A Dependently Typed Framework for Static Analysis of Program Execution Costs

Edwin Brady and Kevin Hammond

School of Computer Science,
University of St Andrews, St Andrews, Scotland.
Email: eb,kh@dcs.st-and.ac.uk.
Tel: +44-1334-463253, Fax: +44-1334-463278

Abstract. This paper considers the use of dependent types to capture information about dynamic resource usage in a static type system. Dependent types allow us to give *(*explicit) proofs of properties with a program; we present a dependently typed core language TT, and define a framework within this language for representing size metrics and their properties. We give several examples of size bounded programs within this framework and show that we can construct proofs of their size bounds within TT. We further show how the framework handles recursive higher order functions and sum types, and contrast our system with previous work based on sized types.

1 Background and Motivation

Obtaining accurate information about the run-time time and space behaviour of computer software is important in a number of areas. One of the most significant of these is embedded systems. Embedded systems are becoming an increasingly important application area: today, more than 98% of *all* processors are used in embedded systems and the number of processors employed in such systems is increasing year on year. At the same time, the complexity of embedded software is growing apace. Assembly language, until recently the development language of choice, has consequently been supplanted by C/C++, and there is a growing trend towards the use of even higher-level languages. This trend towards increased expressivity is, however, in tension with the need to understand the dynamic run-time behaviour of embedded systems. Such understanding is critical for the construction of resource-bounded software.

Because there is a need for strong guarantees concerning the run-time behavior of embedded software to be available *at compile-time*, existing approaches have usually either focused on restricting the programming language so that only resource-bounded programs are expressible, or else relied on painstaking, and often manual and inaccurate, post-facto performance measurement and analysis. However, restricting the language deprives the programmer of many useful abstraction mechanisms (c.f. [22, 27, 28]). Conversely effective program analyses work at a low level of abstraction, and thus cannot deal effectively with high-level abstraction mechanisms, such as polymorphism,

higher-order functions (e.g. `fold`), algebraic data types (e.g. `Either`), and general recursion.

In this paper we develop a framework based on *dependent types* which is capable of expressing dynamic execution costs in the type system. We focus on a strict, purely functional expression language and exemplify our approach with reference to the size of a data structure. The approach is, however, general and should, in due course, be readily extensible to other metrics such as dynamic heap allocation, stack usage or time.

A key feature of a dependently typed setting is that it is possible to express more complex properties of programs than the usual simply typed frameworks in use in languages such as Standard ML or Haskell. In fact, computation is possible at the type level, and it is also possible to expose proof requirements that must be satisfied. These capabilities are exploited in the framework we present here to allow static calculation of cost bounds; we use type level computation to construct bounds on execution costs. In this way we can *statically* guarantee that costs lie within required limits.

1.1 Dependent Types

The characteristic feature of a dependent type system is that types may be predicated on values. Such systems have traditionally been applied to reasoning and program verification, as in the LEGO [17] and COQ [6] theorem provers. More recent research, however, has led to the use of dependent types in programming itself, for example Cayenne [2] and Epigram [20, 19]. Our aim is to use dependent types to include explicit size information in programs, rather than as an external property. In this way, type checking subsumes checking of these properties.

1.2 Contributions

We have previously used sized type systems such as [15, 24] to represent program execution cost; such systems seem attractive for this purpose because there is a clear link between, for example, data structure size and heap usage. However, there are limits to the expressivity of sized type systems. In particular, there is a limit to the form of expressions we can use to express size, leading to difficulty in giving accurate sizes to higher order functions. In this paper, we explore the benefits of using a dependently typed intermediate language to represent size constraints of a high level program:

– We can express more complex properties than those available in the sized type system; we are not restricted in the constraint language. Since we can write programs at the type level, we can extend the constraint language as we wish. In particular, this gives us more flexibility in expressing the cost of higher order functions. There need be no loss of size information — we can give each program as precise a size predicate as we need.

– With dependent types, we can verify the correctness of constraints given by an external inference system. A program with embedded size constraints is a complete, self-contained, checkable term; correctness of the constraints is verified by the typechecker, a small and well understood (and hence relatively straightforward to verify) program. We do not have to provide soundness or completeness proofs for our framework if we implement it entirely within a system already known to be sound and complete.
– In situations where a sized type inference system is not powerful enough to derive a size recurrence, or where the user requires a weaker constraint on the size, we can allow the user to specify the size constraint by hand and still be able to check it. Dependent types allow us to overcome the limitations of a sized type based inference system — it is always possible for the user to provide hints.
– Where an automated proof construction system is not powerful enough to solve a constraint, we can expose proof obligations to the user, either for the user to solve, or to show that a constraint cannot be satisfied.

It is important to note that we do not use a dependent type system to help *infer* size information; this is left to an external inference system, or to the programmer (or possibly to some extent both). Rather, we use dependent types to verify that the constraints we have are satisfiable.

2 Programming With Dependent Types

We use a strongly normalising type theory with inductive families [9], similar to Luo's UTT [16]. This language, which we call TT, is an enriched lambda calculus, with the usual properties of subject reduction, Church Rosser, and uniqueness of types. The strong normalisation property is guaranteed by allowing only primitive recursion over strictly positive inductive datatypes. This is a dependent type system, with *no syntactic distinction* between types and terms; hence we can have arbitrarily complex terms in types. Full details of TT are given in [4]. For clarity of the presentation here, we use a higher level notation similar to the Epigram notation of [20]. In this section, we give a brief introduction to programming and theorem proving with inductive families.

2.1 Inductive Families

Inductive families are simultaneously defined collections of algebraic data types which can be indexed over values as well as types. For example, we can define a "lists with length" (or vector) type; to do this we first declare a type of natural numbers to represent such lengths, using the natural deduction style notation proposed for Epigram in [20]:

$$\underline{\text{data}} \quad \frac{}{\mathbb{N} : \star} \quad \underline{\text{where}} \quad \frac{}{\mathsf{0} : \mathbb{N}} \quad \frac{n : \mathbb{N}}{\mathsf{s}\, n : \mathbb{N}}$$

It is straightforward to define addition and multiplication by primitive recursion. Then we may make the following declaration of vectors; note that nil only targets vectors of length zero, and cons x xs only targets vectors of length greater than zero:

$$\underline{\text{data}} \quad \frac{A \ : \ \star \quad n \ : \ \mathbb{N}}{\mathsf{Vect}\ A\ n \ : \ \star} \quad \underline{\text{where}} \quad \frac{}{\mathsf{nil} \ : \ \mathsf{Vect}\ A\ 0} \qquad \frac{x \ : \ A \quad xs \ : \ \mathsf{Vect}\ A\ k}{\mathsf{cons}\ x\ xs \ : \ \mathsf{Vect}\ A\ (\mathsf{s}\ k)}$$

We leave A and k as implicit arguments to cons; their type can be inferred from the type of Vect. When the type includes explicit length information like this, it follows that a function over that type will express the invariant properties of the length. For example, the type of the following program **vPlus**, which adds corresponding numbers in each vector, expresses the invariant that the input vectors are the same length as the output:

$$\underline{\text{let}} \quad \frac{xs, ys \ : \ \mathsf{Vect}\ A\ n}{\mathbf{vPlus}\ xs\ ys \ : \ \mathsf{Vect}\ A\ n}$$

$$\mathbf{vPlus} \quad \mathsf{nil} \quad \mathsf{nil} \quad \mapsto \mathsf{nil}$$

$$\mathbf{vPlus}\ (\mathsf{cons}\ x\ xs)\ (\mathsf{cons}\ y\ ys) \mapsto \mathsf{cons}\ (x\ +\ y)\ (\mathbf{vPlus}\ xs\ ys)$$

Unlike in a simply typed language, we do not need to give error handling cases when the lengths of the vectors do not match; the typechecker verifies that these cases are impossible.

2.2 Theorem Proving

The dependent type system of TT also allows us to express properties directly. For example, the following heterogeneous definition of equality, due to McBride [18], is built in to TT (rather than introduced as a datatype, so we omit the data keyword):

$$\frac{a \ : \ A \quad b \ : \ B}{a \ = \ b \ : \ \star} \qquad \frac{A \ : \ \star \quad a \ : \ A}{\mathsf{refl}\ a \ : \ a \ = \ a}$$

This definition introduces an infix type constructor, $=$, parametrised over two types; we can declare equality between any two types, but can only construct an instance of equality between two definitionally equal values in the same type; e.g. refl (s 0) is an instance of a proof that s0 $=$ s0. Furthermore, since equality is an ordinary datatype just like $\mathbb{N}$ and Vect, we can also write programs by case analysis on instances of equality, such as the following program which can be viewed as a proof that s respects equality:

$$\underline{\text{let}} \quad \frac{p \ : \ n \ = \ m}{\mathbf{resp\_s}\ p \ : \ (\mathsf{s}\ n) \ = \ (\mathsf{s}\ m)}$$

$$\mathbf{resp\_s}\ (\mathsf{refl}\ n) \mapsto \mathsf{refl}\ (\mathsf{s}\ n)$$

We can also represent more complex properties, such as the less than or equal relation:

$$\underline{\text{data}} \quad \frac{x, y \ : \ \mathbb{N}}{x \leq y \ : \ \star} \quad \underline{\text{where}} \quad \frac{}{\mathsf{leO} \ : \ 0 \leq y} \qquad \frac{p \ : \ x \leq y}{\mathsf{leS}\ p \ : \ \mathsf{s}\ x \leq \mathsf{s}\ y}$$

Note that x and y can be left implicit, as their types (and even their values) can be inferred from the type of the relation. For example, $\mathsf{leS}\ (\mathsf{leS}\ \mathsf{leO})$ could represent a proof of $\mathsf{s}\ (\mathsf{s}\ 0) \leq \mathsf{s}\ (\mathsf{s}\ (\mathsf{s}\ 0))$.

As with equality, given a proof, we can write programs by recursion over the proof. For example, we can write a safe subtraction function (i.e. the result is guaranteed to be non-negative) by primitive recursion over the proof that the second argument is less than or equal to the first:

$$\underline{\text{let}} \quad \frac{n, m \ : \ \mathbb{N} \qquad p \ : \ m \leq n}{\mathbf{minus}\ n\ m\ p \ : \ \mathbb{N}}$$

$$\begin{array}{l} \mathbf{minus} \quad n \quad 0 \quad (\mathsf{leO}\ n) \ \mapsto \ n \\ \mathbf{minus}\ (\mathsf{s}\ n)\ (\mathsf{s}\ m)\ (\mathsf{leS}\ p) \ \mapsto \ \mathbf{minus}\ n\ m\ p \end{array}$$

The values for the arguments n and m are determined by the indices of leO and leS; no case analysis on the numbers themselves is required. The Curry-Howard isomorphism [8, 13] describes this correspondence between proofs and programs. We will exploit this further in developing our framework; by expressing the size properties explicitly in a program's type, we know that a *type correct* program is also a *size correct* program.

3 Dependent Types for Resource Analysis

In previous work [24] we have extended the basic sized type system by incorporating notions of *time*, for time costs, and *latent costs* to capture cost information for higher-order functions. The notion of *size* is now used to obtain information about bounds on the size of function arguments and results. This can in turn be used to calculate time and space costs of executing a function. For example, given *map* : $(\alpha \xrightarrow{fc} \beta) \rightarrow [\alpha]^n \xrightarrow{mapc} [\beta]^n$, we can deduce that the latent cost for *map*, *mapc*, is proportional to $fc \times n$.

In this paper, we consider the use of dependent types to represent program size; thus we can use sized type inference to give size bounds where possible, and represent these bounds in a dependently typed framework. In this way we can check that both machine generated and user specified bounds are admissible.

3.1 Source Language

Our source language is a strict, higher order functional language with no partial application (to avoid additional complications such as currying, although in future work we may remove this restriction). The exact details are not important to the dependently typed framework we will develop — it suffices to say that the syntax is similar to Haskell. For the moment, we assume that functions are total, and recursion is primitive.

Ultimately, we hope to apply the methods presented to multi-stage Hume programs [11], ensuring the resource properties we specify are preserved between stages.

Our aim is to describe a resource framework in which all source language programs can be represented *homogeneously* along with proofs of their resource bounds, in order to facilitate an automated translation into TT. There are two aspects to consider: representation of datatypes, and representation of functions.

3.2 Representing Datatypes

The key idea behind our framework is that each user defined type is represented within the framework by a type predicated on a natural number, $\mathbb{N}$. Thus we can embed size information explicitly within a type, and represent proofs of size properties directly in TT code via relations such as $=$ and $\leq$. e.g. Given the user defined type of lists ...

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

...we can create a "sized list" type in our dependently typed framework as follows, where the size of the empty list is 0, and the size of the non-empty list is one more than the size of its tail:

$$\underline{\text{data}} \quad \frac{A \ : \ \mathbb{N} \to \star}{\mathsf{List_S}\ A \ : \ \mathbb{N} \to \star} \quad \underline{\text{where}} \quad \frac{}{\mathsf{nil_S} \ : \ \mathsf{List_S}\ A\ 0} \qquad \frac{x \ : \ A\ xn \qquad xs \ : \ \mathsf{List_S}\ A\ xsn}{\mathsf{cons_S}\ x\ xs \ : \ \mathsf{List_S}\ A\ (\mathsf{s}\ xsn)}$$

Note that the element type A, like all types within the framework, is also predicated on a size. We use the convention that sized types (and their constructors) generated from the source language are given the suffix $_\mathsf{S}$.

We can be flexible as to what the size information for a structure is; whether it be high level information such as the above length of list, or the total size of all elements in the list, or more low level information such as the number of heap cells required to store a structure. Within our framework, the *meaning* of the size index of a family is not important, what matters is that the index satisfies the required properties.

3.3 Representing Functions

With dependent types, we can ensure that the size index of a value satisfies the required properties of that value by specifying those properties in the types of functions. In our TT representation of functions, we would like to be able to capture such properties.

To this end, we define the following Size type, which pairs a sized value with a predicate describing the properties that value respects:

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

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

The size constructor takes a value of type $A\ n$, coupled with a proof that A satisfies the required property, specified by the predicate P.

We use $\mathcal{S}\ (n, v)\ :\ A.\ P$ as a shorthand for $\mathsf{Size}\ A\ (\lambda n : \mathbb{N}.\ \lambda v : A\ n.\ P)$.

We translate[1] functions in the source language to a function in TT which returns a Size; to demonstrate this, let us consider the append function on lists, as shown above for Vect and defined in the source language as:

```
append nil ys = ys
append (cons x xs) ys = cons x (append xs ys)
```

Given the size of `List`, the size of the result is the sum of the sizes of the inputs. In our framework, we express the type of this function as follows:

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

The predicate given to Size requires that any result of this function must be paired with a proof that its size (n) is equal to the sum of the input sizes. In the definition of **append** we show that the return values satisfy the predicate by returning the value paired with a proof object. In many cases these proofs can be constructed automatically via the Omega decision procedure; in the following definition of **append**, we indicate where in the term there are proof objects to construct with the "hole" notation $\Box_n$. As a notational convenience, we allow pattern matching let definitions, in order to extract size information and the return value separately from the recursive call:

$$\begin{array}{lll}
\mathbf{append} & \mathsf{nil_S} & ys \mapsto \mathsf{size}\ ys\ \Box_1 \\
\mathbf{append}\ (\mathsf{cons_S}\ x\ xs) & ys \mapsto \underline{\text{let}}\ (\mathsf{size}\ val\ p) = \mathbf{append}\ xs\ ys\ \underline{\text{in}} & \\
 & \qquad \mathsf{size}\ (\mathsf{cons_S}\ x\ val)\ \Box_2 &
\end{array}$$

Given $val\ :\ \mathsf{List_S}\ A\ n$, $xs\ :\ \mathsf{List_S}\ A\ xsn$ and $y\ :\ \mathsf{List_S}\ A\ ysn$, the types of the holes are as follows:

$$\begin{array}{l}
\Box_1\ :\ ysn = 0 + ysn \\
\Box_2\ :\ \mathsf{s}\ n = (\mathsf{s}\ xsn) + ysn
\end{array}$$

Normalising the goals gives the following equalities to prove (note that in the case of $\Box_2$ reduction is possible because $+$ is defined by recursion on its first argument):

$$\begin{array}{l}
\Box_1\ :\ ysn = ysn \\
\Box_2\ :\ \mathsf{s}\ n = \mathsf{s}\ (xsn + ysn)
\end{array}$$

To prove $\Box_1$ is straightforward, by reflexivity ($\mathsf{refl}\ ysn$). We can prove $\Box_2$ by induction, using $p\ :\ n = xsn + ysn$ and a lemma **resp_s** to show that s respects equality. The full definition of **append**, including these proofs, is as follows:

[1] By hand, currently.

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

$$\begin{array}{l} \mathbf{append} \quad \mathsf{nil_S} \quad ys \mapsto \mathsf{size}\ ys\ (\mathsf{refl}\ ysn) \\ \mathbf{append}\ (\mathsf{cons_S}\ x\ xs)\ ys \mapsto \underline{\text{let}}\ (\mathsf{size}\ val\ p) \ = \ \mathbf{append}\ xs\ ys\ \underline{\text{in}} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{size}\ (\mathsf{cons_S}\ x\ val)\ (\mathbf{resp\_s}\ p) \end{array}$$

Although we have filled in the proof details explicitly here, in many cases this can be done automatically. We cannot do this in general, as the problem is the type inhabitation problem, to find a term $a \ : \ A$ for any A — however, for certain classes of A, there is a method for constructing an appropriate a (if it exists). For the examples in this paper, all proof obligations can be discharged using COQ's `omega` tactic, based on Pugh's Omega calculator [23].

4 Examples

We present several examples of functions defined in our framework. These examples have all been implemented in the COQ theorem prover, using the `omega` tactic to solve *all* of the equational constraints. A COQ script implementing these examples can be found on the first author's web page[2].

For these examples, we define representations of booleans and natural numbers. We give the size of a boolean as zero in both cases — if we are not interested in the size of a type, we can give all of its values a size of zero. We could also give the booleans size one, to represent the fact that values occupy a single heap cell. We give the size of a natural number as the magnitude of the number it represents.

$$\underline{\text{data}} \quad \frac{}{\mathsf{Bool_S} \ : \ \mathbb{N} \to \star} \quad \underline{\text{where}} \quad \frac{}{\mathsf{True_S} \ : \ \mathsf{Bool_S}\ 0} \qquad \frac{}{\mathsf{False_S} \ : \ \mathsf{Bool_S}\ 0}$$

$$\underline{\text{data}} \quad \frac{}{\mathsf{Nat_S} \ : \ \mathbb{N} \to \star} \quad \underline{\text{where}} \quad \frac{}{\mathsf{O_S} \ : \ \mathsf{Nat_S}\ 0} \qquad \frac{i \ : \ \mathsf{Nat_S}\ n}{\mathsf{S_S}\ i \ : \ \mathsf{Nat_S}\ (\mathsf{s}\ n)}$$

We also take operations such as $\underline{\text{if}} \ \ldots \ \underline{\text{then}} \ \ldots \ \underline{\text{else}}$ and $\leq$ as primitive.

For each source function, we identify an output size and its relation to the size of the inputs. Given this, we can construct the type of the TT representation, with the return type as a Size structure. Then the TT function itself is constructed by traversing the syntax tree of the source function, with proof objects inserted where necessary.

4.1 Partitioning a list

The `split` function partitions a list into a pair of lists based on a pivot value (values smaller than or larger than the pivot). The source language definition is:

[2] `http://www.dcs.st-and.ac.uk/~eb/`

```
split pivot nil = (nil,nil)
split pivot (cons x xs) = let (l,r) = split pivot xs in
                              if x<=pivot
                                 then (cons x l, r)
                                 else (l, cons x r)
```

With sized types it is difficult to infer an upper bound cost for this function, as the sizes of each element of the pair are considered independently and so inference assumes the worst case for each. However, it should be clear that the operation is size preserving. We use the following (sized) definition of pairs:

$$\underline{\text{data}} \quad \frac{A,\ B\ :\ \mathbb{N} \to \star \qquad n\ :\ \mathbb{N}}{\mathsf{Pair_S}\ A\ B\ \ :\ \mathbb{N} \to \star} \quad \underline{\text{where}} \quad \frac{a\ :\ A\ an \qquad b\ :\ B\ bn}{\mathsf{mkPair_S}\ a\ b\ \ :\ \mathsf{Pair_S}\ A\ B\ (\mathsf{s}\ (\mathsf{s}\ 0))}$$

As well as the usual projections, $\mathbf{fst}$ and $\mathbf{snd}$, we have $\mathbf{fstS}$ and $\mathbf{sndS}$ to project out the size of each component.

We have chosen to represent the size of a pair as 2 (the number of elements). There are other choices we could make here, e.g. the sum of the sizes of the elements. We choose 2 to demonstrate that a function's size predicate can depend on values as well as sizes.

In writing the type of $\mathbf{split}$, we need to identify the output size of interest, and its relation to the input size. This is a size preserving relationship — the output size of interest is obtained from the sizes of each element of the pair which is returned, and we express in the type that the sum of the sizes of these elements is equal to the size of the input list:

$$\underline{\text{let}} \quad \frac{piv\ :\ \mathbb{N} \qquad xs\ :\ \mathsf{List_S}\ \mathsf{Nat_S}\ xsn}{\begin{array}{l}\mathbf{split}\ piv\ xs\ :\ (\mathcal{S}\ (n, v)\ :\ \mathsf{Pair_S}\ (\mathsf{List_S}\ \mathsf{Nat_S})\ (\mathsf{List_S}\ \mathsf{Nat_S}).\\ \qquad\qquad xsn\ =\ (\mathbf{fstS}\ v)\ +\ (\mathbf{sndS}\ v))\end{array}}$$

The implementation of this follows the structure of the source language implementation, subject to managing the Size structure. We leave the proof obligations as holes to be filled in by the Omega calculator.

$$\begin{array}{lll}\mathbf{split}\ piv & \mathsf{nil_S} & \mapsto\ \mathsf{size}\ (\mathsf{mkPair_S}\ \mathsf{nil_S}\ \mathsf{nil_S})\ \Box_1\\ \mathbf{split}\ piv\ (\mathsf{cons_S}\ x\ xs) & & \mapsto\ \underline{\text{let}}\ (\mathsf{size}\ (\mathsf{mkPair_S}\ l\ r)\ p)\ =\ \mathbf{split}\ piv\ xs\ \underline{\text{in}}\\ & & \quad \underline{\text{if}}\ \ (x \leq piv)\\ & & \quad\ \underline{\text{then}}\ \mathsf{size}\ (\mathsf{mkPair_S}\ (\mathsf{cons_S}\ x\ l)\ r)\ \Box_2\\ & & \quad\ \underline{\text{else}}\ \mathsf{size}\ (\mathsf{mkPair_S}\ l\ (\mathsf{cons_S}\ x\ r))\ \Box_3\end{array}$$

The fact that this definition typechecks (as we have verified in Coq) gives us a strong static guarantee about the properties that the function satisfies. As the type of $\mathbf{split}$ specifies the size relationship between the input and the output, a well typed implementation of $\mathbf{split}$ such as this must satisfy that relationship.

Here, allowing the user to specify size information in advance is beneficial; it is easy to check that the size information is correct although it is difficult to infer. Some methods

are proposed to infer a size for **split** — Vasconcelos describes a method based on abstract interpretation in his forthcoming PhD thesis; Hofmann and Jost in [12] are able to infer the appropriate heap space usage using a linear type system, although this method is restricted to functions which admit a linear bound. More recent work (to be described in Jost's forthcoming PhD thesis) allows cost inference for polymorphic and higher order functions.

4.2 Map

Higher order functions present additional complications, in that we need not only the type of the function argument, but also its size information. Consider the `map` function, defined as follows:

```
map f nil = nil
map f (cons x xs) = cons (f x) (map f xs)
```

Recall that for each function, we identify an output size and its relation to the sizes of the inputs. However, we do not know either of these until we know some more information about `f`. Therefore the solution we adopt is to associate a size predicate and a size function with the function argument. This allows us to express the size of the higher order function in terms of the size of its arguments. In the case of **map**, this is not such a great problem, given the size metric we have chosen for lists — the length of the resulting list has no relationship to the function argument:

$$\underline{\text{let}} \quad \frac{\begin{array}{l} P \;:\; \forall n{:}\mathbb{N}.\, B\; n \to \star \qquad fs \;:\; \mathbb{N} \to \mathbb{N} \\ f \;:\; A\; an' \to \mathcal{S}\;(n,v) \;:\; B.\; P\; n\; v\;(fs\; an') \qquad xs \;:\; \mathsf{List}_{\mathsf{S}}\; A\; xsn \end{array}}{\mathbf{map}\; P\; fs\; f\; xs \;:\; \mathcal{S}\;(n,v) \;:\; B.\; n \;=\; xsn}$$

Again, the definition of **map** is built by following the syntax tree of the source function, with the addition of the size structures:

$$\begin{array}{lll} \mathbf{map}\; P\; fs\; f & \mathsf{nil}_{\mathsf{S}} & \mapsto\; \mathsf{size}\; \mathsf{nil}_{\mathsf{S}}\; \square_1 \\ \mathbf{map}\; P\; fs\; f\; (\mathsf{cons}_{\mathsf{S}}\; x\; xs) & & \mapsto\; \underline{\text{let}}\; (\mathsf{size}\; val_1\; p_1) \;=\; f\; x\; \underline{\text{in}} \\ & & \qquad \underline{\text{let}}\; (\mathsf{size}\; val_2\; p_2) \;=\; \mathbf{map}\; P\; fs\; f\; xs\; \underline{\text{in}} \\ & & \qquad \mathsf{size}\; (\mathsf{cons}_{\mathsf{S}}\; val_1\; val_2)\; \square_2 \end{array}$$

Other higher order functions, such as **zipWith** and **filter** follow a similar pattern, **filter** having a different size metric giving an upper bound on the size of the resulting list, rather than an exact size. In the case of **filter** we can even imagine the framework giving a more precise size where some argument is known — for example, the size of `filter isEven [1,3,...]` is clearly zero; by giving a size expression which depends on the value of the list, this size can be statically determined.

If we were to choose a different size metric for lists, for example total heap usage or maximum stack size, we would be faced with the problem of how to relate the size of f to the size of the resulting list. This is a general problem with the handling of higher order functions, as we shall see in the size of **twice**.

4.3 Twice

The `twice` function simply applies a function to its argument twice; i.e., it is the Church numeral 2. Although conceptually much simpler than `map` it presents a greater difficulty, since the size relationship between input and output is not uniform. It is defined in the source language as follows:

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

Intuitively, the effect of this function on the size of x should be twice the effect of f on x. However, this is hard to represent in a sized type system; the limitations of the expression language at the type level make it difficult to give a precise cost to many functions, and this is especially the case with higher-order functions. Here, the function is applied twice, with different sizes in each case, but a sized type system cannot represent this.

To represent this in our framework, we again associate a size predicate and function with f. However, this is not quite enough —- we also need to know that the predicate satisfies a transitivity property (intuitively, verifying that the predicate respects repeated application). We therefore associate *three* additional arguments with f, being P, fs and $Ptrans$ in the following declaration:

$$\underline{\text{let}} \quad \frac{\begin{array}{l} P \ : \ \forall n{:}\mathbb{N}.\, \forall a{:}A\ n.\, \mathbb{N} \to \star \qquad fs \ : \ \mathbb{N} \to \mathbb{N} \\ Ptrans \ : \ P\ as\ a\ (fs\ cs) \to P\ \ bs\ b\ (fs\ as) \to P\ bs\ b\ (fs\ (fs\ cs)) \\ f \ : \ A\ an' \to (\mathcal{S}\ (n, v) \ : \ A.\ P\ n\ v\ (fs\ an')) \end{array}}{\mathbf{twice}\ P\ fs\ Ptrans\ f\ a \ : \ (\mathcal{S}\ (n, v) \ : \ A.\ P\ n\ v\ (fs\ (fs\ an')))}$$

$Ptrans$ is a predicate transformer; it is a predicate level reflection of function composition. We now have enough information in the type to define **twice**:

$$\begin{array}{lll} \mathbf{twice}\ P\ fs\ Ptrans\ f\ x \mapsto & \underline{\text{let}}\ (\mathsf{size}\ val_1\ p_1) & = f\ x\ \underline{\text{in}} \\ & \underline{\text{let}}\ (\mathsf{size}\ val_2\ p_2) & = f\ val_1\ \underline{\text{in}} \\ & \mathsf{size}\ val_2\ \Box_1 & \end{array}$$

Again, despite the complicated type, the definition of **twice** follows the same form as the original definition and the proof obligations are straightforward to discharge via the $Ptrans$ transformer. In this case, $\Box_1$ can be instantiated simply by $Ptrans\ p_1\ p_2$.

This function on its own does not give any direct size information; this should not be surprising since we do not have size information for a specific f. An *application* of **twice** on the other hand will give us this information. For example, given **double**:

$$\underline{\text{let}} \quad \frac{i \ : \ \mathsf{Nat_S}\ in}{\mathbf{double}\ i \ : \ \mathcal{S}\ (n, p) \ : \ \mathsf{Nat_S}.\ n = 2 * in}$$

We can apply **twice** to **double**, and get the obvious cost:

$$\underline{\text{let}} \quad \frac{i \ : \ \mathsf{Nat_S}\ in}{\mathbf{twicedouble}\ i \ : \ \mathcal{S}\ (n, p) \ : \ \mathsf{Nat_S}.\ n = 4 * in}$$

$$\mathbf{twicedouble}\ i \mapsto \mathbf{twice}\ (\lambda a, b{:}\mathbb{N}.\ a = b)\ (\lambda n{:}\mathbb{N}.\ 2 * n)\ \Box_1\ \mathbf{double}\ i$$

The hole for the transitivity proof is left for the `omega` tactic to fill in; the proposition to be proven is $\forall a, b, c{:}\mathbb{N}.\ a = 2 * b \to b = 2 * c \to a = 2 * (2 * c)$, which is solved by `omega` without difficulty, although the proof term itself is non-trivial.

The size predicates and transitivity proof we use are specific to the instance of f. In a more complex case, where f is itself a higher order function, such as (`twice twice`), type correctness requires that the higher order f is applied to a predicate.

4.4 Fold

The `fold` function may be seen as a generalisation of `twice`, applying a function several times across a list. It can be defined in the source language as follows:

```
fold f a nil = a
fold f a (cons x xs) = f x (fold f a xs)
```

Dealing with this in a sized type system presents a difficult problem; we do not know how many times `f` will be applied, nor do we know the content of the list, so an expression language which can only represent size is not strong enough at the type level.

Since `TT` has a more flexible language at the type level, we can implement `fold`, although it presents more difficulty than **twice**. Firstly, f is now a function of two arguments (and hence so is fs). Secondly, and more importantly, **fold** is recursive and the number of times f is applied depends on the input.

Since we have no partial application, we can write down the types of f and fs as functions of two arguments. The size of the result of the fold depends not only on the size effect of f, but also on the input list itself. Therefore, we create a function **foldSize** to be run at the type level which computes the size of the result of folding a list:

$$\underline{\text{let}}\quad \frac{fs\ :\ \mathbb{N} \to \mathbb{N} \to \mathbb{N} \qquad an\ :\ \mathbb{N} \qquad xs\ :\ \mathsf{List_S}\ B\ xsn}{\mathbf{foldSize}\ fs\ an\ xs\ :\ \mathbb{N}}$$

$$\begin{array}{l}\mathbf{foldSize}\ fs\ an\ \ \mathsf{nil_S} \mapsto an\\ \mathbf{foldSize}\ fs\ an\ (\mathsf{cons_S}\ x_{xn}\ xs) \mapsto fs\ xn\ (\mathbf{foldSize}\ fs\ an\ xs)\end{array}$$

Note that the implicit size of the x argument to $\mathsf{cons_S}$ is used to compute the size; this argument is subscripted. The function follows the structure of the original source language `fold` function, but calculating the size of the result from the size function for `f`. When we create the `TT` version of **fold**, we express the result size in terms of **foldSize**:

$$\underline{\text{let}}\quad \frac{\begin{array}{l}P\ :\ \forall n{:}\mathbb{N}.\ A\ n \to \mathbb{N} \to \star \qquad fs\ :\ \mathbb{N} \to \mathbb{N} \to \mathbb{N}\\ f\ :\ A\ an' \to B\ bn' \to \mathcal{S}\ (n, v)\ :\ A.\ P\ n\ v\ (fs\ an'\ bn')\\ a\ :\ A\ an \qquad xs\ :\ \mathsf{List_S}\ B\ xsn\end{array}}{\begin{array}{l}\mathbf{fold}\ P\ fs\ f\ a\ xs\ :\ (\mathcal{S}\ (n, v)\ :\ A.\\ \qquad\qquad\qquad P\ n\ v\ (\mathbf{foldSize}\ fs\ an\ xs)))\end{array}}$$

$$
\begin{array}{lll}
\mathbf{fold}\ P\ fs\ f\ a & \mathsf{nil_S} & \mapsto\ \mathsf{size}\ a\ \Box_1 \\
\mathbf{fold}\ P\ fs\ f\ a\ (\mathsf{cons_S}\ x\ xs) & & \mapsto\ \underline{\mathsf{let}}\ (\mathsf{size}\ val_1\ p_1)\ = \mathbf{fold}\ P\ fs\ f\ a\ xs\ \underline{\mathsf{in}} \\
 & & \quad\ \underline{\mathsf{let}}\ (\mathsf{size}\ val_2\ p_2)\ = \mathbf{f}\ val_1\ a\ \underline{\mathsf{in}} \\
 & & \quad\ \mathsf{size}\ val_2\ \Box_2
\end{array}
$$

However, there is still a problem; how do we provide the required proofs for $\Box_1$ and $\Box_2$? The solution, as with the *Prefl* predicate transformer used in **twice**, is to require additional predicate transformers as arguments for **fold**. We get the type of these predicate transformers simply by observing which properties we need to prove to complete the definition. The full definition is as follows:

$$
\begin{array}{ll}
 & P\ :\ \forall n{:}\mathbb{N}.\ A\ n \to \mathbb{N} \to \star \quad fs\ :\ \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
 & Prefl\ :\ \forall n{:}\mathbb{N}.\ \forall a{:}A\ n.\ P\ n\ a\ n \\
 & Ptrans\ :\ \forall an{:}\mathbb{N}.\ \forall a{:}A\ an.\ \forall bn{:}\mathbb{N}.\ \forall b{:}A\ bn.\ \forall cn{:}\mathbb{N}.\ \forall dn{:}\mathbb{N}. \\
 & \qquad P\ bn\ b\ dn \to P\ an\ a\ (fs\ bn\ bn) \to P\ an\ a\ (fs\ cn\ dn) \\
 & f\ :\ A\ an' \to B\ bn' \to \mathcal{S}\ (n, v)\ :\ A.\ P\ n\ v\ (fs\ an'\ bn') \\
\underline{\mathsf{let}} & a\ :\ A\ an \quad xs\ :\ \mathsf{List_S}\ B\ xsn \\
 & \hline \\
 & \quad \mathbf{fold}\ P\ fs\ Prefl\ Ptrans\ f\ a\ xs\ :\ (\mathcal{S}\ (n, v)\ :\ A. \\
 & \qquad\qquad\qquad\qquad\qquad\qquad P\ n\ v\ (\mathbf{foldSize}\ fs\ an\ xs)))
\end{array}
$$

$$
\begin{array}{lll}
\mathbf{fold}\ P\ fs\ Prefl\ Ptrans\ f\ a & \mathsf{nil_S} & \mapsto\ \mathsf{size}\ a\ (Prefl\ a) \\
\mathbf{fold}\ P\ fs\ Prefl\ Ptrans\ f\ a\ (\mathsf{cons_S}\ x\ xs) & & \\
 & & \mapsto\ \underline{\mathsf{let}}\ (\mathsf{size}\ val_1\ p_1)\ = \mathbf{fold}\ P\ fs\ Prefl\ Ptrans\ f\ a\ xs\ \underline{\mathsf{in}} \\
 & & \quad\ \underline{\mathsf{let}}\ (\mathsf{size}\ val_2\ p_2)\ = \mathbf{f}\ val_1\ a\ \underline{\mathsf{in}} \\
 & & \quad\ \mathsf{size}\ val_2\ (Ptrans\ p_1\ p_2)
\end{array}
$$

On applying **fold**, we are required to provide appropriate proof terms to instantiate *Prefl* and *Ptrans*. This is to be expected — it is only when we apply a higher order function that we know enough about its usage to expose concrete proof obligations.

4.5 Sum Types

An important class of function involves construction and case analysis on sum types. One example is the `Either` type, which represents a choice between two values in two separate types:

```
data Either a b = Left a | Right b
```

The `either` function implements a generic elimination of `Either`, and applies the appropriate function depending on which constructor was used to build the instance of `Either a b`:

```
either :: (a->c) -> (b->c) -> Either a b -> c
either f g (Left l) = f l
either f g (Right r) = g r
```

Again, there is a problem in representing this in a sized type system; there are two functions, `f` and `g`, only one of which will be applied depending on the value of the

`Either` instance. To represent this with sized types we have to be conservative and assume a worst case, that the function which gives the larger size will be applied.

We can overcome this problem in our framework. To represent `Either` we take the size to be the size of the value which is stored:

$$\underline{\text{data}} \quad \frac{A, B \ : \ \mathbb{N} \to \star}{\mathsf{Either_S}\ A\ B \ : \ \mathbb{N} \ \to \ \star} \quad \underline{\text{where}} \quad \frac{a \ : \ A\ an}{\mathsf{Left_S}\ a \ : \ \mathsf{Either_S}\ A\ B\ an} \qquad \frac{b \ : \ B\ bn}{\mathsf{Right_S}\ b \ : \ \mathsf{Either_S}\ A\ B\ bn}$$

As with **fold**, we create a type level function $\mathbf{eitherS}$ which computes the size of the result of $\mathbf{either}$ given its input. We choose to use the same predicate for l, r and $\mathbf{either}$, and define $\mathbf{either}$ as follows:

$$\underline{\text{let}} \quad \frac{\begin{array}{l} P \ : \ \forall n{:}\mathbb{N}.\ C\ n \to \mathbb{N} \to \star \\ ls \ : \ \mathbb{N} \to \mathbb{N} \quad l \ : \ A\ an \ \to \ \mathcal{S}\ (n, v) \ : \ C.\ P\ n\ v\ (ls\ an) \\ rs \ : \ \mathbb{N} \to \mathbb{N} \quad r \ : \ B\ bn \ \to \ \mathcal{S}\ (n, v) \ : \ C.\ P\ n\ v\ (rs\ bn) \\ x \ : \ \mathsf{Either_S}\ A\ B\ xn \end{array}}{\mathbf{either}\ P\ ls\ l\ rs\ r\ x \ : \ \mathcal{S}\ (n, v) \ : \ C.\ P\ n\ v\ (\mathbf{eitherS}\ ls\ rs\ x)}$$

$$\mathbf{either}\ p\ ls\ l\ rs\ r\ \ (\mathsf{Left_S}\ a) \ \mapsto \ l\ a$$
$$\mathbf{either}\ p\ ls\ l\ rs\ r\ (\mathsf{Right_S}\ b) \mapsto \ r\ b$$

It is possible that we could use different predicates for l and r if $\mathbf{eitherS}$ also computed an appropriate predicate for the return type. It simplifies the definition to require the same predicates, however, and if at the call site we want to use specific l and r with different predicates, it is possible to combine the predicates — if l's result satisfies predicate P and r's result satisfies predicate Q, then each satisfies predicate $P \vee Q$.

4.6 Summary

We have found that it is straightforward to translate a first order function into the TT framework by hand, simply by identifying an output size and the relationship it has with the input size. In the examples we have looked at, the required equational constraints can be satisfied using COQ's `omega` tactic.

Higher order functions present more difficulty, as we can not tell anything specific about size from the HOF itself. The examples we have given present a variety of such functions and show how we can overcome this difficulty. The approach we take for higher order functions is to associate the following with each function argument:

- A **size function** which computes the size of the result given the function's input. We can create this for any function by following the syntax tree of the source language function.
- A **size predicate** which specifies the property that the result size (i.e. the result of the size function) must satisfy. We assume that these are provided along with the

source function, having been either specified by the user, or given by an external inference system.
– A number of **predicate transformers** which specify the properties the predicate must satisfy; we can generate these mechanically by observing the properties which need to be proved to complete the definition.

4.7 Towards an Automated Transformation

Having shown how to represent these functions in the framework, we need to consider how to automate the translation from the source language into TT; it is important to note that although the definitions of higher order functions look very complex in the TT framework, the programmer will never need to see these definitions.

We believe that the homogeneous framework we have chosen for TT programs will make automated construction straightforward. In most cases, all that is required is the traversal of the structure of the source program, managing Size structures and identifying proof obligations where necessary. In many cases even proof construction is automatable with the `omega` tactic (or even by simpler proof search methods). In more difficult cases, we envisage a theorem proving interface allowing the user to give hints.

To facilitate the construction of TT terms, we are building a theorem proving library[3] and equipping it with appropriate tactics. In particular, we have implemented tactics for management of the Size structure and identifying the required predicate transformers for higher order functions. Our system keeps track of remaining proof obligations, allowing these to be solved by the user or (in future) by an automated proof search tool such as the Omega calculator. The biggest difficulty we envisage is the presentation of useful diagnostics to the user when the automated tools are unable to solve a proof obligation, whether because the tools are not powerful enough, or because the theorem is unprovable.

5 Related Work

Other than our own work [24], we are aware of three main studies of formally bounded time and space behaviour in a functional setting [5, 14, 26]. All such approaches are based on restricted language constructs to ensure that bounds can be placed on time/space usage, and require considerable programmer expertise to exploit effectively. In their proposal for Embedded ML, Hughes and Pareto [14] have combined the earlier *sized type system* [15] with the notion of *region types* [25] to give bounded space and termination for a first-order strict functional language [14]. This language is however restricted in a number of ways: most notably in not supporting higher-order functions, and in requiring the programmer to specify detailed memory usage through type specifications. The practicality of such a system is correspondingly reduced.

There is active research into programming with dependent types — [1] describes the rationale and gives an example of programming in EPIGRAM; [21] gives an example of

[3] Available from `http://www.dcs.st-and.ac.uk/~eb/TT/`

generic programming with dependent types. Augustsson and Carlsson have used dependent types to verify type correctness properties of an interpreter [3]. Xi and Pfenning have also exploited size properties of dependent types in DML for optimising array lookup [29], using dependent types to guarantee the bounds of an array. However, the form of dependent types permitted by DML is limited to a specific constraint domain (e.g. integers, for representing size, with their usual operations) so it is not possible to compute sizes in the type, as in our framework.

Crary and Weirich [7] have developed a dependent type system that provides an explicit upper bound on the number of steps needed to complete a computation. Space is conservatively bounded by the same bound as time. The language does support higher-order functions, although unlike our system their language of cost functions is limited to using a fixed set of operators. Grobauer's work [10] also applies dependent types, extracting time bounds for DML programs, although this is limited to first-order functions.

Hofmann and Jost have shown in [12] how to obtain bounds on heap space consumption of first-order functional programs based on linear types. They are extending these methods to deal with polymorphic and higher order functions, as described in Jost's forthcoming PhD thesis. Vasconcelos also describes methods which extend the basic sized type inference in his forthcoming PhD thesis, using a method based on abstract interpretation. Unlike our framework, these methods have the limitation that they do not allow bounds to depend on input data (as for example we have done with `either`, and may like to do in any case where we know an argument statically). However, these techniques complement our own work — our framework is intended to check externally specified size bounds. Furthermore our framework builds on these systems in that the bounds can be *programmer specified* as well as inferred, allowing resource bounded programs we would not otherwise be able to write (e.g. `either`, or higher order functions which do not admit a linear bound).

6 Conclusions

We have presented a flexible framework for describing and verifying size metrics for functional programs. By using a dependently typed core language, TT, we are able to make explicit the properties which a program must satisfy *in the type* and hence showing that a program satisfies those properties is simply a matter of typechecking. We have implemented these examples in the Coq theorem prover; by using Coq's `omega` tactic to construct proofs of the equational constraints required by the Size type automatically, we can see that in principle it should be possible to mechanise the construction of TT code from the source programs.

Within this framework, we have the flexibility to use any appropriate size metric, and to extend the language of constraints in order to be able to express more complex bounds, such as those which arise in higher order functions. The size bounds of a higher order function will often depend on its function arguments, and so having a rich language at the type level allows us to express size in terms of this. Although there are some difficulties here, as demonstrated in particular by the **fold** example, these can be overcome

within the framework; the definitions of **fold** and **either** show how we can use type level computation to give a precise cost for higher order functions.

This paper documents the first stage of the design and implementation of a resource-safe intermediate language for multi-stage Hume programs — we have implemented several examples by hand within the framework. By doing so, and in particular by investigating more complex higher order functions such as **fold** and their applications, we hope to be able to derive a method for mechanically constructing TT terms from source language programs. If we wish to use TT as a core language for Hume programs, such a translation method is essential. Nonetheless, we believe that through implementing higher order functions by hand within the framework, we have identified the key features which will need to be considered by an automatic translation — specifically, the separation of a size function and predicate, and the generation of a size function for each source language function.

Acknowledgements

This work is generously supported by EPSRC grants GR/R70545 and EP/C001346/1 and by EU Framework VI IST-510255 (EmBounded).

References

1. T. Altenkirch, C. McBride, and J. McKinna. Why dependent types matter, 2005. Submitted for publication.
2. L. Augustsson. Cayenne — a language with dependent types. In *Proc. ACM SIGPLAN International Conf. on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 239–250. ACM, June 1999.
3. L. Augustsson and M. Carlsson. An exercise in dependent types: A well-typed interpreter, 1999.
4. E. Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, University of Durham, 2005.
5. R. Burstall. Inductively Defined Functions in Functional Programming Languages. Technical Report ECS-LFCS-87-25, Dept. of Comp. Sci., Univ. of Edinburgh, April 1987.
6. Coq Development Team. The Coq proof assistant — reference manual. `http://coq.inria.fr/`, 2004.
7. K. Crary and S. Weirich. Resource bound certification. In *the Symposium on Principles of Programming Languages (POPL '00)*, pages 184–198, N.Y., Jan. 19–21 2000. ACM Press.
8. H. B. Curry and R. Feys. *Combinatory Logic, volume 1*. North Holland, 1958.
9. P. Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4):440–465, 1994.
10. B. Grobauer. *Topics in Semantics-based Program Manipluation*. PhD thesis, BRICS, Department of Computer Science, University of Aarhus, August 2001.
11. K. Hammond and G. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *Proc. Conf. Generative Programming and Component Engineering (GPCE '03)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.
12. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proc. POPL 2003 — 2003 ACM Symp. on Principles of Programming Languages*, pages 185–197. ACM, 2003.

13. W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H.B.Curry: Essays on combinatory logic, lambda calculus and formalism*. Academic Press, 1980. A reprint of an unpublished manuscript from 1969.
14. R. Hughes and L. Pareto. Recursion and Dynamic Data Structures in Bounded Space: Towards Embedded ML Programming. In *Proc. 1999 ACM Intl. Conf. on Functional Programming (ICFP '99)*, pages 70–81, 1999.
15. R. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *Proc. ACM Symp. on Principles of Prog. Langs. (POPL '96)*. ACM Press, 1996.
16. Z. Luo. *Computation and Reasoning – A Type Theory for Computer Science*. International Series of Monographs on Computer Science. OUP, 1994.
17. Z. Luo and R. Pollack. LEGO proof development system: User's manual. Technical report, Department of Computer Science, University of Edinburgh, 1992.
18. C. McBride. *Dependently Typed Functional Programs and their proofs*. PhD thesis, University of Edinburgh, May 2000.
19. C. McBride. Epigram: Practical programming with dependent types. Lecture Notes, International Summer School on Advanced Functional Programming, 2004.
20. C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.
21. P. Morris, C. McBride, and T. Altenkirch. Exploring the regular tree types. In *Types for Proofs and Programs 2004*, 2005.
22. A. Mycroft and R. Sharp. A statically allocated parallel functional language. In *Automata, Languages and Programming*, pages 37–48, 2000.
23. W. Pugh. The Omega Test: a fast and practical integer programming algorithm for dependence analysis. *Communication of the ACM*, pages 102–114, 1992.
24. A. Rebón Portillo, K. Hammond, H.-W. Loidl, and P. Vasconcelos. A Sized Time System for a Parallel Functional Language (Revised). In *Proc. Implementation of Functional Langs.(IFL '02), Madrid, Spain*, number 2670 in Lecture Notes in Computer Science. Springer-Verlag, 2003.
25. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1 Feb. 1997.
26. D. Turner. Elementary Strong Functional Programming. In *Proc. 1995 Symp. on Functl. Prog. Langs. in Education — FPLE '95*, LNCS. Springer-Verlag, Dec. 1995.
27. Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In *Proc. Intl. Conf. on Functional Programming (ICFP '01)*, Florence, Italy, September 2001. ACM.
28. Z. Wan, W. Taha, and P. Hudak. Event-driven FRP. In *Proc. 4th. Intl. Sym. on Practical Aspects of Declarative Languages (PADL '02)*. ACM, Jan 2002.
29. H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, June 1998.