Calculating Sized Types*

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

Abstract

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

1 Introduction

Many program optimisations and analyses, such as array-bound checking, termination analysis, etc, depend on knowing the size of a function's input and output. However, size information can be difficult to compute.

Firstly, accurate size computation requires detecting *size relation* between different inputs of a function. Thus, its computation cannot be easily expressed using the usual "independent attribute analysis", such as traditional abstract interpretation based on collecting semantics.

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

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

There are several kinds of size information that are of interest to us. First and foremost, we are interested in knowing how the output of *append* relates to its input, since this is fundamental to computing "inter-procedural" size information. Let m, n be the size (*ie.*, length) of its parameters xs, ys, respectively, and p be its output size, then we have $m + n = p$. Next, we are also interested in finding how the arguments of recursive calls to *append* are changed with respect to the initial *append*

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

call. In particular, in the successive recursive calls to *append*, the first argument to the recursive call of *append* is monotonically decreasing, while the second remains constant in size. Such information is useful for analysing the termination of a program, as well as termination of transformation process, such as partial evaluation (eg., [7, 6]).

Literature on size computation has mainly concentrated on size checking, instead of inferencing (see [8], [17]). Furthermore, size information computed are usually represented in a form that facilitates a particular optimisation or analysis, but less useful to others. In this paper, we aim to *infer* as accurate size information as possible to various optimisations and analyses. In the case of *append*, we express its size (and its type) as follows:

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

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

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

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

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

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

2 Why Sized Types?

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

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

2.1 Termination Property

Sized types can be used to confirm the *termination property* of recursive functions. (Please refer to Section 9 for more related work.) A recursive function terminates if there exists a set of arguments that can be shown to decrease across recursion. Such proofs using well-founded orderings (or quasi-orderings) are useful for safety-critical systems [8] and also to ensure that the partial evaluation process terminates [6]. To illustrate this, consider the Ackermann function. The recursive-call invariant about its two parameters indicates a well-founded lexicographic ordering. This confirms that its recursive calls will eventually terminate, even though the second parameter has an increasing value.

2.2 Vector-Based Memoisation

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

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

2.3 Bound-Check Elimination

Run-time bound checking on arrays is crucial for ensuring robust programs but can be expensive. More sophisticated compilers may offer an option to eliminate bound checks. However, they do not fare well,

due to their inability to synthesize loop invariants and perform inter-procedural analysis. Sized type can offer a superior solution to this problem.

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

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

3 Background

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

1. The size of a recursive data is its maximum depth (*ie.*, the number of nesting). Thus, the size of a list is its length, and the size of a binary tree is its depth. Also, the size of an array[1] is simply its size. The size of an integer is itself.

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

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

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

A *Presburger formula* (or a *term*) is used to express the relationship among the variables. Fig 1 depicts its syntax. Note that arithmetic expressions are restricted to affine functions of the variables. For example, $2 * x$ is legal, but $x * x$ is not. Semantically, a Presburger formula ϕ defines a set of integer tuples, each element of which satisfies the boolean relation, $\mathcal{B}[\![\, \phi \,]\!]$. Following is some of the examples:

$$\mathcal{B}[\![\, a \geq 0 \wedge b > 0 \wedge a \leq b \,]\!] = \{(a, b) \mid a \geq 0 \wedge b > 0 \wedge a \leq b \}.$$
$$\mathcal{B}[\![\, \exists\, a : a \geq 0 \wedge b > 0 \wedge a \leq b \,]\!] = \{\, b \mid \exists\, a : a \geq 0 \wedge b > 0 \wedge a \leq b \,\}.$$

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

Given a formula ϕ containing variable a. Suppose that $\mathcal{B}[\![\phi]\!] \neq \emptyset$, it is easy to show that $\mathcal{B}[\![(\exists\, a\, :\, \phi)]\!] \neq \emptyset$. In fact, we can obtain $\mathcal{B}[\![(\exists\, a\, :\, \phi)]\!]$ by projecting the result of $\mathcal{B}[\![\phi]\!]$ on all free variables, except a, in ϕ.

The definition of $\mathcal{B}$ is equivalent to the predicate calculus semantics with an interpretation over integer domain. The reader may refer to many related textbooks for detail, such as [11].

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

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

Figure 1: Syntax of Constraints

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

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

4 Type System

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

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

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

A type system is introduced to specify the size relation of every expression in a program. This relation relates the size of (possible values bound to) free variables occurring in an expression to the expression's

output size. Specifically, in the type system, we associate each expression with an environment (a type assumption) and a sized type.

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

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

Size variables are attached to data constructors, such as $[Int^n]^l$, where l refers to the length of a list, and n 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; *ie.*, $\underline{[Int^n]^l} = [Int]$. Size variables are used in formulating the size constraint of an expression.

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

Given two predicates, ψ_1 and ψ_2, we say that ψ_2 *approximates* ψ_1, denoted by $\psi_1 \sqsubseteq \psi_2$, if either $\psi_2 = \top$, or $\psi_1 \Rightarrow \psi_2$ (in logical sense). The latter condition implies that every tuple of values satisfying predicate ψ_1 also satisfies ψ_2. This notion of approximation defines the subtyping relation for our type system.

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

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

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.
  2. Applying α to a sized type (σ, ϕ) results in 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.
  3. Applying α to two annotated types τ_1 and τ_2, denoted by $\tau_1 \overset{\alpha}{\equiv} \tau_2$, checks if τ_1 and τ_2 have the same underlying type. That is, they are the same *modulo size-variable renaming*.
- *Size-variable listing* (ι): This takes in either an annotated type σ or a constraint Φ, and returns the set of free size variables occurring in σ or Φ. Thus, $\iota([Int^i]^j) = \{i, j\}$.
- *Size-variable matching* (eqC): Given two annotated types, σ_1 and σ_2, such that (1) $\iota(\sigma_1) \cap \iota(\sigma_2) = \emptyset$, and (2) $\sigma_1 \overset{\alpha}{\equiv} \sigma_2$, eqC produces a conjunction of equalities that identify the corresponding free size variables between σ_1 and σ_2. For instance, $eqC\ [Int^i]^j\ [Int^k]^l = (i = k) \wedge (j = l)$.

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. Given an assumption Γ and a constraint ϕ, the judgment $\mathcal{C}_\Gamma \vdash \phi$ indicates that ϕ is deducible under the set $\mathcal{C}_\Gamma$.

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

Definition 1 ($\widehat{\phi_\Gamma}$) *Let* $K = \{0, \ldots, k\}$, $\Gamma = \{ x_i :: (\sigma_i, \phi_i)\}_{i \in K}$, *and* ϕ *be a Presburger formula. Suppose that* $\mathcal{C}_\Gamma \vdash \phi$. *Then* $\widehat{\phi_\Gamma} \stackrel{def}{\equiv} \phi \wedge \bigwedge \{\phi_j\}_{j \in S}$, *where* $\{\phi_i\}_{i \in K} = \mathcal{C}_\Gamma$, $S \subseteq K$, *and*

1. *if* $\{\phi_j\}_{j \in S} \cup \{\phi\}$ *is partitioned into two non-empty subsets* Π_1 *and* Π_2, *then* $\iota(\wedge \Pi_1) \cap \iota(\wedge \Pi_2) \neq \emptyset$;
2. *for any* $\phi' \in (\mathcal{C}_\Gamma - \{\phi_j\}_{j \in S})$, $\iota(\widehat{\phi_\Gamma}) \cap \iota(\phi') = \emptyset$.

Property 1 (Property of $\widehat{\phi_\Gamma}$)

1. *If* Γ *can be partitioned into two assumption sets* Γ_1, Γ_2, *then* $\widehat{\phi_\Gamma} = (\widehat{\phi_{\Gamma_1}})_{\Gamma_2}$.
2. *Let* $X \subset \iota(\widehat{\phi_\Gamma})$. $\mathcal{B}[\![\exists X . \widehat{\phi_\Gamma}]\!] \subseteq \mathcal{B}[\![(\exists \widehat{X . \phi})_\Gamma]\!]$.

Proof **[Property of $\widehat{\phi_\Gamma}$]**
Property (1) is true by the commutativity of logical conjunction. For Property (2), since $\iota(\exists X . \phi) \subseteq \iota(\phi)$, the conjuncts in the formula $(\exists \widehat{X . \phi})_\Gamma$ is a subset of the conjuncts in the formula $\widehat{\phi_\Gamma}$. By the semantics of conjunction, we thus have $\mathcal{B}[\![\exists X . \widehat{\phi_\Gamma}]\!] \subseteq \mathcal{B}[\![(\exists \widehat{X . \phi})_\Gamma]\!]$. □

As a simple illustration of the last inequality in Property 1.2, let's assume that $\mathcal{C}_\Gamma = \{ b \geq 0 \}$, and $\phi = a < b$. Then we have $\widehat{\phi_\Gamma} = (a < b) \wedge (b \geq 0)$ and $(\exists \widehat{b . \phi})_\Gamma = (a < b)$. Hence,

$$\begin{aligned}
\mathcal{B}[\![\exists b . \widehat{\phi_\Gamma}]\!] &= \mathcal{B}[\![\exists b . (a < b) \wedge (b \geq 0)]\!] \\
&\subseteq \mathcal{B}[\![\exists b . (a < b)]\!] \\
&= \mathcal{B}[\![(\exists \widehat{b . \phi})_\Gamma]\!]
\end{aligned}$$

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

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

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

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

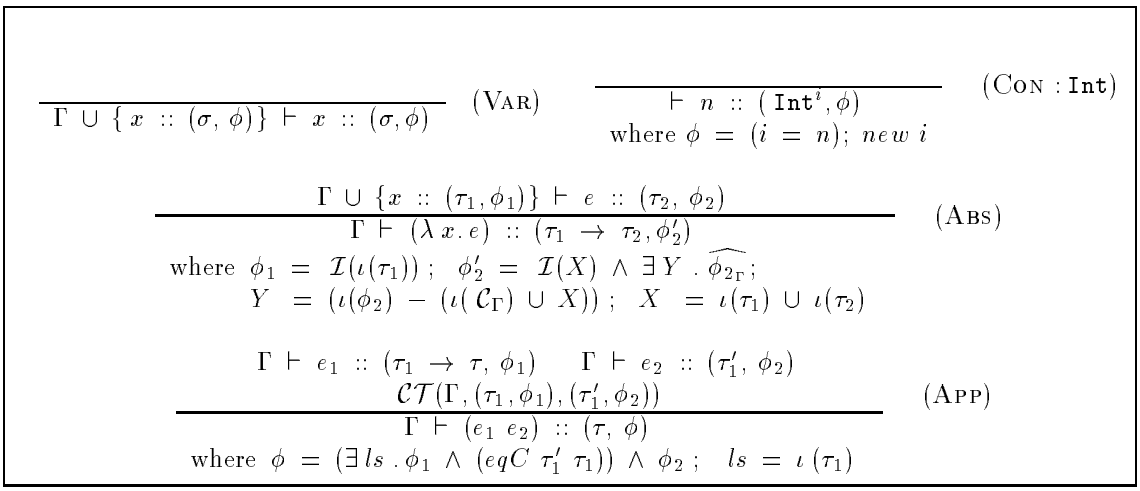

Figure 2: Rules for Checking Sized-Type — Part I

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

$$\frac{new\ x}{\Gamma \vdash^{\text{Pat}} (True, Bool^i) :: \{ x :: (Bool^i, i = 1) \}} \quad (\text{CON} : \mathtt{PAT} : \mathtt{Bool} - \mathtt{True})$$

$$\frac{new\ x}{\Gamma \vdash^{\text{Pat}} (False, Bool^i) :: \{ x :: (Bool^i, i = 0) \}} \quad (\text{CON} : \mathtt{PAT} : \mathtt{Bool} - \mathtt{False})$$

Appendix C details the computation for list types.

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

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

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

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

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

We extend this satisfiability definition to include the assumption set Γ. Given Γ, and a denotational environment ρ, we say ρ *satisfies* Γ if for any $\{ x :: d\} \in \rho$, there exists a binding $\{x :: (\sigma, \phi)\} \in \Gamma$, an integer $k \geq 0$, and a pair ρ', Γ' such that

$$\frac{\Gamma \vdash M :: (\sigma, \Phi) \quad \Gamma \cup \Gamma_1 \vdash N :: (\tau, \phi)}{\Gamma \vdash (\textbf{let } x = M \textbf{ in } N) :: (\tau, \phi')} \quad \text{(Let)}$$
$$\text{where} \quad \phi' = \exists X \,.\, \widehat{\phi_{\Gamma_1}}; \quad X = \iota(\sigma) \cap \iota(\phi); \quad \Gamma_1 = \{ x :: (\sigma, \Phi) \}$$

$$\frac{\Gamma \cup \{x :: (\sigma, \phi)\} \vdash M :: (\sigma', \phi') \quad \Gamma \cup \{ x :: (\sigma, \Phi) \} \vdash N :: (\tau, \phi'')}{\Gamma \vdash (\textbf{letrec } x = M \textbf{ in } N) :: (\tau, \phi'')} \quad \text{(Rec)}$$
$$\text{where} \quad \phi = \mathcal{I}(\iota(\sigma)); \quad \Phi = \mathcal{TC}(\phi') \quad \sigma \stackrel{\alpha}{\equiv} \sigma'$$

$$\frac{\Gamma \vdash M :: (\tau_0, \phi_0) \quad \{x :: (\tau_0, \phi_0)\} \vdash^{\text{Pat}} (P_i, \tau_0) :: \Gamma_i \quad new\ x \quad \Gamma \cup \Gamma_i \vdash M_i :: (\tau_i', \phi_i) \quad \tau_i' \stackrel{\alpha}{\equiv} \tau_j' \quad \forall\, i, j : \ i \neq j \wedge i, j > 0}{\Gamma \vdash \textbf{case } M \textbf{ of } (P_i \rightarrow M_i) :: (\tau'', \exists X \,.\, \phi)} \quad \text{(Case)}$$
$$\text{where} \quad X = (\iota(\phi) - (\iota(\mathcal{C}_\Gamma) \cup \iota(\tau''))); \quad \phi = \phi_0 \wedge \bigvee_i ((eqC\ \tau''\ \tau_i') \wedge \phi_i \wedge \mathcal{C}_{\Gamma_i}); \quad \tau'' = \alpha\, \tau_1'$$

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

$$\frac{\Gamma \vdash e :: (\tau_1 \rightarrow \tau_2, \Phi) \quad X = \iota(\tau_1 \rightarrow \tau_2)}{\Gamma \vdash e :: (\tau_1 \rightarrow \tau_2, \Lambda_S\, X \,.\, \Phi)} \quad \text{(Bind)}$$

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

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

1. $\rho' = \{ x_i :: d_i \}_{i=0}^k \subseteq (\rho - \{x :: d\})$, and
2. $\Gamma' = \{ x_i :: (\sigma_i, \phi_i) \}_{i=0}^k \subseteq (\Gamma - \{ x :: (\sigma, \phi) \})$, and
3. $\mathcal{S}((d, d_0, \ldots, d_k), (\sigma, \sigma_0, \ldots, \sigma_k)) \subseteq \mathcal{B}[\![\widehat{\phi_\Gamma}]\!]$.

We call (ρ', Γ') a *support pair* for d in (ρ, Γ).

Lemma 1 (Soundness of Pattern Matching) *$\{x :: (\tau, \phi)\} \vdash^{\text{Pat}} (P, \tau) :: \Gamma'$ implies for any denotational value d satisfying ϕ under τ, d matches the pattern P if and only if there exists a denotational environment ρ' such that $(\{x :: d\} \cup \rho')$ satisfies $(\{x :: (\tau, \phi)\} \cup \Gamma')$.*

Proof **[Soundness of Pattern matching]**
We prove the case of pattern matching boolean constants. Cases of pattern matching other constructs can be proven similarly.

Without loss of generality, we can rewrite the rule (Con : PAT : Bool – True) as follows:

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

Let d be the denotation that satisfies ϕ under $Bool^i$. Suppose that d pattern matches the pattern, then $d = True$, and size of d will be 1. We thus have $\{x :: d,\ x' :: True\}$ satisfies $\{x :: (Bool^i, \phi),\ x' :: (Bool^i, i = 1)\}$.

Conversely, if there exists a denotational environment ρ' such that $(\{x :: d\} \cup \rho')$ satisfies $(\{x :: (Bool^i, \phi), x' :: (Bool^i, i = 1)\})$, then it must be that $\rho' = \{x' :: True\}$. By satisfiability, d must satisfy $\phi \wedge (i = 1)$. Given that d is of type $Bool$. If d does not pattern match $True$, then d must be $False$, this contradicts the fact that d satisfies $\phi \wedge (i = 1)$. Hence, d pattern matches $True$.

The case for (Con : `PAT : Bool – False`) is proven similarly. □

Theorem 2 (Soundness) *Given an expression M of type $\underline{\sigma_1}$, for some σ_1. If $\vdash M :: (\sigma, \phi)$, then $\underline{\sigma} = \underline{\sigma_1}$, and for any monomorphic instance of M with denotation d, d satisfies ϕ under σ.*

Proof [Soundness]
(Sketch) The proof is by induction on the height of the type derivation. The induction hypothesis is as follows:

> If $\Gamma \vdash M :: (\sigma, \phi)$, then for all ρ such that ρ satisfies Γ, there exists a support pair (ρ', Γ') (both of size $k \geq 0$) for $(\mathcal{E}[\![M]\!]\rho)$ in (ρ, Γ) such that $\mathcal{S}((\mathcal{E}[\![M]\!]\rho, d_0, \ldots, d_k), (\sigma, \sigma_0, \ldots, \sigma_k)) \subseteq \mathcal{B}[\![\widehat{\phi_\Gamma}]\!]$.

Most of the cases in the proof are standard, except for the soundness proof for recursive definition. We shall elaborate on this in Sec. 6. Here, we present the **let** case in detail:

Following the notation used in the type rule for **let**, assume $\Gamma \vdash (\textbf{let } x = M \textbf{ in } N) :: (\tau, \phi')$ and ρ satisfies Γ, for some ρ. We wish to show that there exists a support pair (ρ_{sp}, Γ_{sp}) (both of size $k \geq 0$) for $(\mathcal{E}[\![(\textbf{let } x = M \textbf{ in } N)]\!]\rho)$ in (ρ, Γ) such that $\rho_{sp} = \{ x_i :: d_i \}_{i=0}^{k}$, $\Gamma_{sp} = \{ x_i :: (\sigma_i, \phi_i) \}_{i=0}^{k}$, and

$$\mathcal{S}((\mathcal{E}[\![(\textbf{let } x = M \textbf{ in } N)]\!]\rho, d_0, \ldots, d_k), (\tau, \sigma_0, \ldots, \sigma_k)) \subseteq \mathcal{B}[\![\widehat{\phi'_\Gamma}]\!]$$
$$\Leftrightarrow \quad \mathcal{S}((\mathcal{E}[\![N]\!]\rho[u/x], d_0, \ldots, d_k), (\tau, \sigma_0, \ldots, \sigma_k)) \subseteq \mathcal{B}[\![\widehat{\phi'_\Gamma}]\!] \qquad \text{where } u = \mathcal{E}[\![M]\!]\rho.$$

ρ satisfies Γ

$\Rightarrow$ (induction hypothesis for $\Gamma \vdash M :: (\sigma, \Phi)$)
$\exists$ support pair $(\{ x'_j :: d'_j \}_{j=0}^{a}, \{ x'_j :: (\sigma'_j, \phi'_j) \}_{j=0}^{a})$ for $(\mathcal{E}[\![M]\!]\rho)$ in (ρ, Γ) :
$\mathcal{S}((\mathcal{E}[\![M]\!]\rho, d'_0, \ldots, d'_a), (\sigma, \sigma'_0, \ldots, \sigma'_a)) \subseteq \mathcal{B}[\![\widehat{\Phi_\Gamma}]\!]$

$\Rightarrow$ (Definition of satisfiability)
$\rho[u/x]$ satisfies $(\Gamma \cup \Gamma_1)$ where $\Gamma_1 = \{ x :: (\sigma, \Phi) \}$ and $u = \mathcal{E}[\![M]\!]\rho$

$\Rightarrow$ (induction hypothesis for $\Gamma \cup \Gamma_1 \vdash N :: (\tau, \phi)$)
$\exists$ support pair $(\{x_k :: d_k\}_{k=0}^{b}, \{x_k :: (\sigma_k, \phi_k)\}_{k=0}^{b})$ for $(\mathcal{E}[\![N]\!]\rho[u/x])$ in $(\rho[u/x], \Gamma \cup \Gamma_1)$:
$\mathcal{S}((\mathcal{E}[\![N]\!]\rho[u/x], d_0, \ldots, d_b), (\tau, \sigma_0, \ldots, \sigma_b))$
$\subseteq \mathcal{B}[\![\widehat{\phi_{\Gamma \cup \Gamma_1}}]\!]$
$= \mathcal{B}[\![\widehat{(\widehat{\phi_{\Gamma_1}})_\Gamma}]\!]$ (Property 1.1)

$\Rightarrow$ (Existential quantification introduction)
$\exists \{k_1, \ldots, k_g\} \subseteq \{0, \ldots, b\}$:
$\exists$ support pair $(\{x_{k_m} :: d_{k_m}\}_{m=0}^{g}, \{x_m :: (\sigma_{k_m}, \phi_{k_m})\}_{m=0}^{g})$ for $(\mathcal{E}[\![N]\!]\rho[u/x])$ in $(\rho[u/x], \Gamma \cup \Gamma_1)$:
$\mathcal{S}((\mathcal{E}[\![N]\!]\rho[u/x], d_{k_0}, \ldots, d_{k_g}), (\tau, \sigma_{k_0}, \ldots, \sigma_{k_g}))$
$\subseteq \mathcal{B}[\![\exists X . ((\widehat{\widehat{\phi_{\Gamma_1}})_\Gamma})]\!]$ where $X = \iota(\sigma) \cap \iota(\phi)$
$\subseteq \mathcal{B}[\![\widehat{\phi'_\Gamma}]\!]$ where $\phi' = \exists X . \widehat{\phi_{\Gamma_1}}$ (Property 1.2)

Consider the support pair $(\{x_{k_m} :: d_{k_m}\}_{m=0}^{g}, \{x_{k_m} :: (\sigma_{k_m}, \phi_{k_m})\}_{m=0}^{g})$ found in the last step of the derivation above. If the support pair does not contain the bindings $(\{x :: \mathcal{E}[\![M]\!]\rho\}, \{x :: (\sigma, \Phi)\})$, then we have found the support pair for the **let**-expression in (ρ, Γ). On the other hand, suppose the support pair does contain the binding $(\{x :: \mathcal{E}[\![M]\!]\rho\}, \{x :: (\sigma, \Phi)\})$. Without loss of generality, let this binding be indexed by k_g in the support pair. Since $\iota(\sigma) \cap \iota(\phi') = \emptyset$, we have $\mathcal{S}((\mathcal{E}[\![N]\!]\rho[u/x], d_{k_0}, \ldots, d_{k_g-1}), (\tau, \sigma_{k_0}, \ldots, \sigma_{k_g-1})) \subseteq \mathcal{B}[\![\widehat{\phi'_\Gamma}]\!]$. Thus, $(\{x_{k_m} :: d_{k_m}\}_{m=0}^{k_g-1}, \{x_{k_m} :: (\sigma_{k_m}, \phi_{k_m})\}_{m=0}^{k_g-1})$ for the **let**-expression in (ρ, Γ). □

5 Size Inference for Non-Recursive Functions – An Example

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

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

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

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

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

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

In the *False* branch, we construct an assumption, $\Gamma_1 = \{nv_1 :: (Bool^i,\ i = 0)\}$ from its pattern (*i.e.*, *False*). nv_1 is a new vairable name used to refer to this pattern. We then append Γ_1 to Γ'. 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 append to Γ' the assumption $\Gamma_2 = \{nv_2 :: (Bool^i,\ i = 1)\}$. Here, we encounter the application of *append*. First, we instantiate the sized type for *append* and unbind its size variables, as follows:

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

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

Consistency Test Let (σ_1, ϕ_1) be the sized type of *append*, and (σ_2, ϕ_2) be that of xs. The test $\mathcal{CT}\,(\Gamma, (\tau_1, \phi_1), (\tau_2, \phi_2))$ first renames m_1 in ϕ_1 (and σ_1 too) and j in ϕ_2 (σ_2 too) to a new common variable, say q. In the test, we use an operator, $_\!\uparrow\!\text{-} :: \mathbf{F} \times \mathbf{V}^\star \rightarrow \mathbf{F}$ to perform projection of formulae over a set of variables. Formally, given a formula ϕ that is defined over a set of variables $\mathcal{X}$, let $\mathcal{Y}$ be a subset of $\mathcal{X}$, then $\phi\!\uparrow^{\mathcal{Y}} \stackrel{def}{=} \exists\ x_1 \cdots x_n : \phi$, where $\{x_1, \ldots, x_n\} = \mathcal{X} - \mathcal{Y}$.

With the help of the Omega Calculator, the consistency test is done as follows:

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

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

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

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

$$(\exists\, n_1 \;:\; (\exists\, m_1 \;:\; (m_1 \geq 0 \wedge n_1 \geq 0 \wedge p_1 = m_1 + n_1) \;\wedge\; m_1 = j) \wedge j \geq 0 \wedge n_1 = j) \wedge j \geq 0 \quad (\psi_2)$$

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

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

Lastly, we introduce into ϕ_3 the inherited constraints of the parameters, and bind those local size variables existentially, before passing the constraint to the Omega Calculator for simplification. The final constraint for f is:

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

Remark Note that the use of type variables for polymorphic function does not incur size constraint. This greatly simplifies the type-inference process, as it is not necessary for us to compute any size constraint about the list elements. In the case of monomorphic function over lists or nested lists, we will, in principle, need to compute both the size of the lists as well as their elements/nested lists. This can result in time-consuming computation. In practice, we can choose to assign size variable to the list constructor, but assign $\top$ to its substructure – the list elements or nested lists. This will effectively eliminates size-type inference for substructures. During type inference, we eliminate $\top$'s from the formula by identifying them with the boolean value *True*. The simplified formula will then be fed to the Omega Calculator for computation. Therefore, the cost of conducting type inference can be controlled by the "substructure depth" that is to be analysed.

6 Inferring the Size of Recursive Functions

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

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

1. *Recursive-call constraint* : A relation between the input/output sizes of a recursive call and its immediate caller, and
2. *Termination constraint* : Constraint about the termination condition of the recursive call.

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

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

whereas the termination constraint is computed as follows:

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

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

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

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

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

6.1 Calculating A Generalised Transitive Constraint

A *generalised transitive constraint*, *gtc* for short, describes a relation that holds between a recursive caller and an arbitrarily nested recursive call. Given a recursive-call constraint U, a natural candidate for its *gtc* is the *transitive closure* of U, denoted by U^+. Formally, $U^+ = \bigcup_{k=1}^{\infty} U^k$, where U^k is the composition of k U's. It is a constraint that holds for all recursive calls reachable from the initial call by unfolding, and is characterised by the following important property ([10]):

Property 2 (Transitive Closure) *For all relations P and R such that $R \subseteq P \subseteq R^+$ the following holds: $P = R^+$ if and only if $P \circ R \subseteq P$.*

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

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

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

Using Property 2, we can verify that $\texttt{closure}(U) = U^+$.

Techniques for calculating transitive closure have been described in [10]. In practice, the `closure` operation may not produce the desired result, and we need to approximate the transitive closure by its upper bound, which we call *generalised transitive closure*. This *generalisation* technique will be described in detail in Section 8. Notation-wise, we denote the generalised transitive closure of a recursive-call constraint U by $U^{\oplus}$.

6.2 Incorporating Terminating Constraint

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

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

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

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

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:

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

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

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

Theorem 3 (Soundness – Recursive Case) *Given a recursive function definition $x = M$ with type $\underline{\sigma_1}$, for some σ_1. If we infer that M has sized type (σ, ϕ), then $\underline{\sigma_1} \equiv \underline{\sigma}$, and for any monomorphic instance of M with denotation d, d satisfies ϕ under σ.*

Proof [Soundness of Analysing Recursive Definition]
(Sketch) First, we note that the constraint ϕ is of the form:

$$\phi = B_I \vee \exists\,(v_1, \ldots, v_n\ :\ Up(v_1, \ldots, v_n) \wedge B_I(v_1, \ldots, v_n))$$

where $v_1, \ldots, v_n$ are input and output of x, B_I is the terminating constraint, and Up is the generalised transitive constraint – an upper bound to the transitive closure of recursive-call constraint.

Let F be the functional for the recursive function $x = M$. Then, x is the least upper bound of the following 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:

$$S_i = \cup_{k=0}^{i}\,(B_I \vee \exists\,(v_1, \ldots, v_n\ :\ R^k(v_1, \ldots, v_n) \wedge B_I(v_1, \ldots, v_n)))$$

[2]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 $(Hull\ B) \equiv B$).

with $R^0 = False$. We can then show that, by structural induction, x_0 satisfies S_0 under σ; by fixed-point induction, x_i satisfies S_i under σ for all $i \geq 0$. Hence, we must have x $(= x_\infty)$ satisfies S_∞ under σ, where

$$\begin{aligned}
S_\infty &= \cup_{k=0}^{\infty} (B_I \vee \exists (v_1, \ldots, v_n : R^k(v_1, \ldots, v_n) \wedge B_I(v_1, \ldots, v_n))) \\
&= B_I \vee \exists (v_1, \ldots, v_n : (\cup_{k=0}^{\infty} R^k(v_1, \ldots, v_n)) \wedge B_I(v_1, \ldots, v_n)) \\
&= B_I \vee \exists (v_1, \ldots, v_n : R^+(v_1, \ldots, v_n) \wedge B_I(v_1, \ldots, v_n)) \\
&\subseteq B_I \vee \exists (v_1, \ldots, v_n : Up(v_1, \ldots, v_n) \wedge B_I(v_1, \ldots, v_n)) \\
&= \phi
\end{aligned}$$

□

7 Recursive-call Invariants

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

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

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

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

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

Next, we compute the generalised transitive constraint of U:

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

The above result can be more succinctly expressed as follows:

$$max(1, k^+) \leq n^+ < min(n + k^+ - k, n) \wedge max(k + n^+ - n, 0) \leq k^+ \leq min(n^+, k)$$

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

8 Computing Generalised Transitive Constraints

We have used the `closure` operator provided in the Omega Calculator to help computing the transitive closure for the recursive-call cosntraint U. As noted in [10], 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 terminates and yields a Presburger formula, as in the computation of `closure(`U`)` in Section 6.1. In this case, we check if the formula is indeed the transitive closure. The check is done using Property 2.
2. The computation terminates and yields a constraint that contains symbol `UNKNOWN`.[3] Here, we treat these `UNKNOWN` as our special size value $\top$, and computes an upper bound of the constraint. (The operation `upper_bound` in the Omega Calculator does just this.) We then verify if the resulting constraint is indeed the transitive closure.
3. The computation aborts because the Omega Calculator is unable to determine the transitive closure.

In the first two cases, when `upper_bound(closure(` U`))`[4] fails to produce U^+, the result is a lower bound of U^+ (see Sec. 6.4 of [9]). We propose to overcome this deficiency through a generalisation strategy, which attempts to compute an upper bound to the transitive closure as our desired transitive constraint. 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 4 provides an overview of the strategy for computing generalised transitive constraint. Basically, Steps 2.a and 2.b describe our first strategy, which reduces the number of disjuncts; Step 2 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. While this method can sometimes be computationally expensive, and its result too general, we can reduce both its complexity and generality by an intelligent (and automatic) selection of suitable disjuncts for combination. We illustrate this with a practical example.

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

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

[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 [9] for detail discussion.

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

Figure 4: Strategy for Computing Generalised Transitive Constraints

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

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

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

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

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

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

With the Omega Calculator, we can show that `closure (`U'`)` $= U'^+$. Theorem 4 shows that U' is an upper bound to the transitive closure of U, and therefore $U^{\oplus} = U'$.

It is interesting to see that this invariant constraint can be used to prove the termination of Ackermann function. This is in contrast with the method proposed by Hughes [8], which requires rewriting

the definition of Ackermann function.

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

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

Theorem 4 (Upper Bound to Transitive Closure) *$U^{\oplus}$ is an upper bound U^{+}.*

Proof [Upper Bound on Transitive Closure]
In computing $U^{\oplus}$, the generalisation 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 $R_1 = \bigcup_{k=1}^{n} U^k$, for some $n > 1$. Thus, $U \subseteq R_1$. Next, let S be the result of convex hull computation over R_1. By the definition of convex hull, we must have $U \subseteq R_1 \subseteq S$. In the second case, we existentially quantify some size variables, resulting in a constraint R_2 such that $U \subseteq R_2$. Now, by the monotonicity of the transitive-closure operator, $U^{+} \subseteq S^{+}$ and $U^{+} \subseteq R_2^{+}$. 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 bound to transitive closures.

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

9 Related Work

Our work has been inspired by the work on sized types by Hughes, Pareto and Sabry in [8]. However, instead of performing type-checking (as they did), we have devised a fixed-point method for safely inferring the sizes of the recursive functions. Furthermore, by assigning a logical constraint to each expression, we are able to express size information more accurately in at least two ways: Firstly, their method determines only *one* bound of the size, fixing the other bound at either 0 (for data types) or 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 in [17, 18]. Their type-checking framework is parameterised with constraint domains, of which

size information can be described by a domain of natural numbers with linear equality and inequality constraints. While this has the advantage that analysis efficiency can be controlled by the sophistication of the constraint domain, it requires *insightful type annotations* from programmers. Also, their system does not compute recursive-call invariants for recursive functions, which is very useful for determining local constraints within functions that holds throughout all nested calls. As a result, most recursive functions used by Xi and Pfenning have to be rewritten as local functions. This affects code reuse.

Research into the inference of linear constraints has dated back to 1970's, with the seminal work of Cousot and Halbwachs on imperative languages [4]. It has also received much attention in logic programming community, particularly in the inference of argument sizes for the purpose of termination analysis of logic programs [5]. One such approach is pioneered by Ullman and Van Gelder [16], which has spurred further development in [12, 2, 1, 14, 15]. In this approach, size information is represented via a restricted form of predicate inequalities (usually without disjunction of constraints). For example, constraints of the form $\Sigma_{i \in I} p_i + \gamma_p \geq \Sigma_{j \in J} p_j$ was used in the Mercury system [15], where p_k stands for the size of argument k of predicate p, I is the subset of the set of the input argument positions of predicate p, and J is a subset of the set of its output argument positions. The aim is to solve for an integer value of γ_p, which is the value that will serve as an indicator of the maximum possible growth of arguments. Due to the nature of the termination decision, these proposals for capturing size information are more restrictive and less modular than sized typing: Firstly, only one bound is computed instead of two. Secondly, size relationship between inputs are ignored. Lastly, separate size information for nested data strutures is not captured.

10 Conclusion

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

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

Our preliminary evaluation of sized type inference via the Omega Calculator has gone very well. Based on our new inference method, we have systematically found the sized types for a range of examples including those used in this paper. Experiments are being conducted currently to further improve the

strategy for computing generalised tansitive constraints, so that it can more accurately handle non-linear recursive definitions.

On the theoretical front, it would be of interest to work on an extension to sized typing that could better support lazy languages. Under lazy semantics, the size of partial data structures (that contains $\perp$) should ideally be distinguished from that for $\perp$. The extent to which this could be achieved will help determine both the accuracy and the utility of sized typing in the lazy setting. One such approach is to use data and co-data types to distinguish finite from infinite data structures, as used in [8], but the onus is on the user to make the distinction.

11 Acknowledgment

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

References

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

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

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

[4] 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.

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

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

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

[8] 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.

[9] 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`.

[10] 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.

[11] G.F. Luger and W.A. Stubblefield. *Artificial Intelligence: Structures and Strategies for Complext Problem Solving*. Addison Wesley, 1998.

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

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

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

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

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

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

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

A Some Example Functions

$$
\begin{array}{l}
zip \;::\; [a]^m \;\rightarrow\; [b]^n \;\rightarrow\; [(a,b)]^p \\
\quad \text{s.t. } \mathbf{size}\ \ (0 <= m <= n \wedge p = m) \vee (0 <= n < m \wedge p = n) \\
\qquad \mathbf{inv}\ \ (0 \leq m^+ < m) \wedge (m \leq n + m^+) \wedge (n^+ = n - m + m^+) \\
zip\ \ xs\ \ ys\ \ =\ \mathbf{case}\ xs\ \mathbf{of}\quad Nil \rightarrow Nil \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (x : xs') \rightarrow \mathbf{case}\ ys\ \mathbf{of}\quad Nil \rightarrow Nil \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (y : ys') \rightarrow (x, y)\ :\ zip\ \ xs'\ \ ys'
\end{array}
$$

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

$$
\begin{array}{l}
g\ \ ::\ \ ([Int^u]^i, [Int^v]^j) \rightarrow [Int^w]^\top \\
\quad \text{s.t. } \mathbf{size}\ (i = 0 \wedge j \geq 0 \wedge l = 0) \vee (i \geq 1 \wedge j \geq 0 \wedge l = i - 1) \vee \\
\qquad\qquad (i \geq 1 \wedge j \geq 0 \wedge l = i + j - 1) \\
g\ (x,y)\ =\ \mathbf{case}\ x\ \mathbf{of}\ Nil \rightarrow Nil \\
\qquad\qquad\qquad\qquad\qquad\quad (b : xs) \rightarrow \mathbf{case}\ (b == 0)\ \mathbf{of}\ True \rightarrow append\ xs\ y \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad False \rightarrow xs
\end{array}
$$

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

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

B 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 : Compute the recursive-call constraint
# U := {[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 Generalisation 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 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}
```

C Type rules for Lists

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

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

$$\frac{new\ x}{\Gamma \vdash^{\text{Pat}} (Nil, [\tau]^i) \;::\; \{\, x \;::\; ([\tau]^i, i = 0)\,\}} \qquad (\textsc{Con} : \texttt{PAT} : \texttt{List} - \texttt{Nil})$$

$$\frac{}{\Gamma \vdash^{\text{Pat}} (x, \tau) \;::\; \{\, x \;::\; (\tau, True)\,\}} \qquad (\textsc{Con} : \texttt{PAT} : \texttt{List} - \texttt{Var})$$

$$\frac{\Gamma \vdash^{\text{Pat}} (P_1, \tau') \;::\; \Gamma_1 \quad \Gamma \vdash^{\text{Pat}} (P_2, [\tau'']^j) \;::\; \Gamma_2 \quad new\ x}{\Gamma \vdash^{\text{Pat}} (P_1 : P_2, [\tau]^i) \;::\; \Gamma_1 \cup \Gamma_2 \cup \{\, x \;::\; ([\tau]^i, \phi)\}} \quad \text{where } \begin{array}{lcl} \phi & = & (j+1 = i) \wedge ((eqC\ \tau\ \tau') \vee (eqC\ \tau\ \tau''))\,; \\ \tau' & = & \alpha\,(\tau); \qquad [\tau'']^j \;=\; \alpha([\tau]^i) \end{array} \qquad (\textsc{Con} : \texttt{PAT} : \texttt{List} - \texttt{Cons})$$

D Context Specialisation

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

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

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

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

$$
\begin{array}{l}
zip1 :: [a]^n \rightarrow [b]^m \rightarrow [(a,b)]^p \\
\qquad \textbf{size}\ (n \leq m) \wedge (p = n) \\
zip1\ \ xs\ \ ys = \textbf{case}\ \ xs\ \ \textbf{of}\ \ Nil \rightarrow Nil \\
\qquad\qquad\qquad\qquad\qquad\qquad x : xs' \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{case}\ \ ys\ \ \textbf{of}\ \ \ y : ys' \rightarrow (x,y) : (zip1\ \ xs'\ \ ys')
\end{array}
$$

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

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

$$
\begin{array}{l}
zip2 :: [a]^n \rightarrow [b]^m \rightarrow [(a,b)]^p \\
\qquad \textbf{size}\,(n \geq m \wedge p = m) \\
zip2\ \ xs\ \ ys = \textbf{case}\ \ ys\ \ \textbf{of}\ Nil \rightarrow\ \ Nil \\
\qquad\qquad\qquad\qquad\qquad y : ys' \rightarrow\ \ \textbf{case}\ \ xs\ \ \textbf{of}\ x : xs' \rightarrow (x,y) : (zip2\ \ xs'\ \ ys')
\end{array}
$$

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

When $s_1 = s_2$, our specialisation technique would obtain a specialised code that is similar to $zip1$, but with a size context that is subsumed by $s_1 \leq s_2$. This indicates that maximum specialisation for zip has already been achieved under a weaker context. A scheme that syntactically checks for duplication of specialised programs can help avoid needless code explosion.