Proving Size Bounds with Dependent Types

Edwin Brady and Kevin Hammond
University of St Andrews

WG 2.11 — 27th January 2006

Introduction and Motivation

Obtaining accurate space and time information about computer programs is important in a number of areas. e.g. Embedded systems.

However, there is a trade-off between effective program analyses and high level abstraction mechanisms.

We aim to implement a (multi-stage) functional language which avoids this trade-off, with:

- Strong compile-time guarantees about resource boundedness.
- Useful abstraction mechanisms e.g. higher order functions, recursive data structures, recursive functions.

We are developing a framework based on dependent types.

Why Dependent Types?

Characteristic feature of a dependent type system:

- Types may be predicated on Values.

This allows us to:

- Express size bounds in a function's type.
- Verify correctness of externally specified size constraints (either user specified, or from an external inference system).
- Express arbitrarily complex constraints.
- Expose proof obligations to the user.

The source language need not be dependently typed; we use a dependently typed core language to *explain* source language programs.

Resource Framework

Basic idea 1: We predicate each user defined type on a $\mathbb{N}$.

e.g. A source language List type ...

```
data List a = nil | cons a (List a)
```

... becomes a dependent "List with size" type:

$$\underline{\text{data}} \quad \frac{A \ : \ \mathbb{N} \rightarrow \star \qquad n \ : \ \mathbb{N}}{\mathsf{List_S}\ A \ : \ \mathbb{N} \rightarrow \star}$$

$$\underline{\text{where}} \quad \frac{}{\mathsf{nil_S} \ : \ \mathsf{List_S}\ A\ 0}$$

$$\frac{x \ : \ A\ an \qquad xs \ : \ \mathsf{List_S}\ A\ xsn}{\mathsf{cons_S}\ x\ xs \ : \ \mathsf{List_S}\ A\ (\mathsf{s}\ xsn)}$$

The framework is independent of the meaning we attach to size. We could choose, e.g., length (as here), required heap cells, total size of list and all elements ...

Resource Framework

Basic idea 2: We pair all values with a proof of a predicate.

$$\underline{\text{data}} \quad \frac{X \ : \ \mathbb{N} \longrightarrow \star \qquad P \ : \ \forall n{:}\mathbb{N}.\ X\ n \longrightarrow \star}{\mathsf{Size}\ X\ P \ : \ \star}$$

$$\underline{\text{where}} \quad \frac{val \ : \ X\ n \qquad p \ : \ P\ n\ val}{\mathsf{size}\ val\ p \ : \ \mathsf{Size}\ X\ P}$$

Each function now returns a Size, rather than a simple value. e.g. a sized list, carrying a proof of a length property:

$$\mathsf{size}\ (\mathsf{cons_S}\ x\ xs)\ (\mathsf{refl}\ (\mathsf{s}\ xsn))$$
$$: \ \mathsf{Size}\ (\mathsf{List}\ A)\ (\lambda n{:}\mathbb{N}.\ \lambda v{:}(\mathsf{List}\ A)\ n.\ n = \mathsf{s}\ xsn)$$

refl constructs a reflexive proof of equality.

Example — Sized **append**

Consider, e.g., the append function, defined in our source language as:

```
append :  List a -> List a -> List a
append nil ys = ys
append (cons x xs) ys = cons x (append xs ys)
```

What size properties does this definition satisfy?

How do we translate it into our framework?

Example — Sized **append**

Translated into our framework, **append** returns values paired with a proof object; we use a high level notation similar to EPIGRAM:

$$\underline{\text{let}} \quad \frac{xs \;:\; \mathsf{List_S}\ A\ xsn \qquad ys \;:\; \mathsf{List_S}\ A\ ysn}{\mathbf{append}\ xs\ ys \;:\; \mathsf{Size\,List_S}\ A\ (\lambda n{:}\mathbb{N}.\ \lambda v{:}\mathsf{List_S}\ A\ n.\ n = xsn + ysn)}$$

$$\mathbf{append}\quad \mathsf{nil_S}\quad ys \;\mapsto\; \mathsf{size}\ ys\ \Box_1$$

$$\mathbf{append}\ (\mathsf{cons_S}\ x\ xs)\ ys \;\mapsto\; \underline{\text{let}}\ (\mathsf{size}\ val\ p) = \mathbf{append}\ xs\ ys\ \underline{\text{in}}$$
$$\mathsf{size}\ (\mathsf{cons_S}\ x\ val)\ \Box_2$$

We need to fill in:

$\Box_1 \;:\; ysn = 0 + ysn$

$\Box_2 \;:\; \mathsf{s}\ n = (\mathsf{s}\ xsn) + ysn$

$val \;:\; \mathsf{List_S}\ A\ n$

$xs \;:\; \mathsf{List_S}\ A\ xsn$

$ys \;:\; \mathsf{List_S}\ A\ ysn$

$p \;:\; n = xsn + ysn$

Example — Sized **append**

Translated into our framework, **append** returns values paired with a proof object; we use a high level notation similar to EPIGRAM:

$$\underline{\text{let}} \quad \frac{xs \;:\; \mathsf{List_S}\ A\ xsn \qquad ys \;:\; \mathsf{List_S}\ A\ ysn}{\mathbf{append}\ xs\ ys \;:\; \mathsf{Size\,List_S}\ A\ (\lambda n{:}\mathbb{N}.\ \lambda v{:}\mathsf{List_S}\ A\ n.\ n = xsn + ysn)}$$

$$\mathbf{append}\quad \mathsf{nil_S}\quad ys \;\mapsto\; \mathsf{size}\ ys\ \Box_1$$

$$\mathbf{append}\ (\mathsf{cons_S}\ x\ xs)\ ys \;\mapsto\; \underline{\text{let}}\ (\mathsf{size}\ val\ p) = \mathbf{append}\ xs\ ys\ \underline{\text{in}}$$
$$\mathsf{size}\ (\mathsf{cons_S}\ x\ val)\ \Box_2$$

We need to fill in:

$\Box_1 \;:\; ysn = ysn$

$\Box_2 \;:\; \mathsf{s}\ n = \mathsf{s}\ (xsn + ysn)$

$val \;:\; \mathsf{List_S}\ A\ n$

$xs \;:\; \mathsf{List_S}\ A\ xsn$

$ys \;:\; \mathsf{List_S}\ A\ ysn$

$p \;:\; n = xsn + ysn$

Example — Sized **append**

The full definition of **append** is:

$$\underline{\text{let}} \quad \frac{xs \ : \ \mathsf{List_S}\ A\ xsn \qquad ys \ : \ \mathsf{List_S}\ A\ ysn}{\mathbf{append}\ xs\ ys \ : \ \mathsf{Size}\ \mathsf{List_S}\ A\ (\lambda n{:}\mathbb{N}.\ \lambda v{:}\mathsf{List_S}\ A\ n.\ n = xsn + ysn)}$$

$$\mathbf{append}\quad \mathsf{nil_S}\quad ys \ \mapsto\ \mathsf{size}\ ys\ (\mathsf{refl}\ ysn)$$

$$\mathbf{append}\ (\mathsf{cons_S}\ x\ xs)\ ys \ \mapsto\ \underline{\text{let}}\ (\mathsf{size}\ val\ p) \ =\ \mathbf{append}\ xs\ ys\ \underline{\text{in}}$$

$$\mathsf{size}\ (\mathsf{cons_S}\ x\ val)\ (\mathbf{refl\_s}\ p)$$

refl_s is a function which lifts a proof into a proof of equality of successors.

Higher Order Functions — **twice**

The framework can also deal with higher order functions, e.g. **twice**:

```
twice :  (a -> a) -> a -> a
twice f x = f (f x)
```

However, the size and predicate depend on the specific instance of **f** used, so the type of **twice** in our framework reflects this:

$$\underline{\text{let}} \quad \frac{f \;:\; \forall sa' {:}\mathbb{N}.\; A\; sa' \rightarrow \mathsf{Size}\; A\; (\lambda n{:}\mathbb{N}.\; \lambda v{:}A\; n.\qquad) \qquad a \;:\; A\; sa}{\mathbf{twice} \qquad\qquad f\; a \;:\; \mathsf{Size}\; A\; (\lambda n{:}\mathbb{N}.\; \lambda v{:}A\; n.\qquad)}$$

Higher Order Functions — **twice**

The framework can also deal with higher order functions, e.g. **twice**:

```
twice :  (a -> a) -> a -> a
twice f x = f (f x)
```

However, the size and predicate depend on the specific instance of **f** used, so the type of **twice** in our framework reflects this:

$$P \;:\; \forall n{:}\mathbb{N}.\, \forall a{:}A\ n.\, \mathbb{N} \to \star \qquad sf \;:\; \mathbb{N} \to \mathbb{N}$$

$$\underline{\text{let}} \quad \frac{f \;:\; \forall sa'{:}\mathbb{N}.\, A\ sa \to \mathsf{Size}\ A\ (\lambda n{:}\mathbb{N}.\, \lambda v{:}A\ n.\, P\ n\ v\ (sf\ sa)) \qquad a \;:\; A\ as}{\mathbf{twice}\ P\ sf \qquad f\ a \;:\; \mathsf{Size}\ A\ (\lambda n{:}\mathbb{N}.\, \lambda v{:}A\ n.\, P\ n\ v\ (sf\ (sf\ sa)))}$$

Higher Order Functions — **twice**

The framework can also deal with higher order functions, e.g. **twice**:

```
twice :  (a -> a) -> a -> a
twice f x = f (f x)
```

However, the size and predicate depend on the specific instance of **f** used, so the type of **twice** in our framework reflects this:

$$P \;:\; \forall n{:}\mathbb{N}.\, \forall a{:}A\ n.\, \mathbb{N} \to \star \qquad sf \;:\; \mathbb{N} \to \mathbb{N}$$

$$\underline{\text{let}} \quad \frac{f \;:\; \forall sa'{:}\mathbb{N}.\, A\ sa \to \mathsf{Size}\ A\ (\lambda n{:}\mathbb{N}.\, \lambda v{:}A\ n.\, P\ n\ v\ (sf\ sa)) \qquad a \;:\; A\ as}{\mathbf{twice}\ P\ sf \qquad f\ a \;:\; (\mathsf{Size}\ A\ (\lambda n{:}\mathbb{N}.\, \lambda v{:}A\ n.\, P\ n\ v\ (sf\ (sf\ sa))))}$$

$$\begin{aligned} \mathbf{twice}\ P\ sf \qquad f\ x \;\mapsto\; & \underline{\text{let}}\ (\mathsf{size}\ val_1\ p_1) \;=\; f\ x\ \underline{\text{in}} \\ & \underline{\text{let}}\ (\mathsf{size}\ val_2\ p_2) \;=\; f\ val_1\ \underline{\text{in}} \\ & \mathsf{size}\ val_2\ \square_1 \end{aligned}$$

Higher Order Functions — **twice**

The framework can also deal with higher order functions, e.g. **twice**:

```
twice f x = f (f x)
```

However, the size and predicate depend on the specific instance of **f** used, so the type of **twice** in our framework reflects this:

$$\underline{\text{let}} \quad \frac{P \;:\; \forall n{:}\mathbb{N}.\, \forall a{:}A\ n.\, \mathbb{N} \to \star \qquad sf \;:\; \mathbb{N} \to \mathbb{N}}{\begin{array}{l} trans \;:\; P\ sa_1\ a_1\ (sf\ sa) \to P\ sa_2\ a_2\ (sf\ as_1) \to P\ sa_2\ b_2\ (sf\ (sf\ sa)) \\ f \;:\; A\ sa \to (\mathsf{Size}\ A\ (\lambda n{:}\mathbb{N}.\, \lambda v{:}A\ n.\, P\ n\ v\ (sf\ sa))) \qquad a \;:\; A\ as \end{array}}$$

$$\mathbf{twice}\ P\ sf\ trans\ f\ a \;:\; (\mathsf{Size}\ A\ (\lambda n{:}\mathbb{N}.\, \lambda v{:}A\ n.\, P\ n\ v\ (sf\ (sf\ sa))))$$

$$\begin{array}{rcl} \mathbf{twice}\ P sf\ trans\ f\ x & \mapsto & \underline{\text{let}}\ (\mathsf{size}\ val_1\ p_1) \;=\; f\ x\ \underline{\text{in}} \\ & & \underline{\text{let}}\ (\mathsf{size}\ val_2\ p_2) \;=\; f\ val_1\ \underline{\text{in}} \\ & & \mathsf{size}\ val_2\ (trans\ p_1\ p_2) \end{array}$$

Using **twice**

twice itself gives no concrete size information. When we apply it, however, for example to **double** (where $\mathsf{Nat_S}$ is a sized natural number type)...

$$\underline{\text{let}} \quad \frac{i \;:\; \mathsf{Nat_S}\ in}{\mathbf{double}\ i \;:\; \mathsf{Size}\ \mathsf{Nat_S}\ (\lambda n{:}\mathbb{N}.\ \lambda p{:}\mathsf{Nat_S}\ n.\ n = 2 * in)}$$

...we get the obvious cost:

$$\underline{\text{let}} \quad \frac{i \;:\; \mathsf{Nat_S}\ in}{\mathbf{twicedouble}\ i \;:\; \mathsf{Size}\ \mathsf{Nat_S}\ (\lambda n{:}\mathbb{N}.\ \lambda p{:}\mathsf{Nat_S}\ n.\ n = 4 * in)}$$

$$\mathbf{twicedouble}\ i \;\mapsto\; \mathbf{twice}\ (\lambda a, b{:}\mathbb{N}.\ a = b)\ (\lambda n{:}\mathbb{N}.\ 2 * n)\ \square_1\ \mathbf{double}\ i$$

twice also requires us to provide a transitivity proof in $\square_1$:

$$\square_1 \;:\; \forall a, b, c{:}\mathbb{N}.\ a = 2 * b \to b = 2 * c \to a = 2 * (2 * c)$$

This is solved automatically without difficulty.

Higher Order Functions — **fold**

fold is effectively a more general **twice**, applying a function any number of times. We can express this in our source language as follows:

```
fold :  (a -> b -> a) -> a -> List b -> a
fold f a nil = a
fold f a (cons x xs) = f (fold f a xs) x
```

How might we express this in our framework? We have to account for:

- The size effect of f.
- The property that f's size must satisfy.
- Maintaining this property through the recursive calls.

Higher Order Functions — **fold**

The type of **fold**, in our framework:

$$
\begin{array}{l}
P \;:\; \forall n{:}\mathbb{N}.\, A\ n \to \mathbb{N} \to \star \qquad sf \;:\; \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
Prefl \;:\; \forall n{:}\mathbb{N}.\, \forall a{:}A\ n.\, P\ n\ a\ n \\
Ptrans \;:\; \forall an{:}\mathbb{N}.\, \forall a{:}A\ an.\, \forall bn{:}\mathbb{N}.\, \forall b{:}A\ bn.\, \forall cn{:}\mathbb{N}.\, \forall dn{:}\mathbb{N}. \\
\qquad\qquad P\ bn\ b\ dn \to P\ an\ a\ (sf\ bn\ bn) \to P\ an\ a\ (sf\ cn\ dn) \\
f \;:\; A\ sa \to B\ bn' \to \mathsf{Size}\ A\ (\lambda n{:}\mathbb{N}.\, \lambda v{:}A\ n.\, P\ n\ v\ (sf\ sa\ bn')) \\
a \;:\; A\ an \qquad xs \;:\; \mathsf{List_S}\ B\ xsn
\end{array}
$$

let

$$
\mathbf{fold}\ P\ sf\ trans\ f\ a\ xs \;:\; \mathsf{Size}\ A\ (\lambda n{:}\mathbb{N}.\, \lambda v{:}A\ n.\, P\ n\ v\ (\mathbf{foldCost}\ sf\ A\ B\ f\ a\ xs))
$$

foldCost is a function which calculates the cost of folding a specific list, following the same structure as **fold**.

Conclusions and Further Work

Initial results are encouraging:

- Our framework deals with higher order functions and sum types (also done e.g. **partition**, **map**, **filter**, **either** . . .).
- Complex machinery required, but it allows:
  - Explicit checking of proofs of size bounds.
  - Exposing of more complex proof goals to users.
- In a multi-stage setting, type preservation ensures *property* preservation.

There is much further work to do, e.g.:

- Currently working on a theorem proving library for Haskell, to help automate the building of TT terms.
- Extending the framework to deal with other metrics, and to consider staging.