieved via the *bottom-up fix point method* which begins with the first approximation using the base case. The second size information is meant to describe an invariant relationship between the parameters of an arbitrary recursive call with those of the initial recursive function call. This invariant relation must be obtained via a *top-down fix-point method* which does not make use of the base case in its computation. It begins with a one-step relation, and gradually refining until fix-point for an arbitrary recursive call is obtained.

Let us provide a detailed description of the bottom-up method first. This method starts off with the base case as its first approximation, namely:

$$U_0(I, O) \ = \ Base(I, O)$$

where I, O denotes the size variables of input and output sets respectively. After that, we obtain the next approximation with the help of recursive step, as follows:

$$\begin{array}{l} U_{i+1}(I, O) \ = (\exists\, I', O'.\ Rec(I, O, I', O') \ \wedge \ U_i(I', O')) \\ \qquad\qquad\qquad\qquad \vee \ U_i(I, O) \end{array}$$

where $Rec(I, O, I', O')$ describes a one-step recursive relation between the set of input/output size variables. This step is repeated until a fix-point is detected. In order for the fix-point computation to converge, both hulling and widening operations may be required. We stop when the following fix-point subsumption condition holds, where $\Rightarrow$ denotes logical implication.

$$\exists\, I', O'.\ Rec(I, O, I', O') \ \wedge \ U_i(I', O') \ \Rightarrow \ U_i(I, O)$$

In the case of our running example, the constraint abstraction that can be derived is:

$$Q_{repl}((a,n,c),(b,m)) \Leftarrow case\,\{ \\ a\$ = \{\} \wedge n = 0 \rightarrow b\$ = \{\} \wedge m = 0 \\ a\$ = \{a'\} \sqcup a''\$ \wedge n = n'+1 \rightarrow \\ b\$ = \{c\} \sqcup b''\$ \wedge m = m'+1 \\ \wedge\ Q_{repl}((a'',n',c+1),(b'',m'))\,\}$$

From here, we can derive the following base and recursive relations:

$$repl0((a,n,c),(b,m)) \\ \Leftarrow a\$ = \{\} \wedge n = 0 \wedge b\$ = \{\} \wedge m = 0 \\ replr((a,n,c),(b,m),(a'',n',c'),(b'',m')) \\ \Leftarrow \exists\, a'.a\$ = \{a'\} \sqcup a''\$ \wedge n = n'+1 \\ \wedge\ b\$ = \{c\} \sqcup b''\$ \wedge m = m'+1 \wedge c' = c+1$$

The first approximation is the base case relation itself. The second approximation to the fix-point is obtained as follows:

$$repl0((a,n,c),(b,m)) \\ \Leftarrow a\$ = \{\} \wedge n = 0 \wedge b\$ = \{\} \wedge m = 0 \\ repl1((a,n,c),(b,m)) \\ \Leftarrow (\exists\, a'',n',c',b'',m'. \\ \quad replr((a,n,c),(b,m),(a'',n',c'),(b'',m')) \wedge \\ \quad repl0((a'',n',c'),(b'',m'))) \vee repl0((a,n,c),(b,m)) \\ \Leftarrow \text{(hulling)} \\ \quad n = m \wedge 0{\leq}n{\leq}1 \wedge \forall\, b\$\,.\, b = c$$

After that, a second approximation is obtained by the following steps which involves both a hulling and a widening operation:

$$repl2((a,n,c),(b,m)) \\ \Leftarrow (\exists\, a'',n',c',b'',m'. \\ \quad replr((a,n,c),(b,m),(a'',n',c'),(b'',m')) \wedge \\ \quad repl1((a'',n',c'),(b'',m'))) \vee repl1((a,n,c),(b,m)) \\ \Leftarrow \text{(hulling)} \\ \quad n = m \wedge 0{\leq}n{\leq}2 \wedge \forall\, b\$\,.\, c{\leq}b < c+n \\ \Leftarrow \text{(widening)} \\ \quad n = m \wedge 0{\leq}n \wedge \forall\, b\$\,.\, c{\leq}b < c+n$$

The hulling operation merges the disjunction together, while the widening operation omits an increasing constraint $n \leq 2$. As a result of this step, we have now reached a fix-point which could be checked via the fix-point subsumption condition.

The top-down fix-point method is used to capture sized relationship between the parameters of an arbitrary recursive call with those of the original recursive call. In this step, our first approximation is a one-step recursive relation which relates the parameter of one recursive call with the next call:

$$U_0(I,I') \;=\; RecI(I,I')$$

After that, we obtain the next approximation, as follows:

$$U_{i+1}(I,I') \;=\; (\exists\, A.\, RecI(I,A) \wedge U_i(A,I')) \vee U_i(I,I')$$

This step is repeated until fix-point is detected. We stop when the following fix-point subsumption condition holds:

$$\exists\, A.\, RecI(I,A) \wedge U_i(A,I') \;\Rightarrow\; U_i(I,I')$$

With the *repl* example, the one step relation is obtained by omitting the output of the recurrence, as follows:

$$replr((a,n,c),(a'',n',c')) \\ \Leftarrow \exists\, a'.\, a\$ = \{a'\} \sqcup a''\$ \wedge n = n'+1$$

Following the top-down fix-point procedure, we can obtain the following approximated fix-point for the *repl* function.

$$replr'((a,n,c),(a',n',c')) \\ \Leftarrow (c < c' \leq c+n) \wedge (0 \leq n' < n) \\ \wedge\ (a'\$ \sqsubseteq a\$)$$

6. DISCUSSION

We shall now look at two additional techniques that can make our collection analysis more accurate.

6.1 Bijective Relation

Often, two or more collections may be related to one another in a one-to-one (and onto) bijective relation. A classic example is the *map* function which produces an output for every element of its input collection. Under this scenario, we introduce a special notation $\forall\, a\${\leftrightarrow}r\$.\, P[a,r]$ where $a\$, r\$$ are two collections that are related in a bijective relationship. Such a relationship is more expressive as it implies both $(\forall\, a\$ \exists\, r\$.\, P[a,r])$ and $(\forall\, r\$ \exists\, a\$.\, P[a,r])$. An example of such bijective relation is shown below:

$$addone(xs) \;=\; \textbf{case}\; xs\; \textbf{of}\; Nil \rightarrow Nil \\ (x:xs') \rightarrow (x+1) : addone(xs') \\ addone \;::\; ([Int^a]^m) \rightarrow [Int^r]^n \\ \textbf{size}\;\; m = n \wedge (\forall\, a\${\leftrightarrow}r\$\,.\, r = a+1)$$

where the input collection $a\$$ is in a bijective relation with the output bag $r\$$.

Note that for such bijective relation to hold, a strictly 1-to-1 and onto mapping between elements of two or more collections is essential. As a counter-example, consider:

$$addone'(xs) \;=\; \textbf{case}\; xs\; \textbf{of}\; [x] \rightarrow Nil \\ (x:xs') \rightarrow (x+1) : addone'(xs')$$

While this function is also a (injective) mapping, it is not based on a bijective mapping since there are more elements in the input collection than the output collection. Due to this, we can only obtain the following size relation.

$$addone' \;::\; ([Int^a]^m) \rightarrow [Int^r]^n \\ \textbf{size}\;\; m = n+1 \wedge m > 0 \wedge (\forall\, r\$ \exists\, a\$\,.\, r = a+1)$$

By extending the uniform property rules with two more rules in Figure 6, our type system can automatically generate such bijective relations in its sized type.

6.2 Locally Instantiated Collection

The elements of our collection can be locally instantiated inside quantified relation. Such local instantiation of collection can help us capture more precise relation between collections and their elements. This is made possible by the polymorphic handling of size structures.

For example, consider the *allsplit* function which returns all possible ways of splitting an input list into two sublists.

$$allsplit(xs) \;=\; \textbf{case}\; xs\; \textbf{of} \\ Nil \rightarrow (Nil, Nil) \\ (x:xs') \rightarrow \textbf{let}\; zs = allsplit(xs')\textbf{in} \\ (Nil, x:xs') : addelem(a, zs) \\ allsplit \;::\; [\alpha^a]^m \rightarrow [([\alpha],[\alpha])^d]^n$$

$$\frac{\wedge_{i=1}^{n}\{v_i\$ = \{v_i\}\} \wedge \Phi(v_1,\ldots,v_n,v')}{\forall (v_1\$ \leftrightarrow \cdots \leftrightarrow v_n\$)\,.\,\Phi(v_1,\ldots,v_n,v')} \quad [\forall\text{-Bi}]$$

$$\frac{\begin{array}{c}(v_1\$ = v_1'\$ \sqcup v_1''\$) \wedge \cdots \wedge (v_n\$ = v_n'\$ \sqcup v_n''\$) \wedge \\ \forall v_1'\$ \leftrightarrow \cdots \leftrightarrow v_n'\$\,.\,\Phi'(v_1',\ldots,v_n') \wedge \\ \forall v_1''\$ \leftrightarrow \cdots \leftrightarrow v_n''\$\,.\,\Phi''(v_1''\ldots,v_n'')\end{array}}{\forall v_1\$ \leftrightarrow \cdots \leftrightarrow v_n\$\,.\,\Phi'(v_1,\ldots,v_n) \vee \Phi''(v_1,\ldots,v_n)} \quad [\text{UP-5}]$$

Figure 6: Uniform Property Rules for Bijective Relation

$$\begin{array}{l}\textbf{size}\ \ m+1=n \wedge \\ \quad (\forall\, d\$.\, \exists\, b, n1, c, n2.\ d = ([b]n1, [c]n2) \\ \quad\ \wedge\ n1+n2 = m\ \wedge\ a\$ = b\$ \sqcup c\$)\end{array}$$

With locally instantiated collection for $d\$$, our type system is able to relate inner collections, $b\$$ and $c\$$, to the outer collection $a\$$ via $a\$ = b\$ \sqcup c\$$. Furthermore, the locally instantiated size variables $n2$, $n1$ are also related to the outer size variable m by $n2 + n1 = m$. This is so because the collection $a\$$ has been splitted via different ways into each element pair of the collection, $d\$$. Without locally instantiated collection, sized formulation would be much less precise.

For another example, consider the *addelem* function which is an auxiliary function for *allsplit*. This function takes a list of pairs and adds an element a to the first component of each pair.

$$\begin{array}{l} addelem(a, xs) = \textbf{case}\ xs\ \textbf{of} \\ \qquad\qquad Nil \rightarrow Nil \\ \qquad\qquad ((x,y):xs') \rightarrow (a:x,y) : addelem(a, xs') \\ addelem\ ::\ (\alpha^a, [([\alpha],\beta)^d]^m) \rightarrow [([\alpha],\beta)^g]^n \\ \ \textbf{size}\ \ m = n \wedge (\forall\, d\$ \leftrightarrow g\$.\, \exists\, b, n1, c, e, n2, f. \\ \qquad d = ([b]n1, c) \wedge g = ([e]n2, f) \wedge \\ \qquad c =_t f \wedge e\$ = b\$ \sqcup \{a\} \wedge n2 = n1+1) \end{array}$$

Our type system is able to detect that each pair of input from $d\$$ is bijectively related to another pair of elements from collection $g\$$. Such a bijective relation is more precisely captured when the elements of collections $d\$$ and $g\$$ suitably instantiated to size structures $([b]n1, c)$ and $([e]n2, f)$ respectively. Furthermore, this universal property is able to state that $c =_t f$ and that each $e\$$ is made up of $b\$$ merged with singleton $\{a\}$. As illustrated here, collection relation that are nested (with suitable local instantiation) can allow more precise size analysis to be captured.

7. RELATED WORK

One of the earliest work which captures size information through a type-checking mechanism was proposed by Hughes, Pareto and Sabry [11]. Instead of type-checking, we have proposed a systematic inference method for calculating the size relationships of functions. Furthermore, by assigning a logical constraint to each expression, we are able to express size information more accurately in at least two ways: Firstly, their method determines only *one* bound of the size, fixing the other bound at either 0 (for data types) or ω (for co-data types). On the contrary, we are able to infer tighter bounds by determining both the lower and upper bounds. Secondly, their method still falls in the domain of independent attribute analysis, as opposed to relational analysis. On the other hand, we have not yet considered the handling of lazy languages, and thus unable to prove certain correctness aspects of reactive systems (for example, productivity).

Size information can also be captured using *dependent types*. This technique is advocated by Xi and Pfenning [19, 20]. Their type-checking framework is parameterized with constraint domains, of which size information can be described by a domain of natural numbers with linear equality and inequality constraints. While this has the advantage that analysis efficiency can be controlled by the sophistication of the constraint domain, it requires *insightful type annotations* from programmers.

Research into the inference of linear constraints dates back to 1970's, with the seminal work of Cousot and Halbwachs on imperative languages [3]. Halbwachs has applied linear constraints to synchronization analysis of reactive systems [8, 9]. There, a constraint is described by a set of linear inequalities, and an effective widening operation is employed to obtain fixed point of polyhedra. However, separate size information for nested data structures is not captured.

The work by Le Metayer [13] aims to verify uniform properties among elements in a data structures using abstract interpretation technique. These properties are expected to be provided by programmers. His method was able to demonstrate the verification of orderedness of elements in the output of a sorting program. Properties between input and output can also be verified, but this can be done only if the input re-appears at the output.

In another dimension, we may also compare our sized analysis with other constraint-based type analyses, such as those for binding-time analysis [4], usage analysis[18], and flow analysis[5]. While the handling of type polymorphism is not so straightforward, the methods used in [18, 6, 12] are essentially the same with the need to instantiate type variables; and the need to maintain suitable relationships between existing and the newly instantiated annotations. To achieve lower time-complexity, a number of researchers have recently proposed clever ways to avoid the duplication of constraints when a polymorphic type is instantiated. The concept of *instantiation constraint* was used in [5], while [7] provided *constraint abstractions*.

In this paper, we have shown how a mixed constraint system could be utilized on an enhanced type system where collections can be handled in a systematic way. Scalable constraint-solving techniques for this new class of analyses remains a future research topic.

8. CONCLUSION

We have presented an enhanced sized typing, its formal rules and associated techniques to support systematic inference. Our type system make use of a mixed constraint system, with *arithmetic constraints* for size properties, *bag constraints* to model flow of components, and *term equality constraints* for polymorphism. The different constraint systems work together to allow more accurate size information to be propagated.

Our preliminary implementation of sized type inference (for a first-order language) via the Omega Calculator has gone very well. Based on our new proposal, we have systematically found the sized types for a wide range of examples, with reasonable performance. We are currently extending the system to support collection analysis through mixed constraint solving techniques.

9. ACKNOWLEDGEMENTS

Special thanks to Razvan Musaloiu, Shengchao Qin and anonymous referees of PEPM for their helpful comments on an earlier version of this paper. This work has been supported by Singapore-MIT Alliance and research grants R-252-000-092-112 and R-252-000-138-112

10. REFERENCES

[1] Wei-Ngan Chin and Siau-Cheng Khoo. Calculating sized types. *Higher-Order and Symbolic Computation*, 14(2-3):261–300, 2001.

[2] W.N. Chin and S.C. Khoo. Calculating sized types. In *2000 ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 62–72, Boston, Massachusetts, United States, January 2000.

[3] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Symposium on Principles of Programming Languages*, pages 84–96. ACM Press, 1978.

[4] D Dussart, F Henglein, and C Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In *Static Analysis Symposium*, (LNCS, vol 983, pp. 118-135) Springer-Verlag, 1995.

[5] M Fahndrich and J Rehof. Type-based flow analysis: From polymorphic subtyping to cfl reachability. In *ACM Conference on Principles of Programming Languages*, 2001.

[6] Kevin Glynn, Peter J. Stuckey, Martin Sulzmann, and Harald Sondergaard. Boolean constraints for binding-time analysis. In *Programs as Data Objects (PADO II)*, pages 39–62, Aarhus, Denmark, May 2001.

[7] J Gustavsson and J Svenningsson. Constraint abstractions. In *Programs as Data Objects (PADO II)*, pages 63–83, Aarhus, Denmark, May 2001.

[8] N. Halbwachs. About synchronous programming and abstract interpretation. *Science of Computer Programming, Special Issue on SAS'94*, 31(1), May 1998.

[9] N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.

[10] L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebras: Part I.* North-Holland, 1971.

[11] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *23rd ACM Principles of Programming Languages Conference*, pages 410–423. ACM Press, January 1996.

[12] N Kobayashi. Type-based useless variable elimination. In *2000 PEPM*, pages 84–93, 2000.

[13] Daniel Le Metayer. Proving properties of programs defined over recursive data structure. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–99, La Jolla, California, June 1995. ACM Press.

[14] S. Nishimura. Parametric polymorphic type inference in constraint form. In *Second JSSST Workshop on Programming and Programming Languages*, pages 38—49, March 2000.

[15] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

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

[17] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *4th ACM Conference on Functional programming Languages and Computer Architecture*, pages 1–11, California, June 1995. ACM Press.

[18] K Wansbrough and S. L. Peyton Jones. Once upon a polymorphic type. In *ACM Conference on Principles of Programming Languages*, pages 15–28, 1999.

[19] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *ACM Conference on Programming Language Design and Implementation*, pages 249–257. ACM Press, June 1998.

[20] H. Xi and F. Pfenning. Dependent types in practical programming. In *Symposium on Principles of Programming Languages*, pages 214–227. ACM Press, January 1999.