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 optimizations and analyses, such as array-bounds 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 a *size relation* between different inputs of a function. Secondly, different optimizations 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 size inference. 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 utilizes the Omega Calculator to compute as exact a size information as possible, within the linear arithmetic capability.

1. Introduction

Many program optimizations and analyses, such as array-bounds 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, the computation of the size cannot be easily expressed using the usual "independent attribute analysis"; rather, a "relational analysis" is required.

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

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

There are several kinds of size information that are of interest to us. First and foremost, we are interested in knowing how the output size of *append* relates to the sizes of its two inputs, since this input/output size

* This is a revised and expanded version of the paper presented in the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation.

relationship is fundamental to the computation of "inter-procedural" size information. Let m, n be the sizes (*i.e.*, lengths) of the parameters xs, ys, respectively, and p be the output size, then we have $m + n = p$. Next, we are also interested in finding how the sizes of the argument of the recursive calls to *append* compare to the sizes of the original arguments. In the case of *append*, the size of the first argument to the recursive call of *append* is monotonically decreasing, while the second remains constant in size. Such information is useful for analyzing, among other properties, the termination of the program, or the termination of a transformation process, such as partial evaluation (*eg.*, [9, 15]).

The literature on size computation has mainly concentrated on size checking, instead of size inference ([16, 29]). In this paper, we aim to *infer* as accurate size information as possible for various optimizations and analyses. In the case of *append*, we can express its size (and its type) as follows:

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

Here, we capture two pieces of size information. The first, prefixed by the keyword **size**, relates the size of the function's input to that of its output. The second, prefixed by **inv**, is used for recursive functions. We let m and n be the sizes of the arguments of an (initial) call to *append*, and m^+ and n^+ be the sizes of the arguments of an arbitrary recursive (inner) call to *append*. The constraint expressed by the declaration 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 shares some commonality; *i.e.*, 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, from which kinds of information can be computed. We express size information using Presburger formulae, a class of logical formulae which can be built from affine constraints over integer variables, the logical connetives $\neg$, $\wedge$, $\vee$, and the quantifiers $\exists$ and $\forall$. There are a couple of practical reasons for limiting size information to Presburger formulae: Firstly, there exist efficient tools for solving these formulae. Secondly, Presburger formulae can be used to capture the size information in a wide range of practical applications.

We employ the Omega Calculator [24] for solving and inferring suitable Presburger formulae 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 employ fixed-point techniques to compute a safe approximation of the size of the inputs and outputs of 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* [24] to compute a suitable size. Consequently, we propose an effective algorithm for *inferring*, instead of checking, the required size information of any function.

The outline for this paper is as follows: In the next section, we motivate the use of size information in various applications. Then, in Section 3, we describe the syntax and semantics of the source language. Section 4 describes the constraint system used to capture size information. Section 5 describes the inference algorithm declaratively, using a type system. Type inference is further illustrated through an example in Section 6. In Section 7, we describe how fixed-point computation is performed via the transitive-closure operation. Section 8 describes recursive-call-invariant computation. Section 9 discusses a strategy for obtaining a constraint that approximates fixed-point computation. Section 10 provides general discussion of issues related to our type system, followed by discussion of related work in Section 11. Section 12 concludes the paper by presenting our contributions.

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 optimization, such as elimination of bounds checks, dead codes, and repeated calls. In this section, we motivate the need for sized types by briefly explaining how they can be used to support termination property detection, vector-based memoization, and bounds-check elimination.

2.1. Termination Property

Sized types can be used to confirm the *termination property* of recursive functions. A recursive function terminates if there exists a set of arguments that can be shown to decrease across recursion. Termination proofs normally use well-founded orderings (or quasi-orderings) of the arguments of recursive functions. Termination property is needed by safety-critical systems [16]. It is also used to ensure that the partial evaluation process terminates [9]. To briefly illustrate termination property is detected, consider the following Ackermann function.

$$
\begin{array}{l}
ack :: Int^i \rightarrow Int^j \rightarrow Int^k \\
\qquad \text{s.t. } \mathbf{inv} \ \ (0 \leq j^+ < j \wedge 1 \leq i \wedge i^+ = i) \\
\qquad\qquad\quad \vee (0 \leq i^+ < i \wedge 0 \leq j) \\
ack\ x\ y = \mathbf{case}\ (x,y)\ \mathbf{of}\ \begin{array}[t]{ll} (0,m) & \rightarrow m+1 \\ (n+1,0) & \rightarrow ack\ n\ 1 \\ (n+1,m+1) & \rightarrow ack\ n\ (ack\ (n+1)\ m) \end{array}
\end{array}
$$

The recursive-call invariant about its two parameters indicates a well-founded lexicographic ordering. This invariant confirms that recursive calls to *ack* will eventually terminate, even though the second parameter has an increasing value.

2.2. Vector-Based Memoisation

Sized types can also be used to determine the sizes of the vectors needed to avoid the re-computation of repeated calls, through a technique known as dynamic-sized memoisation [4]. A simple example is the following binomial function with exponential time complexity:

$$
\begin{array}{l}
bin :: (Int^n,\ Int^k) \rightarrow Int^p \\
\qquad \text{s.t. } \mathbf{inv} \ \ (k^+,\ 1 \leq k < n) \wedge (k^+,\ 1 \leq n^+ < n) \\
\qquad\qquad\quad \wedge (k + n^+ \leq n + k^+) \wedge 0 \leq k^+ \\
bin\ (n,k) = \mathbf{case}\ n\ \mathbf{of}\ \begin{array}[t]{ll} 0 & \rightarrow 1 \\ m+1 & \rightarrow \mathbf{if}\ k \leq 0 \,||\, k \geq n\ \mathbf{then}\ 1 \\ & \qquad\quad \mathbf{else}\ bin(m,k-1) + bin(m,k) \end{array}
\end{array}
$$

If the redundant calls of binomial function is 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 can derive the following less readable but much efficient vector-based definition of binomial (using Haskell-like notation).

$$
\begin{array}{l}
bin\ \ (n,k)\ \ = \\
\quad \mathbf{let}\ bin'\ (n^+) = \begin{array}[t]{l} \mathbf{case}\ n^+\ \mathbf{of} \\ 0 \rightarrow (\lambda\, k^+ \rightarrow 1) \\ (m+1) \rightarrow \\ \mathbf{let}\ \begin{array}[t]{lcl} z & = & bin'(m)\,; \\ bn\ k^+ & = & \mathbf{if}\ (k^+ \leq 0 \vee k^+ \geq n^+)\ \ \mathbf{then}\ 1 \\ & & \mathbf{else}\ \ z(k^+ - 1) + z(k^+)\,; \\ l & = & max(0, n^+ - (n-k))\ ; \\ u & = & min\ (n^+, k)\ ; \\ vec & = & array\,(l,u)\,[a := bn\ a \mid a \leftarrow [l..u]] \end{array} \\ \mathbf{in}\ \ (!)\ vec \end{array} \\
\quad \mathbf{in}\ \ \mathbf{if}\,(0 \leq k \wedge max(1,k) \leq n)\ \ \ \mathbf{then}\ \ bin'(n)\,(k)\ \ \mathbf{else}\ \ 1
\end{array}
$$

2.3. Bound-Check Elimination

Run-time bounds checking on arrays is crucial for ensuring robust programs but can be expensive. Many compilers can eliminate bounds checks. However, they do not fare well, due to their inability to synthesize loop invariants and perform inter-procedural analysis. Sized type can give better results than standard techniques.

Consider the following binary search function *bsearch*:

$$
\begin{array}{l}
\mathit{data}\ \mathit{Cmp}\ =\ \mathit{LESS}\ \mid\ \mathit{EQUAL}\ \mid\ \mathit{MORE}\\
\mathit{bsearch}\ \mathit{cmp}\ \mathit{key}\ \mathit{arr}\ =\\
\quad \textbf{let}\ \mathit{look}\ ::\ (\mathit{Int}^u, \mathit{Int}^v)\ \to\ \mathit{Int}^\top\\
\qquad\qquad \text{s.t. } \textbf{inv}\ \ u \leq u^+ \wedge v \geq v^+\\
\qquad \mathit{look}(lo, hi)\ =\ \ \textbf{if}\ (hi >= lo)\\
\qquad\qquad\qquad\qquad\quad \textbf{then let}\ m\ =\ lo + (hi - lo)\ \mathit{div}\ 2\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad x\ =\ \mathit{sub}(arr, m)\\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{in case}\ \mathit{cmp}(key, x)\ \textbf{of}\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{LESS}\ \ \to \mathit{look}(lo, m-1)\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{EQUAL} \to m\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{MORE}\ \ \to \mathit{look}(m+1, hi)\\
\qquad\qquad\qquad\qquad \textbf{else}\ \mathit{NONE}\\
\quad \textbf{in}\ \mathit{look}(0, (\mathit{length}\ arr)\ - 1)
\end{array}
$$

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 sizes 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 invocations of *look*. The result is sufficient to allow the replacement of *sub* by an optimized version (without bound-checking).

An approach to allow bound checks to be safely eliminated was proposed by Xi and Pfenning [29]. In their type-checking framework, it is the responsibility of the user to specify where bound checks can be eliminated. Automatic inference of sized types presents a new challenge. With the help of automatic size type inference, we have recently formulated a new approach to identifying and eliminating redundant bounds check [5]. Our approach is able to identify both totally-redundant and partially-redundant bound checks. In the case of partially-redundant bound checks, we make use of a program specialisation to strengthen the context under which bound checks can be safely eliminated.

3. Language Syntax and Semantics

We apply our technique to a higher-order typed functional language with a strict semantics. Recursive functions in the language are confined to self-recursion for simplicity. Mutual recursion can be encoded into self-recursion by appropriate tagging of input and output, as described in Section 10. The syntax of our source language is shown in Figure 1.

$$
\begin{array}{lcll}
x & \in & \mathbf{Var} & \text{— Variables} \\
n & \in & \mathbf{Int} & \text{— Integer constants} \\
c & \in & \mathbf{Con} & \text{— Data Constructors} \\
p & \in & \mathbf{Pat} & \text{— Patterns} \\
e & \in & \mathbf{Exp} & \text{— Expressions}
\end{array}
$$

$$
\begin{array}{rl}
e ::= & x \mid n \mid c\, e_1 \,\ldots\, e_n \mid \lambda\, x\,.\,e \mid (e_1 \;\; e_2) \\
& \mid \; \mathbf{let} \;\; x \; = \; e_1 \;\; \mathbf{in} \;\; e_2 \; \mid \; \mathbf{letrec} \;\; x \; = \; e_1 \;\; \mathbf{in} \;\; e_2 \\
& \mid \; \mathbf{case} \;\; e_0 \;\; \mathbf{of} \;\; \{c_i\, x_{1_i} \,\ldots\, x_{m_i} \;\rightarrow\; e_i\}_{i=1}^{n}
\end{array}
$$

Figure 1. Syntax of the Source Language

Our method is accurate for only a restricted form of higher-order functions, as will be discussed in Section 10.

We assume that expressions are well-typed under Milner-Damas type system. Only well-formed and well-typed expressions are assigned a meaning. This meaning is defined by the denotational semantics.

The denotational semantics of the language has two semantic domains – one to model values ($\mathcal{U}$) and the other to model the value environment (Env).

$$
\begin{array}{lcl}
\mathcal{U} & = & \mathbf{Int}_\perp \oplus \mathbf{Con}_\perp \oplus [\mathcal{U} + \mathcal{U}] \oplus [\mathcal{U} \times \mathcal{U}] \oplus [\mathcal{U} \rightarrow \mathcal{U}] \\
Env & = & \mathbf{Var} \rightarrow \mathcal{U}
\end{array}
$$

The flat domains of integers and data constructors are denoted by $\mathbf{Int}_\perp$ and $\mathbf{Con}_\perp$ respectively. The definitions of domain operations $_\perp$ (lifting), $+$ (separated sum), $\times$ (cartesian products), $\rightarrow$ (continuous function space), and $\oplus$ (coalesced sum) are standard. We will write a series of cartesian products as one n-ary product. The semantic function $\mathcal{E}$ is shown in Figure 2.

$\rho[v_1/x_1, \ldots, v_k/x_k]$ denotes environment update, and is defined by

$$
(\rho[v_1/x_1, \ldots, v_k/x_k])\; x \;=\; \begin{cases} v_j & \textit{if } \; x \; = \; x_j \;\; \textit{for some} \\ & \qquad\qquad\quad j \; \in \; \{1, \ldots, k\} \\ \rho\, x & \textit{otherwise} \end{cases}
$$

$$
\begin{array}{lcl}
\mathcal{E} :: Exp \rightarrow Env & \rightarrow & \mathcal{U} \\
\mathcal{E} [\![n]\!] \rho & = & n \\
\mathcal{E} [\![x]\!] \rho & = & \rho [\![x]\!] \\
\mathcal{E} [\![c \;\; e_1 \ldots e_n]\!] \rho & = & c \;\; (\mathcal{E} [\![e_1]\!] \rho) \ldots (\mathcal{E} [\![e_n]\!] \rho) \\
\mathcal{E} [\![\lambda x \,.\, e]\!] \rho & = & f \\
 & & \mathit{where} \;\; f(v) \; = \; \mathcal{E} [\![e]\!] \rho[v/x] \\
\mathcal{E} [\![e_1 \;\; e_2]\!] \rho & = & (\mathcal{E} [\![e_1]\!]) \; (\mathcal{E} [\![e_2]\!]) \\
\mathcal{E} [\![\textbf{let} \;\; x \; = \; e_1 \;\; \textbf{in} \;\; e_2]\!] \rho & = & \mathcal{E} [\![e_2]\!] \rho[v/x] \\
 & & \mathit{where} \;\; v \; = \; \mathcal{E} [\![e_1]\!] \rho \\
\mathcal{E} [\![\textbf{letrec} \;\; x \; = \; e_1 \;\; \textbf{in} \;\; e_2]\!] \rho & = & \mathcal{E} [\![e_2]\!] \rho[v/x] \\
 & & \mathit{where} \;\; v \; = \mathit{fix} \; (\lambda x \,.\, e_1) \\
\multicolumn{3}{l}{\mathcal{E} [\![\textbf{case} \;\; e_0 \;\; \textbf{of} \;\; \{c_i \, x_{1_i} \ldots x_{m_i} \rightarrow e_i\}_{i=1}^{n}]\!] \rho \; =} \\
\multicolumn{3}{l}{\qquad\qquad \mathit{case} \; (\mathcal{E} [\![e_0]\!] \rho) \; \mathit{in}} \\
\multicolumn{3}{l}{\qquad\qquad\quad c_1 \, v_{1_1} \ldots v_{m_1} \;\Rightarrow \mathcal{E} [\![e_1]\!] \rho[v_{1_1}/x_{1_1}, \ldots, v_{m_1}/x_{m_1}]} \\
\multicolumn{3}{l}{\qquad\qquad\qquad\qquad\qquad \vdots} \\
\multicolumn{3}{l}{\qquad\qquad\quad c_n \, v_{1_n} \ldots v_{m_n} \;\Rightarrow \mathcal{E} [\![e_n]\!] \rho[v_{1_n}/x_{1_n}, \ldots, v_{m_n}/x_{m_n}]}
\end{array}
$$

Figure 2. Semantics of the Source Language

4. Constraints

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

Figure 3. Syntax of Constraints

Our technique requires manipulation of constraints represented as *Presburger formulae*. Figure 3 depicts their syntax. Note that arithmetic expressions are restricted to affine functions of the variables. For example, $2 * v$ is legal, but $v * v$ is not. When $X = \{v_1, \ldots, v_n\} \subset \mathbf{V}$, we write $\exists\, X.\, \phi$ or $\exists\, v_1 \ldots v_n \,.\, \phi$ instead of $\exists\, v_1. \ldots . \exists\, v_n \,.\, \phi$.

The basic Presburger algorithm that decides the validity (or satisfiability) of Presburger formulae is given by Hilbert and Bernays [14]. D.C. Cooper [6] extends the algorithm to include subtraction and formulae on both positive and negative integers. In this paper, Presburger formulae are simplified and satisfied via the *Omega calculator*[24]. We view our constraint system as a *cylindric algebra*[13], 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[21, 22].

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.

Since formulae are identified up to logical equivalence, free variables appearing in a formula should be defined as those appearing in every logically equivalent formula. Hence, we define the set $fv(\phi)$ of free variables occurring in a formula ϕ by $fv(\phi) = \{\, v \mid \phi \neq_{\mathbf{P}} \exists\, v.\, \phi \,\}$.

DEFINITION 1 (Satisfiability of a Formula). *A formula ϕ is* satisfiable *iff* $\vdash_{\mathbf{P}} \exists fv(\phi).\, \phi$.

Given two formula, ϕ_1 and ϕ_2, we say that ϕ_2 *approximates* ϕ_1, denoted by $\phi_1 \sqsubseteq_{\mathbf{P}} \phi_2$, iff $\phi_1 \vdash_{\mathbf{P}} \phi_2$.

A constraint as defined in Figure 3 can be a formula, ϕ, or a *parameterized* formula of the form $\Lambda_S v_1 \ldots v_k \,.\, \phi$. Λ_S provides a binding mechanism that binds variables in $fv(\phi)$. When the free variables refer to the sizes of the input and output of an expression, the corresponding parameterized formula can be interpreted as constraints on the sizes of the input and output of that expression. So, $\Lambda_S\, v_1 \ldots v_k \,.\, \phi$ can be presented in different forms. For instance, we use the notation $[v_1] \rightarrow [v_2, \ldots, v_k] : \phi$ (this is called the *relational form* in the Omega's terminology) to indicate that ϕ is the constraint between the input of size

v_1 and the output of sizes $v_2, \ldots, v_k$. On the other hand, $[v_1, \ldots, v_k]$: ϕ (this is called the *set form*) denotes a general constraint between the sizes $v_1, \ldots, v_k$, without any differentiation between input and output. In general, this presentation can be written as $X \rightarrow Y : \phi$, where X and Y are (possibly empty) sets of size variables. Lastly, we note that while these presentations may be used with different intentions, they all have equivalent logical semantics, which is exactly that of ϕ.

We use the terminology "formula", "term", and "constraint" interchangeably, when it is clear from the context. Also, we use the notions $[v_1, \ldots, v_k] : \phi$ (and its variants) and $\Lambda_S v_1 \ldots v_k : \phi$ interchangeably.

Given two constraints $\Phi = \Lambda_S X . \phi$ and $\Psi = \Lambda_S Y . \psi$. Then $\Phi \wedge \Psi$ is meaningful when $X = Y$, and the result is $\Lambda_S X . \phi \wedge \psi$. We define $\Phi \vee \Psi$ similarly.

Given a formula ϕ, $\phi[\psi_1/y_1, \ldots, \psi_n/y_n]$ denotes the substitution of $\psi_1, \ldots, \psi_n$ for the free variables $y_1, \ldots, y_n$ in ϕ. For a parameterized formula, substitution applies to its parameters as well. Substitution is written in a manner similar to function application. Thus, given $\Phi = [v_1, \ldots, v_k] : \phi$, we write $\Phi(n_1, \ldots, n_k)$ to denote $\phi[n_1/v_1, \ldots, n_k/v_k]$.

5. Type System

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

A *sized type* is a pair containing an *annotated type* and a *constraint.* The precise set of annotated type expressions is shown in Figure 4.

$$
\begin{array}{lcll}
v, i & \in & \mathbf{V} \cup \{\top\} & \text{— Size Variables} \\
t & \in & \mathbf{TVar} & \text{— Type Variables} \\
D & \in & \mathbf{TypeCon} & \text{— Type Constructors} \\
\sigma & \in & \mathbf{TypeSch} & \text{— Type Schemes} \\
 & ::= & \forall\, t\, .\, \sigma \mid \tau & \\
\tau & \in & \mathbf{Type} & \text{— Type} \\
 & ::= & t \mid \tau \rightarrow \tau \mid D^v\, \tau_1 \ldots \tau_k &
\end{array}
$$

Figure 4. Annotated Type Expressions

We use the notations $[\cdot]$ and $(\cdot, \ldots, \cdot)$ for type constructors of lists and tuples, respectively. Size variables are attached to type constructors, such as $[Int^v]^w$, where w refers to the length of a list, and v refers to the possible integer values of list elements. $[Int]$ is referred to as the *underlying type* of this list, and is described by underlining the annotated type; *i.e.*, $\underline{[Int^v]^w} = [Int]$. Size variables are used to formulate the size constraint of an expression.

The symbol $\top$ denotes the absence of size information. Thus, the domain $D^\top$ 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. While size variables may appear in the constraint, $\top$ may not.

Type variables convey no size information, since we do not know the actual type yet. Consequently, our type system does not calculate any relationship among polymorphically-typed parameters. We leave this interesting issue as one of the future works.

5.1. The Meaning of Size

The term *size* can have various meanings. In Xi and Pfenning's work on dependent type system [30], users can specify a specific meaning of size through a size annotation on the data declaration. To specify the rules for size computation, we require an understanding of what sizes mean. While our work can possibly be extended to support user-specified size annotation, we shall for simplicity assume default interpretation of sizes for data types of interest. In this paper, we focus on deriving the size of lists declared as $[t] = Nil + (t : [t])$. A shorthand notation for a finite list is of the form $[d_1, \ldots, d_k]$, where $d_1, \ldots, d_k$ are elements of some type. Following is the default interpretation of sizes of the relevant data types.

1. The size of an integer is the integer itself.

2. The size of a recursive data consists of its maximum depth (*i.e.*, the number of nestings). 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 the array size.

3. We assign distinct size value to different data of a non-recursive enumerated type. In the case of boolean type, we arbitrarily choose 0 as the size of *False*, and 1 for *True*. Similarly, in Section 2.3, we declare a data type $Cmp = LESS + EQUAL + MORE$. We will assign 0, 1, and 2 to the size of $LESS$, $EQUAL$, and $MORE$ respectively.

[1] 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 does not include formal treatment of arrays.

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

Note that we provide separate size information for different data constructors in an enumerated data type. In contrast to the work of sized typing by Hughes, Pareto, and Sabry [16], this treatment of enumerated type enables us to infer more accurate size information of an expression whose value depends on enumerated data.

Formally, we define the size through the following function $\mathcal{S}$.

DEFINITION 2 (Size function $\mathcal{S}$). *Function $\mathcal{S}$ is a strict function that takes in a data with its annotated type, and returns a formula describing the size of the data.*

$$
\begin{array}{ll}
\mathcal{S} :: (\mathcal{U} \times \textbf{TypeSch}) & \rightarrow \textbf{P} \\
\mathcal{S}\,(\bot :: \sigma) & = \mathit{False} \\
\mathcal{S}\,(n :: \mathit{Int}^v) & = \mathit{chk}_v\,(v = n) \\
\mathcal{S}\,(\mathit{Nil} :: [\tau]^v) & = (\mathit{chk}_v\,(v = 0)) \qquad \forall\, \tau \in \textbf{Type} \\
\mathcal{S}\,([d_1, \ldots, d_k] :: [\tau]^v) & = ((\mathit{chk}_v\,(v = k)) \wedge \bigvee_{j=1}^{k} (\,\mathcal{S}\,(d_j :: \tau))) \\
\mathcal{S}((d_1, d_2) :: (\tau_1, \tau_2)^\top) & = (\,\mathcal{S}(d_1 :: \tau_1)) \wedge (\,\mathcal{S}\,(d_2 :: \tau_2))) \\
\mathcal{S}\,(\mathit{False} :: \mathit{Bool}^v) & = (\mathit{chk}_v\,(v = 0)) \\
\mathcal{S}\,(\mathit{True} :: \mathit{Bool}^v) & = (\mathit{chk}_v\,(v = 1)) \\
\mathcal{S}\,(f :: \tau_1 \rightarrow \tau_2) & = \phi \\
\quad \mathit{where}\ \phi = \bigwedge \{\phi_i \mid & \phi_i \in \textbf{P}\ \ \mathit{and}\, \forall\,(d_1 :: \tau_1) : \forall\,(d_2 :: \tau_2) : \\
& (d_2 = f\ d_1)\ \mathit{implies} \\
& \quad ((\,\mathcal{S}\,(d_1 :: \tau_1)) \wedge (\,\mathcal{S}\,(d_2 :: \tau_2))) \vdash_{\textbf{P}} \phi_i \,\} \\
\mathcal{S}\,(d :: \forall\, t\,.\,\sigma) & = \,\mathcal{S}\,(d :: \sigma) \\
\mathcal{S}\,(d :: t) & = \mathit{True} \\
 & \\
\mathit{chk}_v :: \textbf{P} & \rightarrow \textbf{P} \\
\mathit{chk}_v\ \phi & = \mathit{if}\ (v = \top)\ \mathit{then}\ \ \mathit{True}\ \ \mathit{else}\ \ \phi
\end{array}
$$

In the above definition, function chk_v checks if the variable v is $\top$, and provides no information (*i.e.*, *True*) when that is true. Notice also that we do not assign size information for tuple construction. If we were to assign a size for a pair, it would be 2. Since the type system only accepts well-typed program, it is not useful to know the sizes of tuples.

The size of a function is defined as the strongest constraint which can be satisfied by all pairs of the function's input- and output-sizes.

We treat **TypeSch** as a discrete domain. The fact that $\mathcal{S}$ is monotone can be obtained by structural induction. We omit the proof here.

LEMMA 1. *Function $\mathcal{S}$ is continuous.*

Proof We show the case when the first argument to $\mathcal{S}$ is an element of a chain of functions. Consider a chain of functions $\{f_i :: \tau_1 \rightarrow \tau_2\}_{i>0}$

such that for each i, $\mathcal{S}(f_i :: \tau_1 \to \tau_2) = \phi_i$. Let $f_\infty = \sqcup \{ f_i :: \tau_1 \to \tau_2 \}$. We wish to show that $\mathcal{S}(f_\infty :: \tau_1 \to \tau_2) = \bigsqcup_{\mathbf{P}\, i>0} \{\phi_i\} = \bigvee_{i>0} \{\phi_i\}$.

For any $d :: \tau_1$ and any $d' :: \tau_2$, if $d' = f_\infty d$, then there exists a function f_j for some j such that $d' = f_j\ d$. By the definition of $\mathcal{S}$, we thus have:

$$\begin{array}{ll} (d' = f_\infty\ d) \ \textit{implies}\ (d' = f_j\ d) \text{ for some } j & \\ \quad \textit{implies}\ ((\mathcal{S}(d :: \tau_1)) \wedge (\mathcal{S}(d' :: \tau_2))) \vdash_{\mathbf{P}} \phi_j & \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdash_{\mathbf{P}} \bigvee_{i>0} \{\phi_i\}. \end{array}$$

We next show that $\bigvee_{i>0} \{\phi_i\}$ is the desired strongest constraint. Given any constraint ψ such that for any $d :: \tau_1$ and any $d' :: \tau_2$, $d' = f_\infty\ d$ implies $(\mathcal{S}(d :: \tau_1)) \wedge (\mathcal{S}(d' :: \tau_2)) \vdash \psi$. By monotonicity of $\mathcal{S}$, we must have $\phi_i \sqsubseteq_{\mathbf{P}} \psi$ for all $\phi_i = \mathcal{S}(f_i :: \tau_1 \to \tau_2)$. Thus, $(\bigvee_{i>0} \phi_i) \sqsubseteq_{\mathbf{P}} \psi$. □

We further extend on the definition of $\mathcal{S}$ to work on products of denotation values. That is,

$$\mathcal{S}(d_1 :: \tau_1, \ldots, d_k :: \tau_k) \;=\; \bigwedge_{i=1}^{k} \mathcal{S}(d_i, \tau_i)$$

COROLLARY 2. *In the extended definition, $\mathcal{S}$ is continuous on all its data arguments.*

Proof By Lemma 1 and the distributivity of $\wedge$ over $\vee$ in the algebra **P**. □

Intuitively, if a value, such as an integer, is not $\perp$, then it has a size, and applying $\mathcal{S}$ to this value should produce a formula that is satisfiable (since this data is itself an evidence for the satisfiability of the formula). However, this intuition does not extend straightforwardly to a function. It is possible to define a function that never terminates. In this case, the size of this function should be equivalent to *False*. Connecting this intuition with the definition of satisfiability in Def. 1, we have the following tautology:

PROPERTY 1. *For any $d :: \mathcal{U}$ with annotated type τ. Suppose that $\phi = \mathcal{S}(d :: \tau)$ and $d \neq \perp$. Then either $\exists fv(\phi).\ \phi =_{\mathbf{P}}$ True or $\exists\, fv(\phi).\ \phi =_{\mathbf{P}}$ False.*

We capture two kinds 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 inter-procedural analysis. Therefore, we compute the size relation between a function's input and output, before computing recursive-call invariant.

5.2. Type Rules

Our type system specifies the size relation of every expression in the language. 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.

The subtyping relation for our type system is simply the approximation relation in our constraint system:

$$\frac{(\sigma, \phi) \quad \phi \sqsubseteq_P \psi}{(\sigma, \psi)}$$

Figures 5 and 6 list the inference rules for inferring the sized type of an expression. To avoid cluttering of symbols, we include a "where"-clause beneath a rule whenever necessary. The "where"-clause defines new symbols to be used in either the premises or the conclusion.

A type usually has an inherent size constraint associated with it. For instance, if the (annotated) type is of the form $[t]^i$, then it is true that $(chk_i\ (i \geq 0))$ (chk is defined in Def. 2). That is, if $i \in \mathbf{V}$, the length of a list must be non-negative. Next, for $Bool^j$, we have $(chk_j\ (j = 0 \vee j = 1))$. The *inherent constraint* associated with an annotated type σ is denoted by $\mathcal{I}(\sigma)$.

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

- *Size-variable renaming* (α): This is an overloaded operation serving various purposes:
  1. Applying α to an annotated type (or type scheme), τ (or σ), returns a new annotated type with the same underlying type, but new free size variables. Note that renaming $\top$ returns $\top$, and renaming a variable in $\mathbf{V}$ returns a new variable in $\mathbf{V}$.
  2. Applying α to a sized type (σ, ϕ) results in a consistent renaming of size variables occurring in σ. 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.
- *Size-variable listing* (ι): This takes in either an annotated type σ or a formula ϕ, and returns the set of free size variables (from $\mathbf{V}$) occurring in σ or ϕ. Thus, $\iota([Int^i]^j) = \{i, j\}$. In the case of

formulae, ι works just like function *fv* defined in Section 4. We also extend ι to take in a type assumption as argument: Given $\Gamma = \{x_i :: (\sigma_i, \phi_i)\}_{i=1}^n$, we define $\iota(\Gamma) = \bigcup_{i=1}^n \{\iota(\sigma_i) \cup \iota(\phi_i)\}$.

– *Size-variable matching* (eqC): Given two annotated types, σ_1 and σ_2, with identical underlying type. If $\iota(\sigma_1) \cap \iota(\sigma_2) = \emptyset$, then eqC produces a conjunction of equalities that identify the corresponding free size variables between σ_1 and σ_2. For instance, $eqC\ [Int^i]^j\ [Int^k]^l = chk_i\ (chk_k\ (i = k)) \wedge chk_j\ (chk_l\ (j = l))$.

A *type assumption* Γ binds program variables to their sized types. We write $\mathcal{C}_\Gamma$ to denote the collection of all *constraints* occurring in Γ. For example, given an assumption $\Gamma = \{\ b :: (Bool^i,\ i = 0 \vee i = 1),\ xs :: ([a]^j,\ j \geq 0)\}$, we have $\mathcal{C}_\Gamma = \{i = 0 \vee i = 1,\ j \geq 0\}$. We call this set of constraint the *contextual constraint*. When the context is clear, we also use $\mathcal{C}_\Gamma$ to denote the conjunction of all these constraints. Γ is *consistent* iff $\mathcal{C}_\Gamma$ is satisfiable.

New and unique size variables (from $\mathbf{V}$) are generated during size inference. This is achieved via either the application of function *new* or via size-variable renaming. These size variables are associated with some (sub-)expressions. As the focus of type inference shifts from one expression to its enclosing expression, those size variables associated with the sub-expression become "irrelevant", and need to be "localized", so that type inference will only deal with those size variables associated with the enclosing expression. This localization of size variable is achieved via existential quantification.

There are three places in the type rules where irrelevant size variables are localized. They appear respectively in the rules `(Abs)`, `(Let)` and `(Case)`.

Now we describe type rules in detail, bearing in mind that the usual (un-annotated) types for expressions have been provided, and the rules are used to infer constraints about the size.

The rules for variables and integer constants are obvious. There are two rules for constructing a list: For constructing an empty list, we set the length of the list to be 0. For adding an element argument to the beginning of a list argument, we infer that the length of the new list is one element longer than the list argument. Furthermore, the size of the list element is a *disjunction* of the size of the element argument and the size of elements in the list argument. This disjunction is achieved by equating the size variables of the resulting list element to *either* the size variables of the element argument *or* that of the elements of the list argument.

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

$$\frac{}{\Gamma \cup \{\, x \,::\, (\sigma,\, \phi)\} \,\vdash\, x \,::\, (\sigma, \phi)} \qquad \texttt{(Var)}$$

$$\frac{}{\vdash\, n \,::\, (\texttt{Int}^i, \phi)} \qquad \texttt{(Con : Int)}$$
$$\text{where } \phi \,=\, chk_i \;\; (i \,=\, n);\; new\; i$$

$$\frac{}{\vdash\, Nil \,::\, ([\tau]^i, \phi)} \qquad \texttt{(Con : List}_1\texttt{)}$$
$$\text{where } \phi \,=\, chk_i \;\; (i \,=\, 0);\; new\; i$$

$$\frac{\Gamma \,\vdash\, e_1 \,::\, (\tau, \phi_1) \qquad \Gamma \,\vdash\, e_2 \,::\, ([\tau']^j, \phi_2)}{\Gamma \,\vdash\, (e_1 \,:\, e_2) \,::\, ([\tau'']^a,\, \phi)} \qquad \texttt{(Con : List}_2\texttt{)}$$
$$\begin{array}{lll} \text{where } \phi & = & chk_a \;\, (chk_j \;(a \,=\, j+1)) \,\wedge \\ & & \quad \phi_3 \,\wedge\, \phi_2 \,\wedge\, \phi_1 \\ \phi_3 & = & (eqC\; \tau''\; \tau) \,\vee\, (eqC\; \tau''\; \tau') \\ [\tau'']^a & = & \alpha([\tau']^j) \end{array}$$

$$\frac{\Gamma \,\cup\, \{x \,::\, (\tau_1, \phi_1)\} \,\vdash\, e \,::\, (\tau_2,\, \phi_2)}{\Gamma \,\vdash\, (\lambda\, x.\, e) \,::\, (\tau_1 \,\rightarrow\, \tau_2, \phi_2')} \qquad \texttt{(Abs)}$$
$$\begin{array}{lll} \text{where } \phi_1 & = & \mathcal{I}(\tau_1 \,\rightarrow\, \tau_2) \\ \phi_2' & = & \exists\, X \,.\, \phi_2 \,\wedge\, \phi_1 \wedge \mathcal{C}_\Gamma \\ X & = & \iota(\phi_2) \,-\, (\iota(\tau_1 \,\rightarrow\, \tau_2) \,\cup\, \iota(\Gamma)) \end{array}$$

$$\frac{\Gamma \,\vdash\, e_1 \,::\, (\tau_1 \,\rightarrow\, \tau,\, \phi_1) \qquad \Gamma \,\vdash\, e_2 \,::\, (\tau_2,\, \phi_2) \qquad \mathcal{CT}(\Gamma, (\tau_1, \phi_1), (\tau_2, \phi_2))}{\Gamma \,\vdash\, (e_1\; e_2) \,::\, (\tau,\, \phi_1 \,\wedge\, (eqC\; \tau_2\; \tau_1) \,\wedge\, \phi_2)} \qquad \texttt{(App)}$$

Figure 5. Rules for Checking Sized Type — Part I

abstraction body. The result of inference is denoted by ϕ_2. As the abstraction may be applied later on in some other context, we make ϕ_2 to be context-independent by (1) including (via conjunction) the contextual constraint to obtain the constraint $\phi_2 \wedge \phi_1 \wedge\ \mathcal{C}_\Gamma$, and (2) localizing those "irrelevant" size variables occurring in ϕ_2 through existential quantification. This set of quantified size variables is denoted by the symbol X.

The "irrelevant" size variables occurring in ϕ_2 are generated when the size of the abstraction body is inferred; they will not be referenced outside the abstraction body, and therefore need not remain free. For example, if a list is constructed and returned from the abstraction body, size variables will be generated during the list construction. Once constructed, only those size variables associated with the entire list is of relevant outside the abstraction, but not the size variables generated during the construction of this list.

$$\frac{\Gamma \vdash e_1 :: (\sigma, \phi_1) \quad \Gamma \cup \Gamma_1 \vdash e_2 :: (\tau, \phi)}{\Gamma \vdash (\textbf{let } x = e_1 \textbf{ in } e_2) :: (\tau, \phi \wedge \mathcal{C}_{\Gamma_1})} \quad \texttt{(Let)}$$
$$\text{where } \Gamma_1 = \{ x :: (\sigma, \exists X . \phi_1) \} \quad X = \iota(\phi_1) - (\iota(\sigma) \cup \iota(\Gamma))$$

$$\frac{\Gamma \cup \{x :: (\tau, \Phi)\} \vdash e_1 :: (\tau, \Phi) \quad \Gamma \cup \{x :: (\forall t.\tau, \Phi)\} \vdash e_2 :: (\tau, \phi_2)}{\Gamma \vdash (\textbf{letrec } x = e_1 \textbf{ in } e_2) :: (\tau, \phi_2)} \quad \texttt{(Rec)}$$

$$\frac{\Gamma \vdash e_0 :: (\tau_0, \phi_0) \quad \vdash^{\text{Pat}} (P_i, \tau_0) :: \Gamma_i \;\; \forall i \in \{1, \ldots, n\} \quad \Gamma \cup \Gamma_i \vdash e_i :: (\tau_i, \phi_i) \;\; \forall i \in \{1, \ldots, n\}}{\Gamma \vdash \textbf{case } e_0 \textbf{ of } (c_i\, x_{1_i} \ldots x_{m_i} \to e_i)_{i=1}^n :: (\tau', \exists X . \phi)} \quad \texttt{(Case)}$$
$$\text{where } X = \bigcup_{i=1}^n \{\iota(\phi_i') - (\iota(\tau_0) \cup \iota(\Gamma))\} \quad \phi = \phi_0 \wedge \bigvee_i ((eqC\; \tau'\; \tau_i) \wedge \phi_i') \quad \tau' = \alpha\, \tau_1 \,; \quad \phi_i' = \phi_i \wedge \mathcal{C}_{\Gamma_i}$$

$$\frac{\Gamma \vdash e :: (\tau_1 \to \tau_2, \phi)}{\Gamma \vdash e :: (\tau_1 \to \tau_2, \Lambda_S X . \phi)} \quad \texttt{(Bind)}$$
$$\text{where } X = \iota(\tau_1 \to \tau_2)$$

$$\frac{\Gamma \vdash e :: (\tau_1 \to \tau_2, \Lambda_S X . \phi)}{\Gamma \vdash e :: (\tau_1' \to \tau_2', \phi')} \quad \texttt{(Unbind)}$$
$$\text{where } (\tau_1' \to \tau_2', \phi') = \alpha(\tau_1 \to \tau_2, \phi)$$

Figure 6. Rules for Checking Sized Type — Part II

The rule `(App)` specifies that the constraint for a function application is formed by combining the abstraction constraint and the argument constraint. The combination is accomplished by identifying the size variables of the abstraction input and the argument. The term $\mathcal{CT}(\Gamma, (\tau_1, \phi_1), (\tau_2, \phi_2))$ is a *consistency test*. This ensures that the size of the actual argument applied satisfies the sized constraint stipulated by the function's formal. It is defined as follows:

$$
\begin{array}{lcl}
\mathcal{CT}(\Gamma,\, (\tau_1, \phi_1),\, (\tau_2,\, \phi_2)) & = & \psi_2 \sqsubseteq_{\mathbf{P}} \psi_1 \\
\quad \textit{where } \psi_1 & = & \exists\, X \,.\, (\phi_1 \wedge (\mathit{eqC}\ \ \iota(\tau_1)\ \ Z)) \\
\qquad\qquad \psi_2 & = & \exists\, Y \,.\, (\phi_2 \wedge \mathcal{C}_\Gamma \wedge (\mathit{eqC}\ \ \iota(\tau_2)\ \ Z)) \\
\qquad\qquad X & = & \iota(\phi_1)\,;\ \ Y \ = \ \iota(\phi_2)\,;\ \ Z \ = \ \iota(\alpha(\tau_1))
\end{array}
$$

For a **let**-expression, we first infer the sized type (σ, ϕ_1) of its local declaration. We then existentially quantify the "irrelevant" size variables occurring in ϕ_1, and use the result to infer the sized type of the body. Finally, the fact that the size of the body relies on the knowledge about the size of the local declaration is made explicit by including the local constraints in the result.

Just like the case for inferring the size of an abstraction, the "irrelevant" size variables refer to those generated in ϕ_1, and will not be used outside the local declaration.

In the rule `(Rec)`, we infer the sized type of a local recursive function. This is accomplished via fixed-point computation, which will be elaborated in Section 7. We then use the result about the local declaration to infer ϕ_2, the size of the **letrec**-body. In contrast to the rule `(Let)`, it is not necessary to include the size of local declaration in the final result, because this size information will be included when the local function is used in the body.

We discuss the inference of **case**-expressions in the following 4 paragraphs. To begin with, we infer the sized type of the test; the inference result is then used in the inference of the branches. At each branch, we build a local assumption, and then infer the size of this branch under this assumption. The final size of a branch is the conjunction of the size inferred and the constraints in the assumption.

The constraints produced at all branches are collected via disjunction to obtain a constraint for the **case**-expression. Now, the size variables generated at each of the branches become irrelevant, and we therefore existentially quantify them. These variables include those generated from the patterns, as well as those generated when inferring the branches.

Note that the contextual constraint in each branch are included in the final result. This is useful as pattern matching reveals some information about the test. For example, consider the following expression:

$$
\begin{array}{lll}
\textbf{case}\ \ i\ \ \textbf{of} & 0 & \rightarrow\ \ 1 \\
 & (n+1) & \rightarrow\ \ 2
\end{array}
$$

Conventional type inference will tell us the $i\ ::\ \mathit{Int}$. However, the pattern used in the **case** tells us that i is expected to be non-negative. This information is captured in the local environment of the second branch, which states that the size of i is one larger than that of n, and the size of n is non-negative. However, this fact is not used by the expression in any of the branches, and therefore it will only be captured by including the contextual constraint.

We require that patterns in a **case**-expression be mutually exclusive and exhaustive, and employ a set of inference rules called $\vdash^{\text{Pat}}$ to compute the sized types of each pattern. Figure 7 shows the rules for pattern matching. The function *newVar* introduces a new variable from **Var**, which is then bound to a size type to form a type assumption.

$$\frac{}{\begin{array}{l}\vdash^{\text{Pat}} (\mathit{True}, \mathit{Bool}^i) :: \\ \quad \{\, x :: (\mathit{Bool}^i,\ \mathit{chk}_i\ (i = 1))\,\} \\ \quad \text{where} \quad newVar\ x\end{array}} \qquad (\mathtt{PAT:Bool_1})$$

$$\frac{}{\begin{array}{l}\vdash^{\text{Pat}} (\mathit{False}, \mathit{Bool}^i) :: \\ \quad \{\, x :: (\mathit{Bool}^i,\ \mathit{chk}_i\ (i = 0))\,\} \\ \quad \text{where} \quad newVar\ x\end{array}} \qquad (\mathtt{PAT:Bool_2})$$

$$\frac{}{\vdash^{\text{Pat}} (x, \tau) :: \{\, x :: (\tau, \mathit{True})\,\}} \qquad (\mathtt{PAT:Var})$$

$$\frac{}{\begin{array}{l}\vdash^{\text{Pat}} (\mathit{Nil}, [\tau]^i) :: \{\, x :: ([\tau]^i,\ \mathit{chk}_i\ (i = 0))\,\} \\ \quad \text{where} \quad newVar\ x\end{array}} \qquad (\mathtt{PAT:List_1})$$

$$\frac{\begin{array}{c}\vdash^{\text{Pat}} (P_1, \tau') :: \Gamma_1 \\ \vdash^{\text{Pat}} (P_2, [\tau'']^j) :: \Gamma_2\end{array}}{\begin{array}{l}\vdash^{\text{Pat}} (P_1 : P_2, [\tau]^i) :: \{\, x :: ([\tau]^i, \phi)\} \\ \begin{array}{lll}\text{where}\ \phi & = & (\mathit{chk}_i\ (\mathit{chk}_j\ (j + 1 = i))) \\ & & \wedge\ (((\mathit{eqC}\ \tau\ \tau')\ \wedge\ \mathcal{C}_{\Gamma_1}) \vee \\ & & \quad ((\mathit{eqC}\ \tau\ \tau'')\ \wedge\ \mathcal{C}_{\Gamma_2})) \\ \tau' & = & \alpha\,(\tau) \\ [\tau'']^j & = & \alpha([\tau']^i)\ ;\ newVar\ x\end{array}\end{array}} \qquad (\mathtt{PAT:List_2})$$

$$\frac{}{\begin{array}{l}\vdash^{\text{Pat}} (0, \mathit{Int}^i) :: \{\, x :: (\mathit{Int}^i,\ \mathit{chk}_i\ (i = 0))\,\} \\ \quad \text{where} \quad newVar\ x\end{array}} \qquad (\mathtt{PAT:Int_1})$$

$$\frac{\vdash^{\text{Pat}} (n, \mathit{Int}^j) :: \Gamma}{\begin{array}{l}\vdash^{\text{Pat}} (n+1, \mathit{Int}^i) :: \{\, x :: (\mathit{Int}^i, \phi)\} \\ \begin{array}{lll}\text{where}\ \phi & = & \mathit{chk}_i\ (\mathit{chk}_j\ ((j + 1 = i) \wedge \\ & & \quad (j \geq 0)))\ \wedge\ \mathcal{C}_{\Gamma}\ ; \\ \mathit{Int}^j & = & \alpha\,(\mathit{Int}^i); \quad newVar\ x\end{array}\end{array}} \qquad (\mathtt{PAT:Int_2})$$

Figure 7. Type Rules for Pattern Matching

Going back to the discussion of the type rules, we now discuss the last two rules shown in Figure 6. The binding rule `(Bind)` is only applied to the constraint of λ-abstraction. It parameterizes the constraint with the size variables associated with the annotated type of the lambda abstraction. On the other hand, the unbinding rule `(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 only when the associated lambda expression is applied.

Note that the binding rule applies to all λ-abstractions encountered, and the unbinding rule applies to all function applications. This is different from the use of generalization/instantiation rules used in polymorphic type inference, and the reason for being different is because each function application will have different sets of arguments and result, and these need to be captured by different sets of size variables.

Since both `(Bind)` and `(Unbind)` are used in fixed places of a program during sized-type inference, our rules can be made syntax-directed.

5.3. Soundness

Soundness of the sized-type system is defined with respect to the semantics function $\mathcal{E}$ defined in Section 3 and the size function $\mathcal{S}$ given in Definition 2.

DEFINITION 3 (Satisfiability). *Given a denotational value d of an annotated type σ, we say d* satisfies *a constraint ϕ* under *σ if $\mathcal{S}(d :: \sigma) \sqsubseteq_{\mathbf{P}} \exists X . \phi$, where $X = \iota(\phi) - \iota(\sigma)$.*

LEMMA 3. *Satisfiability is admissible.*

Proof By the continuity of function $\mathcal{S}$ (Lemma 1). □

We also extend the satisfiability to work with parameterized formula, instead of formula. Given $\Phi = \Lambda_S X . \phi$, we say "$d$ satisfies Φ under σ" if d satisfies ϕ under σ.

When an assumption set Γ is given, and ϕ may contain some "free variables" which are further constrained by this assumption, we need to extend the satisfiability definition to Γ.

Given an assumption set Γ and a denotational environment ρ, we say ρ *satisfies* Γ if

$$\begin{array}{l} \mathcal{S}(d_1 :: \sigma_1, \ldots, d_n :: \sigma_n) \sqsubseteq_{\mathbf{P}} \exists X . \mathcal{C}_\Gamma \\ \quad \textit{where } X = \iota(\mathcal{C}_\Gamma) - (\bigcup_{i=1}^{n} \{\iota(\sigma_i)\}); \\ \qquad\qquad \rho = \{x_i :: d_i\}_{i=1}^{n}; \quad \Gamma = \{x_i :: (\sigma_i, \phi_i)\}_{i=1}^{n}. \end{array}$$

COROLLARY 4. *Satisfiability over an assumption set is admissible.*

Proof By Corollary 2 and the continuity of an assumption set. □

LEMMA 5 (Soundness of Pattern Matching). *If* $\vdash^{\text{Pat}} (P,\tau) :: \Gamma$, *then for any denotational value* d *of annotated type* τ, *if* d *matches the pattern* P, *then* $\mathcal{S}(d :: \tau) \sqsubseteq_{\mathbf{P}} \exists X . \mathcal{C}_\Gamma$ *where* $X = \iota(\mathcal{C}_\Gamma)$ - $\iota(\tau)$.

Proof The proof is by induction on the height of the type derivation. We prove the case of pattern matching a list. Cases of pattern matching other constructs can be proven similarly.

We show the soundness of rule (PAT:List$_1$), let d be a denotation of an annotated type $[\tau]^i$. Suppose that d pattern matches *Nil*, then d must be (the denotational value) *Nil*. We have $\Gamma = \{x :: ([\tau]^i, chk_i\ (i=0))\}$. So, $\mathcal{C}_\Gamma = chk_i\ (i=0)$. Now, $\mathcal{S}(Nil :: [\tau]^i) = chk_i\ (i=0)$ $=_{\mathbf{P}} \mathcal{C}_\Gamma$. Since $\iota(\mathcal{C}_\Gamma) - \iota(\tau) = \emptyset$, we arrive at the proof.

To prove the soundness of the rule (PAT:List$_2$), let d be a denotation of an annotated type $[\tau]^i$. Suppose that d pattern matches the pattern $(P_1 : P_2)$, then d must be a non-empty list. Let $d = [d_1,\ldots,d_n]$ for some $n > 0$, then d_1 matches P_1, and $[d_2,\ldots,d_n]$ matches P_2. Assuming that $\vdash^{\text{Pat}} (P_1,\tau') :: \Gamma_1$ and $\vdash^{\text{Pat}} (P_2,[\tau'']^j) :: \Gamma_2$. From the induction hypothesis, we have

$$\mathcal{S}(d_1 :: \tau') \sqsubseteq_{\mathbf{P}} \exists X . \mathcal{C}_{\Gamma_1} \textit{ where } X = \iota(\mathcal{C}_{\Gamma_1}) - \iota(\tau')$$

and

$$\begin{aligned}\mathcal{S}([d_2,\ldots,d_n] :: [\tau'']^j) = {} & (chk_j\ (j=n-1)) \wedge \\ & \textstyle\bigvee_{k=2}^{n} (\mathcal{S}(d_k :: \tau'')) \sqsubseteq_{\mathbf{P}} \exists Y . \mathcal{C}_{\Gamma_2} \\ \textit{where } Y = {} & \iota(\mathcal{C}_{\Gamma_2}) - \iota(\tau'').\end{aligned}$$

$$\begin{aligned}
&\textit{So, } \mathcal{S}([d_1,\ldots,d_n] :: [\tau]^i) \\
= \quad & chk_i\ (i = n) \wedge \textstyle\bigvee_{k=1}^{n} \mathcal{S}(d_k :: \tau) \\
=_{\mathbf{P}} \quad & \exists j . chk_i\ (chk_j\ (j+1 = i)) \wedge chk_j\ (j=n-1) \wedge \\
& \qquad (\mathcal{S}(d_1 :: \tau) \vee \textstyle\bigvee_{k=2}^{n} \mathcal{S}(d_k :: \tau)) \\
=_{\mathbf{P}} \quad & \exists V . chk_i(chk_j\ (j+1 = i)) \wedge \\
& \qquad (((eqC\ \tau\ \tau') \wedge chk_j\ (j=n-1) \wedge \mathcal{S}(d_1 :: \tau')) \vee \\
& \qquad ((eqC\ \tau\ \tau'') \wedge chk_j\ (j=n-1) \wedge \textstyle\bigvee_{k=2}^{n} \mathcal{S}(d_k :: \tau''))) \\
& \quad \textit{where } V = \iota(\tau') \cup \iota(\tau'') \cup \{j\} \\
\sqsubseteq_{\mathbf{P}} \quad & \exists V . chk_i\ (chk_j\ (j+1 = i)) \wedge (((eqC\ \tau\ \tau') \wedge \exists X . \mathcal{C}_{\Gamma_1}) \vee \\
& \qquad ((eqC\ \tau\ \tau'') \wedge \exists Y . \mathcal{C}_{\Gamma_2})) \\
& \quad \textit{where } V = \iota(\tau') \cup \iota(\tau'') \cup \{j\} \\
& \qquad\qquad X = \iota(\mathcal{C}_{\Gamma_1}) - \iota(\tau') ;\ Y = \iota(\mathcal{C}_{\Gamma_2}) - \iota(\tau'') \\
\sqsubseteq_{\mathbf{P}} \quad & \textit{(Since variables in } X, Y \textit{ do not occur anywhere else in the formula)} \\
& \exists Z . chk_i\ (chk_j\ (j+1 = i)) \wedge (((eqC\ \tau\ \tau') \wedge \mathcal{C}_{\Gamma_1}) \vee \\
& \qquad ((eqC\ \tau\ \tau'') \wedge \mathcal{C}_{\Gamma_2})) \\
& \quad \textit{where } Z = V \cup X \cup Y ;\ V = \iota(\tau') \cup \iota(\tau'') \cup \{j\} \\
& \qquad\qquad X = \iota(\mathcal{C}_{\Gamma_1}) - \iota(\tau') ;\ Y = \iota(\mathcal{C}_{\Gamma_2}) - \iota(\tau'')
\end{aligned}$$

$$
\begin{aligned}
=_{\mathbf{P}} \quad & \textit{(Since } V \cup X \cup Y = \iota(\mathcal{C}_\Gamma) - \iota(\tau)\textit{)} \\
& \exists Z.\ \mathcal{C}_\Gamma \\
& \quad \textit{where } Z = \iota(\mathcal{C}_\Gamma) - \iota(\tau) \\
& \qquad\qquad \Gamma = \{ x :: ([\tau]^i, \phi)\} \\
& \qquad\qquad \phi = chk_i\ (chk_j\ (j + 1 = i)) \wedge (((eqC\ \tau\ \tau') \wedge \mathcal{C}_{\Gamma_1}) \\
& \qquad\qquad\qquad \vee ((eqC\ \tau\ \tau'') \wedge \mathcal{C}_{\Gamma_2}))
\end{aligned}
$$

□

THEOREM 6 (Soundness). *Given an expression e of type $\underline{\sigma}$, for some σ. If $\vdash e :: (\sigma, \phi)$, then for any monomorphic instance of e with denotation d, d satisfies ϕ under σ.*

Proof In the proof, we assume that the denotation of the expression is not $\perp$. If it were $\perp$, we would have $\mathcal{S}(\perp) = \mathit{False}$, and the result would hold vacuously.

The proof is by induction on the height of the type derivation. The induction hypothesis is as follows:

If $\Gamma \vdash e :: (\sigma, \phi)$, then for all ρ such that ρ satisfies Γ and $\mathcal{E} [\![e]\!] \rho \neq \perp$,

$$\mathcal{S}(\mathcal{E}[\![e]\!]\rho :: \sigma, d_1 :: \sigma_1, \ldots, d_n :: \sigma_n) \sqsubseteq_{\mathbf{P}} \exists X.\ \phi \wedge \mathcal{C}_\Gamma,$$

where $\rho = \{x_i :: d_i\}_{i=1}^n$, *and*

$$X = \iota(\phi \wedge \mathcal{C}_\Gamma) - (\iota(\tau) \cup \textstyle\bigcup_{i=1}^n \{\iota(\sigma_i)\}).$$

The case of inferring the size of self-recursive function requires proving that the validity of fixed-point computation, and we leave it until Section 7 (Theorem 7). The proofs for all the other cases are standard. Here, we present the **let** case in detail:

Following the notation used in the type rule for **let**, assume $\Gamma \vdash (\textbf{let}\ x = e_1\ \textbf{in}\ e_2) :: (\tau, \phi \wedge \mathcal{C}_{\Gamma_1})$ and ρ satisfies Γ, for some $\rho = \{x_i :: d_i\}_{i=1}^n$. We wish to show that:

$$
\begin{aligned}
& \mathcal{S}(\mathcal{E} [\![(\textbf{let}\ x = e_1\ \textbf{in}\ e_2)]\!]\ \rho :: \tau, d_1 :: \sigma_1, \ldots, d_n :: \sigma_n) \\
& \qquad\qquad\qquad\qquad \sqsubseteq_{\mathbf{P}} \exists X.\ \phi \wedge \mathcal{C}_{\Gamma_1} \wedge \mathcal{C}_\Gamma \\
& \textit{where } X = \iota(\phi \wedge \mathcal{C}_{\Gamma_1} \wedge \mathcal{C}_\Gamma) - (\iota(\tau) \cup \textstyle\bigcup_{i=1}^n \{\iota(\sigma_i)\})
\end{aligned}
\qquad (\star)
$$

Equivalently,

$$
\begin{aligned}
& \mathcal{S}(\mathcal{E} [\![e_2]\!]\rho[u/x] :: \tau, d_1 :: \sigma_1, \ldots, d_n :: \sigma_n) \sqsubseteq_{\mathbf{P}} \exists X.\ \phi \wedge \mathcal{C}_{\Gamma_1} \wedge \mathcal{C}_\Gamma \\
& \quad \text{where } u = \mathcal{E} [\![e_1]\!]\ \rho \\
& \qquad\qquad X = \iota(\phi \wedge \mathcal{C}_{\Gamma_1} \wedge \mathcal{C}_\Gamma) - (\iota(\tau) \cup \textstyle\bigcup_{i=1}^n \{\iota(\sigma_i)\})
\end{aligned}
$$

ρ satisfies Γ

$$
\begin{aligned}
\Rightarrow \quad & \textit{(induction hypothesis for } \Gamma \vdash e_1 :: (\sigma, \phi_1)\textit{)} \\
& \mathcal{S}(\mathcal{E} [\![e_1]\!]\ \rho :: \sigma, d_1 :: \sigma_1, \ldots, d_n :: \sigma_n) \sqsubseteq_{\mathbf{P}} \exists Y.\ \phi_1 \wedge \mathcal{C}_\Gamma \\
& \qquad\quad \textit{where } Y = \iota(\phi_1 \wedge \mathcal{C}_\Gamma) - (\iota(\sigma) \cup \textstyle\bigcup_{i=1}^n \{\iota(\sigma_i)\}) \\
& \qquad\qquad\qquad = \quad \textit{(Since } \iota(\Gamma) = \iota(\mathcal{C}_\Gamma) \cup \textstyle\bigcup_{i=1}^n \{\iota(\sigma_i)\}\textit{)} \\
& \qquad\qquad\qquad\quad (\iota(\phi_1) \cup \iota(\Gamma)) - (\iota(\tau) \cup \textstyle\bigcup_{i=1}^n \{\iota(\sigma_i)\})
\end{aligned}
$$

$$
\begin{array}{l}
\Rightarrow \quad \textit{(Since } (\bigcup_{i=1}^{n} \{\iota(\sigma_i)\}) \subseteq \iota(\Gamma)\textit{, and therefore } X_1 \subseteq Y\textit{)} \\
\quad \mathcal{S}(\,\mathcal{E}\,[\![\, e_1 \,]\!]\, \rho \,::\, \sigma\,,\, d_1 \,::\, \sigma_1, \ldots, d_n \,::\, \sigma_n) \sqsubseteq_{\mathbf{P}} \exists\, Y_1 \,.\, \exists\, X_1 \,.\; \phi_1 \wedge \mathcal{C}_\Gamma \\
\qquad\qquad \textit{where } \; Y_1 \,=\, Y \,-\, X_1 \,; \;\; X_1 \,=\, \iota(\phi_1) \,-\, (\iota(\sigma) \,\cup\, \iota(\Gamma))
\end{array}
$$

$$
\begin{array}{rl}
\text{Now,} \;\; Y_1 & = Y \,-\, X_1 \\
& = ((\iota(\phi_1) \,\cup\, \iota(\Gamma)) \,-\, (\iota(\tau) \,\cup\, \bigcup_{i=1}^{n} \{\iota(\sigma_i)\})) \,-\, X_1 \\
& = \quad \textit{(Since } \iota(\Gamma) \,=\, \iota(\,\mathcal{C}_\Gamma) \,\cup\, \bigcup_{i=1}^{n} \{\iota(\sigma_i)\}\textit{)} \\
& \quad ((\iota(\phi_1) \,\cup\, \iota(\,\mathcal{C}_\Gamma)) \,-\, (\iota(\tau) \,\cup\, \bigcup_{i=1}^{n} \{\iota(\sigma_i)\})) \,-\, X_1 \\
& = \quad \textit{(Since } \iota(\,\mathcal{C}_\Gamma) \,\cap\, X_1 \,=\, \emptyset\textit{)} \\
& \quad ((\iota(\exists\, X \,.\, \phi_1) \,\cup\, \iota(\,\mathcal{C}_\Gamma)) \,-\, (\iota(\tau) \,\cup\, \bigcup_{i=1}^{n} \{\iota(\sigma_i)\}))
\end{array}
$$

Hence, we have

$$
\begin{array}{l}
\mathcal{S}(\,\mathcal{E}\,[\![\, e_1 \,]\!]\, \rho \,::\, \sigma\,,\, d_1 \,::\, \sigma_1, \ldots, d_n \,::\, \sigma_n) \sqsubseteq_{\mathbf{P}} \exists\, Y_1 \,.\, (\exists\, X_1 \,.\; \phi_1) \,\wedge\, \mathcal{C}_\Gamma \\
\qquad \textit{where } \; Y_1 \,=\, ((\iota(\exists\, X \,.\, \phi_1) \,\cup\, \iota(\,\mathcal{C}_\Gamma)) \,-\, (\iota(\tau) \,\cup\, \bigcup_{i=1}^{n} \{\iota(\sigma_i)\})) \\
\Rightarrow \quad \textit{(Definition of satisfiability)} \\
\quad \rho[u/x] \text{ satisfies } (\Gamma \,\cup\, \Gamma_1) \quad \text{where } \Gamma_1 = \{\, x \,::\, (\sigma, \exists\, X_1 \,.\, \phi_1) \,\} \text{ and} \\
\qquad\qquad\qquad\qquad\qquad\qquad u \;\; = \mathcal{E}\,[\![\, e_1 \,]\!]\, \rho \\
\Rightarrow \quad \textit{(induction hypothesis for } \Gamma \,\cup\, \Gamma_1 \,\vdash\, e_2 \,::\, (\tau,\, \phi)\textit{)} \\
\quad \mathcal{S}\,(\,\mathcal{E}\,[\![\, e_2 \,]\!]\, \rho[u/x] \,::\, \tau,\, u \,::\, \sigma, d_1 \,::\, \sigma_1, \ldots, d_n \,::\, \sigma_n) \\
\qquad\qquad\qquad\qquad \sqsubseteq_{\mathbf{P}} \exists\, Z \,.\; \phi \wedge \mathcal{C}_{\Gamma \,\cup\, \Gamma_1} \\
\qquad \textit{where } \; Z \,=\, \iota(\phi \wedge \mathcal{C}_{\Gamma \,\cup\, \Gamma_1}) \,-\, (\iota(\tau) \,\cup\, \iota(\sigma) \,\cup \bigcup_{i=1}^{n} \{\iota(\sigma_i)\}) \\
\Leftrightarrow \quad \textit{(Associativity of Conjunction)} \\
\quad \mathcal{S}\,(\,\mathcal{E}\,[\![\, e_2 \,]\!]\, \rho[u/x] \,::\, \tau,\, u \,::\, \sigma, d_1 \,::\, \sigma_1, \ldots, d_n \,::\, \sigma_n) \\
\qquad\qquad\qquad\qquad \sqsubseteq_{\mathbf{P}} \exists\, Z \,.\; \phi \,\wedge\, \mathcal{C}_{\Gamma_1} \wedge \mathcal{C}_\Gamma \\
\qquad \textit{where } \; Z \,=\, \iota(\phi \,\wedge\, \mathcal{C}_{\Gamma_1} \wedge \mathcal{C}_\Gamma) \\
\qquad\qquad\qquad\qquad -\, (\iota(\tau) \,\cup\, \iota(\sigma) \,\cup \bigcup_{i=1}^{n} \{\iota(\sigma_i)\}) \\
\Leftrightarrow \quad \textit{(Definition of } \mathcal{S} \textit{ on } n\textit{-tuple)} \\
\quad \mathcal{S}(u \,::\, \sigma) \,\wedge\, \mathcal{S}\,(\,\mathcal{E}\,[\![\, e_2 \,]\!]\, \rho[u/x] \,::\, \tau, d_1 \,::\, \sigma_1, \ldots, d_n \,::\, \sigma_n) \\
\qquad\qquad\qquad\qquad\qquad \sqsubseteq_{\mathbf{P}} \exists\, Z \,.\; \phi \,\wedge\, \mathcal{C}_{\Gamma_1} \wedge \mathcal{C}_\Gamma \\
\Rightarrow \quad \textit{(Existential introduction)} \\
\quad \exists\, X \,.\, (\,\mathcal{S}(u \,::\, \sigma) \,\wedge\, \mathcal{S}\,(\,\mathcal{E}\,[\![\, e_2 \,]\!] \rho[u/x] \,::\, \tau,\, d_1 \,::\, \sigma_1, \ldots, d_n \,::\, \sigma_n)) \\
\qquad\qquad\qquad\qquad \sqsubseteq_{\mathbf{P}} \exists\, X \,.\, \exists\, Z \,.\; \phi \,\wedge\, \mathcal{C}_{\Gamma_1} \wedge \mathcal{C}_\Gamma \\
\qquad \textit{where } \; X \,=\, \iota(\sigma) \\
\Leftrightarrow \quad \textit{(since } X \,\cap\, \iota(\cup_{i=1}^{n} \sigma_i) \,=\, \emptyset\textit{)} \\
\quad (\exists\, X \,.\; \mathcal{S}(u \,::\, \sigma)) \,\wedge\, \mathcal{S}\,(\,\mathcal{E}\,[\![\, e_2 \,]\!] \rho[u/x] \,::\, \tau,\, d_1 \,::\, \sigma_1, \ldots, d_n \,::\, \sigma_n) \\
\qquad\qquad\qquad\qquad \sqsubseteq_{\mathbf{P}} \exists\, X \,.\, \exists\, Z \,.\; \phi \,\wedge\, \mathcal{C}_{\Gamma_1} \wedge \mathcal{C}_\Gamma
\end{array}
$$

By Property 1, we have either $\exists\, X \,.\; \mathcal{S}(u \,::\, \sigma) \,=\, \textit{False}$ or $\exists\, X \,.\; \mathcal{S}(u \,::\, \sigma) \,=\, \textit{True}$. In the former case, the above inequality is vacuously true, and we have attained the proof. In the latter case, we get

$$
\begin{array}{l}
\mathcal{S}\,(\,\mathcal{E}\,[\![\, e_2 \,]\!]\, \rho[u/x] \,::\, \tau, d_1 \,::\, \sigma_1, \ldots, d_n \,::\, \sigma_n) \\
\qquad\qquad\qquad\qquad \sqsubseteq_{\mathbf{P}} \exists\, X \,.\, \exists\, Z \,.\; \phi \,\wedge\, \mathcal{C}_{\Gamma_1} \wedge \mathcal{C}_\Gamma \\
\qquad\qquad =_{\mathbf{P}} \exists\, W . \, \phi \,\wedge\, \mathcal{C}_{\Gamma_1} \wedge \mathcal{C}_\Gamma \;\; \textit{where } W \,=\, Z \,\cup\, X
\end{array}
$$

$$
\begin{array}{rl}
\text{Lastly,} \;\; W & = \;\; Z \,\cup\, X \\
& = \;\; (\iota(\phi \,\wedge\, \mathcal{C}_{\Gamma_1} \wedge \mathcal{C}_\Gamma) \;\, -\, (\iota(\tau) \,\cup\, \iota(\sigma) \,\cup \bigcup_{i=1}^{n} \{\iota(\sigma_i)\})) \,\cup\, \iota(\sigma) \\
& = \;\; \iota(\phi \,\wedge\, \mathcal{C}_{\Gamma_1} \wedge \mathcal{C}_\Gamma) \,-\, (\iota(\tau) \,\cup\, \cup \bigcup_{i=1}^{n} \{\iota(\sigma_i)\})
\end{array}
$$

Hence, we obtain the inequality expressed in $(\star)$. $\square$

6. Size Inference for Non-Recursive Functions

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

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

The function f defined above depends on *append*. The sized type of *append* is assumed known.

The annotated type of f is $Bool^i \rightarrow [a]^j \rightarrow [a]^k$, for some size variables $i,\ j,\ k\ \in \mathbf{V}$, and type variable a. Consider the process of inferring the type of the right-hand side of the definition of f. The assumption at this point consists of:

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

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 variable name used to refer to this pattern. The size constraint for this branch is now the constraint of xs: $\Gamma \cup \Gamma_1 \vdash xs :: ([a]^j,\ j \geq 0)$.

In the *True* branch, we obtain the local 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:

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

We begin by deducing the sized type for (*append xs*). Here, we need to perform a consistency test for the application argument. This is validated with the help of the Omega Calculator.

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

$$(\exists\, m_1\ .\,(m_1 \geq 0 \wedge n_1 \geq 0 \wedge p_1 = m_1 + n_1) \wedge m_1 = j) \wedge j \geq 0$$

Next, we deduce the constraint for ((*append xs*) *xs*). After consistency test, the constraint is:

$$\begin{array}{l} (\exists\, n_1\ .\,(\exists\, m_1\ .\ (m_1 \geq 0 \wedge n_1 \geq 0 \wedge p_1 = m_1 + n_1) \\ \qquad\qquad \wedge\ m_1 = j) \wedge j \geq 0 \wedge n_1 = j) \wedge j \geq 0 \end{array} \qquad (\psi_2)$$

Generating Constraints for Case-Expression To combine both the *True* and *False* branches, we introduce a common new size variable s for both branches and obtain the following constraint for the **case** expression:

$$\exists\, p_1 \;.\; (s = j \wedge j \geq 0 \wedge i = 0) \vee (s = p_1 \wedge \psi_2 \wedge i = 1) \qquad (\psi_3)$$

As s captures the size of both branches, which is the result of evaluating the function f, we equate it to k. Lastly, we introduce into ψ_3 the inherent 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:

$$\Lambda_S\, i,j,k \;.\; (i = 0 \wedge j \geq 0 \wedge k = j) \;\vee (i = 1 \wedge j \geq 0 \wedge k = j + j)$$

Remark Note that the use of type variables for a polymorphic function does not incur a size constraint. This greatly simplifies the type-inference process, as it removes the need to compute any size constraint about the list elements. In the case of monomorphic function over lists or nested lists, we will, in principle, wish to compute both the sizes of the lists and their elements/nested lists. This may result in time-consuming computation. In practice, we can choose to assign a size variable from $\mathbf{V}$ to the list constructor, but assign $\top$ to its substructure – the list elements or nested lists. This assignment can either be passed as a parameter to our type system or be annotated by the user on their programs. During type inference, any term in **BExp** that involves $\top$ will be replaced by *True*. While the resulting sized type is still safe, this will effectively eliminate size-type inference for substructures, and can help simplify the formula to be fed to the Omega Calculator. Consequently, the cost of conducting type inference can be controlled by the "substructure depth" that is to be analyzed.

7. Size Inference for Recursive Functions

In the type rule for `(Rec)`, we infer the sized type of a recursive function by computing a safe approximation of the *fixed point* of the sized type of the function body. This involves two steps:

1. Deriving the constraint associated with the recursive function body.

2. Computing a safe approximation of the fixed point of the constraint derived in the first step.

7.1. Deriving Constraint of Recursive Function Body

The objective of the first step is to calculate the parent/child relationship of all the recursive calls as exhibited in the function body. For example, consider performing size inference on the definition of *append* in Section 1. We refer the left-hand side of the definition, *append xs ys*, as the *parent call*, and the recursive call *append* xs' *ys* in the right-hand side of the definition as the *child call*. The parent/child relationship is thus the change in both the argument sizes and the size of the application result.

To derive this relation, we create *unique* size variables for the recursive function at each of its recursive-call sites. At each of these call sites, we derive a constraint that captures the call context. Again, consider the definition of *append*. Let the annotated type of *append* be $[t]^m \rightarrow [t]^n \rightarrow [t]^p$. During the inference of the recursive call to *append*, we instantiate the annotated type of *append* to $[t]^{i_1} \rightarrow [t]^{i_2} \rightarrow [t]^{i_3}$, where i_1, i_2 and i_3 are new size variables. A constraint associated with the function body can then be inferred by our type rules, and simplified by the Omega Calculator to the following disjunctive normal form:

$$\begin{array}{l}(m = 0 \wedge n \geq 0 \wedge n = p) \vee \\ (m = i_1 + 1 \wedge i_1 \geq 0 \wedge i_2 = n \wedge n \geq 0 \wedge p = i_3 + 1 \wedge i_3 \geq 0)\end{array}$$

Intuitively, this constraint captures the result of a single-step unfolding of a recursive call. We therefore call it *single-step constraint*. It consists of two components, separated by a disjunctive operator ($\vee$):

1. *Recursive-call constraint* : This expresses the relationship between newly instantiated size variables of the function at its recursive-call sites and those of its parameters and results.

2. *Termination constraint* : This expresses the termination condition of the recursive call.

For the above single-step constraint, the recursive-call constraint is as follows:

$$\begin{array}{rl}U = [m, n, p] \rightarrow [i_1, i_2, i_3] : & m = i_1 + 1 \wedge i_1 \geq 0 \wedge i_2 = n \wedge \\ & n \geq 0 \wedge p = i_3 + 1 \wedge i_3 \geq 0\end{array}$$

whereas the following is the termination constraint:

$$B_I = [m, n, p] : m = 0 \wedge n \geq 0 \wedge n = p$$

How do we derive these two pieces of information? From the single-step constraint, the termination constraint can be retrieved from those disjuncts that do not involve instantiated size variables of recursive calls, and the recursive-call constraint is retrieved from disjuncts that depend on those instantiated size variables. It is clear that the recursive-call constraint contains the parent/child relationship of interest.

In general, our method for deriving parent/child relationship (and recursive-call constraint) works in the presence of partial applications of recursive functions, and multiple recursive calls. To illustrate this, consider the following variant of binomial function bin':

$$\begin{array}{l} bin' :: Int^n \rightarrow Int^k \rightarrow Int^p \\ bin' \; n \; k \;\; = \mathbf{case}\; n \;\mathbf{of}\;\; 0 \rightarrow 1 \\ \qquad\qquad\qquad\qquad\quad m+1 \rightarrow \mathbf{if}\; k \;\leq\; 0 \,||\, k \;\geq\; n \;\mathbf{then}\; 1 \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else\;let}\; f \;\; = \; bin' \; m \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\;\; \mathbf{in}\;\; (f \;\; (k-1)) \;\; + \;\; (f \;\; k) \end{array}$$

Syntactically, $(bin' \; m)$ appears to be the only recursive call in the definition. Semantically, however, both $(f \; (k-1))$ and $(f \; k)$ are real recursive calls the sizes of which are used to develop the parent/child relation.

We shall demonstrate that, by consistently assigning new size variables to all functions involved – either directly or indirectly – in recursive applications, and keeping track of the final sets of size variables used in the final (fully-applied) calls, we can derive the desired relationship.

Let's assume that, during size inference, the annotated type of bin' as appeared in the type assumption is $Int^a \;\rightarrow\; Int^b \;\rightarrow Int^c$. During the inference of the size of the call $bin' \; m$, size variables of bin' are renamed (according to the type rules) to $Int^{a_1} \rightarrow \; Int^{b_1} \;\rightarrow\; Int^{c_1}$. Assuming that the size variable of m is denoted by m too, the result of inferring the size of $bin' \; m$, and subsequently f, is as follows:

$$\begin{array}{lll} (bin' \; m) & :: & (Int^{b_1} \;\rightarrow\; Int^{c_1}, \; m \; = \; a_1) \\ f & :: & (Int^{b_1} \;\rightarrow\; Int^{c_1}, \; \Lambda_S \; b_1 \,.\, \Lambda_S \; c_1 \,.\, (m \; = \; a_1)) \end{array}$$

Next, for each application of f in the **let**-body, we instantiate f with new size variables. For $f \; (k-1)$, we have the annotated type of f being $Int^{b_2} \;\rightarrow\; Int^{c_2}$; for $f \; k$, we have $Int^{b_3} \;\rightarrow\; Int^{c_3}$. Consequently, we obtain

$$\begin{array}{lll} f \; (k-1) & :: & (Int^{c_2}, \; m \; = \; a_1 \; \wedge \; k-1 \; = \; b_2) \\ f \;\; k & :: & (Int^{c_3}, \; m \; = \; a_1 \; \wedge \; k \; = \; b_3) \end{array}$$

Notice that size variables of bin' have been consistently replaced everytime bin' is either directly or indirectly applied. This systematic replacement of size variables for bin' can be described using a tree structure as follows:

$$
\begin{array}{l}
Int^{a} \rightarrow Int^{b} \rightarrow Int^{c} \\
\quad \textit{renamed to } \underline{Int^{a_1}} \rightarrow Int^{b_1} \rightarrow Int^{c_1} \quad \textit{at call } (bin'\ m) \\
\qquad\qquad\qquad\qquad \textit{renamed to } \underline{Int^{b_2}} \rightarrow \underline{Int^{c_2}} \quad \textit{at call } (f\ (k-1)) \\
\qquad\qquad\qquad\qquad \textit{renamed to } \underline{Int^{b_3}} \rightarrow \underline{Int^{c_3}} \quad \textit{at call } (f\ k)
\end{array}
$$

From this tree structure, we can obtain the size variables used by those final recursive calls of bin'. The annotated types associated with these size variables are underlined. Hence, the size variables used by $f\ (k-1)$ are a_1, b_2, and c_2, and that used by $f\ k$ are a_1, b_3 and c_3. In deriving a single-step constraint, we do not existentially quantify these size variables. The constraint resulted from inferring the size of the right-hand side of bin' (and after simplification) is:

$$
\begin{array}{l}
\exists\, m : (n = 0 \wedge p = 1) \vee \\
\qquad (n > 0 \wedge m+1 = n \wedge \\
\qquad\quad ((k \leq 0 \vee k \geq n) \wedge p = 1) \vee \\
\qquad\quad ((k > 0 \wedge k < n) \wedge (m = a_1 \wedge p = c_2 + c_3 \wedge \\
\qquad\qquad\qquad\qquad\qquad (k-1 = b_2 \wedge k = b_3)))
\end{array}
$$

Now that the single-step constraint for bin' has been derived, we need to retrieve the recursive-call constraint. First, as mentioned earlier, we delete all termination constraints from the single-step constraint. This yields:

$$
\begin{array}{l}
\exists\, m : (n > 0 \wedge m+1 = n \wedge \\
\qquad\quad ((k > 0 \wedge k < n) \wedge (m = a_1 \wedge p = c_2 + c_3 \wedge \\
\qquad\qquad\qquad\qquad\qquad (k-1 = b_2 \wedge k = b_3)))
\end{array}
$$

Next, as this constraint includes constraints of multiple recursive calls, we need to determine the relationship between the parent call and each *individual* recursive call, before combining them to form the final recursive-call constraint.

To do that, we create as many copies of the above constraint as there are final recursive calls. For each copy, we capture the constraint relating the parent call and a *single* recursive child call. This is achieved by keeping the size variables of both the parent call and that child call, and existentially quantifying the size variables of all other recursive calls. Let S_1 and S_2 be these two constraints (for the two recursive child calls of bin'). We thus have:

$$
\begin{array}{l}
S_1 = \exists\, b_3, c_3 : \exists\, m : \\
\qquad (n > 0 \wedge m+1 = n \wedge \\
\qquad\quad ((k > 0 \wedge k < n) \wedge (m = a_1 \wedge p = c_2 + c_3 \wedge \\
\qquad\qquad\qquad\qquad\qquad (k-1 = b_2 \wedge k = b_3))) \\
S_2 = \exists\, b_2, c_2 : \exists\, m :
\end{array}
$$

$$
\begin{array}{l}
(n > 0 \wedge m+1 = n \wedge \\
\quad ((k > 0 \wedge k < n) \wedge (m = a_1 \wedge p = c_2 + c_3 \wedge \\
\qquad\qquad\qquad\qquad\qquad (k-1 = b_2 \wedge k = b_3)))
\end{array}
$$

Finally, we form the recursive-call constraint by taking the union of the parent/child relationship of all recursive calls:

$$S_{bin'} = S_1 \vee S_2$$

7.2. Computing a Fixed Point of a Constraint

Once the recursive-call constraint is derived, the next step is to compute a safe approximation of the fixed point of the sized type of the recursive function. The least fixed point of a formula is also called its *transitive closure*, borrowing the terminology from Kelly, Pugh, *et al.* [19]. We shall use the terms least fixed point and transitive closure interchangeably.

Before defining transitive closure of a formula, we define the *formula-composition operation* as follows:

DEFINITION 4 (Formula-Composition Operation). *Let $\Phi = X \rightarrow Y : \phi$ and $\Psi = Z \rightarrow W : \psi$ be two constraints written in relation form. Suppose that X, Y, Z and W are all sets of identical cardinality, then*

$$\Phi \circ \Psi \stackrel{def}{=} X \rightarrow W : (\phi \wedge \psi \wedge (eqC\ Y\ Z))$$

DEFINITION 5 (Transitive Closure of a Formula). *Let $\Phi = X \rightarrow Y : \phi$ be a constraint such that the cardinality of the sets X and Y are identical. The transitive closure of Φ is defined as*

$$\Phi^{+} = \sqcup_{\mathbf{P}\, i>0}\, \Phi^{i} \quad \text{where } \begin{array}{lcl} \Phi^{1} & = & \Phi \\ \Phi^{n+1} & = & \Phi \circ \Phi^{n} \end{array}$$

In principle, the closure operation provided by the Omega Calculator allows us to compute the transitive closure of the size constraint. In practice, however, such operation may fall short in producing the desired transitive closure. There are several reasons for this; one such reason is that the transitive closure of a constraint may not be expressible as a Presburger formula.

We thus devise a generalization technique to compute an upper bound of the transitive closure. The result is a fixed point, but not necessarily the least, of the constraint. We shall describe the generalization technique in Section 9. In this section, we discuss how a fixed point is computed.

A fixed point is computed in two steps:

1. Calculating a *generalized transitive constraint* for the recursive-call constraint.

2. Incorporating termination constraints into the generalized transitive constraint.

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

7.2.1. *Calculating A Generalized Transitive Constraint*
A *generalized transitive constraint*, *gtc* for short, describes a relation (*i.e.*, formula) that holds between a parent call and an arbitrarily nested recursive child call. Given a recursive-call constraint U, a natural candidate for its *gtc* is the transitive closure of U, denoted by U^+. It is a constraint that holds for all recursive calls reachable from the initial call by unfolding, and is characterized by the following important property (adapted from [19]):

PROPERTY 2 (Transitive Closure). *Given two constraints Φ and Ψ such that $\Phi = X \rightarrow Y : \phi$, $\Psi = X \rightarrow Y : \psi$ and X, Y have identical cardinality. If $\Psi \sqsubseteq_{\mathbf{P}} \Phi \sqsubseteq_{\mathbf{P}} \Psi^+$, then the following holds: $\Phi =_{\mathbf{P}} \Psi^+$ if and only if $\Phi \circ \Psi \sqsubseteq_{\mathbf{P}} \Phi$.*

Semantically, U^+ is an invariant about size, capturing the constraint between the size of an initial call and that of an arbitrary recursive call. This is built on the idea of unfolding recursive calls an arbitrary number of times.

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

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

By checking the result of $\texttt{closure}(U)$ against Property 2, we verify that $\texttt{closure}(U) = U^+$.

Techniques for calculating transitive closure have been described in the work on the Omega Calculator [19]. As the transitive-closure computation can be expensive, in practice, it is possible to set a limit on the resources needed by the Omega Calculator to perform transitive closure. Consequently, the `closure` operation may not produce the desired result, and we need to approximate the transitive closure by its upper bound, which we call *generalized transitive constraint*. This *generalization* technique will be described in detail in Section 9. We denote the generalized transitive constraint of a recursive-call constraint U by $U^{\oplus}$.

7.2.2. *Incorporating Termination Constraint*
To completely capture the size of a function, we need to include termination constraints into our final constraint. Let $U^{\oplus} = [a_1, \ldots, a_n] \rightarrow [a'_1, \ldots, a'_n] : \phi$ be the generalized transitive constraint, and $B_I = [a_1, \ldots, a_n] : \psi$ be the termination constraint. There are two scenarios to consider:

1. When the 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 termination constraint. We denote it by B_I.

2. When the arguments of a nested recursive call match a degenerate case, and thus terminate the recursive invocation. The constraint is expressed as follows:

$$\begin{aligned} B_R &= [a_1, \ldots, a_n] : \exists\, b_1, \ldots, b_n \,.\, U^{\oplus}(a_1, \ldots, a_n, b_1, \ldots, b_n) \\ &\qquad\qquad\qquad\qquad\quad \wedge\ B_I(b_1, \ldots, b_n) \\ \textit{where} \quad & U^{\oplus}(a_1, \ldots, a_n, b_1, \ldots, b_n) \ = \ \phi[b_1/a'_1, \ldots, b_n/a'_n]. \end{aligned}$$

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:

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

The result is already the sized type of *append*. But, it can be further simplified by computing its convex hull[2]: *Hull* $B = [m, n, p] : m \geq 0 \wedge n \geq 0 \wedge p = m + n$.

[2] A convex hull of a constraint $\Lambda_S v_1 \ldots v_n . \phi$ is the least upper bound of $\Lambda_S v_1 \ldots v_n . \phi$ which is of the form

$$\Lambda_S v_1 \ldots v_n . \bigwedge_{i=1}^{k} \{a_{1i} v_1 + \ldots + a_{ni} v_n \leq b_i | a_{1i}, \ldots, a_{ni}, b_i \in \mathcal{Z}\}.$$

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

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

THEOREM 7 (Soundness – Recursive Case). *Given a recursive function definition $x = M$ of type $\underline{\sigma}$. If we infer that M has sized type (σ, Φ), then for any monomorphic instance of M with denotation d, d satisfies Φ under σ.*

Proof Let $v_1, \ldots, v_n$ be the size variables associated with the input and output of x. Let $B_I = [v_1, \ldots, v_n] : \psi$ be the termination constraint, $R = [v_1, \ldots, v_n] \rightarrow [v_1', \ldots, v_n'] : \phi$ be the recursive-call constraint for M, and $R^{\oplus}$ be the generalized transitive constraint. We assume that it can be shown that $R^{\oplus}$ is an upper bound to R^+ (this will be proven in Theorem 8 of Section 9).

First, we note that the constraint Φ derived for the recursive function definition is of the form:

$$\begin{aligned} \Phi = [v_1, \ldots, v_n] \, : \, & B_I(v_1, \ldots, v_n) \vee \\ & \exists\, (u_1, \ldots, u_n \,.R^{\oplus}(v_1, \ldots, v_n, u_1, \ldots, u_n) \wedge B_I(u_1, \ldots, u_n)) \end{aligned}$$

Let F be the functional for the recursive function $x = M$. Then, x is the least upper bound of the series of functions: $x_i = \sqcup_{k=1}^{i} F_k(\perp)$.

Let R be the recursive-call constraint for M. We define a series of constraints as follows:

$$\begin{aligned} S_i = \textstyle\bigsqcup_{\mathbf{P}}{}_{k=1}^{i} (& B_I(v_1, \ldots, v_n) \vee \\ & \exists\, (u_1, \ldots, u_n \,.\, R^k(v_1, \ldots, v_n, u_1, \ldots, u_n) \wedge B_I(u_1, \ldots, u_n))) \end{aligned}$$

with $R^1 = R$ and $R^{k+1} = R \circ R^k$ (refer to Definitions 4 and 5). We can then show that, by the structural induction hypothesis, x_1 satisfies S_1 under σ; by the fixed-point induction hypothesis, x_i satisfies S_i under σ for all $i > 0$. Now, by the admissibility of satisfiability (Lemma 3), we must have x $(= x_\infty)$ satisfies S_∞ under σ, where

$$\begin{aligned} S_\infty = \; & \textstyle\bigsqcup_{\mathbf{P}\,k>0} (B_I(v_1, \ldots, v_n) \vee \exists\, (u_1, \ldots, u_n \,.\, R^k(v_1, \ldots, v_n, u_1, \ldots, u_n) \\ & \qquad \wedge B_I(u_1, \ldots, u_n))) \\ =_{\mathbf{P}} \; & B_I(v_1, \ldots, v_n) \vee \\ & \exists\, (u_1, \ldots, u_n \,.(\textstyle\bigsqcup_{\mathbf{P}\,k>0} R^k(v_1, \ldots, v_n, u_1, \ldots, u_n)) \wedge B_I(u_1, \ldots, u_n)) \\ =_{\mathbf{P}} \; & B_I(v_1, \ldots, v_n) \vee \\ & \exists\, (u_1, \ldots, u_n \,.R^+(v_1, \ldots, v_n, u_1, \ldots, u_n) \wedge B_I(u_1, \ldots, u_n)) \\ \sqsubseteq_{\mathbf{P}} \; & B_I(v_1, \ldots, v_n) \vee \\ & \exists\, (u_1, \ldots, u_n \,.R^{\oplus}(v_1, \ldots, v_n, u_1, \ldots, u_n) \wedge B_I(u_1, \ldots, u_n)) \\ =_{\mathbf{P}} \; & \Phi \end{aligned}$$

□

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

The actual computation is also almost identical to the computation for *gtc* in Section 7.2.1. Since we do not need to determine the size relation of the final output and initial input here, there is no need to incorporate termination constraints in our computation, and the output size relation is ignored.

As an example, consider the variant of binomial function bin' defined in Section 7. To derive its recursive-call constraint for the purpose of computing the recursive-call invariant, we modify the original recursive-call constraints S_1 and S_2 by "dropping" those computations related to output size:

$$\begin{aligned}
U_1 &= \exists\, p,\ c_2 \ :\ S_1 \\
&= \exists\, p,\ c_2 \ :\ \exists\, b_3,\ c_3 \ :\ \exists\, m \ : \\
&\quad (n > 0 \land m+1 = n \land \\
&\quad ((k > 0 \land k < n) \land (m = a_1 \land p = c_2 + c_3 \land \\
&\qquad\qquad\qquad\qquad (k-1 = b_2 \land k = b_3)))) \\
U_2 &= \exists\, p,\ c_3 :\ S_2 \\
&= \exists\, p,\ c_3 \ :\ \exists\, b_2,\ c_2 \ :\ \exists\, m \ : \\
&\quad (n > 0 \land m+1 = n \land \\
&\quad ((k > 0 \land k < n) \land (m = a_1 \land p = c_2 + c_3 \land \\
&\qquad\qquad\qquad\qquad (k-1 = b_2 \land k = b_3))))
\end{aligned}$$

The desired recursive-call constraint for bin' is then obtained by, again, the *union* of all parent/child relationships. After simplification, we obtain the following result:

$$\begin{aligned}
U_{bin'} &= U_1 \lor U_2 \\
&= [n,k] \to [n',k'] \ :\ (n' = n-1 \land k' = k-1 \land 1 \le k < n) \\
&\qquad\qquad\qquad\qquad \lor\ (n' = n-1 \land k' = k \land 1 \le k < n)
\end{aligned}$$

Next, we compute the generalized transitive constraint of U_{bin}:

$$U^{\oplus}_{bin'} = [n,k] \rightarrow [n^+,k^+] : \\ (k^+,\ 1 \le k < n) \wedge (k^+,\ 1 \le n^+ < n) \wedge \\ (k + n^+ \le n + k^+) \wedge 0 \le k^+$$

The above result can be more succinctly expressed as follows:

$$U^{\oplus}_{bin'} = [n,k] \rightarrow [n^+,k^+] : \\ max(1,k^+) \le n^+ < min(n + k^+ - k, n) \\ \wedge\ max(k + n^+ - n, 0) \le k^+ \le min(n^+, k)$$

This result is the same as that obtained using the original definition of *bin* (Section 2). Moreover, in their earlier work [4], 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 concerns between specification and constraint solving enables the analyzer (either the programmer or the compiler) to concentrate on formulating the constraint, without being unduly concerned with the operational detail of actually solving the problem.

9. Computing Generalized Transitive Constraints

We have used the `closure` operator provided in the Omega Calculator to help compute the transitive closure of the recursive-call constraint U. As noted by the developers of the Omega Calculator [19], the transitive-closure computation becomes complicated in the presence of multiple disjuncts. Thus, in practice, there are three possible results from such calculation using the Omega Calculator:

1. The computation `closure(`U`)` terminates and yields a Presburger formula. In this case, we check whether the formula is indeed the transitive closure. The check is done using Property 2 (Section 7.2.1).

2. The computation terminates and yields a constraint that contains the symbol `UNKNOWN`.[3] Here, we replace these `UNKNOWN` by a safe approximation – the value *True*, resulting in an upper bound of the original constraint. (This replacement is done by the operation `upper_bound` in the Omega Calculator.)

3. The computation aborts because the Omega Calculator is unable to determine the transitive closure.

[3] The special constraint `UNKNOWN` represents one or more additional constraints that are not known to the Omega Library. Such constraints can arise from the need to approximate during computation, such as the uses of transitive closure. We refer the readers to the related paper [18] for detail discussion.

In the first two cases, when `upper_bound(closure(`U`))`[4] fails to produce U^+, the result is a lower bound of U^+ (see Section 6.4 of [18]). We propose to overcome this deficiency through a generalization strategy, which attempts to compute an upper bound to the transitive closure as our desired transitive constraint. Specifically, our tactic is to *reduce the number of disjuncts by combining related disjuncts into one, where appropriate.*

In the last case, when the computation has been aborted, we reduce the complexity of the constraint U by *reducing the number of dependencies between size variables*, thus eliminating the constraints for some input/output sizes. Elimination of these constraints is obtained by existentially quantifying the pertinent size variables.

Figure 8 provides an overview of the strategy for computing a generalized transitive constraint. Basically, Steps 2.a to 2.c describe our first strategy, which reduces the number of disjuncts; step 2.d describes our second strategy, which reduces the number of size-variable dependency.

Reducing the number of disjuncts A well-known method for combining multiple disjuncts is the *convex hull* computation. Computation of convex hull is described in Schrijver's *Theory of Linear and Integer Programming* [25].

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.

Consider the Ackermann function *ack*. Inferring its recursive-call constraint produces the following constraint in disjunctive normal form:

$$\begin{aligned}[i,j] \to [i',j'] : \; & (1+i'=i \wedge j'=1 \wedge j=0 \wedge i>0) \vee \\ & (i'=i \wedge 1+j'=j \wedge i>0 \wedge j>0) \vee \\ & (1+i'=i \wedge i>0 \wedge j>0)\end{aligned}$$

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

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

Our heuristic attempts to identify disjuncts that are "similar" in form for merging. Two disjuncts are similar if each variable in each

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

Given a constraint U:

Step 0. $\quad SecondTime \leftarrow$ False ;
$\qquad originalU \leftarrow U$;
Step 1. $\quad R \leftarrow$ `upper_bound(closure(`U`))`.
Step 2. $\quad$ If (R is a formula) , then
Step 2.a. $\qquad$ If $R == U^{+}$, then $U^{\oplus} \leftarrow R$, halt.
Step 2.b. $\qquad$ If *SecondTime* then Goto Step 2.d.
Step 2.c. $\qquad$ If the number of disjuncts in R can be reduced, then
Step 2.c.1. $\qquad\quad U \leftarrow$ apply convex-hull operation heuristically on R,
$\qquad\quad SecondTime \leftarrow$ True.
$\qquad\quad$ Goto Step 1.
Step 2.c.2. $\qquad$ Else, $U^{\oplus} \leftarrow$ *True*, halt.
Step 2.d. $\quad$ Else, $U \leftarrow originalU$;
Step 2.d.1. $\qquad$ if the number of dependencies in U can be reduced, then
Step 2.d.2. $\qquad\quad U \leftarrow$ reduce dependency of U,
$\qquad\quad originalU \leftarrow U$.
$\qquad\quad SecondTime \leftarrow$ False.
$\qquad\quad$ Goto Step 1.
Step 2.d.3. $\qquad$ Else $U^{\oplus} \leftarrow$ *True*, halt.

Figure 8. Strategy for Computing Generalized Transitive Constraints

disjuncts does not depend on different variables. The following table illustrates the "similarity" between the disjuncts of `closure(`U`)` stated above:

	disjunct 1		disjuncts 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				

Notice from the table that, the last three disjuncts of `closure(`U`)` share some similarity: all have their size variable i bounded below by some constant factor of i', and vice versa. Consequently, we combine these disjuncts through a convex-hull operation. Together with the first disjunct, we obtain the following more general constraint U':

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

With the Omega Calculator, we can show that `closure (`U'`)` $= U'^{+}$. Theorem 8 shows that U' is an upper bound of 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, Pareto and Sabry [16], which requires rewriting the definition of Ackermann function.

What happens if `closure`(U') above is still not a transitive closure? From the empirical result, we employ the second tactic (reducing the number of dependencies), instead of repeating the existing one. This is because the `closure` operation has already made an attempt to perform the transitive-closure operation. If it is still not possible to obtain the desired transitive closure or its upper bound, it is usually the case that the result has too many disjuncts. This makes further reduction of disjunction time-consuming, with little prospect for obtaining a sufficiently accurate upper bound later on.

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 a constraint between these size variables, we *quantify them existentially*. A common technique we use is to first existentially quantify the size variables associated with the output of a function, and then existentially quantify the size variables associated with one of the input.

As an example, consider the function *look* defined locally in *bsearch* (Section 2.3). The usual calculation of the recursive-call invariant yields *True*, due to its inability to relate the two inputs. However, by reducing the number of dependencies between the two inputs, we are able to obtain the more precise recursive-call invariant shown in Section 2.3.

THEOREM 8 (Upper Bound to Transitive Closure). *$U^{\oplus}$ is an upper bound of U^{+}.*

Proof When computing $U^{\oplus}$, the generalization method applies when the `closure(`U`)` either yields a lower bound of U^{+} or the computation aborts. In the first case, we let R_1 = `closure(`U`)`, and we have $R_1 = \bigsqcup_{\mathbf{P}}{}_{k=1}^{n} U^k$, for some $n > 1$. Thus, $U \sqsubseteq_{\mathbf{P}} R_1$. Next, let S be the result of the convex-hull computation over R_1. By the definition of the convex hull, we must have $U \sqsubseteq_{\mathbf{P}} R_1 \sqsubseteq_{\mathbf{P}} S$. In the second case, we existentially quantify some size variables, resulting in a constraint R_2 such that $U \sqsubseteq_{\mathbf{P}} R_2$. Now, by the monotonicity of the transitive-closure operator, $U^+ \sqsubseteq_{\mathbf{P}} S^+$ and $U^+ \sqsubseteq_{\mathbf{P}} R_2^+$. Thus, both S^+ and R_2^+ are upper bounds of U^+. □

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

Termination Our method always terminates, because (1) calls to the Omega Calculator always terminate, (2) steps 2.a to 2.c are executed only twice, and (3) step 2.d is iterated a maximum of n times, where n is the number of size variables available in U.

10. Enhancements

In this section, we point to some enhancements that can be made to the type system.

Dealing with mutual recursion Our system works well on self-recursive function. To treat mutual recursion, we encode each set of mutual-recursive functions as a single self-recursive function, and feed this function to our type system for calculation. As an example, consider the following *odd* and *even* functions:

$$\begin{array}{llll} odd :: Int \rightarrow Bool & & even :: Int & \rightarrow Bool \\ odd\ 0 & = False & even\ 0 & = True \\ odd\ (n+1) & = even\ n & even\ (n+1) & = odd\ n \end{array}$$

These can be encoded as the following *oe* function:

$$\begin{array}{ll} oe :: (Bool^i,\ Int^j) & \rightarrow Bool^k \\ oe\ (True, 0) & = False \\ oe\ (True, n+1) & = oe\ (False, n) \\ oe\ (False, 0) & = True \\ oe\ (False, n+1) & = oe\ (True, n) \end{array}$$

Here, the argument to *oe* is a pair, the first component of which is a tag indicating the original function that is being called. Thus, applying the function *odd* to a number n has been encoded in *oe* with input $(True, n)$, and applying *even* to n has been encoded by call to *oe* with input $(False, n)$. The sized type computed for *oe* is shown below;

it actually calculates the results of both the encoded *odd* and *even* functions.

$$
\begin{array}{l}
[i,j] \rightarrow [k] : \\
\quad ((i=1) \wedge (k=0) \wedge \exists\,(\,\alpha : 2\,\alpha = j \wedge 0 <= j)) \vee \\
\quad ((i=1) \wedge (k=1) \wedge \exists\,(\,\alpha : 0 = 1+j+2\,\alpha \wedge 1 <= j)) \vee \\
\quad ((i=0) \wedge (k=0) \wedge \exists\,(\,\alpha : 0 = 1+j+2\,\alpha \wedge 1 <= j)) \vee \\
\quad ((i=0) \wedge (k=1) \wedge \exists\,(\,\alpha : 2\,\alpha = j \wedge 0 <= j))
\end{array}
$$

Handling polymorphic types Since the type system ignores polymorphic data (the types of which are represented by type variables), it does not incorporate any placeholder in the size constraint of a polymorphic function to capture the manipulation of such data. Consequently, when a polymorphic function is instantiated (*i.e.*, it is applied), some size information is lost during the calculation. For instance, consider the polymorphic projection function *fst* and *snd*, which respectively returns the first and second component of a tupled argument. In order to make use of these functions, we require them to be declared monomorphically. For the function $fst :: (Int^i, Int^j) \rightarrow Int^k$, we then derive its size to be $[i,j] \rightarrow [k] : i = k$.

Note that this does not mean that our type system will be rendered useless when dealing with polymorphic functions. In practice, it can compute useful size information about these functions. The size information of the *append* function provided in Section 1 is one such example.

We believe that the size of polymorphic data can be better handled by a separate set of subtyping relations. We are currently working in this direction.

Polymorphic recursion In inferring the size of recursive function, we systematically instantiate a recursive function with new size variables at each recursive call. This idea is similar to the treatment of recursive function as polymorphic recursion. Notice that we only introduce new size variables (and not new un-annotated types) at each recursive calls. Instantiation at each recursive calls is necessary since different recursive calls can, and do have different call contexts. Introducing new size variables at each call enables capturing of these contextual constraints.

It has been shown by Henglein [12] that type inference algorithm for polymorphic recursion is incomplete. For our type system, our type inference algorithm is incomplete for another reason: the limitation of the Omega's transitive-closure computation over multiple disjuncts [19].

Higher-order functions The type system handles a restricted form of higher-order functions. Specifically, it requires that the function body be available in order to infer the size of a function. Consequently, while it is able to calculate rather accurately the size of the value resulting from either full or partial application of an already-defined function, it fails to compute much meaningful information about a function parameter (because of the unknown function value.)

When performing inference on a function definition with function parameters, our type system attempts to infer the sizes of these function parameter from their uses within the function definition. It does not extend this inference beyond the definition body, thus limiting the inference to local (within the function body) constraint. One plausible solution is to add the notion of higher-order constraint to capture the size of a higher-order function parameter. A higher-order constraint can be applied when the actual function argument is available. Manipulation of higher-order constraints is outside the scope of Presburger formulae, and requires a more sophisticated solvers (or a combination of different solvers) than the Omega Calculator. This work is currently in progress.

11. Related Work

Our work has been inspired by the work on sized types by Hughes, Pareto and Sabry [16]. However, instead of performing type-checking (as they did), we have devised a fixed-point method for safely inferring the sizes of 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 ω (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 [29, 30]. 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. Also, their system does not compute the recursive-call invariants for recursive functions, which are 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 affects code reuse.

Research into the inference of linear constraints dates back to 1970's, with the seminal work of Cousot and Halbwachs on imperative languages [7]. Halbwachs has applied linear constraints to synchronization analysis of reactive systems [10, 11]. In their work, a constraint is described by a set of linear inequalities, and the technique relies on an effective widening operation to obtain fixed point of polyhedra. Our method differs from their work in the following aspects: (1) Their fixed-point computation includes the base case computation at each iteration, whereas we exclude that from the generalized-transitive-constraint computation. Consequently, they can sometimes provide more accurate results, should the base-case constraint be needed to help restrict the explosion of search space. (2) Their method captures only input/output size relations, but not recursive-call invariants. We have found recursive-call invariants to be useful in some applications (*e.g.*, [5]). (3) The use of polyhedra restricts the solution to linear inequalities, instead of general Presburger formulae. Consequently, their solution can sometimes by less accurate than ours, when the precise solution needs to be expressed using existential quantification. (4) A polyhedra alone cannot be used to express disjunctive constraints. Thus, it cannot be used to describe the lexicographical ordering of input to *ack* function. Jeannet, Halbwachs and Raymond [17] recently described a way to statically partition a linear relation to improve the precision of their analysis. This new method is similar to the introduction of disjunction in a linear relation; further investigation is needed for comparison.

Linear constraints have also received much attention in logic programming community, particularly in the inference of argument sizes for the purpose of termination analysis of logic programs [8]. One such approach is pioneered by Ullman and Van Gelder [28], which has spurred substantial development [2, 3, 23, 26, 27]. 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 $\Sigma_{i \in I} p_i + \gamma_p \geq \Sigma_{j \in J} p_j$ were used in the Mercury system [27], where p_k stands for the size of argument k of predicate p, I is a 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 the minimum integer value of γ_p, which serves as an indicator of the maximum possible growth of arguments. Due to the nature of the termination analysis problem, 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 structures is not captured.

King *et al.* also look into the use of linear constraints in program analysis, both in computing argument-size relationships [1] and in time-complexity analysis [20]. Their work is in the context of constraint logic programs. The main purpose of this analysis is to detect termination. The technique employed is the polyhedra analysis proposed by Cousot and Halbwachs. While their method supports input/output relationships, it is in first-order setting, and the lack of a type framework limits their analysis to unnested data structures. They also do not attempt to capture recursive-call invariants in their analysis; neither do they employ disjunction in their computation.

12. Conclusion

We have presented a sized typing, its formal rules and associated techniques to support automatic inference. Our main contributions are:

- We have designed a *sized typing* that can cater to various optimizations and analyses. It is powerful since it supports inter-procedural analysis (via size constraints) and captures loop invariants across recursion (via invariant constraints).

- 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 type-checking frameworks that require 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 [24], augmented with *selective generalizations* to achieve good results (Section 9).

- We showed the soundness of 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. Our approach directly benefits from the practically useful performance of the Omega Calculator. Since it is relatively easy to replace the constraint solver, our approach is also well-hedged for future improvement in constraint-solving technology.

Our preliminary implementation 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 generalized transitive 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 of $\perp$. The extent to which this can be achieved will help determine both the accuracy and the utility of sized typing in the lazy setting. One approach is to use data and co-data types to distinguish finite from infinite data structures [16], but the onus is on the user to make the distinction.

13. Acknowledgments

We would like to thank Neil Jones for his critical review of the earlier part of the work, and Hongwei Xi, Masami Hagiya, Luke Ong, Zhenjiang Hu, and Tyng-Ruey Chuang for their insightful comments and valuable suggestions to the first draft of this paper. We also thank the anonymous referees for their comments, some of which were both spirited and detailed, and helped to bring the article to its present form. Last, but not least, we thank Dana N. Xu for her help in constructing a prototype of the type system. This work was supported by the research grants RP3972716 and RP3982693.

References

1. F. Benoy and A. King. Inferring argument size relationships with CLP($\mathcal{R}$). In *Logic Programming Synthesis and Transformation*, Springer-Verlag, August 1997.
2. A. Bossi, N. Cocco, and M. Fabris. Typed norms. In *European Symposium on Programming '92*, LNCS 582, pages 73–92, 1992.
3. 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.
4. 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.
5. W.N. Chin, S.C. Khoo, and Dana N. Xu. Deriving pre-conditions for array bound check elimination. To appear in the proceedings of 2001 Symposium on Programs as Data Objects, May 2001.

6. D.C. Cooper. Programs for mechanical program verification. In *Machine Intelligence 6*, pages 43–59, 1971.
7. 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.
8. D. De Schreye and S. Decorte. Termination of logic programs: The never-ending story. *Journal of Logic Programming*, 19/20:199–260, 1994.
9. A.J. Glenstrup and N.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.
10. N. Halbwachs. About synchronous programming and abstract interpretation. *Science of Computer Programming, Special Issue on SAS'94*, 31(1), May 1998.
11. 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.
12. F. Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.
13. L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebras: Part I.* North-Holland, 1971.
14. D. Hilbert and P. Bernays. *Grundlagen der Mathematik I.* Berlin: Springer-Verlag, 1968.
15. C.K. Holst. Finiteness analysis. In *5th ACM Functional Programming Languages and Computer Architecture Conference*, Cambridge, Massachusetts, LNCS 523, pages 473–495, August 1991.
16. 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.
17. B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numerical properties. In *International Symposium on Static Analysis, SAS'99*, LNCS 1694, pages 39–50, Berlin Heidelberg: Springer, 1999.
18. 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`.
19. 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.
20. A. King, K. Shen, and F. Benoy. Lower-bound time-complexity analysis of logic programs. In Jan Maluszynski, editor, *International Symposium on Logic Programming*, pages 261 – 276. MIT Press, November 1997.
21. S. Nishimura. Parametric polymorphic type inference in constraint form. In *Second JSSST Workshop on Programming and Programming Languages*, pages 38—49, Japan, March 2000.
22. M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.
23. L. Plumer. Termination proofs for logic programs. In *Lecture Notes in Artificial Intelligence*, volume 446. Springer Verlag, 1990.
24. W. Pugh. The Omega Test: A fast practical integer programming algorithm for dependence analysis. *Communications of ACM*, 8:102–114, 1992.
25. A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, New York, 1986.

26. 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.
27. C. Speirs, Z. Somogyi, and H. Sondergaard. Termination analysis of Mercury. In *Fourth International Symposium on Static Analysis*, Paris, France, LNCS 1302, pages 157–171, 1997.
28. J.D. Ullman and A. Van Gelder. Efficient tests for top-down termination of logical rules. *Journal of ACM*, 35(2):345–373, 1988.
29. 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.
30. H. Xi and F. Pfenning. Dependent types in practical programming. In *Symposium on Principles of Programming Languages*, pages 214–227. ACM Press, January 1999.

Appendix

A. An Example Run of the Omega Calculator

```
# Omega Calculator [v1.1, Nov 96]:
# # Here, we attempt to find the recursive-call invariant
# # for Ackermann function
#
## Step 1 : Computet he recursive-call constraint
# U1 := {[i,j] -> [i1,j1] :
#         exists (k,k1 :
#       (i > 0 & j = 0 & i1 + 1 = i & j1 = 1 & k1 = k) or
#       (i > 0 & j > 0 & i1 = i & j1+1 = j &
#       exists (u : k1 = u)) or (i > 0 & j > 0 &
#           exists (u : j1 = u) & i1+1 = i & k1 = k)) } ;
#
## Step 2 : Compute its transitive closure
# Fst := U1+ ;
#
# Fst ;
{[i,j] -> [i,j1] : 0 <= j1 < j && 1 <= i} union
 {[i,j] -> [i1,j1] : 0 <= i1 < i && 1 <= j} union
 {[i,j] -> [i1,1] : 0 <= i1 < i && 0 <= j} union
 {[i,0] -> [i1,j1] : 0 <= i1 <= i-2}
#
## Step 3 : Check for transitive closure validity fails:
# (Fst compose U1) subset Fst ;
False
```

```
## Step 4 : Constraint Generalization using Hull operation
# MFst := {[i,j] -> [i,j1] : 0 <= j1 < j && 1 <= i} union
#  hull ({[i,j] -> [i1,j1] : 0 <= i1 < i && 1 <= j} union
#  {[i,j] -> [i1,1] : 0 <= i1 < i && 0 <= j} union
#  {[i,0] -> [i1,j1] : 0 <= i1 <= i-2}) ;
#
## Step 5 : Compute the transitive closure for MFst
# Snd := MFst+ ;
#
## Step 6 : Check for its validity succeeds.
# (Snd compose U1) subset Snd ;
True
#
# This is the final recursive-call invariant
# Snd ;
{[i,j] -> [i,j1] : 0 <= j1 < j && 1 <= i} union
 {[i,j] -> [Out_1,j1] : 0 <= Out_1 < i && 0 <= j}
```