# Calculating Sized Types

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

**Abstract**

Many program optimisations and analyses, such as array-bound checking, termination analysis, etc, 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 *size relation* between different inputs of a function. Secondly, different optimisations and analyses may require slightly different size information, and thus slightly different computation. Literature in size computation has mainly concentrated on size checking, instead of inferencing. In this paper, we provide a generic framework on which different size variants can be expressed and computed. We also describe an effective algorithm for *inferring*, instead of checking, size information. Size information are expressed in terms of Presburger formulae, and our algorithm utilises the Omega Calculator to compute as exact a size information as possible, within the linear arithmetic capability.

## 1 Introduction

Many program optimisations and analyses, such as array-bound checking, termination analysis, etc, 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 *size relation* between different inputs of a function. Thus, its computation cannot be easily expressed using the usual "independent attribute analysis", such as traditional abstract interpretation based on collecting semantics.

Secondly, different optimisations and analyses may require slightly different size information, and thus slightly different computation. For instance, consider the following *append* function:

$$\begin{array}{lll} append\ xs\ ys & = \textbf{case}\ xs\ \textbf{of} & \\ & \quad Nil & \rightarrow ys \\ & \quad (x : xs') & \rightarrow x\ :\ (append\ xs'\ ys) \end{array}$$

There are several kinds of size information that are of interest to us. First and foremost, we are interested in knowing how the output of *append* relates to its input, since this is fundamental to computing "inter-procedural" size information. Let $m$, $n$ be the size (*ie.*, length) of its parameters $xs$, $ys$, respectively, and $p$ be its output size, then we have $m + n = p$. Next, we are also interested in finding how the arguments of recursive calls to *append* are changed with respect to the initial *append* call. In particular, in the successive recursive calls to *append*, the first argument to the recursive call of *append* is monotonically decreasing, while the second remains constant in size. Such information is useful for analysing the termination of a program, as well as termination of transformation process, such as partial evaluation (eg., [8, 7]).

Literature on size computation has mainly concentrated on size checking, instead of inferencing (see [9], [17]). Furthermore, size information computed are usually represented in a form that facilitates a particular optimisation or analysis, but less useful to others. In this paper, we aim to *infer* as accurate size information as possible to various optimisations and analyses. In the case of *append*, we express its size (and its type) as follows:

$$\begin{array}{l} append\ ::\ [t]^m\ \rightarrow\ [t]^n\ \rightarrow\ [t]^p \\ \quad \text{s.t. } \textbf{size}\ \ m\ \geq\ 0\ \wedge\ n\ \geq\ 0\ \wedge\ p\ =\ m\ +\ n \\ \qquad\quad \textbf{inv}\ \ 0\ \leq\ m^+\ <\ m\ \wedge\ n^+\ =\ n \end{array}$$

Here, we capture two pieces of size information. The first, prefixed by the keyword **size**, relates the size of a function's input and output. The second, prefixed by **inv**, applies only to recursive functions. Here, we let $m$ and $n$ be the argument sizes of an (initial) call to *append*, and $m^+$ and $n^+$ be the argument sizes of an arbitrary recursive (inner) call to *append*. The constraint describes the call invariant for all nested recursive calls (*i.e.*, monotonic decrease of $m^+$ and constancy of $n^+$), with respect to the initial call.

While the amount of size information may appear overwhelming, it is worth noting that the computation of these kinds of information share some commonality; ie., a major portion of the computation can be reused. This prompts us to find a representation that acts as a base for representing size information, on which various desired information can be computed.

We express size information using Presburger formulae. This enables us to capture linear relations among various size variables. There are a couple of practical reasons for limiting size information to Presburger formulae: Firstly, there exists efficient tools for performing test over these formulae. Secondly, Presburger formulae can be used to capture the size information for a wide range of practical applications. Here, we employ the the Omega Calculator [13] in our size computation. The entire computation is divided into two steps: The first step computes the relation between input/output size of a non-recursive function, as well as the size change during one unfolding of a recursive function. In the second step, we compute the desired fixed point $s$ in size for recursive functions.

Corresponding to the use of fixed-point theory in computing the size of a recursive function, we interpret the underlying Presburger formulae as automorphic relations, and adapt the *transitive-closure operation* [13] to compute a suitable size. Consequently, we propose an effective algorithm for *inferring*, instead of checking, the required size information of any function.

Functions used in this paper for demonstration are listed in Appendix A. The outline of the paper is as follows: In the next section, we motivate the use of size information in various applications. Then, in Section 3, we provide brief background for the work. Section 4 describes the inference algorithm declaratively using a type system. This is illustrated through an example in Section 5. In Section 6, we describe how fixed point computation is performed via the transitive-closure operation. Section 7 describes recursive-call-invariant computation. Section 8 discusses a strategy for obtaining a constraint that approximates fixed-point computation. Section 9 discusses on related work. Section 10 concludes the paper by listing our contribution.

## 2 Why Sized Types?

Sized types have a wide variety of uses. They can be used to prove some size-related properties, such as termination. They can also be used to support program optimisation. Using sized types, unnecesary checks, dead-codes and repeated calls can be eliminated, where appropriate.

To highlight the versatility of sized types, consider a set of commonly used functions in Appendix A. As we shall see later, our new size inference method is able to calculate appropriate sized types for these functions. In this section, we motivate the need for these sized types by briefly explaining how they may be used.

### 2.1 Termination Property

Sized types can be used to confirm the *termination property* of recursive functions. (Please refer to Section 9 for more related work.) A recursive function terminates if there exists a set of arguments that can be shown to decrease across recursion. Such proofs using well-founded orderings (or quasi-orderings) are useful for safety-critical systems [9] and also to ensure that the partial evaluation process terminates [7]. To illustrate this, consider the Ackermann function. The recursive-call invariant about its two parameters indicates a well-founded lexicographic ordering. This confirms that its recursive calls will eventually terminate, even though the second parameter has an increasing value.

### 2.2 Vector-Based Memoisation

Sized types may also be used to determine the bounds of integer parameters. Such bounds may be required to decide on a suitably sized vector that could help avoid the recomputation of repeated calls, through a technique known as dynamic-sized memoisation [3]. A simple example is the binomial function with exponential time complexity. If the redundant calls of binomial function were eliminated, the complexity becomes quadratic. How can this be done? The recursive-call invariant for binomial function states that parameter $k^+$ is bounded below by $max(0,\ n^+ + k - n)$, and above by $min(n^+, k)$. Using this bound, the dynamic-sized memoisation technique of [3] could derive the following less readable but much efficient vector-based definition of binomial (using Haskell-like notation).

$$
\begin{array}{l}
bin\ \ (n,k)\ \ = \\
\textbf{let}\ \ bin'\ (n^+)\ = \\
\qquad \textbf{case}\ \ n^+\ \textbf{of} \\
\qquad 0\ \rightarrow\ (\lambda\, k^+\ \rightarrow\ 1) \\
\qquad (m+1)\ \rightarrow \\
\qquad \textbf{let}\ \ z\ =\ bin'(m)\,; \\
\qquad\qquad bn\ k^+\ =\ \textbf{if}\ (k^+\ \leq\ 0\ \vee\ k^+\ \geq\ n^+) \\
\qquad\qquad\qquad\qquad \textbf{then}\ \ 1 \\
\qquad\qquad\qquad\qquad \textbf{else}\ \ z(k^+ - 1)\ +\ z(k^+)\,; \\
\qquad\qquad l\ =\ max(0, n^+ - (n-k))\,; \\
\qquad\qquad u\ =\ min\,(n^+, k)\ \ ; \\
\qquad\qquad vec\ =\ array\,(l,u)\,[a\ :=\ bn\ a\ \mid\ a\ \leftarrow\ [l..u]] \\
\qquad \textbf{in}\ \ (!)\ vec \\
\textbf{in}\ \ \textbf{if}\,(0\ \leq\ k\ \wedge\ max(1,k)\ \leq\ n) \\
\qquad \textbf{then}\ \ bin'(n)\,(k)\,\textbf{else}\ \ 1
\end{array}
$$

### 2.3 Bound-Check Elimination

Run-time bound checking on arrays is crucial for ensuring robust programs but can be expensive. More sophisticated compilers may offer an option to eliminate bound checks. However, they do not fare well, due to their inability to synthesize loop invariants and perform inter-procedural analysis. Sized type can offer a superior solution to this problem.

Consider the binary search function *bsearch*. The array access function *sub* requires a bound check to ensure that the index is within bounds. Here, the size context of the initial call to *look* is $(l_0\ =\ 0\ \wedge\ h_0\ =\ i\ -\ 1)$, where $l_0$, $h_0$ are the size of the two arguments of *look*, and $i$ is the size of the array. Combining this context, the recursive-call invariant for *look*, and the computation for local variable $m$, we can determine that $0\ \leq\ m\ \leq\ h_0$ for all nested invocation of *look*. This result is sufficient to allow the replacement of *sub* by an optimised version (without bound-checking).

An approach to allow bound checks to be safely eliminated was proposed by Xi and Pfenning in [17]. In their type-checking framework, it is the responsibility of the user to specify where bound checks can be eliminated, and where they cannot. Automatic inference of sized types presents a new challenge. Apart from sized-type inference, the decision on whether bound checks can be safely eliminated should also be (automatically) decided through program specialisation. A brief description of how to achieve this feat is presented in Appendix B.

## 3 Background

The term *size* can have various meanings. In [18], users can specify different meaning of sizes through a size annotation on the data declaration. For ease of presentation, we fix the meaning of size to be the following :

1. The size of a recursive data is its maximum depth (*ie.*, the number of nesting). Thus, the size of a list is its length, and the size of a binary tree is its depth. Also,

the size of an array[1] is simply its size. The size of an integer is itself.

2. For any non-recursive enumerated data, the size is used to distinguish one such data from another. Hence, we choose an enumeration as its size. For example, for a boolean value, we choose 0 as the size of *False*, and 1 for *True*.

3. The size of a function is a relation between its input and output sizes.

We apply our technique to higher-order strict functional languages with self-recursive function definitions. Our technique requires manipulation of constraints, which are called *integer tuple relations.* An *integer tuple relation* is a relation whose domain consists of integer $k$-tuples and whose range consists of integer $k'$-tuples, for some fixed $k$ and $k'$. An example of a relation from 2-tuples to 1-tuples is $[n, m] \rightarrow [p] : m \geq 0 \wedge n \geq 0 \wedge p = m + n$. As seen previously, this describes the relationship between the input and output of *append* function.

A *Presburger formula* (or a *term*) is used to express the relationship among the variables. Fig 1 depicts its syntax. Note that arithmetic expressions are restricted to affine functions of the variables. For example, $2 * x$ is legal, but $x * x$ is not. Semantically, a Presburger formula $\phi$ defines a set of integer tuples, each element of which satisfies the boolean relation, $\mathcal{B}\,[\![\, \phi \,]\!]$. This definition is no different from that of predicate calculus.

$$
\begin{array}{lll}
\Phi \in \mathbf{C} & & \text{— Constraints} \\
\quad ::= \Lambda_S\, v \,.\, \Phi \mid \phi & & \\
\phi \in \mathbf{F} & & \text{— Presburger Formulae (terms)} \\
\quad ::= b \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\, \phi \mid \exists\, v : \phi & & \\
b \in \mathbf{BExp} & & \text{— Boolean Expression} \\
\quad ::= \mathit{True} \mid \mathit{False} \mid a_1 = a_2 \mid a_1 \neq a_2 & & \\
\qquad \mid a_1 < a_2 \mid a_1 > a_2 \mid a_1 \leq a_2 \mid a_1 \geq a_2 & & \\
a \in \mathbf{AExp} & & \text{— Arithmetic Expression} \\
\quad ::= c \mid v \mid c * a \mid a_1 + a_2 \mid -\, a & &
\end{array}
$$

*where* $c \in \mathcal{Z}$ is an integer (a constant)
$v \in \mathbf{V}$ is a variable

Figure 1: Syntax of Constraints

A constraint as defined in Fig 1 can be a formula, $\phi$, or a *parameterised* formula of the form $\Lambda_S v_1 \ldots v_k \,.\, \phi$. $\Lambda_S$ provides a binding mechanism that binds variables $v_1, \ldots, v_k$ occurring in $\phi$. A parameterised formula of $k$ parameters can be interpreted as either a set of $k$-tuples of integers, or a relation between two tuples of independent and dependent variables. So, $\Lambda_S\, v_1 \ldots v_k.\, \phi$ has several interpretations, such as $[v_1] \rightarrow [v_2, \ldots, v_k] : \phi$ (a relation), or $[v_1, \ldots, v_k] : \phi$ (a set), *etc.*. Nevertheless, given a set of integers $\{n_1, \ldots, n_k\}$, we say that a tuple $(n_1, \ldots, n_k)$ *satisfies* $\Lambda_S\, v_1 \ldots v_k \,.\, \phi$ (or simply $\phi$) if $(n_1, \ldots, n_k) \in \mathcal{B}\,[\![\, \phi \,]\!]$.

Lastly, we use the terminology formula, term, and constraint interchangeably, when it is clear from the context.

[1] We treat one-dimensional arrays with 0-based indexing in this paper. It is easy to handle arrays with explicit lower and upper bounds. Moreover, multi-dimensional arrays can also be handled using tuples of size information. For simplicity, this paper ignores these extensions.

Also, we use the notion $[v_1, \ldots, v_k] : \phi$ (and its variants) and $\Lambda_S v_1 \ldots v_k : \phi$ interchangeably.

## 4 Type System

In this paper, we describe a size computation for a subset of functional programs. The programs we deal with contain only *self-recursive functions* for the sake of presentation clarity.

We assume that the input program to our sized-type system is well-typed (under the Milner-Damas type system). Therefore, the type of each expression in the program is available during size computation.

We capture two pieces of size information: the *size relation* between a function's input and output; and the *recursive-call invariant* about the input to *nested* recursive calls. Although there is no need to capture the output size of a function in this invariant, the output size of an auxiliary function is still required for interprocedural analysis. Therefore, we compute the size relation between a function's input and output, before computing recursive-call invariant.

A type system is introduced to specify the size relation of every expression in a program. This relation relates the size of (possible values bound to) free variables occurring in an expression to the expression's output size. Specifically, in the type system, we associate each expression with an environment (a type assumption) and a sized type.

A *sized type* is a pair containing an *annotated type* and a *constraint.* The precise set of annotated type expressions is inductively generated over infinite sets of constructor $D$, type variable $t \in \mathit{TVar}$ and size variable $v \in \mathbf{V}$:

$$
\begin{array}{lll}
\sigma & ::= \forall\, t \,.\, \sigma \mid \tau & \text{— Type Scheme} \\
\tau & ::= t \mid \tau \rightarrow \tau & \text{— Type} \\
 & \quad \mid D^{\mathbf{V}}\, \tau_1 \ldots \tau_k \mid D^{\top}\, \tau_1 \ldots \tau_k &
\end{array}
$$

Size variables are attached to data constructors, such as $[\mathit{Int}^n]^l$, where $l$ refers to the length of a list, and $n$ refers to the possible integer values of list elements. $[\mathit{Int}]$ is referred to as the *underlying type* of this list, and is described by underlining the annotated type; *ie.*, $\underline{[\mathit{Int}^n]^l} = [\mathit{Int}]$. Size variables are used in formulating the size constraint of an expression.

$\top$ denotes the absence of size information. Thus, $D^{\top}$ denotes the absence of size information for the constructor $D$. Moreover, $\top$ is also used when the size information is too "complicated" to compute, such as expressing multiplication of two size variables. When $\top$ appears at the top-level of a (Disjunctive Normal Form) formula, it obeys the following rules: $\top \wedge \psi = \psi$, $\top \vee \psi = \top$, and $\neg(\top) = \top$.

Given two predicates, $\psi_1$ and $\psi_2$, we say that $\psi_2$ *approximates* $\psi_1$, denoted by $\psi_1 \sqsubseteq \psi_2$, if either $\psi_2 = \top$, or $\psi_1 \Rightarrow \psi_2$ (in logical sense). The latter condition implies that every tuple of values satisfying predicate $\psi_1$ also satisfies $\psi_2$. This notion of approximation defines the subtyping relation for our type system.

Figures 2 and 3 lists the inference rules for inferring the sized type of an expression.

A type may have an inherent size constraint associated with it. For instance, if the (annotated) type is of the form $[t]^i$, then it is true that $i \geq 0$, as the length of a list must be non-negative. Next, for $\mathit{Bool}^j$, we have $(j = 0 \vee j = 1)$. The *inherited constraint* associated with size variable $i$ is denoted by $\mathcal{I}(i)$ (we ignore the type associated with $i$ to reduce the clustering of symbols). Extending this notation to

$$\frac{}{\Gamma \cup \{x :: (\sigma, \phi)\} \vdash x :: (\sigma, \phi)} \quad (\textsc{Var}) \qquad \frac{}{\vdash n :: (\texttt{Int}^i, \phi)} \quad (\textsc{Con} : \texttt{Int})$$
$$\text{where } \phi = (i = n);\ \mathit{new}\ i$$

$$\frac{\Gamma \cup \{x :: (\tau_1, \phi_1)\} \vdash e :: (\tau_2, \phi_2)}{\Gamma \vdash (\lambda\, x.\, e) :: (\tau_1 \rightarrow \tau_2, \phi_2')} \quad (\textsc{Abs})$$
$$\text{where } \phi_1 = \mathcal{I}(\iota(\tau_1))\,;\quad \phi_2' = \mathcal{I}(X) \wedge \exists\, Y\,.\, \widehat{\phi_{2\Gamma}}\,;$$
$$Y = (\iota(\phi_2) - (\iota(\mathcal{C}_\Gamma) \cup X))\,;\quad X = \iota(\tau_1) \cup \iota(\tau_2)$$

$$\frac{\Gamma \vdash e_1 :: (\tau_1 \rightarrow \tau, \phi_1) \qquad \Gamma \vdash e_2 :: (\tau_1', \phi_2) \qquad \mathcal{CT}(\Gamma, (\tau_1, \phi_1), (\tau_1', \phi_2))}{\Gamma \vdash (e_1\ e_2) :: (\tau, \phi)} \quad (\textsc{App})$$
$$\text{where } \phi = (\exists\, ls\,.\, \phi_1 \wedge (eqC\ \tau_1'\ \tau_1)) \wedge \phi_2\,;\quad ls = \iota\,(\tau_1)$$

Figure 2: Rules for Checking Sized-Type — Part I

a set of size variables $\mathcal{X}$, we denote by $\mathcal{I}(\mathcal{X})$ the conjunction of all inherited constraints associated with variables in $\mathcal{X}$.

A *type assumption* $\Gamma$ binds program variables to their sized types. We use $\mathcal{C}_\Gamma$ to denote the collection of all constraints occurring in $\Gamma$. Given an assumption $\Gamma$ and a constraint $\phi$, the judgment $\Gamma \vdash \phi$ indicates that $\phi$ is deducible under the assumption $\Gamma$.

Furthermore, given $\Gamma \vdash \phi$, we write $\widehat{\phi_\Gamma}$ to denote conjunction of $\phi$ and the context enclosing $\phi$. For instance, let $\mathcal{C}_\Gamma = \{a :: a \geq b,\ b :: b \geq 0,\ c :: c \geq 0\}$ and $\phi = d < a + 5$, then $\widehat{\phi_\Gamma} = d < a + 5 \wedge a \geq b \wedge b \geq 0$.

Some operations are used in the type rules to manipulate size variables. They are:

- *Size-variable renaming* ($\alpha$): This is an overloaded operation serving various purposes:
  1. Applying $\alpha$ to an annotated type (or type scheme), $\tau$ (or $\sigma$), returns a new annotated type with the same underlying type, but new free size variables.
  2. Applying $\alpha$ to a sized type $(\sigma, \phi)$ results in consistent renaming of size variables occurring in $\sigma$. Thus, $\alpha\,([Int^i]^j,\ j \geq 1 \wedge i > 1) = ([Int^k]^l,\ l \geq 1 \wedge k > 1)$, for some new variables $k$ and $l$.
  3. Applying $\alpha$ to two annotated types $\tau_1$ and $\tau_2$, denoted by $\tau_1 \stackrel{\alpha}{\equiv} \tau_2$, checks if $\tau_1$ and $\tau_2$ have the same underlying type. That is, they are the same *modulo size-variable renaming.*
- *Size-variable listing* ($\iota$): This takes in either an annotated type $\sigma$ or a constraint $\Phi$, and returns the set of free size variables occurring in $\sigma$ or $\Phi$. Thus, $\iota([Int^i]^j) = \{i, j\}$.
- *Size-variable matching* ($eqC$): Given two annotated types, $\sigma_1$ and $\sigma_2$, such that (1) $\iota(\sigma_1) \cap \iota(\sigma_2) = \emptyset$, and (2) $\sigma_1 \stackrel{\alpha}{\equiv} \sigma_2$, $eqC$ produces a conjunction of equalities that identify the corresponding free size variables between $\sigma_1$ and $\sigma_2$. For instance, $eqC\ [Int^i]^j\ [Int^k]^l = (i = k) \wedge (j = l)$.

The rule for variables is obvious. In the rule for abstraction, we assume that the parameter is associated with its inherent constraint, and infer the sized type of the abstraction body.

In the rule for application, the term $\mathcal{CT}(\Gamma, (\tau_1, \phi_1), (\tau_1', \phi_2))$ is a *consistency test.* This ensures that the actual argument applied satisfies the sized type stipulated by the function's formal. Its implementation is detailed in Sec 5. Successful function application is modeled by instantiating (through existential quantification) the size variables associated with the function's domain.

For **let**-expressions, we compute the sized types of its locals, and use them in computing the sized type of the body. Finally, the result is enriched with a constraint pertaining to the locals, and the size variables associated to the locals are existentially quantified.

To infer the sized type of a recursive function, we assume that the function is constrained by the inherited constraints of its parameters and results, and compute the constraint associated with a *single* unfolding of the recursive call; the result is denoted by $\phi'$. We then perform a fixed-point operation ($\mathcal{TC}(\phi')$) to determine the constraint $\Phi$ for the recursive function. The detailed computation procedure will be provided in Section 6. Briefly, this involves performing a key step using a *transitive closure operation.* Besides fixed-point computation, inferring a **letrec**-expression is trivial.

For **case**-expressions, we compute the sized type of its test, which is then used to deduce the sized type of each branch. We require that patterns be matched against test result are mutually exclusive and exhaustive, and employ a set of inference rules called $\vdash^{\text{Pat}}$ to compute the sized types of each pattern. The rules for pattern matching boolean constants are as follows:

$$\frac{\mathit{new}\ x}{\Gamma \vdash^{\text{Pat}} (True, Bool^i) :: \{\, x :: (Bool^i, i = 1)\,\}}$$

$$(\textsc{Con} : \texttt{PAT} : \texttt{Bool} - \texttt{True})$$

$$\frac{\mathit{new}\ x}{\Gamma \vdash^{\text{Pat}} (False, Bool^i) :: \{\, x :: (Bool^i, i = 0)\,\}}$$

$$(\textsc{Con} : \texttt{PAT} : \texttt{Bool} - \texttt{False})$$

We refer the reader to [4] for the inference rules for other constructs, such as list type.

For each branch, we collect the inferred sized type and the constraints obtained from pattern matching at that branch (ie., $\mathcal{C}_{\Gamma_i}$ for branch $i$). The size constraint for **case** expression is then the disjunction of the size constraints at all branches, with local size variables existentially quantified.

$$\frac{\begin{array}{c}\Gamma \vdash M :: (\sigma, \Phi) \\ \Gamma \cup \Gamma_1 \vdash N :: (\tau, \phi)\end{array}}{\Gamma \vdash (\textbf{let}\ x = M\ \textbf{in}\ N) :: (\tau, \phi')} \quad (\textsc{Let})$$

$$\text{where} \quad \phi' = \exists X . \widehat{\phi_{\Gamma_1}}\,; \quad X = \iota(\sigma) \cap \iota(\phi)\,; \quad \Gamma_1 = \{ x :: (\sigma, \Phi) \}$$

$$\frac{\begin{array}{c}\Gamma \cup \{x :: (\sigma, \phi) \} \vdash M :: (\sigma', \phi') \\ \Gamma \cup \{ x :: (\sigma, \Phi) \} \vdash N :: (\tau, \phi'')\end{array}}{\Gamma \vdash (\textbf{letrec}\ x = M\ \textbf{in}\ N) :: (\tau, \phi'')} \quad (\textsc{Rec})$$

$$\text{where} \quad \phi = \mathcal{I}(\iota(\sigma))\,; \quad \Phi = \mathcal{TC}(\phi') \qquad \sigma \overset{\alpha}{\equiv} \sigma'$$

$$\frac{\begin{array}{c}\Gamma \vdash M :: (\tau_1, \phi_0) \quad \{x :: (\tau_1, \phi_0)\} \vdash^{\text{Pat}} (P_i, \tau_1) :: \Gamma_i \quad new\ x \\ \Gamma \cup \Gamma_i \vdash M_i :: (\tau_i', \phi_i) \\ \tau_i' \overset{\alpha}{\equiv} \tau_j' \quad \forall\, i \neq j\end{array}}{\Gamma \vdash \textbf{case}\ M\ \textbf{of}\ (P_i \rightarrow M_i) :: (\tau'', \exists X . \phi)} \quad (\textsc{Case})$$

$$\begin{array}{ll}\text{where} & X = (\iota(\phi) - (\iota(\mathcal{C}_\Gamma) \cup \iota(\tau'')));\\ & \phi = \bigvee_i ((eqC\ \tau''\ \tau_i') \wedge \phi_i \wedge \mathcal{C}_{\Gamma_i}); \quad \tau'' = \alpha\, \tau_1'\end{array}$$

$$\frac{\Gamma \vdash M :: (\sigma, \phi) \qquad t \notin FV(\Gamma)}{\Gamma \vdash M :: (\forall\, t . \sigma, \phi)} \quad (\textsc{Gen}) \qquad \frac{\Gamma \vdash M : (\forall\, t . \sigma, \phi_1)}{\Gamma \vdash (M : \sigma[\tau/t], \phi_1 \wedge \phi_2)} \quad (\textsc{Inst})$$

$$\text{where } \iota(\tau) \equiv \iota(\phi_2)$$

$$\frac{\begin{array}{c}\Gamma \vdash e :: (\tau_1 \rightarrow \tau_2, \Phi) \\ X = \iota(\tau_1 \rightarrow \tau_2)\end{array}}{\Gamma \vdash e :: (\tau_1 \rightarrow \tau_2, \Lambda_S\, X\ .\ \Phi)} \quad (\textsc{Bind})$$

$$\frac{\Gamma \vdash e :: (\tau_1 \rightarrow \tau_2, \Lambda_S\, X\ .\ \phi)}{\Gamma \vdash e :: (\tau_1' \rightarrow \tau_2', \phi')} \quad (\textsc{Unbind})$$

$$\text{where } X = \iota(\tau_1 \rightarrow \tau_2)\,; \quad (\tau_1' \rightarrow \tau_2', \phi') = \alpha(\tau_1 \rightarrow \tau_2, \phi)$$

Figure 3: Rules for Checking Sized-Type — Part II

We have the usual rules for type generalisation (GEN) and instantiation (INST). Furthermore, (BIND) is applied to the constraint of $\lambda$-abstraction. It parameterises the constraint with the size variables associated with the annotated type of the lambda abstraction. On the other hand, (UNBIND) undoes the job of (BIND), and instantiates (with new size variables) a parameterized constraint into a Presburger formula for further manipulation. This rule is used when the associated lambda expression is applied.

Since both (BIND) and (UNBIND) are used in fixed situation during sized-type inference, our rules can therefore be made syntax-directed.

Soundness of the sized-type system is defined with respect to the standard (monomorphic) denotational definition, $\mathcal{E}$, of a strict language. Given a denotational value $d$ of an annotated type $\sigma$ with $\iota(\sigma) = \{v_1, \ldots, v_k\}$, we define a function $\mathcal{S}$ such that $\mathcal{S}(d, \sigma)$ returns a (possibly singleton) set of tuples, of which each tuple $(n_1, \ldots, n_k)$ denotes a size of $d$, and $v_i = n_i, \forall\, i \in \{1, \ldots, k\}$.

We say $d$ *satisfies* a constraint $\phi$ *under* $\sigma$ if $\iota(\phi) \subseteq \iota(\sigma)$ and $\mathcal{S}(d, \sigma) \subseteq \mathcal{B}[\![\, \phi \,]\!]$.

We extend this satisfiability definition to include the assumption set $\Gamma$. Given $\Gamma$, and a denotational environment $\rho$, we say $\rho$ *satisfies* $\Gamma$ if

> For any subset (assuming of size $k \geq 1$) of bindings $\rho_1 = \{x_i : d_i\}_{i=1}^k \subseteq \rho$, and its corresponding assumption set $\Gamma_1 = \{x_i : (\sigma_i, \phi_i)\}_{i=1}^k \subseteq \Gamma$, if $(\cup_i \{\iota(\phi_i)\}) \cap \iota(\mathcal{C}_{\Gamma - \Gamma_1}) = \emptyset$, then $\mathcal{S}((d_1, \ldots, d_k), (\sigma_1, \ldots, \sigma_k)) \subseteq \mathcal{B}[\![\, \bigwedge_i \phi_i \,]\!]$.

We then call $(\rho_1, \Gamma_1)$ a *support pair* for $d_i, \forall\, i \in \{1, \ldots, k\}$.

We also note that $\mathcal{S}((d_1, \ldots, d_k), (\sigma_1, \ldots, \sigma_k)) \subseteq \mathcal{B}[\![\, \bigwedge_i \phi_i \,]\!] = \mathcal{B}[\![\, \bigwedge_i \widehat{\phi_{i_\Gamma}} \,]\!]$.

**Theorem 1 (Soundness)** *Given an expression $M$ of type $\underline{\sigma_1}$, for some $\sigma_1$. If $\vdash M :: (\sigma, \phi)$, then $\underline{\sigma} = \underline{\sigma_1}$, and for any monomorphic instance of $M$ with denotation $d$, $d$ satisfies $\phi$ under $\sigma$.*

Due to space constraint, we omit all the proofs in this paper, and refer the reader to our technical paper [4] for detail.

## 5 Size Inference for Non-Recursive Functions – An Example

We show how type inference is done for the following non-recursive function:

$$\begin{array}{llll} f\ b\ xs & = & \textbf{case}\ b\ \textbf{of} & False \rightarrow xs \\ & & & True \rightarrow append\ xs\ xs \end{array}$$

The function $f$ defined above depends on *append*; we assume that the sized type of *append* is known.

The annotated type of $f$ is $Bool^i \rightarrow [a^\top]^j \rightarrow [a^\top]^k$, for some $i, j, k$. Consider the process of inferring the type of the rhs of its definition. The assumption at this point consists of:

$$\begin{array}{rl}\Gamma = \{ & b :: (Bool^i, i = 0 \vee i = 1), \quad xs :: ([a^\top]^j, j \geq 0), \\ & append :: ([t]^m \rightarrow [t]^n \rightarrow [t]^p, \\ & \qquad \Lambda_S\, m, n, p\ .(m \geq 0 \wedge n \geq 0 \\ & \qquad\qquad \wedge\, p = m + n)) \quad \}\end{array}$$

The type inference rule stipulates that we assign a new variable, $x$ say, to the case-test $b$. So, we update $\Gamma$ to include the binding $\{x :: (Bool^i, i = 0 \vee i = 1)\}$. Let's call this new assumption $\Gamma'$.

In the *False* branch, we construct an assumption, $\Gamma_1 = \{nv_1 :: (Bool^i,\ i = 0)\}$ from its pattern (*i.e.*, *False*). $nv_1$ is a new vairable name used to refer to this pattern. We then append $\Gamma_1$ to $\Gamma'$. The size constraint for this branch is now the constraint of $xs$: $\Gamma' \cup \Gamma_1 \vdash xs :: ([a^\top]^j, j \geq 0)$.

In the *True* branch, we append to $\Gamma'$ the assumption $\Gamma_2 = \{nv_2 :: (Bool^i,\ i = 1)\}$. Here, we encounter the application of *append*. First, we instantiate the sized type for *append* and unbind its size variables, as follows:

$$\begin{aligned} append :: (\ & [a^\top]^{m_1} \rightarrow [a^\top]^{n_1} \rightarrow [a^\top]^{p_1}, \\ & m_1 \geq 0 \wedge n_1 \geq 0 \wedge p_1 = m_1 + n_1) \end{aligned}$$

We begin by deducing the sized type for (*append xs*). Here, we need to perform consistency test for the application argument.

**Consistency Test** Let $(\sigma_1, \phi_1)$ be the sized type of *append*, and $(\sigma_2, \phi_2)$ be that of $xs$. The test $\mathcal{CT}\ (\Gamma, (\tau_1, \phi_1), (\tau_2, \phi_2))$ first renames $m_1$ in $\phi_1$ (and $\sigma_1$ too) and $j$ in $\phi_2$ ($\sigma_2$ too) to a new common variable, say $q$. In the test, we use an operator, $\_\uparrow^{-} :: \mathbf{F} \times \mathbf{V}^\star \rightarrow \mathbf{F}$ to perform projection of formulae over a set of variables. Formally, given a formula $\phi$ that is defined over a set of variables $\mathcal{X}$, let $\mathcal{Y}$ be a subset of $\mathcal{X}$, then $\phi\uparrow^{\mathcal{Y}} \stackrel{def}{=} \exists\, x_1 \cdots x_n : \phi$, where $\{x_1, \ldots, x_n\} = \mathcal{X} - \mathcal{Y}$. With the help of the Omega Calculator, the consistency test is done as follows:

$$\begin{array}{lll} 1. & C_2 & \stackrel{def}{=} (\widehat{\phi_{2_\Gamma}} \wedge \mathcal{I}(\{q\}))\uparrow^{\{q\}} \quad = \quad q \geq 0 \\ 2. & C_1 & \stackrel{def}{=} (\widehat{\phi_{1_\Gamma}} \wedge \mathcal{I}(\{q\}))\uparrow^{\{q\}} \\ & & = \exists\, n_1, p_1, : q \geq 0 \wedge n_1 \geq 0 \wedge p_1 = q + n_1 \\ & & = q \geq 0 \\ 3. & C_2 & \sqsubseteq \ C_1 \equiv True. \end{array}$$

**Generating Constraints for Function Application** The constraint for (*append xs*), after renaming the size variables of *append*, is

$$\begin{aligned} (\exists\, m_1 \ :\ & (m_1 \geq 0 \wedge n_1 \geq 0 \wedge p_1 = m_1 + n_1) \wedge m_1 = j) \\ & \wedge j \geq 0 \end{aligned}$$

We next deduce the constraint for ((*append xs*) *xs*). After passing the consistency test, the constraint is:

$$\begin{aligned} (\exists\, n_1 \ :\ & (\exists\, m_1 \ :\ (m_1 \geq 0 \wedge n_1 \geq 0 \wedge p_1 = m_1 + n_1) \\ & \wedge m_1 = j) \wedge j \geq 0 \wedge n_1 = j) \wedge j \geq 0 \end{aligned} \qquad (\psi_2)$$

**Generating Constraints from Conditionals** Now, combining both *True* and *False* branches, we introduce a common new size variable $s$ for both branches and obtain the following constraint for the **case** expression:

$$\begin{aligned} \exists\, p_1 \ :\ & (s = j \wedge j \geq 0 \wedge i = 0) \\ & \vee (s = p_1 \wedge \psi_2 \wedge i = 1) \end{aligned} \qquad (\psi_3)$$

Lastly, we introduce into $\phi_3$ the inherited constraints of the parameters, and bind those local size variables existentially, before passing the constraint to the Omega Calculator for simplification. The final constraint for $f$ is:

$$\begin{aligned} \Lambda_S\ i, j, s\ .\ & (i = 0 \wedge j \geq 0 \wedge s = j) \\ & \vee (i = 1 \wedge j \geq 0 \wedge s = j + j) \end{aligned}$$

**Remark** Note that the type-inference process can be greatly simplified if we are only interested in the size of the lists, and not their elements. In the example above, we assign size variable to the list constructor, but $\top$ to the its substructure – the list elements. This effectively eliminates sized-type inference for list elements. During type inference, we eliminate $\top$'s from the formula by identifying them with the boolean value *True*. The simplified formula will be fed to the Omega Calculator for computation. Therefore, the cost of conducting type inference can be controlled by the "substructure depth" that is to be analysed.

## 6 Inferring the Size of Recursive Functions

In the type rule for (REC), we infer the sized type of a recursive function by computing a *fixed-point* for the sized type of the function body. We denote this inference by the operation $\mathcal{TC}$. An essential step in our fixed-point computation is the *transitive-closure operation*, as proposed by Kelly, Pugh, et al. in their work on the Omega Calculator [13]. In principle, the transitive-closure operation allows us to compute the least fixed-point (called the transitive closure) for the size constraint. In practice, however, such operation may fall short in producing the necessary transitive closure. We thus devise a generalisation technique (as described in Section 8) to compute an upper bound for the transitive closure. The result is a fixed-point, but not necessarily the least, of the constraint.

It is shown in (REC) that the fixed-point operation $\mathcal{TC}$ is applied on $\phi'$, the sized type for a single unfolding of the recursive call. $\phi'$ is derived under the assumption that we have no prior information about the recursive function, except the constraint inherited from its input/output type (which is denoted by $\phi$). $\phi'$ may consist of two pieces of information:

1. *Recursive-call constraint*[2]: A relation between the input/output sizes of a recursive call and its immediate caller, and

2. *Termination constraint* : Constraint about the termination condition of the recursive call.

As an example, in the definition of *append* in Sec. 1, let the annotated type of *append* be $[t]^m \rightarrow [t]^n \rightarrow [t]^p$, and the annotated type of the recursive *append call* (in the rhs of the definition of *append*) be $[t]^{i_1} \rightarrow [t]^{i_2} \rightarrow [t]^{i_3}$. The recursive-call constraint can be computed to be:

$$\begin{aligned} U\ & :: [m, n, p] \rightarrow [i_1, i_2, i_3] \\ & = \exists\, i : (m \geq 0) \wedge (i \geq 0 \wedge i + 1 = m) \\ & \quad \wedge (n \geq 0) \wedge (p \geq 0) \wedge (i_1 \geq 0 \wedge i_1 = i) \\ & \quad \wedge (i_2 \geq 0 \wedge i_2 = n) \wedge (i_3 \geq 0 \wedge i_3 + 1 = p) \end{aligned}$$

whereas the termination constraint is computed as follows:

$$\begin{aligned} B_I\ & :: [m, n, p] \\ & = (m = 0) \wedge (n \geq 0) \wedge (p = n) \end{aligned}$$

The fixed-point computation $\mathcal{TC}$ comprises two steps:

[2] For easy presentation, we assume that the function under inference is linearly self-recursive. In the situation where there are multiple recursive calls in a function body, we infer the constraint for each recursive call separately, by replacing the other recursive calls with some existentially quantified size variables (to denote the result of their applications). These individually inferred constraints are then combined, through disjunction, to form the overall recursive-call constraint.

1. Calculating a generalised transitive constraint for the recursive-call constraint.
2. Incorporating a terminating constraint into the generalised transitive constraint.

We describe these two steps in detail next, using the *append* function as illustration.

### 6.1 Calculating A Generalised Transitive Constraint

A *generalised transitive constraint*, *gtc* for short, describes a relation that holds between a recursive caller and an arbitrarily nested recursive call. Given a recursive-call constraint $U$, a natural candidate for its *gtc* is the *transitive closure* of $U$, denoted by $U^+$. Formally, $U^+ = \bigcup_{k=1}^{\infty} U^k$, where $U^k$ is the composition of $k$ $U$'s. It is a constraint that holds for all recursive calls reachable from the initial call by unfolding, and is characterised by the following important property ([11]):

**Property 1 (Transitive Closure)** *For all relations $P$ and $R$ such that $R \subseteq P \subseteq R^+$ the following holds: $P = R^+$ if and only if $P \circ R \subseteq P$.*

Semantically, $U^+$ is an invariant about size, capturing the relationship between the size of an initial call and that of an arbitrary recursive call. The transitive-closure operation is intimately related to least fixed point computation; both are built on the idea of unfolding recursive calls an arbitrary number of times.

The Omega Calculator provides a *transitive-closure operation*, (we call it `closure` to avoid clashing of terminology) for calculating the transitive closure of a relation. For the case of *append*, Omega calculation yields the following:

$$\begin{aligned}&\texttt{closure}(U) :: [m,n,p] \rightarrow [m_1,n_1,p_1]\\ &\quad = (0 \leq m_1 < m) \wedge (m \leq p + m_1) \wedge (n \geq 0)\\ &\qquad \wedge (n_1 = n) \wedge p_1 = p - m + m_1\end{aligned}$$

Using Property 1, we can verify that $\texttt{closure}(U) = U^+$.

Techniques for calculating transitive closure have been described in [11]. In practice, there are three possible results from such calculation using the Omega Calculator:

1. The computation terminates and yields a Presburger formula, just like the computation of $\texttt{closure}(U)$ above. In this case, we check if the formula is indeed the transitive closure. The check is done using Property 1.
2. The computation terminates and yields a constraint that contains symbol `UNKNOWN`. Here, we treat these `UNKNOWN` as our special size value $\top$, and computes an upper bound of the constraint. (The operation `upper_bound` in the Omega Calculator does just this.) We then verify if the resulting constraint is indeed the transitive closure.
3. The computation aborts because the Omega Calculator is unable to determine the transitive closure.

In the first two cases, when $\texttt{upper\_bound(closure(}\ U\texttt{))}$[3] fails to produce $U^+$, the result is a lower bound of $U^+$ (see Sec. 6.4 of [10]). We propose to overcome this deficiency through a generalisation strategy, which attempts to compute an upper bound to the transitive closure as our desired transitive constraint. Failing that, we choose $\top$ to be the size of the recursive function.

In the last case, when the computation has been aborted, we reduce the complexity of the constraint $U$ by eliminating the constraints for some input/output sizes. Elimination of these constraints is obtained by existentially quantifying the related size variables.

This technique for computing *gtc* is described in greater detail in Section 8.

As will be proven in Section 8, a generalised transitive constraint for a recursive-call constraint $U$ is *an upper-bound of the transitive closure* of $U$. Such a constraint will be denoted by $U^{\oplus}$.

[3] We note here that when $\texttt{closure(}U\texttt{)}$ returns a formula containing *no* `UNKNOWN` symbol, we have $\texttt{upper\_bound(closure(}U\texttt{))} = \texttt{closure(}U\texttt{)}$. Thus, we can always subject the result of a `closure` operation to an `upper_bound` operation.

### 6.2 Incorporating Terminating Constraint

To completely capture the size constraint about a function, we need to include a termination constraint into our final constraint. There are two scenarios for consideration:

1. When arguments of the initial function call match a degenerate case; *i.e.*, *no* recursive call is ever invoked. This situation is already captured by the terminating constraint. We denote it by $B_I :: [a_1, \ldots, a_n]$, where $a_i$ for $i \in \{1, \ldots, n\}$ are the size variables for the initial call.
2. When arguments of a nested recursive call match a degenerate case, and thus terminate the recursive invocation. The constraint is expressed as follows:

$$\begin{aligned}&B_R :: [a_1, \ldots, a_n]\\ &\quad = \exists\, b_1, \ldots, b_n \,:\, (U^{\oplus})(a_1, \ldots, a_n, b_1, \ldots, b_n)\\ &\qquad\qquad\qquad\qquad \wedge B_I(b_1, \ldots, b_n)\end{aligned}$$

By combining both scenarios (through disjunction), we are able to calculate the sized type for recursive functions quite accurately. For the case of *append*, we have:

$$\begin{aligned}&B :: [m,n,p]\\ &\quad = B_I \vee B_R \quad \text{where}\\ &\qquad B_I \;:: [m,n,p]\\ &\qquad\quad = (m = 0) \wedge (n \geq 0) \wedge (p = n)\\ &\qquad B_R \;:: [m,n,p]\\ &\qquad\quad = \exists\, m_1,\, n_1,\, p_1 \,:\, U^{\oplus}(m,n,p,m_1,n_1,p_1)\\ &\qquad\qquad\qquad\qquad \wedge B_I(m_1,n_1,p_1)\\ &\quad = ((m = 0) \wedge (n \geq 0) \wedge (p = n))\\ &\qquad \vee ((m \geq 1) \wedge (n \geq 0) \wedge (p = m + n))\end{aligned}$$

The result is already the sized type of *append*. But, it can be further simplified by computing its convex hull[4]: $\mathit{Hull}\ B = [m,n,p] : m \geq 0 \wedge n \geq 0 \wedge p = m + n$.

We now state soundness of our size analysis for the case of recursive function definition.

**Theorem 2 (Soundness – Recursive Case)** *Given a recursive function definition $x = M$ with type $\underline{\sigma_1}$, for some $\sigma_1$. If we infer that $M$ has sized type $(\sigma, \phi)$, then $\underline{\sigma_1} \equiv \underline{\sigma}$, and for any monomorphic instance of $M$ with denotation $d$, $d$ satisfies $\phi$ under $\sigma$.*

[4] We can use ($\mathit{Hull}\ B$) as the size relation for *append*, without compromising the accuracy of the result, since it can automatically be checked that ($\mathit{Hull}\ B) \equiv B$).

## 7 Recursive-call Invariants

In the previous section, we show that the proposed type system can be used to infer (1) the sized type of non-recursive functions; and (2) both the recursive-call constraints and termination constraints of recursive functions. In this section, we show that this information is adequate for the computation of call invariants of recursive functions.

Computing recursive-call invariant is very similar to computing size relation: it can be inferred by taking the transitive closure of the initial recursive-call constraint. In computing size relation, we take into consideration the output size, whereas in computing recursive-call invariants, we require only the input relation. Even so, size relation of auxiliary calls may still be needed. Hence, it is natural that we compute size relation before recursive-call invariant.

The actual computation is similar to the computation for *gtc* in 6.1. Since we do not need to determine the size relation of final output and initial input here, there is no need to incorporate a terminating constraint in our computation, and the output size relation is ignored.

As an example, we compute an recursive-call invariant for *bin* function, defined in Appendix A. We let $(n, k)$ denote the initial-call arguments, and $(n_1, k_1)$ represent the arguments of the recursive calls. The recursive-call constraint of is as follows:

$$\begin{aligned} U_{bin} &:: [n,k] \rightarrow [n_1,k_1] \\ &= (n > 0 \wedge k > 0 \wedge k < n) \wedge && \text{– The context} \\ &\quad ((n_1 = n - 1 \wedge k_1 = k - 1) \vee && \text{– argument change} \\ &\quad\ (n_1 = n - 1 \wedge k_1 = k)). \end{aligned}$$

Next, we compute the generalised transitive constraint of $U$:

$$\begin{aligned} U^{\oplus}_{bin} &:: [n,k] \rightarrow [n^+,k^+] \\ &= (k^+,\ 1 \leq k < n) \wedge (k^+,\ 1 \leq n^+ < n) \wedge \\ &\quad (k + n^+ \leq n + k^+) \wedge 0 \leq k^+ \end{aligned}$$

The above result can be more succinctly expressed as follows:

$$\begin{aligned} & max(1, k^+) \leq n^+ < min(n + k^+ - k, n) \\ & \wedge max(k + n^+ - n, 0) \leq k^+ \leq min(n^+, k) \end{aligned}$$

In [3], Chin and Hagiya also managed to obtain the above constraint, but relied on several advanced techniques. In contrast, the Omega Calculator offers tremendous assistance in deriving the invariant. This separation of concern for specification and constraint solving enables the analyser (either the programmer or the compiler) to concentrate on formulating the constraint, without being unduly concerned with the operational detail of constraint solving.

## 8 Computing Generalised Transitive Constraints

In this section, we elaborate on the process for obtaining an upper bound to the transitive closure of our recursive-call constraint.

As noted in [11], transitive closure computation becomes complicated in the presence of multiple disjuncts. To attain successful transitive closure, we choose a combination of the following tactics:

1. Reduce the number of disjuncts by combining related disjuncts into one, where appropriate.
2. Reduce the number of dependencies between size variables.

Figure 4 provides an overview of the strategy for computing generalised transitive constraint.

**Reducing the number of disjuncts** A well-known method for combining multiple disjuncts is the *convex hull* computation. While this method can sometimes be computationally expensive, and its result too general, we can reduce both its complexity and generality by an intelligent (and automatic) selection of suitable disjuncts for combination. We illustrate this with a practical example.

As an example, consider the Ackermann function defined in Appendix A. The following recursive-call constraint $U$ describes the relationship between the two inputs of *ack* (for three recursive calls):

$$\begin{aligned} U &:: [i,j] \rightarrow [i_1,j_1] \\ &= \exists\, k, k1 : ((i > 0 \wedge j = 0 \wedge i_1 + 1 = i \\ &\qquad\qquad \wedge j_1 = 1 \wedge k1 = k) \vee \\ &\qquad\qquad (i > 0 \wedge j > 0 \wedge i_1 = i \wedge j_1 + 1 = j \\ &\qquad\qquad \wedge (\exists\, u : k1 = u)) \vee \\ &\qquad\qquad (i > 0 \wedge j > 0 \wedge (\exists\, u : j_1 = u) \\ &\qquad\qquad \wedge i_1 + 1 = i \wedge k1 = k)) \end{aligned}$$

The Omega Calculator fails to compute its transitive closure, but instead yields the following lower bound constraint:

$$\begin{aligned} \mathtt{closure}(U) &:: [i,j] \rightarrow [i_1,j_1] \\ &= (0 \leq j_1 < j \wedge 1 \leq i \wedge i_1 = i) \\ &\quad \vee (0 \leq i_1 < i \wedge 1 \leq j) \\ &\quad \vee (0 \leq i_1 < i \wedge 0 \leq j \wedge 1 = j_1) \\ &\quad \vee (0 \leq i_1 \leq i - 2 \wedge 0 = j)\,; \end{aligned}$$

Our heuristic template attempts to identify conjuncts that are "similar" in form for merging. Two conjuncts are similar if each variable in each conjunts does not depend on different variables. The following template shows the similarity of the conjuncts of `closure(`$U$`)` stated above:

| | Conjunct 1 | | Conjuncts 2, 3, and 4 | |
|---|---|---|---|---|
| | LB | UB | LB | UB |
| $i$ | $i_1$ | $i_1$ | $i_1$ | - |
| $j$ | $j_1$ | - | - | - |
| $i_1$ | $i$ | $i$ | - | $i$ |
| $j_1$ | - | $j$ | - | - |
| Legend: | LB – Lower bound | | | |
| | UB – Upper bound | | | |

As the last three conjuncts of `closure(`$U$`)` are found to be identical in form, we combine them through a convex-hull operation. Together with the first conjunct, we obtain the following more general constraint $U'$ below:

$$\begin{aligned} U' &:: [i,j] \rightarrow [i_1,j_1] \\ &= (0 \leq j_1 < j \wedge 1 \leq i \wedge i_1 = i) \\ &\quad \vee (0 \leq i_1 < i \wedge 0 \leq j) \end{aligned}$$

With the Omega Calculator, we can show that `closure` $(U') = U'^{+}$. Theorem 3 shows that $U'$ is an upper bound to the transitive closure of $U$, and therefore $U^{\oplus} = U'$.

It is interesting to see that this invariant constraint can be used to prove the termination of Ackermann function. This is in contrast with the method proposed by Hughes [9], which requires rewriting the definition of Ackermann function.

Given a constraint $U$:

| | |
|---|---|
| Step 1. | $R \leftarrow$ `upper_bound(closure(`$U$`))`. |
| Step 2. | If $R$ is a formula, then |
| Step 2.a. | If $R == U^+$, then $U^\oplus \leftarrow R$, halt. |
| Step 2.b. | If the number of disjuncts in $R$ can be reduced, then |
| Step 2.b.1. | $U \leftarrow$ apply comvex-hull operation heuristically on $R$, Goto Step 1. |
| Step 2.b.2. | Else, $U^\oplus \leftarrow \top$, halt. |
| Step 2.c. | Else, if the number of dependencies in $U$ can be reduced, then |
| Step 2.c.1. | $U \leftarrow$ reduce dependency of $U$, Goto Step 1. |
| Step 2.c.2. | Else $U^\oplus \leftarrow \top$, halt. |

Figure 4: Strategy for Computing Generalised Transitive Constraints

**Reducing the number of dependencies** The second tactic we adopt is to reduce the number of dependencies between size variables by *ignoring some of these size variables.* Instead of computing the constraint for these size variables, we *quantify them existentially*. A common technique we use is to first existentially quantify the size variables associated with output of a function, followed by existentially quantifying the size variables associated with one of the input.

As an example, consider the function *look* defined locally in *bsearch* (Appendix A). Usual calculation of recursive-call invariant yields $\top$, due to its inability to relate the two inputs. However, by reducing the number of dependency between the two inputs, we are able to obtain its recursive-call invariant, as shown in Appendix A.

**Theorem 3 (Upper Bound to Transitive Closure)** *$U^\oplus$ is an upper bound $U^+$.*

In summary, this technique has worked quite well in practice, and enabled us to generate useful upper bound to transitive closures.

**Termination** Our method always terminate, because (1) calls to the Omega Calculator always terminates, and (2) at each iteration of calls to the Omega Calculator, our constraint gets "reduced" either in the number of size variables under computation or in the total number of disjunctive clauses.

## 9 Related Work

Our work has been inspired by the work on sized types by Hughes, Pareto and Sabry in [9]. However, instead of performing type-checking (as they did), we have devised a fixed-point method for safely inferring the sizes of the recursive 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 $\omega$ (for co-data types). On the contrary, we are able to infer tighter bounds by determining both the lower and upper bounds (for instance, the recursive-call invariant for *bin*). Secondly, their method still falls into the domain of independent attribute analysis. It is thus difficult for their system to express direct relationship between two inputs (as in the case of lexicographic ordering for the two inputs of Ackermann function). On the other hand, we have not yet developed inference technique for handling 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 in [17, 18]. Their type-checking framework is parameterised 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. Also, their system does not compute recursive-call invariants for recursive functions, which is very useful for determining local constraints within functions that holds throughout all nested calls. As a result, most recursive functions used by Xi and Pfenning have to be rewritten as local functions. This sacrifices code reuse.

Research into the inference of linear constraints has dated back to 1970's, with the seminal work of Cousot and Halbwachs on imperative languages [5]. It has also received much attention in logic programming community, particularly in the inference of argument sizes for the purpose of termination analysis of logic programs [6]. One such approach is pioneered by Ullman and Van Gelder [16], which has spurred further development in [12, 2, 1, 14, 15]. In this approach, size information is represented via a restricted form of predicate inequalities (usually without disjunction of constraints). For example, constraints of the form $\Sigma_{i \in I}\, p_i + \gamma_p \geq \Sigma_{j \in J}\, p_j$ was used in the Mercury system [15], where $p_k$ stands for the size of argument $k$ of predicate $p$, $I$ is the subset of the set of the input argument positions of predicate $p$, and $J$ is a subset of the set of its output argument positions. The aim is to solve for an integer value of $\gamma_p$, which is the value that will serve as an indicator of the maximum possible growth of arguments. Due to the nature of the termination decision, these proposals for capturing size information are more restrictive and less modular than sized typing: Firstly, only one bound is computed instead of two. Secondly, size relationship between inputs are ignored. Lastly, separate size information for nested data strutures is not captured.

## 10 Conclusion

We have presented an enhanced sized typing, its formal rules and associated techniques to support automatic inference.

Our main contributions are:

- We have designed an *enhanced sized typing* that can cater to various optimisations and analyses. It is powerful as it supports interprocedural analysis (via size constraints) and captures loop invariants across recursion (via invariant constraint).
- We have formulated an *inference method* for sized typing. To the best of our knowledge, this has not been achieved before. Previous works were limited to a type-checking framework that requires insightful type annotations from programmers.
- A key result of this paper is a new *fixed-point method* to help infer suitable sized constraints for recursive functions. We base this on the transitive-closure operation of [13], augmented with *selective generalisations* to achieve impressive results. (Sec. 8)
- We described the soundness of our our sized type system.
- We confirmed the *feasibility* of our inference method with the help of the linear arithmetic constraint-solving capability of the Omega Calculator (examples are shown in [4]). Our approach directly benefits from the superior performance of the Omega Calculator. As it is relatively easy to replace the constraint solver, our approach is also well-hedged for future improvement in constraint-solving technology.

Our preliminary evaluation of sized type inference via the Omega Calculator has gone very well. Based on our new inference method, we have systematically found the sized types for a range of examples including those used in this paper. Experiments are being conducted currently to further improve the strategy for computing generalised tansitive constraints, so that it can more accurately handle non-linear recursive definitions.

On the theoretical front, it would be of interest to work on an extension to sized typing that could better support lazy languages. Under lazy semantics, the size of partial data structures (that contains $\perp$) should ideally be distinguished from that for $\perp$. The extent to which this could be achieved will help determine both the accuracy and the utility of sized typing in the lazy setting. One such approach is to use data and co-data types to distinguish finite from infinite data structures, as used in [9], but the onus is on the user to make the distinction.

## 11 Acknowledgment

We would like to thank Neil Jones for his critical review of this piece of work. We also thank Hongwei Xi, Masami Hagiya, Luke Ong, Zhenjiang Hu, Tyng Ruey, and the anonymous referees for their insightful comments and valuable suggestions to an earlier draft of this paper. Last, but not least, we thank Xu Na for her help in constructing a prototype of the type system. This work was supported by the research grant RP3972716.

## References

[1] A. Bossi, N. Cocco, and M. Fabris. Typed norms. In *European Symposium on Programming '92*, LNCS, vol 582, pp. 73–92, 1992.

[2] A. Brodsky and Y. Sagiv. Inference of inequality constraints in logic programs. In *Proceedings of the 10th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 227–240, 1991.

[3] W.N. Chin and M. Hagiya. A bounds inference method for vector-based memoisation. In *2nd ACM Intl. Conference on Functional Programming*, pages 176–187, Amsterdam, Holland, June 1997. ACM Press.

[4] W.N. Chin and S.C. Khoo. Calculating sized types. In *Technical Report*, DISCS, NUS, December 1999.

[5] 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.

[6] D. De Schreye and S Decorte. Termination of logic programs: The never-ending story. *Journal of Logic Programming*, 19/20:199–260, 1994.

[7] Arne J. Glenstrup and Neil D. Jones. BTA algorithms to ensure termination of off-line partial evaluation. In *Andrei Ershov Second Intl Conference "Perspective of System Informatics" (LNCS 1110)*, 1996.

[8] Carsten Kehler Holst. Finiteness analysis. In *5th ACM Functional Programming Languages and Computer Architecture Conference*, Cambridge, Massachusetts, (LNCS, vol 523, pp. 473–495), August 1991.

[9] 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.

[10] P. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The omega library version 1.1.0 interface guide. Technical report, University of Maryland, College Park, November 1996. `http://www.cs.umd.edu/projects/omega`.

[11] P. Kelly, W. Pugh, E. Rosser, and T. Shpeisman. Transitive closure of infinite graphs and its applications. Technical report, umiacs-tr-95-48, University of Maryland, College Park, April 1994.

[12] Lutz Plumer. Termination proofs for logic programs. In *Lecture Notes in Artificial Intelligence*, volume 446. Springer Verlag, 1990.

[13] William Pugh. The omega test: A fast practical integer programming algorithm for dependence analysis. *Communications of ACM*, 8:102–114, 1992.

[14] K. Sohn and A. Van Gelder. Termination detection in logic programs using argument sizes (extended abstract). In *Proceedings of the 10th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 216–226, 1991.

[15] C. Speirs, Z. Somogyi, and Sondergaard H. Termination analysis of mercury. In *Fourth International Symposium on Static Analysis*, Paris, France, (LNCS, vol 1302, pp. 157–171), 1997.

[16] J.D. Ullman and A. Van Gelder. Efficient tests for topdown termination of logical rules. *Journal of ACM*, 35(2):345–373, 1988.

[17] 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.

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

## A Some Example Functions

```
zip :: [a]^m → [b]^n → [(a,b)]^p
    s.t. size (0 <= m <= n ∧ p = m) ∨
              (0 <= n < m ∧ p = n)
         inv  (0 ≤ m⁺ < m) ∧ (m ≤ n + m⁺) ∧
              (n⁺ = n − m + m⁺)
zip xs ys  = case xs of
  Nil → Nil
  (x : xs') → case ys of
              Nil       → Nil
              (y : ys') → (x, y) : zip xs' ys'
```

```
bsearch :: ((a, a)^⊤ → Bool^⊤) →
           a → (Arr^i a) → Int^j
bsearch cmp key arr =
  let look :: (Int^u, Int^v) → Int^⊤
        s.t. inv  lo ≤ lo⁺ ∧ hi ≥ hi⁺
      look(lo, hi) =
       if (hi >= lo)
       then let m = lo + (hi − lo) div 2
                x = sub(arr, m)
            in case cmp(key, x) of
                 LESS  → look(lo, m − 1)
                 EQUAL → m
                 MORE  → look(m + 1, hi)
       else NONE
  in look(0, (length arr) − 1)
```

```
g :: ([Int^u]^i, [Int^v]^j) → [Int^w]^⊤
  s.t. size (i = 0 ∧ j ≥ 0 ∧ l = 0) ∨
            (i ≥ 1 ∧ j ≥ 0 ∧ l = i − 1) ∨
            (i ≥ 1 ∧ j ≥ 0 ∧ l = i + j − 1)
g (x, y) = case x of
  Nil      → Nil
  (b : xs) → case (b == 0) of
               True  → append xs y
               False → xs
```

```
ack :: Int^i → Int^j → Int^k
    s.t. inv (0 ≤ j⁺ < j ∧ 1 ≤ i ∧ i⁺ = i) ∨
             (0 ≤ i⁺ < i ∧ 0 ≤ j)
ack x y = case (x, y) of
           (0, m)         → m + 1
           (n + 1, 0)     → ack n 1
           (n + 1, m + 1) → ack n (ack (n + 1) m)
```

```
bin :: (Int^n, Int^k) → Int^p
    s.t. inv (k⁺, 1 ≤ k < n) ∧
             (k⁺, 1 ≤ n⁺ < n) ∧
             (k + n⁺ ≤ n + k⁺) ∧ 0 ≤ k⁺
bin (n, k) = case n of
  0 → 1
  m + 1 →
   if k ≤ 0 || k ≥ n then 1
     else bin(m, k − 1) + bin(m, k)
```

## B Context Specialisation

One popular way to optimise code is via program specialisation. Partial evaluation is a well-known example of this approach that exploits known (static) arguments to help derive more efficient code.

Sized types can contribute towards more program specialisation, through the interprocedural propagation of size constraints. Specialised size constraint can help eliminate redundant branches (dead code), and may also reduce unnecessary tests.

Consider the *zip* function (in Appendix A) that combines two lists into a list of pairs. Such a function may be called under different size constraints. For example, given the call $(zip\ \ e_1\ \ e_2)$, the sizes of the two actual arguments, respectively $s_1$ and $s_2$, may be related in some way. Three possible specialised contexts on this relationship are (i) $s_1 \leq s_2$, (ii) $s_1 \geq s_2$, or (iii) $s_1 = s_2$. The specialised program that could be derived may vary depending on such sized contexts.

When $s_1 \leq s_2$, our specialisation technique obtains:

```
zip1 :: [a]^n → [b]^m → [(a, b)]^p
        size (n ≤ m) ∧ (p = n)
zip1 xs ys = case xs of
               Nil → Nil
               x : xs' →
                case ys of
                 y : ys' → (x, y) : (zip1 xs' ys')
```

(The annotations on *pair* constructor and type variables $a$ and $b$ are all $\top$, and thus ignored here.) Successful elimination of a branch has helped to eliminate a piece of dead code and also the need for a tag-check on variable *ys*.

When $s_1 \geq s_2$, our specialisation technique obtains:

```
zip2 :: [a]^n → [b]^m → [(a, b)]^p
        size (n ≥ m ∧ p = m)
zip2 xs ys = case ys of
               Nil → Nil
               y : ys' →
               case xs of
               x : xs' → (x, y) : (zip2 xs' ys')
```

Again a piece of dead code has been eliminated, but note that a swap of the two case statements has been made to allow the smaller sized list to be tested first. Such swaps (based on size information) would have to be built into context specialisation to maximise the opportunities for optimisation.

When $s_1 = s_2$, our specialisation technique would obtain a specialised code that is similar to *zip*1, but with a size context that is subsumed by $s_1 \leq s_2$. This indicates that maximum specialisation for zip has already been achieved under a weaker context. A scheme that syntactically checks for duplication of specialised programs can help avoid needless code explosion.