View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Radboud Repository

Radboud Repository

Radboud University Nijmegen

PDF hosted at the Radboud Repository of the Radboud University Nijmegen

The following full text is an author's version which may differ from the publisher's version.

For additional information about this publication click this link.
http://hdl.handle.net/2066/36493

Please be advised that this information was generated on 2017-12-06 and may be subject to change.

Static inference of polynomial size-aware types

Olha Shkaravska*, Marko van Eekelen
{O.Shkaravska, M.vanEekelen}@cs.ru.nl

Security of Systems Department
Institute for Computing and Information Sciences
Radboud University Nijmegen

Abstract. We propose a static size analysis procedure that combines term-rewriting and type checking to automatically obtain output-on-input size dependencies for first-order functions. Attention is restricted to functions for which the size of the result is strictly polynomial, not necessarily monotonous, in the sizes of the arguments.
To infer a size dependency, the procedure generates hypotheses for increasing degrees of polynomials. For each degree, to define a hypothetical polynomial one needs to know its values on a finite collection of points (subject to some geometrical condition). To compute the value of a size polynomial in a certain point we use a term-rewriting system generated by a standard size-annotations inference procedure.
We have proven that if a function with a given input terminates at run-time on a meaningful stack and heap then the static ('compile-time') term rewriting system of the size inference also terminates, on the integers representing the sizes of the corresponding inputs.
The term rewriting system may terminate at compile-time when the underlying function definition does not at run-time. This makes the theoretical applicability of the proposed approach larger than the previous state-of-the-art, where run-time testing was used to generate hypothetical polynomials. Also, the practical applicability is improved due to increased efficiency since the term rewriting system at compile time abstracts from many computations that are done at run-time.

1 Introduction

Embedded systems or server applications often have limited resources available. Therefore, it can be important to know in advance how much time or memory a computation is going to take, for instance to determine how much memory should at least be put in a system to enable all desired operations.

This helps to prevent abrupt termination for both sorts of hardware: small devices (like mobile phones or Java cards), or powerful computers on which one runs memory exhaustive computation (GRID, model generation). Static size information helps in efficient memory management, since it allows to decrease

* This research is sponsored by the Netherlands Organisation for Scientific Research (NWO), project Amortized Heap Space Usage Analysis (AHA), grantnr. 612.063.511.

fragmentation. From the point of view of security, size verification is useful in avoiding "Denial of Service" attacks that exploit memory exhaustion.

Decisions in these (and related) problems are based on formally verified upper bounds of the resource consumption. However, an advanced detailed analysis of these bounds requires knowledge of the sizes of the data structures used throughout the program [vESvK+07]. Trivially, the time it takes to iterate over a list depends on the size of that list. In this paper we focus on the task of automatically deriving the exact output-on-input size dependencies of function definitions in a program. The ratio behind exactness is explained later in this section.

Size dependencies can be represented in function *types*. We focus on shapely functions, where shapely means that the size relations are exactly polynomial (not necessarily monotonous). The size of a list is its number of nodes (its length).

Consider an example. The function `diffplus`, given two lists of length n and m respectively

- returns a list with the length $n - m$, if $n > m$.
- non-terminates, otherwise:

```
diffplus ([], ys)          = diffplus ([], ys)
diffplus ((x:xs), [])      = (x:xs)
diffplus ((x:xs), (y:ys))  = diffplus(xs, ys)
```

For instance, on the lists `[1, 2, 3]`, `[5, 6]` it returns `[3]`, and on the lists `[1, 2, 3]`, `[5, 6, 7]` it non-terminates.

The *size-aware type* of a function expresses the relation between its argument and result sizes if terminates:

$$\texttt{diffplus} : [\,\alpha\,]^n \times [\,\alpha\,]^m \rightarrow [\,\alpha\,]^{n-m}$$

In general, all lists at the input side,have an associated size variable. At the output side, all lists have an associated polynomial that determines the size of the output list. These polynomials are defined in terms of the input size variables.

The size dependencies we study are not necessarily linear, as in Pareto's approach [Par98], or monotonic as in [VH04] and for polynomial interpretations [MP06].

In [SvKvE07a] we introduced a type system with static type checking and with dynamic type inference based on the fact that a polynomial is defined by a finite number of points on its graph. Running tests of a given function definition on an appropriate set of inputs (see [vKSvE07] for the principles of choice of inputs) one obtains a finite collection of test data that defines a system of linear equations. Its solution is the vector of coefficients of the polynomial expressing the hypothetical size dependency. There are several size polynomials if the output type is a nested list. The polynomials annotate corresponding underlying first-order types and these sized types are *checked* by a type-checker. If the type-checker rejects the first-order type, one continues the procedure for a polynomial of a higher degree.

The procedure non-terminates in two cases: either a function definition under consideration does not terminate on the proposed test inputs, or it is not well-typed.

In this paper we "lift" the non-termination issue from the level of run-time execution of full program code to the level of compile-time type constraints. This makes the approach uniform and we get more control over termination. Given a first-order function definition, a standard type-inference procedure ends up with a set of (recurrent) equational constraints for size dependencies. In general, constraints derived by a standard type-inference procedure may be non-linear and multivariate (see [SvKvE07b] for the example). It is very unlikely that there exists a solver that solves all types of them. However, once one assumes that the solution is a polynomial, one may use the constraints to calculate values of the size functions(s), compute the coefficients and type-check the resulting polynomials.

Calculating values of the size functions(s) can be done with a rewriting tool that performs breadth-first search. As an illustration of that we will show what the results are of using `Maude` [CDH+07] for a small example.

Note, that both, the old run-time test-based annotation-inference procedures, and the presented new, static term-rewriting one, have as advantage that one eventually solves a system of *linear equations* avoiding non-linearity.

To illustrate the new static approach, continue with the **`diffplus`** example. Suppose, its size dependency $?p(n,m)$ must be inferred. Type inference produces a system of equations

$$\begin{array}{rl} n=0 & \vdash ?p(n,\,m) = ?p(n,\,m) \\ m=0 & \vdash ?p(n,\,m) = n \\ & \vdash ?p(n,\,m) = ?p(n-1,\,m-1) \end{array}$$

that can be easily interpreted as a program (or, formally, a term-rewriting system), computing $?p$. Using, for instance, diagonal parsing of the space $\mathbf{N}^2$, we see that we can compute $?p(0,0)=0$, $?p(1,\,0)=1$ $?p(1,\,1)=0$. This is sufficient to obtain $a,\ b,\ c$ for $?p(n,m)=an+bm+c$. Solving the corresponding system

$$\begin{array}{ll} 0\cdot a+0\cdot b+c & = ?p(0,0)=0 \\ a+0\cdot b+c & = ?p(1,0)=1 \\ a+b+c & = ?p(1,1)=0 \end{array}$$

gives $a=1,\ b=-1,\ c=0$. So, $?p(n,m)=n-m$. Note, that due to diagonal search the nodes satisfy the configuration that assures the uniqueness of the solution of the system of linear equations for $a,\ b,\ c$.

It is important that we analyse *strict*, precise size dependencies. In this case test values lie exactly on the graph of the size function (a hypothetical polynomial). One can calculate the coefficients of the hypothetical polynomial due to this fact.

The rest of this paper is organized as follows. In section 2 we recapitulate the first-order language over sized lists, its type system and the run-time test-based annotation inference procedure, given in [SvKvE07a] and in [vKSvE07] respectively.

In the section 3 we define a term-rewriting system computing size dependencies. We prove that if a function body terminates on a meaningful stack and

heap then term-rewriting terminates on the integers representing the sizes of inputs. At the end of the section we show that term-rewriting may terminate when the underlying function definition does not. This makes the applicability of the proposed approach larger than its previous version, where we use run-time testing to generate hypothetical polynomials.

Both test-based and term-rewriting inference procedures are semi-terminating. The test-based (term-rewriting-based) inference procedure stops with the correct annotating polynomials if the given function definition (term-rewriting) terminates and is typeable. Otherwise both procedures work infinitely. One may want to find a condition that allows to stop inference, after a certain polynomial degree is reached, with the negative answer. It is very unlikely that for any function definition (in general, or shapely) using *integer arithmetic* one can find a maximal degree d (*a stopping criterion*) such that if the procedure fails to give a positive answer on polynomials of degree less or equal to d, then one rejects it as not typeable for sure. Also, we sketch an extension of the methodology to function definitions with non-strict size dependencies, such as `insertionsort`.

2 Type System

This section briefly describes the existing strict size-aware type system for a functional language and accompanying type checking procedure [SvKvE07a] that we use in the inference procedure. This also motivates our approach to type inference.

2.1 Size-aware Types

The zero-order types we consider are integers, *strictly* sized lists of integers, *strictly* sized lists of *strictly* sized lists, etc. A strict list of length n is a list exactly of length n (not of some length up to n, as, e.g. in *sized* types of Pareto [Par98]). For lists of lists the element lists have to be of the same size and in fact it would be more precise to speak about matrix-like structures, e.g. the type $[[\texttt{Int}]^3]^2$ is given to a list which two elements are both lists of exactly three integers, such as `[[2,5,3], [7,1,6]]`.

$$\mathit{Types}\ \tau ::= \texttt{Int} \mid \alpha \mid [\tau]^p \qquad \alpha \in \mathit{TypeVar}$$

Here p denotes a *size expression*, i.e. a polynomial in size variables.

$$\mathit{SizeExpr}\ p ::= \mathcal{Q} \mid n \mid p + p \mid p - p \mid p * p \qquad n \in \mathit{SizeVar}$$

As usual $\mathcal{Q}$ denotes the set of all rational numbers. As size expressions we consider polynomials with rational coefficients that are not necessary integer. Only those of them who map non-negative integers into non-negative integers have a semantic in the type system[1]. An example of a size expression with non-integer coefficients is the polynomial for `progression` function:

[1] In the earlier version of the type-system we considered only integer polynomials.

```
progression []     = []
progression (x:xs) = (x:xs) ++ (progression xs)
```

The type of this function is $[\,\alpha\,]^n \to [\,\alpha\,]^{\frac{n^2+n}{2}}$

We do not have partial applications and higher-order types. First-order types are functions from tuples of zero-order types to zero-order types.

$$FTypes\ \tau^f ::= \tau_1 \ \ldots\ \tau_k \to \tau_{k+1}$$

For example, the type of `diffplus`, $[\,\texttt{Int}\,]^n \times [\,\alpha\,]^m \to [\,\alpha\,]^{n-m}$ is a first-order type. In well-formed first-order types, the argument types are annotated only by size variables and the result type is annotated by size expressions in these variables. Type and size variables occurring in the result type should also occur in the argument types. Thus, the type of `diffplus` is a well-formed type, whereas $[\,\alpha\,]^{n+m} \to [\,\alpha\,]^{2*n}$ is not, because the argument is annotated by a size expression that is not a variable.

2.2 Typing system

Previously, we have developed a sound size-aware type system and a type checking procedure for a first-order functional language with call-by-value semantics [SvKvE07a]. The language supports lists and integers and standard constructs for pattern matching, if-then-else branching, and let-binding.

The typing rules follow the intuition on how sizes are created and changed during evaluation. The construction of a list gives a list that is one element longer than its tail. The `then` and `else` branches of the if-statement are required to yield the same size. The same holds for the `nil` and `cons` branches of pattern matching, but that rule also takes into account that the matched list is known to be empty in the `nil` branch: when matching a list of size s, if the `cons` branch has size $n * 4$, the `nil` branch can have size $0 * 4 = 0$ because, there $n = 0$.

As in [SvKvE07a] the formal rules are designed conventionally for ML-like syntax. Recall that an empty list `[]` is denoted by nil, a list `x:xs` is presented as $\mathsf{cons}(x, xs)$, and pattern matching and `case`-expressions both correspond to a `match`-construct. But, still, everywhere in examples we use Haskell-like syntax.

In the formal rules, a context Γ is a mapping from zero-order program variables to zero-order types, a signature Σ is a mapping from function names to first-order types, and D is a set of Diophantine equations that keeps track of which lists are empty. A typing judgment is a relation of the form $D;\ \Gamma\ \vdash_\Sigma e\!:\!\tau$ which means that if the free program variables of the expression e have the types defined by Γ, and the functions called have the types defined by Σ, and the size constraints D are satisfied, then e will be evaluated to a value of type τ, if it terminates. For example:

$$\frac{D \vdash p = p' + 1}{D;\ \Gamma,\ hd\!:\tau,\ tl\!:[\,\tau\,]^{p'}\ \vdash_\Sigma \mathsf{cons}(hd, tl)\!:[\,\tau\,]^p}\ \textsc{Cons}$$

$$\frac{\Gamma(x) = \texttt{Int} \qquad D;\ \Gamma\ \vdash_\Sigma e_t : \tau \qquad D;\ \Gamma\ \vdash_\Sigma e_f : \tau}{D;\ \Gamma\ \vdash_\Sigma \mathsf{if}\ x\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f : \tau}\ \textsc{If}$$

$$\frac{\begin{array}{c} p = 0,\ D;\ \Gamma,\ x : [\tau']^p\ \vdash_\Sigma e_{\mathsf{nil}} : \tau \\ hd, tl \notin dom(\Gamma) \qquad D;\ \Gamma, hd : \tau',\ x : [\tau']^p,\ tl : [\tau']^{p-1}\ \vdash_\Sigma e_{\mathsf{cons}} : \tau \end{array}}{D; \Gamma, x : [\tau']^p\ \vdash_\Sigma \mathsf{match}\ x\ \mathsf{with}\ \begin{array}{l} \mid \mathsf{nil} \Rightarrow e_{\mathsf{nil}} \\ \mid \mathsf{cons}(hd, tl) \Rightarrow e_{\mathsf{cons}} \end{array} : \tau}\ \textsc{Match}$$

Size-aware type checking eventually amounts to checking entailments of the form $D \vdash p = p'$, which means that $p = p'$ is derivable from D in the axiomatics of the ring of integers. Because p and p' are known polynomials of universally quantified size variables, comparing them is straightforward. For instance, for the `diffplus` function we obtain $m = 0 \vdash n - m = n$ (in the `cons, nil` branch) and $\vdash n - m = (n-1) + (m-1)$ (in the `cons, cons` branch).

We formulated a syntactical condition sufficient to make type checking decidable for this system [SvKvE07a]. We allow *pattern matching and case expressions only for function parameters and variables bound to them by other pattern matchings and case expressions* For instance, `diffplus` and `progression` satisfy this condition, since here only program arguments are matched. Case expressions on tails (of tails of ...) of function arguments are allowed as well:

```
f (x:xs) = case xs of
             [] -> ...
             (xx:xxs) -> ...
```

We prohibit constructs like

```
f x  = case g(x) of
         [] -> ...
         (xx:xxs) -> ...
```

This syntactic condition ensures that all equations in D in an inference tree have a very simple form $n - c = 0$, where c is an integer. In fact D defines substitution of size variables by constants of the form $n = c$.

2.3 Run-time test-based inference procedure

In [vKSvE07] we gave a detailed description of the procedure that infer polynomial size dependencies based on the fact that a polynomial is defined by a finite collection of points. The procedure is implemented and there is an available on-line demo, see `www.aha.cs.ru.nl`.

The procedure generates hypotheses for an increasing degree of size polynomials. For each degree, hypotheses for all polynomial size expressions in the output type are determined. The resulting size-aware type is checked using the size-aware type checking procedure. Thus:

1. Infer the underlying type (without sizes) using standard type inference;
2. Annotate the underlying type with size variables;
3. Assume the degree of the polynomial;

4. For every output size annotation: determine which tests are needed, do the required series of test runs and compute the polynomial coefficients based on the test results;
5. Annotate the type with the size expressions found;
6. Check the annotated type;
7. If checking fails, repeat from step 4 assuming a higher degree.

Below we show the procedure in pseudo-code. The *TryIncreasingDegrees* function generates *GetSizeAwareType* and checks *CheckSizeAwareType* hypotheses of increasing degrees. A size expression is derived by selecting a node configuration *GetNodeConf*, running the tests for these nodes *RunTests*, and deriving the size polynomial from the test results *DerivePolynomial*.

Note that if the assumed degree is lower than the true degree, then the derived polynomials may be wrong. It will be later rejected by a type checker, or the nodes where the size annotations are fully defined cannot be determined correctly. It may happen that the node configuration has "too many" points where the size expression is undefined so the test results do not provide enough information to uniquely infer the inner polynomial(s). In that case one increases the degree and continue inference.

Function: TRYINCREASINGDEGREES
Input: a degree m, a function definition `f`
Output: the size-aware type of that function

```
TryIncreasingDegrees(m, f) =
  let type  = InferUnderlyingType(f)
      atype = AnnotateWithSizeVariables(type)
      vs    = GetOutputSizeVariables(atype)
      stype = GetSizeAwareType(m, f, atype, vs, [ ])
  in  if (CheckSizeAwareType(stype, f)) then stype
      else TryIncreasingDegrees(m+1, f)
```

Function: GETSIZEAWARETYPE
Input: a degree, m the function definition `f` with its annotated type, a list of unknown size annotations, and the polynomials already derived
Output: the size-aware type of that function if the degree is high enough

```
GetSizeAwareType(m, f, atype, [ ], ps) =
  AnnotateWithSizeExpressions(atype, ps) // The End
GetSizeAwareType(m, f, atype, v:vs, ps) =
  let nodes   = GetNodeConf(m, atype, ps)
      results = RunTests(f, nodes)
      p       = DerivePolynomial(m, v, atype, results)
  in  GetSizeAwareType(m, f, atype, vs, p:ps)
```

If a type is rejected, this can mean two things. First, the assumed degree was too low and one of the size expressions has a higher degree. That is why the procedure continues for a higher degree. Another possibility is that one of

the size expressions is not a polynomial (the function definition is not shapely) or that the type cannot be checked due to incompleteness. In that case the procedure *will not terminate.* Fortunately, in practice a suitable stopping criterion may be a reasonable upper bound on the degree of size polynomials. However, theoretically, it is very unlikely that for any function definition (in general, or shapely) using *integer arithmetic* one can find a maximal degree d (*a stopping criterion*) so that if the procedure fails to give a positive answer on polynomials of degree less or equal to d, then one rejects it as not typeable for sure.

If the function is well-typable, the procedure will eventually find the correct size-aware type and terminate.

In this paper we consider the same schema, with procedure RunTests replaced by RunTermRewriting using, instead of run-time testing, a static analysis with a generated term rewriting system. Implementation of RunTermRewriting seems to be much simpler, than for RunTests, especially if one uses an external rewriting tool. In particularly, for RunTests we had to generate input lists from given sizes and underlying type.

3 Term rewriting system computing values of hypothetical polynomials

In this section we show that if a function body terminates, then one can compute values of the hypothetical size polynomials using not run-time testing but the system of constraints generated by the standard inference procedure.

Informally, one thinks about the constraints as of a *t*erm *r*ewriting *s*ystem (further, t.r.s.). It is obtained from the constraints just by substituting the symbol "=" by "$\rightarrow$". The t.r.s. has a terminating reduction path once the function body terminates.

In section 3.1, we define precisely, what we mean by "standard annotation inference", and construct a t.r.s. computing size polynomials. In section 3.2, we recapitulate the notion of a meaningful value, using a heap-aware semantics of types, and operational semantics of the language. We show, that if an operation-semantics tree is finite then there is a terminating reduction for the term rewriting system, on the integers representing the sizes of the inputs.

In this way we prove the main result of this section:

Theorem 1. *If a computation of the well-typed function terminates, then there exists a reduction sequence for the term-rewriting system that terminates on integers that represent the sizes of inputs.*

Finally, in section 3.3 we show that term rewriting may terminate when the underlying function definition does not. Thus, in practice term rewriting significantly extends the applicability of the proposed non-standard annotation inference.

3.1 Standard annotation inference and term rewriting system

We want to construct a term-rewriting system that computes a value of size polynomials on given integer arguments.

For instance, for the body of `diffplus` function, given an inference tree for its underlying type, and the typing judgment input:

$$\emptyset;\ x\colon[\,\alpha\,]^n,\ y\colon[\,\alpha\,]^m\ \vdash_\Sigma e\colon[\,\alpha\,]^{?p},$$

one wants to obtain the system of constraints

$$\begin{array}{l} n=0\ \vdash ?p(n,\,m) = ?p(n,\,m)\\ m=0 \vdash ?p(n,\,m) = n\\ \qquad\quad \vdash ?p(n,\,m) = ?p(n-1,\,m-1)\end{array}$$

and the term-rewriting system

$$\begin{array}{ll} ?p(0,\,m) & \to ?p(0,\,m)\\ ?p(n,\,0) & \to n\\ ?p(n,\,m) & \to ?p(n-1,\,m-1)\end{array}$$

For the `progression` function one may want to obtain the following t.r.s.:

$$\begin{array}{ll} ?p(0) & \to 0\\ ?p(n) & \to n + ?p(n-1)\end{array}$$

or an equivalent one, for the desugared let-form of the function body, where an auxiliary non-terminal $?q$ is introduced to denote the annotation of the type of the bound expression `progression xs` in the body `let z=progression xs in (x:xs)++(progression xs)`:

$$\begin{array}{ll} ?p(0) & \to 0\\ ?p(n) & \to n + ?q(n-1)\\ ?q(n-1) & \to ?p(n-1)\end{array}$$

Generation of constraints and term rewriting system:

Input: 1) an underlying-type inference tree for a (sub)expression e (of the body e_f of the function f with unknown output size annotations $?p_i$) with the root $x_1\colon[\ldots[\,\tau_{10}\,]\,],\ldots\ \vdash_\Sigma e\colon[\ldots[\,\tau_0\,]\ldots]$, where $\tau_0, \tau_{10}, \ldots$ are either `Int` or a size variable α;
2) a typing judgment $D; x_1\colon[\ldots[\,\tau_{10}\,]^{g_{1k_1}}\ldots]^{g_{11}},\ldots\ \vdash_\Sigma e\colon[\ldots[\,\tau_0\,]^{?g_l}\ldots]^{?g_1}$ with free size variables n_t and arithmetic expressions g_{js}, possibly containing symbols $?p_i$, and auxiliary non-terminals.

Output: 1) the system of constraints $\mathcal{D}$ w.r.t. $?g_1, \ldots, ?g_l$,
formed by conditions of the form $D \vdash ?g_i(\ldots, n_t, \ldots) = P_i$,
where P_i some arithmetical expressions,
containing, maybe, $?p_1, \ldots, ?p_l$ and auxiliary non-terminals,
2) the term rewriting system $\mathcal{T}$ with rules of the form
$\vdash ?g_i(\ldots, n_t, \ldots)[D] \ \rightarrow \ P_i[D]$,
where $t[D]$ is a result of substituting of free size variables
in a term t by integer constants, determined by D.

The set of constraints is generated inductively together with the annotation-inference tree for a given expression. All the type placeholders τ assume canonical forms of types. The canonical form of a type $[\ldots[\,\tau_0\,]^{p_l}\ldots]^{p_1}$, given D, means that $D \vdash p_i = 0$, for $i > 1$, once $D \vdash D \vdash p_1 = 0$. It is based on an observation that semantically all lists $[\ldots[\,\tau_0\,]^{p_l}\ldots]^0$ represent the same list $[\ldots[\,\tau_0\,]^0\ldots]^0$.

1. If e is a constant expression $e \equiv c$ then one applies IConst rule

$$\frac{}{D;\ \Gamma\ \vdash_{\Sigma} c\!:\!\mathtt{Int}}\ \textsc{IConst}$$

Then the "nestedness" of the output list is $l = 0$, $\tau_0 \equiv \mathtt{Int}$ and the set of constraints is empty: $\mathcal{D} = \emptyset$.

2.

$$\frac{D \vdash [\ldots[\,\tau_0\,]^{?g_l}\ldots]^{?g_1} = \tau'}{D;\ \Gamma,\ x\!:\tau'\ \vdash_{\Sigma} x\!:\![\ldots[\,\tau_0\,]^{?g_l}\ldots]^{?g_1}}\ \textsc{Var}$$

Then $\tau' \equiv [\ldots[\,\tau_0\,]^{g_l}\ldots]^{g_1}$ for some g_i and

$$\mathcal{D} = \{D \vdash ?g_1 = g_1, \ldots, D \vdash ?g_l = g_l\}.$$

3.

$$\frac{D \vdash ?g_1 = 0}{D;\ \Gamma\ \vdash_{\Sigma} \mathsf{nil}\!:\![\ldots[\,\tau_0\,]^{?g_l}\ldots]^{?g_1}}\ \textsc{Nil}$$

Then

$$\mathcal{D} = \{D \vdash ?g_1 = 0; ?g_2 = 0\ldots, ?g_l = 0\}.$$

4. In the Cons-rule

$$\frac{D \vdash ?g_1 = g_1 + 1}{D;\ \Gamma,\ hd\!:\tau',\ tl\!:[\,\tau'\,]^{g_1}\ \vdash_{\Sigma} \mathsf{cons}(hd, tl)\!:\![\ldots[\,\tau_0\,]^{?g_l}\ldots]^{?g_1}}\ \textsc{Cons}$$

$\tau' \equiv [\ldots[\,\tau_0\,]^{g_l}\ldots]^{g_2}$ for some g_i and

$$\mathcal{D} = \{D \vdash ?g_1 = g_1 + 1,\ D \vdash ?g_2 = g_2,\ \ldots,\ D \vdash ?g_l = g_l\}.$$

5. Recall, that in the function application rule, with $\Sigma(f) = \tau_1^\circ \times \ldots \times \tau_k^\circ \rightarrow \tau_{k+1}$ the symbol $*$ denotes the substitution from the size-variables of $(\tau_1^\circ \times \cdots \times \tau_k^\circ)$ to size expressions of $\tau_1{}' \times \cdots \times \tau_k{}'$. If $\tau_i^\circ = [\ldots[\,\alpha_i\,]^{n^\circ_{il_i}}\ldots]^{n^\circ_{i1}}$ then τ_i' must be of the form $[\ldots[\,\tau_i''\,]^{p_{il_i}}\ldots]^{p_{i1}}$ for some τ_i''. We demand that *all n°_{ij} be different size variable names.*

In the function application rule

$$\frac{\Sigma(f) = \tau_1^\circ \times \ldots \times \tau_k^\circ \to \tau_{k+1} \qquad D \vdash ?\tau = *(\tau_{k+1})}{D;\ \Gamma, x_1 : {\tau_1}', \ldots, x_k : {\tau_n}' \ \vdash_\Sigma f(x_1, \ldots, x_k) \mathrel{?\!\!:} \tau} \text{ FunApp}$$

$\tau \equiv [\ldots[\tau_0]^{?g_l} \ldots]^{?g_1}$. Then $*(\tau_{k+1})$ is of the form $[\ldots[\tau_0]^{g_l} \ldots]^{g_1}$, and

$$\mathcal{D} = \{D \vdash ?g_1 = g_1; ?g_2 = g_l \ldots, ?g_l = g_l\}.$$

6.
$$\frac{\Gamma(x) = \mathtt{Int} \qquad D;\ \Gamma \ \vdash_\Sigma e_t \mathrel{?\!\!:} \tau \qquad D;\ \Gamma \ \vdash_\Sigma e_f \mathrel{?\!\!:} \tau}{D;\ \Gamma \ \vdash_\Sigma \mathsf{if}\ x\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f : \tau} \text{ If}$$

Then

$$\mathcal{D} = \mathcal{D}(D;\ \Gamma \ \vdash_\Sigma e_t \mathrel{?\!\!:} \tau) \bigcup \mathcal{D}(D;\ \Gamma \ \vdash_\Sigma e_f \mathrel{?\!\!:} \tau)$$

with $?\tau$ for $[\ldots[\tau_0]^{?g_l} \ldots]^{?g_1}$.

7.
$$\frac{\begin{array}{c} g = 0,\ D;\ \Gamma,\ x\colon [\tau']^g \ \vdash_\Sigma e_{\mathsf{nil}} \mathrel{?\!\!:} \tau \\ hd, tl \notin dom(\Gamma) \qquad D;\ \Gamma, hd : \tau',\ x\colon [\tau']^g,\ tl\colon [\tau']^{g-1} \ \vdash_\Sigma e_{\mathsf{cons}} \mathrel{?\!\!:} \tau \end{array}}{\begin{array}{l} D;\ \Gamma,\ x\colon [\tau']^g \ \vdash_\Sigma \mathsf{match}\ x\ \mathsf{with}\ \mid \mathsf{nil} \Rightarrow e_{\mathsf{nil}} \qquad \mathrel{?\!\!:} \tau \\ \qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{cons}(hd, tl) \Rightarrow e_{\mathsf{cons}} \end{array}} \text{ Match}$$

Then

$$\mathcal{D} = \mathcal{D}(D, g = 0;\ \Gamma,\ x\colon [\tau']^g \ \vdash_\Sigma e_{\mathsf{nil}} \mathrel{?\!\!:} \tau) \bigcup \mathcal{D}(D;\ \Gamma,\ hd : \tau',\ x\colon [\tau']^g, tl : [\tau']^{g-1} \ \vdash_\Sigma e_{\mathsf{cons}} \mathrel{?\!\!:} \tau)$$

with $?\tau$ for $[\ldots[\tau_0]^{?g_l} \ldots]^{?g_1}$. Note that due to the syntactical condition g is a function of the form $n - c$.

8. In the Let-rule

$$\frac{x \notin dom(\Gamma) \qquad D;\ \Gamma \ \vdash_\Sigma e_1 \mathrel{?\!\!:} \tau_x \qquad D;\ \Gamma,\ x : \tau_x \ \vdash_\Sigma e_2 \mathrel{?\!\!:} \tau}{D;\ \Gamma \ \vdash_\Sigma \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mathrel{?\!\!:} \tau} \text{ Let}$$

we introduce extra nonterminals $?q_i$ for the annotations of the type τ_x of the bound expression e_1. Then

$$\mathcal{D} = \mathcal{D}\big(D;\ \Gamma \ \vdash_\Sigma e_1 \mathrel{?\!\!:} \tau_x\big) \bigcup \mathcal{D}\big(D;\ \Gamma, x\colon \tau_x \ \vdash_\Sigma e_2 \mathrel{?\!\!:} \tau\big).$$

9. Without loss of generality one may think that the LetFun rule

$$\frac{\begin{array}{c} \Sigma(f) = \tau_1^\circ \times \cdots \times \tau_n^\circ \to \tau_{k+1} \\ \mathsf{True};\ x_1 : \tau_1^\circ, \ldots, x_k : \tau_k^\circ \ \vdash_\Sigma e_1 : \tau_{k+1} \qquad D;\ \Gamma \ \vdash_\Sigma e_2 : \tau' \end{array}}{D;\ \Gamma \ \vdash_\Sigma \mathsf{letfun}\ f(x_1, \ldots, x_k) = e_1\ \mathsf{in}\ e_2 : \tau'} \text{ LetFun}$$

the type for bound function is already inferred (otherwise run the procedure for this function independently). So,

$$\mathcal{D} = \mathcal{D}\big(D;\ \Gamma \ \vdash_\Sigma e_2 \mathrel{?\!\!:} \tau\big),$$

w.r.t. to a signature, containing the type for f.

The term-rewriting system $\mathcal{T}$ is obtained from $\mathcal{D}$ straightforwardly: for any $D \vdash ?g = P$ one obtains $?g[D] \rightarrow P[D]$.

The construction of $\mathcal{T}$ for $e = e_f$ defines the term-rewriting system for the unknown size annotations $?p_i$ of f.

Note the following. Let one have a family of functions $p_i : \mathtt{Int} \times \ldots \times \mathtt{Int} \rightarrow \mathtt{Int}$, such that if $p_i(n_1^0, \ldots, n_{d_i}^0) = n^0$, then for the integers $n_1^0, \ldots, n_{d_i}^0$ there is a reduction path in $\mathcal{T}$ terminating with n^0. We will not prove that this family satisfies $\mathcal{D}$. It is up to type checker to check if it is indeed the fact ($\mathcal{D}$ must be extended with the constraints $D \vdash C$ from instances of function-application rules).

3.2 Operational semantics generates a reduction path

In our semantic model, the purpose of the heap is to store lists. Therefore, it essentially is a finite collection of locations l that can store list elements. A location is the address of a cons-cell each consisting of a hd-field, which stores the value of the list element, and a tl-field, which contains the location of the next cons-cell of the list (or the NULL address). Formally, a program value is either an integer constant, a location, or the null-address and a heap is a finite partial mapping from locations and fields to such program values:

$$\mathit{Val}\ v \ ::= \ c \mid \ell \mid \mathtt{NULL} \qquad \ell \in \mathit{Loc} \qquad c \in \mathtt{Int}$$

[2]

$$\mathit{Heap}\ h \ : \ \mathit{Loc} \rightharpoonup \{\mathtt{hd}, \mathtt{tl}\} \rightharpoonup \mathit{Val}$$

We will write $h[\ell.\mathtt{hd} := v_h,\ \ell.\mathtt{tl} := v_t]$ for the heap equal to h everywhere but in ℓ, which at the hd-field of ℓ gets value v_h and at the tl-field of ℓ gets value v_t.

The semantics w of a program value v is a set-theoretic interpretations with respect to a specific heap h and a ground type $\tau^\bullet$, via the four-place relation $v \models^h_{\tau^\bullet} w$. Integer constants interprets themselves, and locations are interpreted as non-cyclic lists.

$$\begin{array}{lll}
i & \models^h_{\mathtt{Int}} & i \\
\mathtt{NULL} & \models^h_{[\tau^\bullet]^0} & \mathtt{[]} \\
\ell & \models^h_{[\tau^\bullet]^n} & w_{\mathtt{hd}} :: w_{\mathtt{tl}} \ \text{iff} \ n \geq 1, \ell \in \mathit{dom}(h), \\
& & \qquad h.\ell.\mathtt{hd} \models^{h|_{\mathit{dom}(h)\setminus\{\ell\}}}_{\tau^\bullet} w_{\mathtt{hd}}, \\
& & \qquad h.\ell.\mathtt{tl} \models^{h|_{\mathit{dom}(h)\setminus\{\ell\}}}_{[\tau^\bullet]^{n-1}} w_{\mathtt{tl}}
\end{array}$$

where $h|_{\mathit{dom}(h)\setminus\{\ell\}}$ denotes the heap equal to h everywhere except for ℓ, where it is undefined.

When a function body is evaluated, a frame store maintains the mapping from program variables to values. It only contains the actual function parameters, thus

[2] To avoid overhead with notations we treat integer values as integer literals. Ideally, one considers integer values i rather than literals c.

preventing access beyond the caller's frame. Formally, a frame store is a finite partial map from variables to values:

$$Store\ s\ :\ ExpVar \rightharpoonup Val$$

An operational-semantics judgment $s;\, h;\, \mathcal{C} \vdash e \rightsquigarrow v;\, h'$ informally means that at a store s and a heap h with a set of closures $\mathcal{C}$ an expression e terminates and evaluates to the value v at the heap h'. Using heaps and frame stores, and maintaining a mapping $\mathcal{C}$ from function names to bodies for the functions definitions encountered, the operational semantics of expressions is defined by rules in the usual way.

In general, say, for a binary function definition over integer lists one shows the following consistency property: if $(x_1, v_1),\ (x_2, v_2);\, h;\, \mathcal{C} \vdash e_f \rightsquigarrow v;\, h'$, where v_i points to the list of length n_i in h and v point to a list of length m in h', the corresponding term-rewriting terminates on $?p(n_1, n_2) \to^* m$.

Examples of consistency between function evaluation and term rewriting We will give two concrete examples of this general consistency.

Consider as first example the `diffplus` function definition. We will show that if $s(x)$ and $s(y)$ point to lists $[1, 2]$ and $[3]$ in a heap h respectively, the t.r.s. for `diffplus` terminates on $?p(2, 1) \to^* 1$.

1. Since [1, 2] is not empty the operational semantics (a full definition of these semantics is given in [SvKvE07b]) will require us to construct the reduction for the `cons`-branch sub-expression.
2. Since [3] is non-empty one constructs the reduction for the second `cons`-branch.
3. The (recursive) function call `diffplus`([2], []) gives $?p(2,\ 1) \to ?p(1, 0)$.
4. Now unfold the body of `diffplus` according to the op.sem. rule.
5. Since [2] is not empty one constructs the reduction for the `cons`-branch sub-expression.
6. Since [] is empty one constructs the reduction for the `nil`-branch sub-expression, that is [].
7. Thus $?p(1, 0) \to 1$.

As a second example, illustrating how this should work for expressions with `let`-bindings, we consider now `progression` on the list $[1, 2]$ that returns $[1, 2, 2]$.

1. Since $[1, 2]$ is not empty one constructs the reduction for the `cons`-branch sub-expression:

   ```
   let ys=progression(xs)
   in x++ys
   ```

   Let $?q$ be an auxiliary non-terminal for the size annotation of bound expression, which is `progression`([2]).
2. (Recursive) function call `progression`[2] gives $?q(1) \to ?p(1)$.

3. Call of ++, for which its size annotation are known, gives $?p(2) \rightarrow 2+?q(1)$
4. Now we need to continue a reduction path for $?q(1) \rightarrow ?p(1)$. We unfold the body of `progression` according to the op.sem. rule.
5. Since [2] is not empty one constructs the reduction for the `cons`-branch sub-expression.
6. (Recursive) function call `progression`[] gives $?q(0) \rightarrow ?p(0)$.
7. Call of ++ with the known size annotation gives $?p(1) \rightarrow 1+?q(0)$
8. Now we need to continue a reduction path for $?q(0) \rightarrow ?p(0)$. We unfold the body of `progression` according to the op.sem. rule.
9. One enters the `nil`-branch that returns an empty list, so we have $?p(0) \rightarrow 0$.
10. Altogether: $?p(2) \rightarrow 2+?p(1) \rightarrow 2+1+?p(0) \rightarrow 2+1+0 \rightarrow 3$.

Proof of consistency of term rewriting with respect to the operational semantics The *consistency* of term rewriting with respect to the operational semantics means that firstly the term rewriting system terminates when the evaluation terminates according to the operational semantics and secondly the size of the operational result is equal to the value of the result of term rewriting.

To be able to apply formal induction on the height of the operational-semantics tree for a given expression one needs to consider a stronger consistency statement, than formulated above.

This general statement looks as follows:

Lemma 1. *Let f be a function with unknown size annotations and a collection of free size variables $\{n_t\}_t$. Let $\mathcal{T}$ be the term-rewriting system, constructed along with the annotation inference tree for its body e_f.*

Let (induction assumptions)

- *e be a sub-expression of e_f,*
- *$s;\, h;\, \mathcal{C} \;\vdash\; e \;\rightsquigarrow\; v;\, h'$,*
- *s be a* meaningful *store: for any $x \in dom(s)$, there are some integers $m^0_{xj'}$ for some set-theoretical value w_x such that*

$$s(x) \models^h_{[\ldots[\,\mathtt{Int}\,]^{m^0_{xk_x}}\ldots]^{m^0_{x1}}} w_x,$$

- *there exist for the return value v some integers m^{0_i} yielding a set-theoretic value w*

$$v \models^{h'}_{[\ldots[\,\mathtt{Int}\,]^{m^0_l}\ldots]^{m^0_1}} w,$$

- *a judgment $D;\ \Gamma\ \vdash_\Sigma e\!:\!\tau$, be s.t.*
  - *it is a node in the annotation-inference tree for e_f,*
  - *$\tau = [\ldots[\,\mathtt{Int}\,]^{?g_l}\ldots]^{?g_1}$,*
  - *$\Gamma(x) = [\ldots[\,\mathtt{Int}\,]^{g_{xk_x}}\ldots]^{g_{x1}}$,*
  - *n^0_t are integer numbers, s.t. they satisfy D and $g_{xj}(\ldots, n^0_t, \ldots) \rightarrow^* m^0_{xj}$ in $\mathcal{T}$,*

then there is a reduction sequence in $\mathcal{T}$: $?g_i(\ldots, n^0_t, \ldots) \rightarrow^* m^0_i$.

Proof. Induction on the height of the operation-semantics tree for $s;\, h;\, \mathcal{C} \;\vdash\; e \;\rightsquigarrow\; v;\, h'$.

1. With the rule
$$\frac{c \in \texttt{Int}}{s;\, h;\, \mathcal{C} \;\vdash\; c \;\rightsquigarrow\; c;\, h}\ \textsc{OSICons}$$
one has $e = c$, $l = 0$ and there are no $?g_i$-s to prove their termination.
2. With the rule
$$\frac{}{s;\, h;\, \mathcal{C} \;\vdash\; x \;\rightsquigarrow\; s(x);\, h}\ \textsc{OSVar}$$
one applies the corresponding typing rule Var, with
$$\mathcal{T} \supseteq \{\ldots, ?g_i[D] \to g_{xi}[D], \ldots\},$$
The last lemma's assumption claims that $g_{xi}(\ldots n_t^0 \ldots) \to^* m_{xi}^0$, from what follows that $?g_i(\ldots n_t^0 \ldots) \;\to^*\; m_{xi}^0 \;=\; m_i^0$ as well, and the reduction for $?g_i(\ldots n_t^0 \ldots)$ takes one step more.
3. With the rule
$$\frac{}{s;\, h;\, \mathcal{C} \;\vdash\; \mathsf{nil} \;\rightsquigarrow\; \texttt{NULL};\, h}\ \textsc{OSNil}$$
one has $\mathcal{T} \supseteq \{\ldots, ?g_i[D] \to 0, \ldots\}$. It means that for all instantiations of variables, satisfying D, $?g_i$ reduces to zero. Therefore, $?g_i(\ldots n_t^0 \ldots) \to 0 = m_i^0$.
4. With the rule
$$\frac{s(hd) = v_{\texttt{hd}} \qquad s(tl) = v_{\texttt{tl}} \qquad \ell \notin dom(h)}{s;\, h \;\vdash\; \mathsf{cons}(hd, tl) \;\rightsquigarrow\; \ell;\, h[\ell.\texttt{hd} := v_{\texttt{hd}},\; \ell.\texttt{tl} := v_{\texttt{tl}}]}\ \textsc{OSCons}$$
one has $\mathcal{T} \supseteq \{?g_1[D] \to g_1[D] + 1, ?g_2[D] \to g_2[D], \ldots ?g_l[D] \to g_l[D]\}$, where $tl\colon \ldots[\,[\,\texttt{Int}\,]^{g_l} \ldots]^{g_1}$. Therefore, $?g_i(\ldots n_t^0 \ldots) \to^* g_i(\ldots n_t^0 \ldots) = m_i^0$ for $i > 1$ and $?g_1(\ldots n_t^0 \ldots) \to^* g_1(\ldots n_t^0 \ldots) + 1 = m_1^0$.
5. Consider the call of a function f' with known size annotations in $\tau_1^0 \times \ldots \times \tau_k^0 \to \tau_{k+1}$, where $\tau_{k+1} = [\ldots[\,\tau_{k+1,0}\,]^{G_l} \ldots]^{G_1}$ with free size variables z_{js}.
$$\frac{\begin{array}{c} s(x_1) = v_1 \;\ldots\; s(x_m) = v_n \qquad \mathcal{C}(f') = (y_1, \ldots, y_n) \times e_{f'} \\ [y_1 := v_1, \ldots, y_n := v_n];\, h;\, \mathcal{C} \;\vdash\; e_{f'} \;\rightsquigarrow\; v;\, h' \end{array}}{s;\, h;\, \mathcal{C} \;\vdash\; f'(x_1, \ldots, x_n) \;\rightsquigarrow\; v;\, h'}\ \textsc{OSFunApp}$$
One has $\mathcal{T} = \{\ldots ?g_i[D] \to q_i[D], \ldots\}$, where $*(\tau_{k+1}) = [\ldots[\,\tau_0\,]^{q_l} \ldots]^{q_1}$. Therefore, $?g_i(\ldots n_t^0 \ldots) \to q_i(\ldots n_t^0 \ldots) =^{\mathrm{def}} G_i(\ldots, g_{js}(\ldots, n_t^0, \ldots) \ldots) \to^* G_i(\ldots, m_{js}, \ldots)$, where $G_i(\ldots, m_{js}, \ldots)$ is a ground term, since G_i is a known arithmetic expression over $z_{js} := g_{js}(\ldots, n_t^0, \ldots)$. Due to the soundness of the type system, $G_i(\ldots, m_{js}, \ldots) = m_i^0$.
6. Consider the call of the function with unknown size annotations $?p_i$. One applies induction (on the heights of e) in this case:
$$\frac{\begin{array}{c} s(x_1) = v_1 \;\ldots\; s(x_m) = v_n \qquad \mathcal{C}(f) = (y_1, \ldots, y_n) \times e_f \\ [y_1 := v_1, \ldots, y_n := v_n];\, h;\, \mathcal{C} \;\vdash\; e_f \;\rightsquigarrow\; v;\, h' \end{array}}{s;\, h;\, \mathcal{C} \;\vdash\; f(x_1, \ldots, x_n) \;\rightsquigarrow\; v;\, h'}\ \textsc{OSFunApp}$$

Indeed, the assumptions of the lemma hold for

$$[y_1 := v_1, \ldots, y_n := v_n];\ h;\ \mathcal{C}\ \vdash\ e_f\ \rightsquigarrow\ v;\ h'$$

and

$$D; \ldots x_j : [\ldots [\,\tau_{j0}\,]^{g_{jk_j}} \ldots]^{g_{j1}} \ldots\ \vdash_{\Sigma} e_f : \tau.$$

The reduction sequence for $?p_i(\ldots, g_{js}(\ldots, n_t^0, \ldots) \ldots) \to^* m_i^0$ is constructed on this pair. Then one needs to apply one more reduction: $?g_i(\ldots n_t^0 \ldots) \to ?p_i(\ldots, g_{js}(\ldots, n_t^0, \ldots) \ldots)$.

7. One applies the induction reasoning in the case

$$\frac{s(x) \neq 0 \qquad s;\ h;\ \mathcal{C}\ \vdash\ e_1\ \rightsquigarrow\ v;\ h'}{s;\ h;\ \mathcal{C}\ \vdash\ \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \rightsquigarrow\ v;\ h'}\ \textsc{OSIfTrue}$$

since the assumptions of the lemma hold for the operation semantics judgment $s;\ h;\ \mathcal{C}\ \vdash\ e_1\ \rightsquigarrow\ v;\ h'$ and $D; \Gamma\ \vdash_{\Sigma} e_1 : \tau$. Thus, the reduction sequence is obtained on these data.

8. The `false` branch is similar:

$$\frac{s(x) = 0 \qquad s;\ h;\ \mathcal{C}\ \vdash\ e_2\ \rightsquigarrow\ v;\ h'}{s;\ h;\ \mathcal{C}\ \vdash\ \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \rightsquigarrow\ v;\ h'}\ \textsc{OSIfFalse}$$

9.

$$\frac{s;\ h;\ \mathcal{C}\ \vdash\ e_1\ \rightsquigarrow\ v_1;\ h_1 \qquad s[x := v_1];\ h_1;\ \mathcal{C}\ \vdash\ e_2\ \rightsquigarrow\ v;\ h'}{s;\ h;\ \mathcal{C}\ \vdash\ \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \rightsquigarrow\ v;\ h'}\ \textsc{OSLet}$$

The reduction path is obtained on $s[x := v_1];\ h_1;\ \mathcal{C}\ \vdash\ e_2\ \rightsquigarrow\ v;\ h'$ and $D; \Gamma,\ x : ?\tau_x\ \vdash_{\Sigma} ?\tau$. To apply induction assumption for this pair one needs to show, that annotations of $?\tau_x$ reduce to the integers, representing the sizes of the value v_1. This is done by induction assumption, on the pair $s;\ h;\ \mathcal{C}\ \vdash\ e_1\ \rightsquigarrow\ v_1;\ h_1$ and $D; \Gamma\ \vdash_{\Sigma} e_1 ?\tau_x$.

10.

$$\frac{s(x) = \mathtt{NULL} \qquad s;\ h;\ \mathcal{C}\ \vdash\ e_1\ \rightsquigarrow\ v;\ h'}{s;\ h;\ \mathcal{C}\ \vdash\ \mathsf{match}\ x\ \mathsf{with}\ \begin{array}{l} \mid \mathsf{nil} \Rightarrow e_1 \\ \mid \mathsf{cons}(hd, tl) \Rightarrow e_2 \end{array}\ \rightsquigarrow\ v;\ h'}\ \textsc{OSMatch-Nil}$$

The reduction sequence is obtained for the pair $s;\ h;\ \mathcal{C}\ \vdash\ e_1\ \rightsquigarrow\ v;\ h'$, where $s(x) = \mathtt{NULL}$, and $D, g_{x1} = 0; \Gamma\ \vdash_{\Sigma} e_1 ?\tau$, since they satisfy the conditions of the lemma.

11.

$$\frac{\begin{array}{c} h.s(x).\mathtt{hd} = v_{\mathtt{hd}} \qquad h.s(x).\mathtt{tl} = v_{\mathtt{tl}} \\ s[hd := v_{\mathtt{hd}}, tl := v_{\mathtt{tl}}];\ h\ \vdash\ e_2\ \rightsquigarrow\ v;\ h' \end{array}}{s;\ h;\ \mathcal{C}\ \vdash\ \mathsf{match}\ x\ \mathsf{with}\ \begin{array}{l} \mid \mathsf{nil} \Rightarrow e_1 \\ \mid \mathsf{cons}(hd, tl) \Rightarrow e_2 \end{array}\ \rightsquigarrow\ v;\ h'}\ \textsc{OSMatch-Cons}$$

The reduction sequence is obtained for the pair $s;\ h;\ \mathcal{C}\ \vdash\ e_2\ \rightsquigarrow\ v;\ h'$ and $D; \Gamma, hd : \tau',\ x : [\,\tau'\,]^{g_{x1}},\ tl : [\,\tau'\,]^{g_{x1}-1}\ \vdash_{\Sigma} e_2 ?\tau$, since they satisfy the conditions of the lemma.

12.

$$\frac{s;\ h;\ \mathcal{C}[f := ((x_1,\ldots,x_n)\times e_1)]\ \vdash\ e_2\ \rightsquigarrow\ v;\ h'}{s;\ h;\ \mathcal{C}\ \vdash\ \mathsf{letfun}\ f((x_1,\ldots,x_n)) = e_1\ \mathsf{in}\ e_2\ \rightsquigarrow\ v;\ h'}\ \text{OSLetFun}$$

The reduction sequence is obtained for the pair $s;\ h;\ \mathcal{C}[f := ((x_1,\ldots,x_n)\times e_1)] \vdash e_2 \rightsquigarrow v;\ h'$, and $D;\Gamma \vdash_\Sigma e_2 \mathrel{?}\tau$, since they satisfy the conditions of the lemma.

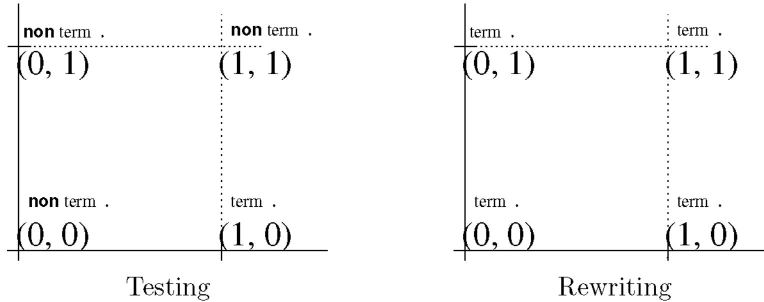
Q.E.D.

3.3 Term rewriting beats run-time testing

In [vKSvE07], for *terminating* function definitions, we have shown that to infer polynomials $?p_l$ of degree m over size variables $n_1,\ldots,n_k$, for an output type $[\ldots[\alpha]^{?p_l}\ldots]^{?p_1}$, one can find enough test data in k-ary cube with side $[0,\ldots,lm]$

Recall, that the vector of input sizes should satisfy certain, nontrivial for $k > 1$ condition that ensures that the corresponding system of linear equation has a unique solution. For instance, for 2-variate case, $k = 2$, and a linear dependency, $m = 1$, $an_1 + bn_2 + c$ it means that one needs *three* test nodes, that do not lie on the same line on the plane.

It is easy to see, that due to non-termination `diffplus` does not provide enough points on the square $[0, 1] \times [0, 1]$:



However, the t.r.s. does terminate on all nodes in this cube. Thus, term rewriting instead of run-time testing provides a good improvement for the inference algorithm, increasing its applicability. Moreover, annotation inference becomes now *completely static*.

We have earlier implemented a run-time test-based version of the inference procedure. To adapt it for static term-rewriting one should use any rewriting system that provides a search of all terminating paths on given data. We have chosen to use `Maude` [CDH+07].

The `Maude` input file for `diffplus` example looks as follows:

```
mod DIFFPLUS is
protecting INT .
```

```
op diffplus : Int Int -> Int .

vars N M Z : Int .
rl diffplus(0, M) => diffplus(0, M) .
rl diffplus(N, 0) => N .
rl diffplus(N, M) => diffplus((N - 1),  (M - 1)) .
endm
```

The `Maude`-output on **`diffplus`** for (0, 0), (1, 0), (1, 1) gives the following information, respectively:

```
search [1, 100] in DIFFPLUS : diffplus(0, 0) =>! Z .

Solution 1 (state 1)
states: 3  rewrites: 5 in 0ms cpu (0ms real) (~ rewrites/second)
Z --> 0
.......
search [3, 10] in DIFFPLUS : diffplus(1, 0) =>! Z .

Solution 1 (state 1)
states: 3  rewrites: 4 in 0ms cpu (0ms real) (~ rewrites/second)
Z --> 1

No more solutions.
states: 13  rewrites: 35 in 0ms cpu (0ms real) (~ rewrites/second)

.......

search [5, 100] in DIFFPLUS : diffplus(1, 1) =>! Z .

Solution 1 (state 2)
states: 4  rewrites: 8 in 0ms cpu (0ms real) (~ rewrites/second)
Z --> 0

No more solutions.
states: 103  rewrites: 305 in 4ms cpu (1ms real) (76230 rewrites/second)
```

The parameters $[x, y]$ behind the **`search`** command define the maximum amount of solutions and the maximal depth of the search respectively.

4 Future work

Bound for degree of a size polynomial The presented annotation-inference procedure terminates when an analyzed function is well-typed and the term-rewriting system calculates enough values of this polynomial for generating the hypothesis.

One may be interested to solve the following problem (which we call a *stopping criterion* for the procedure). Given a function definition find a degree d, such that if the function is shapely, then the degree of its size polynomial(s) is less or equal to d. We do not know the answer on this question yet, if one discusses the decidability in integer arithmetic.

Non-shapely programs The current hypothesis generation procedure relies on the limitation to shapely programs; output sizes need to be exactly polynomial in the input size. In practice many programs are not shapely, but still have a polynomial upper bound. Consider inserting an element in a set. This increases the set size by one only if the element was not in it. Its actual upper bound is:

$$\texttt{insertionsort} : [\,\alpha\,]^n \times \alpha \rightarrow [\,\alpha\,]^{n+1}$$

To extend our approach to such upper bounds, we are studying program transformations that transform an unshapely function into a shapely function with the strict size dependency corresponding to an upper bound of the size dependency of the original function. For instance, the `insertionsort` function would be transformed into a shapely function that always inserts the element. We believe that in many practical cases the testing approach combined with program transformations will succeed in providing good upper bounds.

General data structures In this paper, we presented the procedure for a simple functional language over lists. We plan to extend and implement the procedure for an existing language with more general data structures. Good candidates are XML transformation languages. This is a necessary step towards size analysis of object-oriented programs.

5 Conclusion

We presented for the first time a static type inference method for non-linear non-monotonous polynomial size-aware types. This static type inference method uses a term-rewriting system to solve the type constraints that are derived from the function definitions.

In this paper it is proven that the static type inference method terminates at least at those cases where the existing run-time type inference method terminated. It is furthermore shown that there are cases where static type inference does terminate while the corresponding run-time inference did not.

The proposed static type inference method is not only theoretically more powerful due to better termination properties but also more efficient in practice since it abstracts from a great deal of run-time computation.

References

[CDH+07] Manuel Clavel, Francisco Durán, Joe Hendrix, Salvador Lucas, José Meseguer, and Peter Csaba Ölveczky. The maude formal tool environment. In Till Mossakowski, Ugo Montanari, and Magne Haveraaen, editors, *CALCO*, volume 4624 of *Lecture Notes in Computer Science*, pages 173–178. Springer, 2007.

[MP06] J-Y Marion and R. Péchoux. Resource analysis by sup-interpretation. In *FLOPS*, volume 3945 of *Lecture Notes in Computer Science*, pages 163–176. Springer, 2006.

[Par98] L. Pareto. *Sized Types*. Chalmers University of Technology, 1998. Dissertation for the Licentiate Degree in Computing Science.

[SvKvE07a] O. Shkaravska, R. van Kesteren, and M. van Eekelen. Polynomial size analysis for first-order functions. In S. Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications (TLCA'2007), Paris, France*, volume 4583 of *LNCS*, pages 351 – 366. Springer, 2007.

[SvKvE07b] O. Shkaravska, R. van Kesteren, and M. van Eekelen. Polynomial size analysis of first-order functions. Technical Report ICIS-R07004, Radboud University Nijmegen, December 2007.

[vESvK+07] M. van Eekelen, O. Shkaravska, R. van Kesteren, B. Jacobs, E. Poll, and S. Smetsers. AHA: Amortized Heap Space Usage Analysis. In Marco Morazán, editor, *Trends in Functional Programming 8: Selected Papers of the 8th International Symposium on Trends in Functional Programming (TFP07), New York, USA*. Intellect Publishers, UK, 2007. to appear.

[VH04] P. B. Vasconcelos and K. Hammond. Inferring cost equations for recursive, polymorphic and higher-order functional programs. In P. Trinder, G. Michaelson, and R. Peña, editors, *Implementation of Functional Languages: 15th International Workshop, IFL 2003, Edinburgh, UK, September 8–11, 2003. Revised Papers*, volume 3145 of *Lecture Notes in Computer Science*, pages 86–101. Springer-Verlag, Berlin, 2004.

[vKSvE07] R. van Kesteren, O. Shkaravska, and M. van Eekelen. Inferring static non-monotonically sized types through testing. In Rachid Echahed, editor, 16th *International Workshop on Functional and (Constraint) Logic Programming (WFLP07), Paris, France*, pages 123 – 139. CNAM, France, 2007.