Proving the Correctness of Reactive Systems Using Sized Types

John Hughes Lars Pareto* Amr Sabry*

Department of Computer Science
Chalmers University
412 96 Göteborg

{ rjmh, pareto, sabry }@cs.chalmers.se

Abstract

We have designed and implemented a type-based analysis for proving some basic properties of reactive systems. The analysis manipulates rich type expressions that contain information about the sizes of recursively defined data structures. Sized types are useful for detecting deadlocks, nontermination, and other errors in embedded programs. To establish the soundness of the analysis we have developed an appropriate semantic model of sized types.

1 Embedded Functional Programs

In a reactive system, the control software must continuously react to inputs from the environment. We distinguish a class of systems where the embedded programs can be naturally expressed as functional programs manipulating streams. This class of programs appears to be large enough for many purposes [2] and is the core of more expressive formalisms that accommodate asynchronous events, non-determinism, etc.

The fundamental criterion for the correctness of programs embedded in reactive systems is *liveness*. Indeed, before considering the properties of the output, we must ensure that there is *some* output in the first place: the program must continuously react to the input streams by producing elements on the output streams. This latter property may fail in various ways:

- the computation of a stream element may depend on itself creating a "black hole," or
- the computation of one of the output streams may demand elements from some input stream at different rates, which requires unbounded buffering, or
- the computation of a stream element may exhaust the physical resources of the machine or even diverge.

*Supported by NUTEK (Swedish National Board for Industrial and Technical Development) grant 5321-93-2833.

To appear in: The 23rd Annual ACM Symposium on Principles of Programming Languages, January 1996

To support the use of high-level functional languages in embedded systems we have developed an analysis that checks the fundamental correctness property of an embedded functional program, *i.e.*, that the computation of each stream element terminates. The main component of our analysis is a (non-standard) type system that can express bounds on the sizes of recursive data structures. Experience with the implementation indicates that the system works remarkably well for small but realistic programs.

The next section motivates the use of sized types for reasoning about reactive systems. Section 3 introduces the syntax and semantics of a small functional language with sized types. The next two sections present the type inference rules, establish their soundness with respect to our semantic model of types, and give some examples that illustrate some of their strengths and weaknesses. Sections 6 and 7 deal with the details of the implementation and our experience in using it. Before concluding we review related work.

2 Sized Types

Various basic properties of reactive systems can be established using a notion of "sized types." We informally motivate this notion and its use in reasoning about stream computations using some examples.

2.1 Productivity

In a conventional lazy functional language, the datatype of streams of natural numbers would be defined as:

$$\textbf{data}\ \textsc{St} = \mathsf{Mk}\ \textsc{Nat}\ \textsc{St}.$$

The declaration introduces a new constructor Mk which, given a natural number and a stream produces a new stream, *i.e.*, it has type $\textsc{Nat} \to \textsc{St} \to \textsc{St}$. Two sample programs that use this datatype are:

$$\begin{array}{rcll} head\,(\mathsf{Mk}\ n\ s) & = & n & \text{of type } \textsc{St} \to \textsc{Nat} \\ tail\,(\mathsf{Mk}\ n\ s) & = & s & \text{of type } \textsc{St} \to \textsc{St} \end{array}$$

A more interesting program is:

$$\textbf{letrec}\ ones\ =\ \mathsf{Mk}\ 1\ ones\ \textbf{in}\ ones$$

which computes an infinite stream of 1's. In other words, for any natural number i, a request for the first i elements of the stream is *guaranteed* to be processed in finite time: the program is *productive* [5, 17].

A slight modification of the program to:

$$\textbf{letrec}\ ones' = \mathsf{Mk}\ 1\ (tail\ ones')\ \textbf{in}\ ones'$$

is *not* productive; it cannot compute the first i elements of the stream for any $i > 1$. To understand the problem, assume that after unfolding the recursion i times, we obtain a stream s with $i+1$ elements. The recursive call then computes $(tail\ s)$ which has i elements and adds one element to produce a stream that has $i+1$ elements: each recursive call is attempting to construct a stream with no more elements than the previous call and no new elements are added to the stream after the first one.

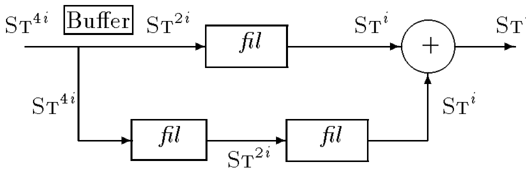
Our intuitive analysis of the behavior of the programs can be formalised using a (non-standard) type system. The system includes a new family of types ST^i (streams with at least i elements), which can express the more informative type $\forall i.\ \mathrm{NAT} \rightarrow \mathrm{ST}^i \rightarrow \mathrm{ST}^{i+1}$ for the constructor Mk. Consequently, for the previous programs, we get the intuitive types $\forall i.\ \mathrm{ST}^{i+1} \rightarrow \mathrm{NAT}$ and $\forall i.\ \mathrm{ST}^{i+1} \rightarrow \mathrm{ST}^i$ for *head* and *tail* respectively. The type of *ones* is $\forall i.\ \mathrm{ST}^i$ which indicates that it is a stream with an infinite number of elements. The program *ones'* is (rightly) rejected.

2.2 Memory Leaks

Using sized types, we can not only guarantee that some streams are productive, but also establish that some functions are unsafe as their computation requires unbounded space. For example, consider the program:

$$\begin{array}{l}\textbf{letrec}\ fil\ (\mathsf{Mk}\ n_1\ (\mathsf{Mk}\ n_2\ s)) = \mathsf{Mk}\ (avg\ n_1\ n_2)\ (fil\ s)\\ \textbf{in}\ \lambda s.stream\text{-}add\ (fil\ s)\ (fil\ (fil\ s))\end{array}$$

The function *fil* represents an idealised digital filter and should have type $\forall i.\mathrm{ST}^{2i} \rightarrow \mathrm{ST}^i$ since it requires two elements from its input stream to compute one element of its output stream. Using the following diagram, we can informally calculate lower bounds on the sizes of all the streams used in the program:



The figure reveals that for every i output elements, we need $4i$ input elements. These latter elements are all consumed along the bottom path. However only $2i$ elements are consumed along the top path and the remaining $2i$ elements must be buffered. The size information reveals that the program would be unsafe in a reactive system as it is impossible to implement a buffer of size $2i$ for all i.

2.3 Termination

In the above example, we claimed that the type of *fil* is $\forall i.\mathrm{ST}^{2i} \rightarrow \mathrm{ST}^i$. In other words, we claimed that given 2 elements, the function *fil* is *guaranteed* to produce one element in finite time. By inspection of the definition of *fil* it is apparent that such a claim can only be justified if we can prove that the function *avg* terminates!

In general we might have to prove that an arbitrary function of the natural numbers terminates. To this end our system must include approximations NAT_i to the datatype NAT of natural numbers. Intuitively an approximation NAT_i represents numbers with *at most* i constructors.[1] Using this new family of sized types, we can prove that the type of *avg* is $\forall kl.\mathrm{NAT}_{2k} \rightarrow \mathrm{NAT}_{2l} \rightarrow \mathrm{NAT}_{k+l}$. Not only does such a type guarantee the termination of the function, but it also provides useful information.

3 Syntax & Semantics

For the purpose of presentation, we restrict our attention to a small prototypical lazy functional language.

3.1 Syntax

A program consists of a number of global declarations of user-defined datatypes followed by a term. We distinguish two kinds of datatype declarations:

$$\begin{array}{rcl}\textbf{data}\ D\ \overline{t_i} & = & \mathsf{Con}_1\ \overline{\tau_j} + \ldots + \mathsf{Con}_m\ \overline{\tau_k}\\ \textbf{codata}\ D\ \overline{t_i} & = & \mathsf{Con}_1\ \overline{\tau_j} + \ldots + \mathsf{Con}_m\ \overline{\tau_k}\end{array}$$

Each declaration introduces a datatype name D possibly parametrised by a vector of type variables $\overline{t_i}$. The right hand side of the declaration introduces a number of term constructors for the datatype $\mathsf{Con}_1, \ldots, \mathsf{Con}_m$ and specifies the types of their arguments. For example, possible declarations for natural numbers, streams, and lists are:

$$\begin{array}{rcl}\textbf{data}\ \mathrm{NAT} & = & \mathsf{Zero} + \mathsf{Succ}\ \mathrm{NAT}\\ \textbf{data}\ \mathrm{FSTREAM}\ t & = & \mathsf{FMk}\ t\ (\mathrm{FSTREAM}\ t)\\ \textbf{codata}\ \mathrm{STREAM}\ t & = & \mathsf{Mk}\ t\ (\mathrm{STREAM}\ t)\\ \textbf{data}\ \mathrm{LIST}\ t & = & \mathsf{Nil} + \mathsf{Cons}\ t\ (\mathrm{LIST}\ t)\\ \textbf{codata}\ \mathrm{ILIST}\ t & = & \mathsf{INil} + \mathsf{ICons}\ t\ (\mathrm{ILIST}\ t)\end{array}$$

Intuitively a **data** declaration signals that the user is interested in the *finite* elements of the datatype. In contrast a **codata** declaration signals that the user is also interested in the *infinite* elements of the datatype. Thus NAT is the set of natural numbers, FSTREAM is the set of all finite streams (the empty set!), STREAM is the set of all finite and infinite streams, LIST is the set of all finite lists, and ILIST is the set of all finite and infinite lists.

With each datatype name, *e.g.*, NAT or STREAM, our system associates a family of types, *e.g.*, NAT_0, NAT_1, ..., NAT_ω or $\mathrm{STREAM}^0\ t$, $\mathrm{STREAM}^1\ t$, ..., $\mathrm{STREAM}^\omega\ t$. The types with natural indices represent the elements of the datatype with the given size bound. The types with the special index ω represent the "limit" of the natural approximations. For example, the type NAT_3 represents the natural numbers $\{0, 1, 2\}$ and NAT_ω represents the entire set of natural numbers. Similarly the type $\mathrm{STREAM}^3\ t$ represents all streams with at least 3 elements of type t and $\mathrm{STREAM}^\omega\ t$ represents all infinite (productive) streams with elements of type t.

---

[1] During our current experiments, we use the inductive definition of the natural numbers (built from 0 and Succ) and prove the termination of arithmetic functions, *e.g.*, addition, from basic principles. In future versions of the system, we intend to provide a library of the common arithmetic functions with their appropriate types.

The precise set of type expressions is inductively generated over infinite sets of datatype names D, type variables $t \in TVar$ and size variables $i \in SVar$:

$$
\begin{array}{lcll}
\sigma & ::= & \forall t.\sigma \mid \forall i.\sigma \mid \tau & \text{(Type Schemes)} \\
\tau & ::= & t \mid \tau \to \tau \mid D_S\ \overline{\tau_k} \mid D^S\ \overline{\tau_k} \mid D_\omega\ \overline{\tau_k} \mid D^\omega\ \overline{\tau_k} &
\end{array}
$$

In addition to the usual quantification over type variables, type schemes may quantify over size variables. Type expressions τ include type variables, function types, and indexed datatype names.

The size indices S can either be the special index ω or some function of size variables. To get precise types, it is tempting to allow arbitrary functions of size variables. For example a precise type for *factorial* would be $\forall i.\mathrm{NAT}_i \to \mathrm{NAT}_{i!}$. However to get a workable system we will restrict ourselves to linear relationships among size variables.

Restriction 3.1 *The size indices can only be linear functions of the size variables:*

$$S \ ::= \ n \in \mathbb{N} \mid i \mid S + S \mid n * S$$

Thus the best type we can express for the *factorial* function is $\forall i.\mathrm{NAT}_i \to \mathrm{NAT}_\omega$.

Finally, the set of terms is inductively generated over infinite sets of variables $x \in Var$, and constructors $\mathsf{Con} \in Con$:

$$
\begin{array}{lcll}
M & ::= & x \mid \lambda x.M \mid M M \mid \mathsf{Con}\ M\ \ldots\ M & \\
 & \mid & \mathbf{case}\ M\ \mathbf{of}\ (P \to M)\ \ldots\ (P \to M) & \\
 & \mid & \mathbf{letrec}\ x = M\ \mathbf{in}\ M & \\
 & & & \\
P & ::= & \mathsf{Con}\ x_1\ \ldots\ x_n & \text{(Patterns)}
\end{array}
$$

The full language also allows mutually recursive definitions but we do not consider them in the remainder of this paper. We require that the patterns in a **case**-expression are exhaustive. The language has more context-sensitive restrictions that will be introduced and motivated during the development of the semantics.

3.2 Semantics of Expressions

We specify the semantics of the terms using a conventional denotational model. The universe $\mathbb{U}$ of denotations is isomorphic to the (coalesced) sum of the following domains:[2]

$$\mathbb{U} \cong Con_\bot \oplus\ [\mathbb{U} + \mathbb{U}] \oplus\ [\mathbb{U} \times \mathbb{U}]_\bot \oplus\ [\mathbb{U} \to \mathbb{U}]_\bot$$

The domain $Con_\bot$ is the flat domain of constructors. The definitions of the domain operations $_\bot$ (lifting), $+$ (separated sum), $\oplus$ (coalesced sum), $\times$ (cartesian product), and $\to$ (continuous function space) are standard. We will use n-ary products as abbreviations of sequences of binary ones. The symbol $\sqsubseteq$ denotes the approximation ordering on the universe of values. Given the universe of values, the semantic function $\mathcal{E}$ that maps a term, and an environment to an element of $\mathbb{U}$. Environments map variables to denotations.

[2]Most semantic models of types include an element **wrong** that formalises run-time errors. We exclude this element from our universe of values because our type system rejects strictly more expressions than a conventional one, *i.e.*, it cannot accept any expression whose evaluation causes run-time errors.

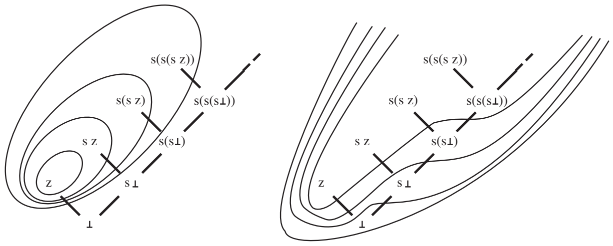


Figure 1: Sample Semantic Types

Definition 3.1 (Meaning Function $\mathcal{E}$) *The definition is standard:*[3]

$$
\begin{array}{rcl}
\mathcal{E} : M \times Env & \to & \mathbb{U} \\
Env : Vars & \to & \mathbb{U}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E}\ (\mathsf{Con}\ M_1 \ldots M_n)\ \ \rho & = & (\mathsf{Con}, \mathcal{E}\ \ M_1\ \ \rho, \ldots, \mathcal{E}\ \ M_n\ \ \rho) \\
\mathcal{E}\ \ x\ \ \rho & = & \rho(x) \\
\mathcal{E}\ \ (\lambda x.M)\ \ \rho & = & f \in [\mathbb{U} \to \mathbb{U}] \\
 & & \mathit{where}\ f(u) = \mathcal{E}\ \ M\ \ \ \rho[u/x] \\
\mathcal{E}\ \ (M\ \ N)\ \ \rho & = & app(\mathcal{E}\ \ M\ \ \ \rho, \mathcal{E}\ \ N\ \ \ \rho) \\
\mathcal{E}\ \ \mathbf{letrec}\ x = M\ \mathbf{in}\ N\ \ \rho & = & \mathcal{E}\ \ N\ \ \ \rho[u/x]\ \mathit{where} \\
 & & u = \ {}_{i \geq 0}\ app^i(f, \bot) \\
 & & \mathit{and}\ f(u) = \mathcal{E}\ \ M\ \ \ \rho[u/x]
\end{array}
$$

$$
\begin{array}{rcl}
app : \mathbb{U} \times \mathbb{U} & \to & \mathbb{U} \\
app(f, u) & = & \begin{array}{ll} f(u) & \mathit{if}\ f \in [\mathbb{U} \to \mathbb{U}] \\ \bot & \mathit{otherwise} \end{array} \\
app^0(f, u) & = & u \\
app^{i+1}(f, u) & = & f(app^i(f, u))
\end{array}
$$

3.3 The Universe of Types

A type is a subset of $\mathbb{U}$ with special properties. To motivate these special properties we examine the meaning of the stream *ones*. According to the semantics in the previous section, the denotation of the stream of 1's is the limit of the following chain:

$$
\begin{array}{rcll}
\mathit{ones} & = & \bot & \in \mathrm{ST}^0 \\
 & & \sqcup\ (\mathsf{Mk}, 1, \bot) & \in \mathrm{ST}^0, \mathrm{ST}^1 \\
 & & \sqcup\ (\mathsf{Mk}, 1, (\mathsf{Mk}, 1, \bot)) & \in \mathrm{ST}^0, \mathrm{ST}^1, \mathrm{ST}^2 \\
 & & \ldots & \ldots
\end{array}
$$

In a conventional semantics for types, there would be *one* set corresponding to the stream datatype, and this set would include all the elements that appear in the chain. For our purposes, we have a family of sets corresponding to each of the types ST^i, and each set only contains part of the chain. For example the set ST^2 must not include $\bot$ or $(\mathsf{Mk}, 1, \bot)$ as we cannot compute the second element of either of these streams in finite time.

A consequence of this property is that not all types include $\bot$, which immediately invalidates the standard seman-

[3]In the definition, we omit the clause for **case**-expressions as well as the explicit isomorphisms mapping elements of the summands into the universe and vice-versa.

tics of types based on ideals [11, 12], or intervals [3].[4] In contrast to these systems, we define the universe of types $\mathbb{U}_T$ to be the collection of all *upward*-closed subsets of $\mathbb{U}$. A set $\mathcal{T}$ is upward-closed if whenever $x \in \mathcal{T}$ then for every $y \sqsupseteq x$, we have $y \in \mathcal{T}$. For example Figure 1 illustrate both the approximations to the **data** and **codata** declarations of natural numbers.

Proposition 3.1 *The universe of types $\mathbb{U}_T$ is a complete lattice ordered by the subset relation; the least upper bound operation is the set-theoretic union, and the greatest lower bound operation is the set-theoretic intersection.*

In order to give a semantics to type expressions we must associate an upward-closed set with each type expression in the language. For composite type expressions, *e.g.*, functions, sums, and products, we introduce corresponding type operations on the universe of types: $\square$, $\boxplus$, and $\boxtimes$. The set $\mathcal{T}_1 \boxplus \mathcal{T}_2$ is defined to be $\{\langle 0, u\rangle \mid u \in \mathcal{T}_1\} \cup \{\langle 1, u\rangle \mid u \in \mathcal{T}_2\}$. The operation $\boxtimes$ is the regular cartesian product. The set $\mathcal{T}_1 \square \mathcal{T}_2$ is defined to be $\{f \in \mathbb{U} \mid \forall x \in \mathcal{T}_1 . app(f, x) \in \mathcal{T}_2\}$.

3.4 Continuity and Ordinals

Given the precise definition of the universe of types and its associated operations, we can now study the semantics of user-defined datatypes. To motivate the definitions we will first examine the semantics of the following three datatypes:

$$\begin{aligned}
\textbf{data}\ \textsc{Nat} &= \mathsf{Zero} + \mathsf{Succ}\ \textsc{Nat} \\
\textbf{codata}\ \textsc{St} &= \mathsf{Mk}\ \textsc{Nat}\ \textsc{St} \\
\textbf{data}\ \textsc{Sp}\ t_1\ t_2 &= \mathsf{Null} \\
&\quad + \mathsf{Put}\ t_2\ (\textsc{Sp}\ t_1\ t_2) \\
&\quad + \mathsf{Get}\ (t_1 \to (\textsc{Sp}\ t_1\ t_2))
\end{aligned}$$

The first two datatypes should be familiar. The datatype $\textsc{Sp}$ is the datatype of *stream processors* from the Fudgets library [6]. This datatype describes three kinds of stream processors:

1. the Null processor,
2. the processor ($\mathsf{Put}\ t_2\ (\textsc{Sp}\ t_1\ t_2)$) that can output an element of type t_2 and then becomes a new stream processor, and
3. the processor ($\mathsf{Get}\ (t_1 \to (\textsc{Sp}\ t_1\ t_2))$) that expects a value of type t_1 and then becomes a new stream processor.

For each datatype D we must associate a set with each of its approximations $0, 1, 2, \ldots$ and ω. The conventional way of explaining the meaning of recursive datatype declarations is by taking the fixed point of a functional over the universe of types. The functionals are easily derivable from the declarations using a function $\mathcal{D}$ whose definition is omitted.

$$\begin{aligned}
\mathcal{D}\ \textsc{Nat} &= F_n \\
\mathcal{D}\ \textsc{St} &= F_s \\
\mathcal{D}\ \textsc{Sp}\ \ \mathcal{T}_1\ \mathcal{T}_2 &= F_p
\end{aligned}$$

where:

$$\begin{aligned}
F_n(\mathcal{T}) &= \{\mathsf{Zero}\} \boxplus (\{\mathsf{Succ}\} \boxtimes \mathcal{T}) \\
F_s(\mathcal{T}) &= \{\mathsf{Mk}\} \boxtimes \textsc{Nat} \boxtimes \mathcal{T} \\
F_p(\mathcal{T}) &= \{\mathsf{Null}\} \\
&\quad \boxplus (\{\mathsf{Put}\} \boxtimes \mathcal{T}_2 \boxtimes \mathcal{T}) \\
&\quad \boxplus (\{\mathsf{Get}\} \boxtimes (\mathcal{T}_1 \square \mathcal{T}))
\end{aligned}$$

We can now calculate for each datatype the meaning of its approximations $0, 1, 2, \ldots$ in a straightforward manner. Simply put, the meaning of the ith approximation to D is the i-fold application of the functional $\mathcal{D}\ D$ to either the bottom element ($\emptyset$) or the top element ($\mathbb{U}$) of the lattice of types. The choice of the initial element depends on whether the datatype was declared as a **data** or **codata**. Thus the meaning of $\textsc{Nat}_2$ is (ignoring injections and tags):

$$\begin{aligned}
F_n^2(\emptyset) &= F_n(F_n(\emptyset)) \\
&= F_n(\{\mathsf{Zero}\} \boxplus (\{\mathsf{Succ}\} \boxtimes \emptyset)) \\
&= F_n(\{\mathsf{Zero}\} \boxplus \emptyset) \\
&= F_n(\{\mathsf{Zero}\}) \\
&= \{\mathsf{Zero}\} \boxplus (\{\mathsf{Succ}\} \boxtimes \{\mathsf{Zero}\}) \\
&= \{\mathsf{Zero}, (\mathsf{Succ}, \mathsf{Zero})\}
\end{aligned}$$

We invite the reader to trace the calculation of $F_s^2(\mathbb{U})$ which is the meaning of $\textsc{St}^2$.

At this point, one might expect that the meaning of the ωth approximation to the datatype is simply the least (for **data**) or greatest (for **codata**) fixed point of the corresponding functional. However in a complete lattice such as our lattice of types, the fixed point of a functional F is calculated as follows:

1. If the functional F is not monotone, then the limit may not exist.
2. If the functional F is monotone but not continuous, then the least fixed point of F is $F^\delta(\emptyset)$ for some ordinal δ where [4:p. 106, Ex. 4.13]:

   $$\begin{aligned}
   F^0(u) &= u \\
   F^{i+1}(u) &= F(F^i(u)) \\
   F^\lambda(u) &= \cup_{j<\lambda} F^j(u) \qquad (\lambda \text{ is a limit ordinal})
   \end{aligned}$$

   The greatest fixed point is calculated by a similar argument using the dual lattice.
3. If the functional F is continuous then the ordinal δ in the previous case is the first limit ordinal ω.

In other words we should not expect the meaning of the ωth approximation to coincide with the least or greatest fixed point of F unless F is continuous. In our running examples, the two functionals F_n and F_s are indeed continuous and the meanings of $\textsc{Nat}_\omega$ and $\textsc{St}^\omega$ can be computed as $_{k<\omega} F_n^k(\emptyset)$ and $_{k<\omega} F_s^k(\mathbb{U})$ respectively.

In contrast the functional F_p is monotone but not continuous and its ωth approximation does not correspond to its least fixed point. This rather technical point has important practical consequences.

Example: Consider the term:

[4] Semantic models based on retracts [16] are not appropriate for another reason; they do not support the form of implicit polymorphism common in the programming languages that we are using.

$$\begin{array}{lrcl} \textbf{letrec}\ \mathit{rid}\ n & = & \textbf{case}\ n\ \textbf{of} & \\ & & \mathsf{Zero} & \rightarrow \mathsf{Null} \\ & & \mathsf{Succ}\ p & \rightarrow \mathsf{Put}\ p\ (\mathit{rid}\ p) \\ \textbf{in}\ \mathsf{Get}\ (\lambda n.\mathit{rid}\ n) & & & \end{array}$$

which represents a stream processor that when given a number n, outputs n elements and then becomes the Null processor. Although the evaluation of the stream processor clearly terminates when given an argument of type NAT_ω, our system cannot handle such a term as the following argument shows.
First we note that in our example the stream processors manipulate natural numbers so the types of the constructors are as follows (where we abbreviate $(\text{SP}_i\ \text{NAT}_\omega\ \text{NAT}_\omega)$ as SPN_i).

$$\begin{array}{rcl} \mathsf{Null} & :: & \forall i.\text{SPN}_{i+1} \\ \mathsf{Put} & :: & \forall i.\text{NAT}_\omega \rightarrow \text{SPN}_i \rightarrow \text{SPN}_{i+1} \\ \mathsf{Get} & :: & \forall i.(\text{NAT}_\omega \rightarrow \text{SPN}_i) \rightarrow \text{SPN}_{i+1} \end{array}$$

Given these types, we note that the meaning of $\text{SPN}_1 = \{\mathsf{Null}\}$. In general the meaning of SPN_i is the set of stream processors that terminate after at most $(i-1)$ Put and Get operations. The set SPN_ω is the union of all the sets SPN_i. In other words it is the set of processes that terminate within a fixed number of operations *regardless of the inputs they receive*. Clearly (Get $(\lambda n.\mathit{rid}\ n)$) is not in the set SPN_ω. On the other hand for each $n \in \text{NAT}_\omega$, $(\mathit{rid}\ n)$ *is* in SPN_ω. It follows that (Get $(\lambda n.\mathit{rid}\ n)$) is in $\text{SPN}_{\omega+1}$. Because the functional F_p is not continuous $\text{SPN}_{\omega+1}$ is *not* identical to SPN_ω. ■

We now have a few alternatives:

1. Allow the declaration of the datatype SP and accept the term (Get $(\lambda n.\mathit{rid}\ n)$). This means that we must be prepared to handle indices that range over ordinals.
2. Allow the declaration of the datatype SP but reject any programs that would require ordinal indices, *i.e.*, reject the term (Get $(\lambda n.\mathit{rid}\ n)$).
3. Reject the declaration of the datatype SP because it defines a non-continuous functional.

For simplicity reasons, we have opted for the third alternative. This choice demands that we find simple syntactic criteria that ensure that all datatype declarations yield continuous functionals. We begin by identifying the problematic positions in a declaration.

Definition 3.2 *A type expression τ occurs in a non-$\cup$-continuous position if:*

(U1) *τ occurs in either argument to $\rightarrow$,*

(U2) *τ occurs in an argument to a codata name, or*

(U3) *τ occurs in argument τ_i to a data name D and τ_i occurs in a non-$\cup$-continuous position in the right hand side of the declaration of D.*

Similary a type expression τ occurs in a non-$\cap$-continuous position if:

(I1) *τ occurs in the left argument to $\rightarrow$,*

(I2) *τ occurs in argument τ_i to a data or codata name and τ_i occurs in a non-$\cap$-continuous position in the right hand side of the declaration of the name.*

The restriction on datatype declarations is now simple.

Restriction 3.2 *In a data declaration the declared type cannot occur in a non-$\cup$-continuous position. In a codata declaration the declared type cannot occur in a non-$\cap$- continuous position.*

We can easily verify our previous claims regarding the continuity of NAT, ST, and SP. For more interesting examples, consider the following declarations:

$$\begin{array}{rcl} \textbf{data}\ \text{ORD} & = & \mathsf{Zero} \\ & & +\ \mathsf{Succ}\ \text{ORD} \\ & & +\ \mathsf{Lim}\ (\text{STREAM ORD}) \\ \textbf{data}\ \text{TREE}\ t & = & \mathsf{Leaf}\ t\ +\ \mathsf{Node}\ (\text{LIST}\ (\text{TREE}\ t)) \\ \textbf{data}\ \text{LIST}\ t & = & \mathsf{Nil}\ +\ \mathsf{Cons}\ t\ (\text{LIST}\ t) \end{array}$$

In the first declaration, ORD occurs as an argument to a **codata** name thus violating condition (U2) above. Indeed ORD is a canonical non-continuous type. In the declaration of TREE, the declared type (TREE t) is passed as an argument to LIST potentially violating condition (U3). To ensure that the condition is not violated we check that t does not occur in non-$\cup$-continuous position in the right hand side of the declaration of LIST. Although ORD and SP are not accepted when declared as **data**, they would be accepted when declared as **codata**. This means that although we cannot reason about the termination of stream processors, we can prove that they do not deadlock.

The major consequence of our restriction is that we can now treat all "infinite" sizes as identical to ω. In other words, we can use identities like $\omega = \omega + 1$ when reasoning about programs.

3.5 Semantics of Types

We can now specify the precise semantics of type expressions.

Definition 3.3 (Type Semantics) *The function $\mathcal{I}$ maps a type expression and two environments β and γ to a semantic type:*

$$\begin{array}{rcl} \mathcal{I} : \sigma \times Env_1 \times Env_2 & \rightarrow & \mathbb{U}_T \\ Env_1 : TVar & \rightarrow & \mathbb{U}_T \\ Env_2 : SVar & \rightarrow & \mathbb{N} \end{array}$$

$$\begin{array}{rcl} \mathcal{I}\ \forall t.\sigma\quad \beta\gamma & = & \cap_{\tau \in \mathbb{U}_T}\ \mathcal{I}\ \sigma\quad \beta[\tau/t]\gamma \\ \mathcal{I}\ \forall i.\sigma\quad \beta\gamma & = & \cap_{k \in \mathbb{N}}\ \mathcal{I}\ \sigma\quad \beta\gamma[k/i] \\ \mathcal{I}\ t\quad \beta\gamma & = & \beta(t) \\ \mathcal{I}\ \tau_1 \rightarrow \tau_2\quad \beta\gamma & = & \mathcal{I}\ \tau_1\quad \beta\gamma \ \square\ \mathcal{I}\ \tau_2\quad \beta\gamma \\ \mathcal{I}\ D_S\ \overline{\tau_k}\quad \beta\gamma & = & F^l(\emptyset) \\ \mathcal{I}\ D_\omega\ \overline{\tau_k}\quad \beta\gamma & = & \cup_{i<\omega}\ F^i(\emptyset) \\ \mathcal{I}\ D^S\ \overline{\tau_k}\quad \beta\gamma & = & F^l(\mathbb{U}) \\ \mathcal{I}\ D^\omega\ \overline{\tau_k}\quad \beta\gamma & = & \cap_{i<\omega}\ F^i(\mathbb{U}) \end{array}$$

where in the last 4 clauses l is the value of the size expression S in the environment γ and F is the functional corresponding to the datatype name D, i.e., $F = \mathcal{D}\ D\ \ \overline{\mathcal{I}\ \tau_k}\ \beta\gamma$.

Proposition 3.2 *The function $\mathcal{I}$ is well-defined.*

Proof Sketch. It is easy to verify that each of the type constructors $\boxplus$, $\boxtimes$, and $\square$ maps upward-closed sets to upward-closed sets. ■

3.6 Testing for $\perp$

A major property of our semantic model is that it is possible to test whether $\perp$ is included in a type, which is crucial if we are to reject programs that may diverge. Given that types are upward-closed sets, the test of whether a type τ includes $\perp$ is quite simple: $\perp \in \tau$ if and only if $\tau = \mathbb{U}$. We define two mutually recursive predicates *empty* and *all* that decide whether the denotation of type expression τ is the empty set or the universe $\mathbb{U}$ respectively:

$$\begin{array}{rll} all\,(D^0) & & \\ all\,(\tau_1 \to \tau_2) & \text{if} & all\,(\tau_2) \text{ or } empty\,(\tau_1) \\ \\ empty\,(D_0) & & \\ empty\,(\tau_1 \to \tau_2) & \text{if} & empty\,(\tau_2) \text{ and } notempty\,(\tau_1) \\ \\ notempty\,(D^S) & & \\ notempty\,(D^\omega) & & \\ notempty\,(\tau_1 \to \tau_2) & \text{if} & notempty\,(\tau_2) \text{ or } empty\,(\tau_1) \end{array}$$

The predicates are sound (but not complete) with respect to our semantic model.

3.7 ω-Types

When a value is too large for its size to be expressed in our restricted language, we give it a size of ω. For example, the type we can actually give *factorial* is $\forall i.\textsc{Nat}_i \to \textsc{Nat}_\omega$. But now we face a problem if we want to compute further with the result. Suppose for example we want to type *factorial* (*factorial* n). To type the outer application of *factorial* we need to give it the type $\textsc{Nat}_\omega \to \textsc{Nat}_\omega$, instantiating i to ω as it were. But our size variables range only over naturals, so it is not valid to do so! Yet in this case the result is clearly sound.

Our goal then is to extend the system to allow the instantiation of size variables to ω. To this end, let us define substitution of ω for a size variable i. The interesting cases are:

$$\begin{array}{rcll} D_S[\omega/i] & = & D_\omega & \text{if } i \in FV(S) \\ & & D_S & \text{otherwise} \\ D^S[\omega/i] & = & D^\omega & \text{if } i \in FV(S) \\ & & D^S & \text{otherwise} \end{array}$$

Thus we simplify sizes involving ω directly to ω; this is justified since our size expressions can only express strictly monotonic functions of their free variables.

We immediately encounter a problem: it is not always sound to instantiate size variables to ω. For example, suppose f has the type $\forall i.\textsc{Stream}^\omega \textsc{Nat}_i \to \textsc{Unit}$. Then we know that f terminates for every *bounded* sequence. But to give f the type $\textsc{Stream}^\omega \textsc{Nat}_\omega \to \textsc{Unit}$ we must show that it terminates for *every* sequence. This need not be the case: consider the function that searches for the first element whose value is less than its position in the stream.

We aim therefore to find syntactic conditions on types that guarantee that our proposed extension is sound. Let π_i be a type expression indexed by a size variable i and abbreviate $\pi_i[\omega/i]$ as π_ω. We want to guarantee that $(\forall i.\pi_i) \rhd \pi_\omega$. Or semantically speaking we want to guarantee that $(\ _i\, \pi_i) \subseteq \pi_\omega$. If this is the case we say π_i is *ω-instantiable.*

We say π_i is *monotonic* if $i \leq j$ implies $\pi_i \subseteq \pi_j$, and *anti-monotonic* in the dual case. These can be recognised easily: π_i is monotonic if i only occurs positively, and anti-monotonic if i only occurs negatively. The definitions of positive and negative occurrences are as usual, with the extension that an occurrence in the size of a **data** type is considered positive, and an occurrence in the size of a **codata** type is considered negative.

Clearly monotonic types are ω-instantiable, because the ω instance is bigger than all the finite ones, but this is hardly interesting. More interesting are the anti-monotonic examples, such as $\textsc{Stream}^i\ \textsc{Bool}$, or those which are mixed, such as $\textsc{Stream}^i\ \textsc{Nat}_i$ or $\textsc{Nat}_i \to \textsc{Nat}_{2i}$. Monotonicity is much too strong a condition to impose. We will therefore define a slightly stronger condition than ω-instantiability, which we can use successfully as an induction hypothesis.

Definition 3.4 *An indexed type π_i is ω-*undershooting *if $\ _i\ \ _{j \geq i}\, \pi_j \subseteq \pi_\omega$. Likewise π_i is ω-*overshooting *if $\pi_\omega \subseteq\ _i\ \ _{j \geq i}\, \pi_j$.*

Intuitively, while ω-instantiability says that the sequence of π_is "tends below" π_ω, ω-undershooting says that the same holds for every suffix of that sequence. Clearly:

- ω-undershooting types are ω-instantiable,
- monotonic types are ω-undershooting. The converse is not necessarily true: for example $\textsc{Stream}^i\ \textsc{Bool}$ is ω-undershooting even though it is not monotonic,
- anti-monotonic types are ω-overshooting. Again the converse is not true: $\textsc{Nat}_i$ is ω-overshooting but it is not anti-monotonic,
- constant types, *e.g.*, the set containing Zero, type variables, and the datatype $\textsc{Unit}$ are ω-undershooting and ω-overshooting, and
- sums and products of ω-overshooting and undershooting types are respectively ω-overshooting and undershooting.

These definitions were motivated by the following result.

Proposition 3.3 *If π_i is ω-overshooting and π'_i is ω-undershooting, then $\pi_i \square \pi'_i$ is ω-undershooting.*

Proof. We show that if $f \in\ _i\ \ _{j \geq i}\, \pi_j \square \pi'_j$ then $f \in \pi_\omega \square \pi'_\omega$. Take an arbitrary $x \in \pi_\omega$: we must show that $f\ x \in \pi'_\omega$. But π_i is ω-overshooting, so $x \in\ _{j \geq i_x}\, \pi_j$ for some i_x. We also know $f \in\ _{j \geq i_f}\, \pi_j \square \pi'_j$ for some i_f. So $f\ x \in\ _{j \geq max(i_x, i_f)}\, \pi'_j$, and therefore since π'_i is ω-undershooting we have $f\ x \in \pi'_\omega$. ■

Thus to check that a function type is ω-instantiable, we need only check that its argument types are ω-overshooting and its result type is ω-undershooting. For example, $\textsc{Nat}_i \to \textsc{Nat}_{2i}$ and $\textsc{Nat}_i \to \textsc{Stream}^i t \to t$ are both ω-instantiable.

$$\frac{}{\sigma \rhd \sigma} \qquad \frac{\sigma_1 \rhd \sigma_2 \qquad \sigma_2 \rhd \sigma_3}{\sigma_1 \rhd \sigma_3}$$

$$\frac{}{D_i\overline{\tau_k} \rhd D_j\overline{\tau_k}} \ \text{if } i \leq j \text{ or } j = \omega \qquad \frac{}{C_i\overline{\tau_k} \rhd C_j\overline{\tau_k}} \ \text{if } j \leq i \text{ or } i = \omega$$

$$\frac{\sigma_3 \ \rhd \ \sigma_1 \qquad \sigma_2 \ \rhd \ \sigma_4}{\sigma_1 \to \sigma_2 \ \rhd \ \sigma_3 \to \sigma_4}$$

$$\frac{\tau_i \rhd \tau_i' \ (i \text{ in covariant position}) \qquad \tau_j' \rhd \tau_j \ (j \text{ in contravariant position})}{D_k \ \tau_1 \ \ldots \ \tau_n \rhd D_k \ \tau_1' \ \ldots \ \tau_n'}$$

$$\frac{\tau_i \rhd \tau_i' \ (i \text{ in covariant position}) \qquad \tau_j' \rhd \tau_j \ (j \text{ in contravariant position})}{C^k \ \tau_1 \ \ldots \ \tau_n \rhd C^k \ \tau_1' \ \ldots \ \tau_n'}$$

Figure 2: The subtyping relation

One might expect a dual result to hold for ω-overshooting function types, but it does not. The type $\text{NAT}_\omega \to \text{NAT}_i$ is not ω-overshooting since ${}_i \ {}_{j \geq i}(\text{NAT}_\omega \to \text{NAT}_i)$ is the set of all *bounded* functions, but $\text{NAT}_\omega \to \text{NAT}_\omega$ also contains the unbounded ones. We settle for anti-monotonicity as a sufficient condition for function types to be ω-overshooting; it is possible to weaken the requirement for function types with finite domains, but we do not consider it worthwhile to do so.

It remains to find general conditions for **data** and **codata** types to have these properties. Let F_i be the functional on types which is iterated to define the semantics of the **data** or **codata** type in question. This functional F_i can very well depend on i: for example if the type is $\text{STREAM}^i \ \text{NAT}_i$ then $F_i(\tau) = \{\mathsf{Mk}\} \boxtimes \text{NAT}_i \boxtimes \tau$. All F_i expressible in our language have the following property which is needed to prove Proposition 3.5.

Lemma 3.4 $F_i(X) \cap F_j(Y) = F_i(X \cap Y) \cap F_j(X \cap Y)$.

Let ϕi be a size expression possibly involving i. Then ϕ is either constant or strictly monotonic. We can prove the following propositions.

Proposition 3.5 *For any family of types π_i, if:*

π_i *ω-undershooting implies $F_i(\pi_i)$ ω-undershooting*

then $F_i^{\phi i}(\emptyset)$, $F_i^{\phi i}(\mathbb{U})$, ${}_j F_i^j(\emptyset)$ and ${}_j F_i^j(\mathbb{U})$ are also ω-undershooting.

Proposition 3.6 *For any family of types π_i, if:*

π_i *ω-overshooting implies $F_i(\pi_i)$ ω-overshooting*

then $F_i^{\phi i}(\emptyset)$ and ${}_j F_i^j(\emptyset)$ are also ω-overshooting.

These results give us a simple way to check that a **data** or **codata** type is ω-undershooting, or that a **data** type is ω-overshooting. The size is irrelevant; we simply check that all components have the same property, assuming that recursive occurrences of the **data** or **codata** type have it. For example, NAT_i and $\text{LIST}_i\text{NAT}_i$ are both ω-over- and -undershooting, as is any other type built only from **data** types. Likewise $\text{STREAM}^i \ \text{NAT}_i$ is ω-undershooting, but we cannot conclude that it is ω-overshooting. And indeed it is not as ${}_i \ {}_{j \geq i} \text{STREAM}^j \ \text{NAT}_j$ is the set of productive streams whose elements are eventually less than their position in the stream, which is *not* a superset of $\text{STREAM}^\omega \ \text{NAT}_\omega$, the set of all productive streams. We fall back on anti-monotonicity as a sufficient condition for a codata type to be ω-overshooting, and we doubt that any much weaker condition can be found.

4 Type System

Before presenting the type inference rules, we present a subtype relation $\sigma_1 \rhd \sigma_2$. The motivation is that since we infer bounds on sizes, it should always be possible to relax the current bound to a less accurate one. Without such flexibility it would be impossible to type programs like (**if** M **then** s **else** (Mk 1 s)), as the two branches would have sizes that differ by 1.

The relation $\rhd$ is defined in Figure 2. It is straightforward to show that the semantic counterpart of the subtyping relation is the subset relation.

Lemma 4.1 *If $\sigma_1 \rhd \sigma_2$ then for all environments β and γ, $\mathcal{I} \ \sigma_1 \ \beta\gamma \subseteq \mathcal{I} \ \sigma_2 \ \beta\gamma$.*

The main innovation in our type inference system is the rule for typing recursive declarations. Before presenting the entire set of rules, we first discuss a simple version of the **letrec** rule that does not include the generalisation of free type variables. The type environment Γ maps variables and constructors to type expressions:

$$\frac{\begin{array}{l} \mathit{all}\,(\tau[0/i]) \\ \Gamma \vdash \lambda x.M : \forall i.\tau \to \tau[i+1/i] \\ \Gamma \cup \{x, \forall i.\tau\} \vdash N : \tau_1 \end{array}}{\Gamma \vdash (\textbf{letrec } x = M \textbf{ in } N) : \tau_1} \quad i \notin FV(\Gamma)$$

The rule has three premises. The main one (in the middle) states that a functional defining a recursive value ought to make progress at each recursive call by producing values of the "next" size. Depending on whether the size variable i indexes a **data** declaration or a **codata** declaration, the "next" size could be a bigger or smaller set respectively. Thus the same typing rule can be used to both prove that computations using **data** objects terminate and computations using **codata** objects are productive. The first premise (the "bottom check") ensures that we can start the iteration of

$$\frac{}{\Gamma \cup \{x:\sigma\} \vdash x:\sigma}\ (\text{Var}) \qquad \frac{\Gamma \cup \{x:\tau_1\} \vdash M:\tau_2}{\Gamma \vdash (\lambda x.M):\tau_1 \to \tau_2}\ (\text{Abs}) \qquad \frac{\Gamma \vdash M:\tau_1 \to \tau_2 \qquad \Gamma \vdash N:\tau_1}{\Gamma \vdash MN:\tau_2}\ (\text{App})$$

$$\frac{}{\Gamma \vdash \text{Con} : \sigma_{\mathsf{Con}}}\ (\text{Con})$$

$$\frac{\Gamma \vdash M:\tau_1 \qquad \vdash^{\text{Pat}} (P_i,\tau_1):\Gamma_i \qquad \Gamma \cup \Gamma_i \vdash M_i:\tau_2 \qquad (i \in \{1..n\})}{\Gamma \vdash \mathbf{case}\ M\ \mathbf{of}\ (P_1 \to M_1)\ldots(P_n \to M_n):\tau_2}\ (\text{Case})$$

$$\frac{\Gamma \vdash M:\forall t.\sigma}{\Gamma \vdash M:\sigma[\tau/t]}\ (\text{Inst}) \qquad \frac{\Gamma \vdash M:\sigma \qquad t \notin FV(\Gamma)}{\Gamma \vdash M:\forall t.\sigma}\ (\text{Gen})$$

$$\frac{\Gamma \vdash M:\forall i.\sigma}{\Gamma \vdash M:\sigma[S/i]}\ (\text{InstS}) \qquad \frac{\Gamma \vdash M:\sigma \qquad i \notin FV(\Gamma)}{\Gamma \vdash M:\forall i.\sigma}\ (\text{GenS})$$

$$\frac{\Gamma \vdash M:\sigma_1 \qquad \sigma_1 \rhd \sigma_2}{\Gamma \vdash M:\sigma_2}\ (\text{Coer}) \qquad \frac{\Gamma \vdash M:\forall i.\pi_i}{\Gamma \vdash M:\pi_\omega}\ \pi_i \text{ is } \omega\text{-undershooting} \quad (\omega)$$

$$\frac{\mathit{all}\,(\tau[0/i]) \quad \Gamma \vdash \lambda x.M:\forall i.\tau \to \tau[i+1/i] \quad \Gamma \cup \{x:\forall i\overline{k}.\forall\overline{t}.\tau\} \vdash N:\tau_1}{\Gamma \vdash (\mathbf{letrec}\ x=M\ \mathbf{in}\ N):\tau_1}\ i \notin FV(\Gamma),\quad \overline{k},\overline{t} \subseteq FV(\tau)\backslash FV(\Gamma) \quad (\text{Rec})$$

Figure 3: Rules for Type Inference

the functional in the first place. The last one concludes (using an implicit induction principle) that the recursive object can be safely used at any size.

To motivate the importance of the "bottom check" we present the following example which illustrates that the rule becomes unsound if we omit it.

Example: Consider the term:

$$\begin{aligned} s\ = {} & \mathbf{if}\ \mathit{head}\ s \\ & \mathbf{then}\ \mathsf{Mk\ True}\ s \\ & \mathbf{else}\ \mathsf{Mk\ False}\ s \end{aligned}$$

It is easy to verify that the program diverges (*i.e.*, the stream s is not productive). However assuming that s is of type ST^{i+1}, the right hand side is of type ST^{i+2}. In other words, the functional is making progress at each recursive call as required by the main premise of the **letrec** rule. The failure of the "bottom check" ($\perp \notin \mathrm{ST}^1$) guarantees that the program would be correctly rejected. ■

The rules for type inference are in Figure 3. The rules for variables, procedures, applications, constructors, generalisation and instantiation of type variables do not exploit nor affect the sizes and are standard. The notation σ_{Con} refers to the type of the constructor Con as inferred from the corresponding datatype declaration. The rule for **case**-expressions uses an auxiliary judgement defined as follows:

$$\frac{\mathit{inst}(\sigma_{\mathsf{Con}}) = \tau_1' \to \ldots \to \tau_n' \to \tau}{\vdash^{\text{Pat}} ((\mathit{Con}\ x_1 \ldots x_n), \tau) : \{x_1 : \tau_1', \ldots, x_n : \tau_n'\}}$$

where *inst* provides a generic instance of a type scheme.

The rest of the inference rules rely on the size annotations. First we can generalise and instantiate size variables in the natural way as in (GenS) and (InstS) respectively. Second we can coerce any type to a less precise type by relaxing the bounds on the sizes (Coer). Third we can instantiate size variables that index ω-undershooting type expressions to ω. Finally when typing recursive declarations (Rec), we must ensure that the resulting computations are terminating and/or productive.

The main technical result of the paper is the soundness of the type system.

Theorem 4.2 (Type Soundness) *If we can prove that M has type σ, then the denotation of M is indeed an element of the type denoted by σ.*

Proof. The proof proceeds by induction on the height of the type derivation and proceeds by case analysis on the last rule. The inductive hypothesis states that:

> If $\Gamma \vdash M : \sigma$ then for all ρ, β, γ such that the relation $\mathbb{R}(\Gamma, \rho, \beta, \gamma)$ holds, $\mathcal{E}\ \ M\ \ \rho \in \mathcal{I}\ \ \sigma\ \ \beta\gamma$.

The relation $\mathbb{R}$ is defined as:

$$\mathbb{R}(\Gamma, \rho, \beta, \gamma) \text{ if } \forall\{x:\sigma\} \in \Gamma.\mathcal{E}\ \ x\ \ \rho \in \mathcal{I}\ \ \sigma\ \ \beta\gamma.$$

Most of the cases in the proof are standard. The subtyping case follows by Lemma 4.1. The soundness of the typing rule for the **case**-expression requires that that the patterns are exhaustive. The soundness of the ω-instantiation rule was sketched in Section 3.7. We present the **letrec** case in detail:

Assume $\Gamma \vdash (\mathbf{letrec}\ x = M\ \mathbf{in}\ N) : \tau_1$ and $\mathbb{R}(\Gamma, \rho, \beta, \gamma)$ for some ρ, β, and γ. We want to show that:

$$\begin{aligned} \mathcal{E}\ (\mathbf{letrec}\ x = M\ \mathbf{in}\ N)\ \ \rho = {} & \mathcal{E}\ \ N\ \ \rho[u/x] \\ & \text{where } u = \sqcup_k f^k(\perp), \\ & \text{and } f = \mathcal{E}\ \ \lambda x.M\ \ \rho \\ \in {} & \mathcal{I}\ \ \tau_1\ \ \beta\gamma \end{aligned}$$

By the inductive hypothesis, we get:

$$\begin{aligned} f & = \mathcal{E}\ \ \lambda x.M\ \ \rho \\ & \in \mathcal{I}\ \ \forall i.\tau \to \tau[i+1/i]\ \ \beta\gamma \\ & \in \cap_k\ \mathcal{I}\ \ \tau\ \ \beta\gamma[k/i] \,\square\, \mathcal{I}\ \ \tau\ \ \beta\gamma[k+1/i] \qquad (*) \end{aligned}$$

We now claim that:

$$\forall k \in \mathbb{N}.f^k(\perp) \in \mathcal{I}\ \ \tau\ \ \beta\gamma[k/i] \qquad (\dagger)$$

Modulo the obligation to prove the claim (†), we can establish our result as follows. Let $u = \sqcup_k f^k(\bot)$, then:

$u \sqsupseteq f^k(\bot)$	for all k
$\therefore u \in \mathcal{I}\ \tau\ \ \beta\gamma[k/i]$	for all k, since types are upward closed
$\therefore u \in \cap_k\ \mathcal{I}\ \tau\ \ \beta\gamma[k/i]$	
$\therefore \mathbb{R}(\Gamma \cup \{x, \forall i.\tau\}, \rho[u/x], \beta, \gamma)$	
$\therefore \mathcal{E}\ \ N\ \ \rho[u/x] \in \mathcal{I}\ \tau_1\ \ \beta\gamma$	inductive hypothesis

To complete the argument we must prove our claim (†). The proof proceeds by induction on k. If $k = 0$ then the claim reduces to $\bot \in \mathcal{I}\ \tau\ \ \beta\gamma[0/i]$ which is an immediate consequence of the bottom check. If $k = j + 1$ then:

$f^j(\bot) \in \mathcal{I}\ \tau\ \ \beta\gamma[j/i]$	inductive hypothesis
$\therefore f(f^j(\bot)) \in \mathcal{I}\ \tau\ \ \beta\gamma[j+1/i]$	property (∗) and definition of $\square$

■

5 Using the Type System

Before talking about the actual implementation, we will attempt to gain some intuition about the strengths and weaknesses of our system using some small examples.

5.1 Primitive Recursion: Reverse

Our system is strong enough to prove termination or productivity of functions in primitive recursive form. For example, assuming that the *append* function for lists can be given the type (cf. Section 6):

$$append :: \forall ij.\forall t.\textsc{List}_i\ t \to \textsc{List}_{j+1}\ t \to \textsc{List}_{i+j}\ t$$

we can prove that the naïve *reverse* function can be typed as follows:

```
reverse :: ∀i.∀t.LIST_i t → LIST_i t
reverse xs =
  case xs of
    Nil → Nil
    Cons y ys → append (reverse ys) (Cons y Nil)
```

The derivation of the type proceeds as follows:

- Assume $reverse :: \textsc{List}_i\ t \to \textsc{List}_i\ t$.
- Assume $xs :: \textsc{List}_{i+1}\ t$.
- Check that the Nil branch has type $\textsc{List}_{i+1}\ t$ (direct).
- Assume $y :: t$ and $ys :: \textsc{List}_i\ t$.
- Conclude $reverse\ ys :: \textsc{List}_i\ t$.
- Conclude (Cons y Nil) $:: \textsc{List}_2\ t$ (*not* $\textsc{List}_1\ t$ as you might expect — it has two constructors).
- Instantiate the type of *append*, and conclude that:

  $$append\ (reverse\ ys)\ (\mathsf{Cons}\ y\ \mathsf{Nil}) :: \textsc{List}_{i+1}\ t$$

  as required. Notice that if we had given *append* a less precise type, omitting the '+1', we would have been unable to draw this conclusion.
- The right hand side of *reverse*'s definition has the type $\textsc{List}_{i+1}\ t \to \textsc{List}_{i+1}\ t$ which satisfies the main premise of the typing rule for recursion.
- The bottom check: verify $all\ (\textsc{List}_0\ t \to \textsc{List}_0\ t)$. It holds because $empty\ (\textsc{List}_0\ t)$ holds.

5.2 Ackerman's Function

Indeed, any function defined by a primitive recursion scheme is accepted by our type system. This includes higher-order primitive recursion, so in particular we can type Ackerman's function. But we are obliged to rewrite it in primitive recursive form.

Consider first the usual first-order definition:

```
ack x y =
  case x of
    Zero → Succ y
    Succ x' → case y of
                Zero → ack x' (Succ Zero)
                Succ y' → ack x' (ack x y')
```

We cannot type this recursion as it stands because there is no argument which gets smaller in every recursive call. We have to reformulate the definition, so that it corresponds to the structure of a termination proof. Ackerman's function can be proved terminating by a double induction, first on x, and then, for a fixed x, on y. We therefore re-express it using *two* recursive functions, one recurring on x, and the other (higher-order) recurring on y.

```
ack x   = case x of
            Zero → Succ
            Succ x' → h (ack x')
h f y   = case y of
            Zero → f (Succ Zero)
            Succ y' → f (h f y')
```

These definitions can be given the types:

$$ack :: \forall k.\textsc{Nat}_k \to \textsc{Nat}_\omega \to \textsc{Nat}_\omega$$
$$h :: \forall k.(\textsc{Nat}_\omega \to \textsc{Nat}_\omega) \to \textsc{Nat}_k \to \textsc{Nat}_\omega$$

Of course, since our size expressions can only express linear functions we cannot do better than give the result a size ω. Notice that in order to type the application of h in *ack*, we have to instantiate k to ω.

5.3 Shuffling Lists

If we could only accept definitions in primitive recursive form, our type system would not be very interesting. The following example shows that we can accept a wider class of definitions. It defines a function that "shuffles" a list:

```
shuffle xs =
  case xs of
    Nil → Nil
    Cons x xs' → Cons x (shuffle (reverse xs'))
```

This definition is not in primitive recursive form because the argument of the recursive call of *shuffle* is not *xs'*. But it is the same size as *xs'*, and the type of *reverse* is strong enough to tell us that. We can therefore give *shuffle* the type:

$$shuffle :: \forall i.\forall t.\textsc{List}_i\ t \to \textsc{List}_i\ t$$

which is accepted by our system.

5.4 A Problem: Accumulating Parameters

We showed above how to typecheck naïve *reverse*, but in reality of course we want to use the efficient linear definition with an accumulating parameter:

$$\begin{aligned}
\mathit{reverse}\ \mathit{xs} &= \mathit{rev}\ \mathit{xs}\ \mathsf{Nil} \\
\mathit{rev}\ \mathit{xs}\ \mathit{ys} &= \mathbf{case}\ \mathit{xs}\ \mathbf{of} \\
&\qquad \mathsf{Nil} \rightarrow \mathit{ys} \\
&\qquad \mathsf{Cons}\ x\ \mathit{xs'} \rightarrow \mathit{rev}\ \mathit{xs'}\ (\mathsf{Cons}\ x\ \mathit{ys})
\end{aligned}$$

The types we would like to give these definitions are:

$$\begin{aligned}
\mathit{reverse} &:: \forall i.\forall t.\textsc{List}_i\ t \rightarrow \textsc{List}_i\ t \\
\mathit{rev} &:: \forall ij.\forall t.\textsc{List}_i\ t \rightarrow \textsc{List}_{j+1}\ t \rightarrow \textsc{List}_{i+j}\ t
\end{aligned}$$

Unfortunately this type for *rev* is not accepted by our system! To see why, we trace through the typing of *rev*:

- Assume $\mathit{rev} :: \textsc{List}_i\ t \rightarrow \textsc{List}_{j+1}\ t \rightarrow \textsc{List}_{i+j}\ t$.
- Assume $\mathit{xs} :: \textsc{List}_{i+1}\ t$ and $\mathit{ys} :: \textsc{List}_{j+1}\ t$.
- In the Cons branch, assume $\mathit{xs'} :: \textsc{List}_i\ t$.
- Conclude $(\mathsf{Cons}\ x\ \mathit{ys}) :: \textsc{List}_{j+2}\ t$.
- Now the application of *rev* is well-typed only if $j+2 \leq j+1$! This constraint is unsatisfiable and the program is rejected.

These definitions *can* be typed, but only as:

$$\begin{aligned}
\mathit{reverse} &:: \forall i.\forall t.\textsc{List}_i\ t \rightarrow \textsc{List}_\omega\ t \\
\mathit{rev} &:: \forall i.\forall t.\textsc{List}_i\ t \rightarrow \textsc{List}_\omega\ t \rightarrow \textsc{List}_\omega\ t
\end{aligned}$$

But with this type for *reverse*, the definition of *shuffle* above cannot be typed!

The solution seems to be to allow a limited form of polymorphic recursion: over sizes, but not types. Thus when typing the body of *rev*, we would assume the typing:

$$\mathit{rev} :: \forall j.\textsc{List}_i\ t \rightarrow \textsc{List}_{j+1}\ t \rightarrow \textsc{List}_{i+j}\ t$$

and then type the recursive call by instantiating l to $j+1$. The type of the result, $\textsc{List}_{i+j+1}\ t$, matches the $i+1$st instance of the declared type of *rev*, and typechecking would therefore succeed.

We plan to extend our implementation to allow this kind of polymorphic recursion. It is necessary to do so: similar problems will arise whenever an accumulating parameter is used, and this is an important programming technique which we must be able to type accurately.

5.5 Array Bounds Check

Our sized types can also be used to guarantee that array indices are never out of bounds. We view an array as a function from indices to contents. For example an array of t with 6 elements has type $\textsc{Nat}_6 \rightarrow t$. Our type system guarantees that an array with this type has *at least* 6 elements and is accessed with indices which are *less than* 6.

6 Implementation

To get an algorithm asserting type correctness of programs, we must complement our deduction rules rules with a proof strategy. If we want type inference, our algorithm must include an equation system solver. If we want subtype inference, it must solve systems of inequations.

We distinguish type inference algorithms from those for type checking in that the latter rely on type annotations and do not address minimality of a typing.

We have implemented a *type checker* for our sized type system. It requires all let-bound variables of a program to be annotated with sized type signatures, but infers the types for all other expressions.

Its proof strategy is simple. Uses of (Var), (Abs), (App), (Con), (Case) and (Rec) are structurally determined by the program. Other rules are used as follows:

(Inst),(InstS) Are used together with (Var) and (Con).

(Gen),(GenS) Are used at (Letrec) and at the topmost expression.

(Coer) Is used at (App), (Con) and (Case).

The systems of inequations arising in our type system are predicates of the form

$$\forall \bar{i}.(S_1 \leq S_1' \wedge \cdots \wedge S_n \leq S_n')$$

where S_j,S_j' are size expressions. Such a constraint system is said to be *solvable* if there exists a substitution under which the system holds.

In general, the above constraint system formulation is not the only conceivable one. Nothing in our semantic model stops us from using arbitrarily complicated constraint languages. The restriction lies in our abilities to solve the corresponding constraint systems.

Our work does not address the problem of constraint solving. Instead, we exploit recent developments in constraint solving technology via the omega calculator [9, 13]. The formulation of our constraint systems is to a large extent determined by the capabilities of the omega calculator.

The reason for implementing a type checker rather than a type inference algorithm is twofold:

First, if we want to do type inference, a minimal type must exist for every typeable program. It is easy to formulate a constraint language for which this does not hold. Furthermore, as we suggested above, the prospects of constraint solving strongly depend on the expressiveness of our constraint language. If our constraint language is too powerful, the type inference algorithm will be hopelessly incomplete. Finding the right compromise might be tedious, so one of our design choices is to let the constraint language reflect the best available constraint solver regardless if the language can express minimal types or not.

Second, inference for the (Letrec) rule is tricky. In the general case we have to solve an equation system with functions of size variables as unknowns, the solution has to respect our constraint language and it must be minimal. By letting the programmer introduce the unknown function using a type signature, these problems disappear. Although initial experiments with letrec-inference looked promising we found the additional complications too distracting to be worthwhile.

6.1 Technical overview

There are three major steps in the algorithm:

Hindley Milner Inference We check that the program is type correct in terms of ordinary types.

Size Inference We typecheck the program with our sized type system under the assumption that all type signatures are correct.

Constraint solving We solve all the constraints and verify that our inferred types match the type signatures.

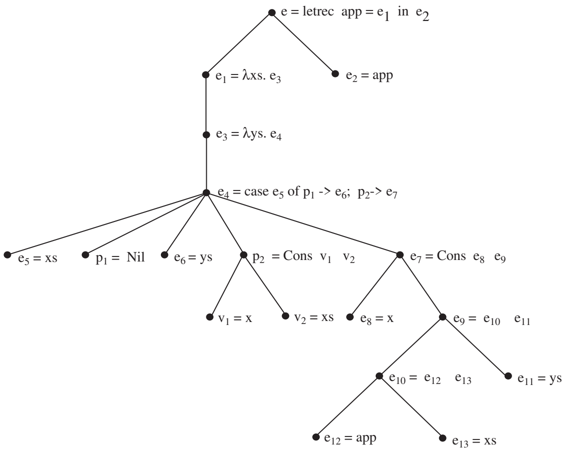
The result of the algorithm is the syntax tree annotated with sized types and a set of error messages. The first is useful in interpreting the latter.

6.2 Example

We proceed through the algorithm step by step and examine the typechecking of the *append* function on lists.

```
letrec
  app :: forall k l a.List#k a -> List#l+1 a -> List#k+l a
  app xs ys = case xs of
              Nil       -> ys
              Cons x xs -> Cons a (app xs ys)
in app
```

We will refer to the nodes of the syntax tree using the names in the figure below. We write $e.a$ for the annotation a of node e.



Hindley Milner inference

The first step is to do standard (Hindley Milner) type inference. We motivate this as follows. The *erasure* $\overline{\sigma}$ of a sized type scheme σ is defined by dropping the size quantifiers and size variables from σ to get an ordinary type. This notion extends to $\overline{\Gamma}$ and $\overline{\Gamma \vdash e\colon\sigma}$. Now, if we erase every judgement in a sized proof tree, then we get a Hindley Milner proof tree.

During inference, we annotate every node of the syntax tree with its Hindley Milner type $e.hm$. For the subtree e_{10} of our example we get

$$e_{10}.hm = \textsc{List}\,\alpha \rightarrow \textsc{List}\,\alpha$$
$$e_{12}.hm = \textsc{List}\,\alpha \rightarrow \textsc{List}\,\alpha \rightarrow \textsc{List}\,\alpha$$
$$e_{13}.hm = \textsc{List}\,\alpha$$

Size inference

The next step is size inference. First we *extend* our ordinary types to sized types: A fresh size variable is added to every type constructor of the type annotations. We refer to the sized type annotations as $e.sz$ which for our example becomes:

$$e_{10}.sz = \textsc{List}_{k20}\,\alpha \rightarrow \textsc{List}_{k21}\,\alpha$$
$$e_{12}.sz = \textsc{List}_{k22}\,\alpha \rightarrow \textsc{List}_{k23}\,\alpha \rightarrow \textsc{List}_{k24}\,\alpha$$
$$e_{13}.sz = \textsc{List}_{k25}\,\alpha$$

Then, we infer our sized types: We traverse the syntax tree and collect *equality* constraints on our extended types to make them appear as in the corresponding proof tree. Consider the typing rule for (App):

$$\frac{\Gamma \vdash M : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash N : \tau_1}{\Gamma \vdash MN : \tau_2}$$

To make our syntax tree annotations match this rule we must unify the codomain type of $e_{12}.sz$ with the type $e_{10}.sz$. Hence we get:

$$\textsc{List}_{k20}\,\alpha \rightarrow \textsc{List}_{k21}\,\alpha = \mathrm{List}_{k23}\,\alpha \rightarrow \textsc{List}_{k24}\,\alpha$$

at the node e_{10}. As we coerce at (App) we add an inequality rather than an equality between the domain type of e_{12} and the type of e_{13} :

$$\textsc{List}_{k25}\,\alpha \leq \textsc{List}_{k22}\,\alpha$$

The nodes e_{12} and e_{13} are both variables so we expect a lookup in Γ possibly followed by an instantiation with fresh variables. At both nodes the type environment Γ includes:

$$\{xs\colon k15, app\colon \textsc{List}_{k30}\,\alpha \rightarrow \textsc{List}_{k31+1}\,\alpha \rightarrow \textsc{List}_{k30+k31}\,\alpha\}$$

so we will simply unify $e_{12}.sz$ and $e_{13}.sz$ with the corresponding types in Γ:

$$\begin{array}{l} \textsc{List}_{k30}\,\alpha \rightarrow \textsc{List}_{k31+1}\,\alpha \rightarrow \textsc{List}_{k30+k31}\,\alpha \quad = \\ \textsc{List}_{k22}\,\alpha \rightarrow \textsc{List}_{k23}\,\alpha \rightarrow \textsc{List}_{k24}\,\alpha \\ \textsc{List}_{k15}\,\alpha = \textsc{List}_{k25}\,\alpha \end{array}$$

The typing of constructors, case expressions and lambda expressions are all variations on the same theme and we do not consider their details. More interesting is letrec. First recall its typing rule:

$$\frac{\begin{array}{c} all\,(\tau[0/i]) \\ i \notin FV(\Gamma), \quad \overline{kt} \subseteq FV(\tau)\backslash FV(\Gamma) \\ \Gamma \vdash \lambda x.M : \forall i.\tau \rightarrow \tau[i+1/i] \quad \Gamma \cup \{x : \forall i\overline{k}\ \overline{t}.\tau\} \vdash N : \tau_1 \end{array}}{\Gamma \vdash (\textbf{letrec}\ x = M\ \textbf{in}\ N) : \tau_1}$$

Note that the letrec rule has a built in (Gen) rule. To understand the strategy of the algorithm, we must consider some general properties of subtype inference: Whenever a constrained type is generalised we must capture its constraints. At instantiation, the captured constraints are added to the constraint set of the typing instantiating it. Now, if we simplify our constraints before generalisation, we will avoid unnecessary constraint instantiations. This is a good thing as constraint manipulation is costly. Although many polymorphic subtype systems have expression forms for bounded quantification, for example $\sigma\,\textbf{where}\,C$ where C is a constraint system [1], our type system does not. As a consequence, simplification must eliminate all constraints, otherwise we can not express the type.

The (Letrec) rule for the implementation has a slightly different appearance than the deduction rule:

$$\frac{\Gamma \cup \{x : \tau\} \vdash M : \tau' \quad \Gamma \cup \{x : x.sig\} \vdash N : \tau_1}{\Gamma \vdash (\textbf{letrec}\ x = M\ \textbf{in}\ N) : \tau_1}$$

$$\begin{array}{rcl} \tau \rightarrow \tau' & = & \textbf{inst}(ft) \\ ft & = & \forall i.\forall\overline{k}.\forall\overline{t}.\tau \rightarrow \tau[^{i+1}/_i] \\ \forall i.\forall\overline{k}.\forall\overline{t}.\tau & = & x.sig \\ aa & = & \langle C, \textbf{FV}(\Gamma), \tau \rightarrow \tau', ft\rangle \end{array}$$

Here **inst** is the polymorphic instantiation function, using fresh variables for all quantifiers. It is used to instantiate ft which is the type scheme used to type the recursion approximator. Note that this formulation of letrec has fused the uses of (Gen) and (Lambda) for the approximator judgement into the rule. The type scheme ft represents the *type family* used in our induction proof. It is defined using the type signature $x.sig$ of the variable x.

The most striking differences of the (Letrec) deduction rule vs. its implementation is the missing conditions for bottom check and generalisation. The reason is simple - they are not checked during size inference. Instead we build an *annotation assumption* aa, which encapsulates everything needed to check these conditions later. An annotation assumption contains: The set of coercions C collected during the proof of M, the free variables of Γ, the inferred type for the recursion approximator and the corresponding type created from the type signature. We will come back to the proofs of annotation assumptions.

Another major difference is that the type signature $x.sig$ is used in the typing of the term N. Hereby, we can continue the inference process assuming that any error lies in the implementation of the function failing to type, not in its specification.

Our little example gives us the following annotation assumption, here updated with the solution to the system of equalities.

$$\langle C, \emptyset, t', s\rangle \quad \text{where}$$

$$\begin{array}{l} C = \{\textsc{List}_{k10+1}\alpha \rightarrow \textsc{List}_{k4}\alpha \rightarrow \textsc{List}_{k6}\alpha \leq \\ \quad \textsc{List}_{k24+1}\alpha' \rightarrow \textsc{List}_{k25+1}\alpha' \rightarrow \textsc{List}_{k24+1+k25}\alpha', \\ \quad \textsc{List}_{k24+1}\alpha' \rightarrow \textsc{List}_{k25+1}\alpha' \rightarrow \textsc{List}_{k24+1+k25}\alpha' \leq \\ \quad \textsc{List}_{k10+1}\alpha \rightarrow \textsc{List}_{k4}\alpha \rightarrow \textsc{List}_{k6}\alpha, \\ \quad \textsc{List}_{k10}\alpha \leq \textsc{List}_{k24}\alpha, \quad \textsc{List}_{k4}\alpha \leq \textsc{List}_{k25+1}\alpha, \\ \quad \alpha \leq \alpha, \quad \textsc{List}_{k24+k25}\alpha \leq \textsc{List}_{k28}\alpha, \\ \quad \textsc{List}_{k28+1}\alpha \leq \textsc{List}_{k6}\alpha, \quad \textsc{List}_{k4}\alpha \leq \textsc{List}_{k6}\alpha\} \end{array}$$

$$\begin{array}{l} t' = (\textsc{List}_{k24}\alpha' \rightarrow \textsc{List}_{k25+1}\alpha' \rightarrow \textsc{List}_{k24+k25}\alpha') \rightarrow \\ \quad \textsc{List}_{k10+1}\alpha \rightarrow \textsc{List}_{k4}\alpha \rightarrow \textsc{List}_{k6}\alpha \end{array}$$

$$\begin{array}{l} s = \forall kl.\forall a.(\textsc{List}_{k}a \rightarrow \textsc{List}_{l+1}a \rightarrow \textsc{List}_{k+l}a) \rightarrow \\ \qquad \textsc{List}_{k+1}a \rightarrow \textsc{List}_{l+1}a \rightarrow \textsc{List}_{k+1+l}a \end{array}$$

Note that the first two constraints form an equality on types. Type equalities added at letrec nodes occasionally involve equalities on size expressions. We use the constraint solving pass to handle such equations. There are six other constraints, two for the alternatives in the case rule, two for the constructor arguments and one for each of the two other applications.

Constraint solving

The constraint systems emerging from our proofs are linear inequalities on natural numbers. The core of our constraint solving pass is the omega calculator [9, 13], a tool for manipulating Presburger formulas. Given a constraint system of integer linear inequations it solves the system and rejects it if unsolvable. The constraint solving pass has 7 stages:

Translation Coercions between types are translated to size constraints using a straightforward implementation of the relation $\rhd$.

Omega test The omega tester either rejects a constraint system or presents a solution to it. A solution consists of a substitution and a simplified constraint system which -in the ideal case- will be empty.

Simplification From our experience, the elimination of redundant constraints in the omega tester needs to be completed with some simple heuristics.

Substitution The resulting substitution is applied to all annotation assumptions.

Annotation check To prove that an assumption about an annotation is correct we unify the type signature with the (pseudo) inferred type, then generalise all variables of the latter with respect to the saved free variables and finally check that the two type schemes are identical.

Bottom check Finally, we bottom check the annotation using a straightforward implementation of the predicate *all*.

There are three possible origins for a constraint. Most are due to the coercions during size inference but some are due to type equivalences. The annotation check equates the body of the annotation with the inferred type by adding two dual coercions to the type constraints. Finally, as our constrained size variables range over naturals rather than integers, we must constrain them to be positive if they are to be solved by the omega calculator.

Let us return to our example. We had 8 coercions in our annotation assumptions, two of them were due to the equivalence in letrec. To equate the annotation with the inferred type we add another two. Translating this set gives us 23 constraints over 8 size variables i.e. 31 constraints in all:

$$\begin{array}{l} \{0 <= k24, \quad 0 <= k10, \quad 0 <= k25, \quad 0 <= k4, \\ \ 0 <= k6, \quad 0 <= k28, \quad 0 <= k, \quad 0 <= l, \\ \ k24 + 1 <= k10 + 1, \quad , \quad k25 + 1 <= k4 \\ \ k6 <= k24 + 1 + k25 \\ \ k10 + 1 <= k24 + 1, \quad k4 <= k25 + 1 \\ \ k24 + 1 + k25 <= k6, \quad k4 <= l + 1 \\ \ k10 <= k24, \quad k4 <= k25 + 1 \\ \ k24 + k25 <= k28, \quad k28 + 1 <= k6 \\ \ k4 <= k6, \quad k <= k24, \quad l + 1 <= k25 + 1 \\ \ k24 + k25 <= k + l, \quad k10 + 1 <= k + 1 \\ \ k + 1 + l <= k6, \quad k24 <= k, \quad k25 + 1 <= l + 1 \\ \ k + l <= k24 + k25, \quad k + 1 <= k10 + 1 \\ \ l + 1 <= k4, \quad k6 <= k + 1 + l\} \end{array}$$

From this constraint system the omega calculator derives the following solution:

$$[k, l, k, k, l, l{+}1, k{+}l{+}1, k{+}l \ / \ k, l, k24, k10, k25, k4, k6, k28]$$

Applied to our annotation assumption we get

$\langle C, \emptyset, t', s\rangle$ where

C is the solved constraint system

$$t' = (\mathrm{LIST}_k\alpha' \to \mathrm{LIST}_{l+1}\alpha' \to \mathrm{LIST}_{k+l}\alpha') \to \mathrm{LIST}_{k+1}\alpha \to \mathrm{LIST}_{l+1}\alpha \to \mathrm{LIST}_{k+l+1}\alpha$$

$$s = \forall kl.\forall a.(\mathrm{LIST}_k a \to \mathrm{LIST}_{l+1} a \to \mathrm{LIST}_{k+l} a) \to \mathrm{LIST}_{k+1} a \to \mathrm{LIST}_{l+1} a \to \mathrm{LIST}_{k+1+l} a$$

As there are no free variables captured by our annotation assumption we generalise all size variables in t' i.e. k, l which are the same as in the signature. Hence we are finished with the annotation check. Remaining is the bottom check. The 0-instantiated type is

$$\mathrm{LIST}_0 a \to \mathrm{LIST}_{l+1} a \to \mathrm{LIST}_l a$$

for which the bottom check holds as the domain type is empty.

7 Using the Implementation

The concrete syntax of our language is similar to the syntax of Haskell [7]. To express our richer set of type expressions, we use `$` to refer to ω and `#` to index both **data** and **codata** declaration names.

7.1 Modularity

As with all type-based analyses, our analysis need not manipulate the entire program at once. It is possible to analyse modules separately and summarise the types of their exported functions in an interface file. Future uses of the functions in the module need only look up the types in the interface file. We have implemented such a rudimentary module system which we illustrate with a small example.

Assume that the modules `Nat` and `Stream` include definitions of typical functions like *add*, *tail*, and *zipWith*. At some previous time, the system has checked the contents of these modules, and summarised the types of their top level definitions in an interface file:

$$\begin{array}{rcl} add & :: & \forall ij.\mathrm{NAT}_i \to \mathrm{NAT}_j \to \mathrm{NAT}_{i+j} \\ tail & :: & \forall i.\forall\alpha.\mathrm{ST}^{i+1}\alpha \to \mathrm{ST}^i\alpha \\ zipWith & :: & \forall i.\forall\alpha\beta\gamma.(\alpha \to \beta \to \gamma) \\ & & \to \mathrm{ST}^i\alpha \to \mathrm{ST}^i\beta \to \mathrm{ST}^i\gamma \end{array}$$

Then we can write the following module:

```
----------------------------------------------------------------
module Fib where

import Nat    -- defines add
import Stream -- defines tail, zipWith

suml :: forall k.Stream#k Nat$ -> Stream#k Nat$ -> Stream#k Nat$
suml = zipWith add

fib1 :: forall k.Stream#k Nat$
fib1 = Mk 0 (Mk 1 (suml fib1 (tail fib1)))

fib2 ::  forall k.Stream#k Nat$
fib2' ::  forall k.Stream#k Nat$
fib2 = Mk 0 fib2'
fib2' = Mk 1 (suml fib2 fib2')
----------------------------------------------------------------
```

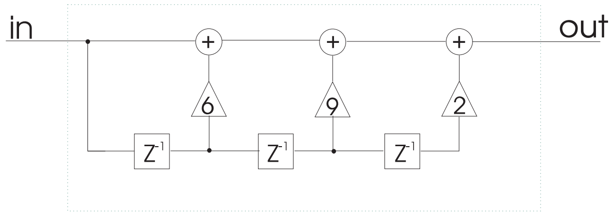


Figure 4: Finite impulse response filter.

When processing the module, *fib1* is rejected as the system can not prove that the application (*tail fib*) will succeed. This is because the structure of the definition does not match the structure of the termination proof for *fib1*. Rewriting the example using two mutually recursive definitions as in *fib2* produces a program that our system accepts.

7.2 Correctness of A Digital Filter

Digital filters are often implemented as a weighted sum of the n last samples of the input signal. Figure 4 illustrates such a filter whose implementation in our language is:

```
----------------------------------------------------------------
module Filter where

import Stream -- defines tail, map, zipWith
import Nat    -- defines add, mul2, mul6, mul9

suml :: forall k.forall a b. Stream#k Nat#a -> Stream#k Nat#b
                             -> Stream#k Nat#a+b
suml = zipWith add

z0 :: forall k.forall a.Stream#k+3 a -> Stream#k a
z1 :: forall k.forall a.Stream#k+2 a -> Stream#k a
z2 :: forall k.forall a.Stream#k+1 a -> Stream#k a
z3 :: forall k.forall a.Stream#k   a -> Stream#k a

z0 = \x -> tail(tail(tail x))
z1 = \x -> tail(tail x)
z2 = tail
z3 = \x -> x

a2 :: forall k l.Stream#k Nat#l -> Stream#k Nat#2*l
a6 :: forall k l.Stream#k Nat#l -> Stream#k Nat#6*l
a9 :: forall k l.Stream#k Nat#l -> Stream#k Nat#9*l

a2 = map mul2
a6 = map mul6
a9 = map mul9

fir :: forall k l.Stream#k+3 Nat#l -> Stream#k Nat#18*l
fir = \i -> (suml (a2 (z3 i))
            (suml (a6 (z2 i))
            (suml (a9 (z1 i))
            (z0 i))))
----------------------------------------------------------------
```

Our system accepts the program. The type of `fir` establishes several points. First the output stream is productive. Second the elements of the output stream are bounded to be no more than 18 times larger than the elements in the input stream. Finally the program will not work unless the environment supplies at least three elements of the input stream.

8 Related Work

The formal notion of productivity is due to Sijtsma [17]. He presents a calculus for proving the productivity of recursive definitions of streams but no automatic analysis. The closest analyses to ours are two recent ones for the estimation of execution times in parallel languages [8, 14]. Both our system and Reistad and Gifford's [14] include similar notions of size and subtyping but nevertheless differ significantly regarding our two main technical contributions. First, our system is the only one that includes a semantic interpretation of the sizes and a proof of soundness. Second, the languages supported by the two systems are different. Reistad and Gifford's system can handle imperative constructs but not user-defined recursive procedures. From our experience, the extension to user-defined recursive procedures is a *major* one that affects the entire system. In contrast the extension to imperative constructs appears to be straightforward.

Our system is also related to several approaches for the formal development of reactive systems using synchronous languages, temporal logics, process calculi, etc [10]. Our system is distinguished by two major properties: productivity and modularity. Indeed in a recent comparison of 18 formal methods for the development of a simple production cell [10], only 6 or 7 implementors could prove the liveness (productivity) of the production cell and only 3 or 4 used a modular solution. Furthermore only 1 system (Focus) combined a proof of liveness with a modular solution *but* only for the *specification* of the program and not for the actual executable code.

On the mathematical side, our approximations to sizes of recursive data structures are apparently related to some hierarchies of recursive functions [15] though the connection is not evident at this point.

9 Conclusion

We have designed and implemented an analysis that guarantees termination and liveness of embedded functional programs. Moreover our analysis can sometimes detect space leaks.

Our immediate goal is to use the analysis to reason about realistic reactive systems written in realistic functional languages for real-time programming [2, 18].

The analysis is based on a new notion of sized types and its associated semantic model which we expect to be applicable in other contexts such as array bounds checking. We believe the same theory and type checking technology can be applied to other programming languages and extended to verify bounds on the run times of computations.

Our longer term goals are to integrate the analysis with a complete compiler so that it can yield more concrete information about the run-time behavior of programs.

References

[1] Aiken, A. S., and Wimmers, E. L. Type inclusion constraints and type inference. In *Functional Programming & Computer Architecture* (June 1993), ACM Press, pp. 31–41.

[2] Broy, M., et al. The design of distributed systems—an introduction to Focus. Tech. Rep. SFB-Bericht Nr. 342/2-2/92 A, Technische Universität München, 1992.

[3] Cartwright, R. Types as intervals. Tech. Rep. TR84-5, Rice University, 1984.

[4] Davey, B. A., and Priestley, H. A. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[5] Dijkstra, E. W. On the productivity of recursive definitions. Personal note EWD 749, University of Texas at Austin, 1980.

[6] Hallgren, T., and Carlsson, M. Programming with Fudgets. In *Advanced Functional Programming* (1995), J. Jeuring and E. Meijer, Eds., Springer Verlag, LNCS 925, pp. 137–182.

[7] Hudak, P., Peyton Jones, S., and Wadler, P. Report on the programming language Haskell, a non-strict purely functional language (version 1.2). *Sigplan Notices* (1992).

[8] Huelsbergen, L., Larus, J. R., and Aiken, A. Using the run-time sizes of data structures to guide parallel-thread creation. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (1994), pp. 79–90.

[9] Kelly, W., et al. *The Omega Library (version 0.91): Interface Guide.* University of Maryland at College Park, 1995.

[10] Lewerentz, C., and Lindner, T. *Formal Development of Reactive Systems: Case Study Production Cell.* Lecture Notes in Computer Science 891. Springer-Verlag, 1995.

[11] MacQueen, D., Plotkin, G., and Sethi, R. An ideal model for recursive polymorphic types. *Information and Control 71*, 1/2 (1986), 95–130.

[12] MacQueen, D., and Sethi, R. A semantic model of types for applicative languages. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (1982), pp. 243–252.

[13] Pugh, W. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM 8* (1992), 102–114.

[14] Reidstad, B., and Gifford, D. K. Static dependent costs for estimating execution time. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (1994), pp. 65–78.

[15] Royer, J. S., and Case, J. *Subrecursive Programming Systems: Complexity and Succinctness.* Boston: Birkhäuser, 1994.

[16] Scott, D. Data types as lattices. *SIAM Journal on Computing 5*, 3 (1976), 522–587.

[17] Sijtsma, B. On the productivity of recursive list definitions. *ACM Transactions on Programming Languages and Systems 11*, 4 (1989), 633–649.

[18] Truvé, S. A new H for real-time programming. Unpublished Manuscript, 1995.