A Verified Resource-Bounded Functional Programming Language

Edwin Brady  Kevin Hammond  James McKinna
School of Computer Science, University of St Andrews, St Andrews, Scotland, KY16 9SX.
Email: eb,kh,james@dcs.st-and.ac.uk

Abstract

This paper studies the problem of constructing formal bounds on program resource usage and other complex properties using full-spectrum dependent types to encode resource usage properties over size information and the associated correctness proofs as program terms in a simple resource-aware functional language, $\text{RAFL}$. Since resource properties and the associated proofs are directly expressed in $\text{RAFL}$ through strong program structures associated with a formal program logic, it follows that correctly specified resource properties of programs written in our language can be formally and automatically verified simply by composing proofs according to the underlying program structure. We illustrate this by constructing a dependently typed interpreter for $\text{RAFL}$ that ensures that the representation of $\text{RAFL}$ terms includes explicit and independently checkable proofs that the required resource properties are satisfied. In this way we are able to construct programs with strong upper bounds on resource usage that can be formally and automatically verified.

Compared with other automatic approaches to bounding resource usage, our work has the twin advantages of flexibility and generality, whilst retaining simplicity and automation. This is achieved through the use of full-spectrum dependent types rather than the more simply typed approaches in previous use. We illustrate the advantages of the approach by considering some complex operations on lists and trees.

1. Introduction and Motivation

We consider the problem of developing programming language notations that both allow formal resource bounds to be specified directly, and that ensure that such resource bounds may be automatically verified. Many applications must be deployed in resource-limited settings. This is most obviously true in the embedded systems domain, where systems must often meet strong constraints on space usage, real-time responsiveness and even power consumption, but database systems, Grid computing, control systems, and even computer games may make similar demands. While average-case estimates may be acceptable in non-realtime situations, under-estimates of space usage, in particular, may result in system crashes or other erroneous behaviours; under-estimates of response times may mean that systems fail to meet hard-realtime requirements; and under-estimates of power consumption may lead to mission failure through system unavailability. There is thus a strong motivation to provide verifiable bounds on resource usage.

1.1 Contributions

The main contributions of this paper are:

1. we define a simple resource-aware pure functional programming language, $\text{RAFL}$, that gives strong static guarantees on resource usage;
2. we describe a verified implementation of a complex type system, i.e. one that goes beyond conventional Hindley-Milner types using a modern dependent type framework; and
3. we provide a practical example of programming with full-spectrum dependent types.

Our use of a full-spectrum dependently-typed implementation language is significant, since it means that we can directly express the properties of our type system through embedded types and still obtain an efficient implementation of our language and type checker.

This paper goes beyond our previous work [11, 14, 19] in considering how resource information may be encoded in the form of dependent types that represent both information about the sizes of program structures and automatically verifiable proofs of the soundness of required properties of those structures. Our use of sizes in types is analogous to that of the Hughes and Pareto sized type system [14, 19]: however, our system permits more flexible definition both of resource properties and of the associated correctness proofs. Our work goes beyond that of Crary and Weirich [6], who have also applied dependent types in a resource-bounded setting in ... It goes beyond that of Igarashi and Kobayashi [15] in ...

By combining a sized type system with a dependently typed implementation language we are able to: i) state certain desired resource properties of a function; ii) construct proofs of these desired properties in the language implementation; and iii) overcome the limitations of automated sized type checking, by allowing a programmer to construct proofs of properties which are too complex to express automatically.

The remainder of this paper is structured as follows: Section 2 introduces our resource-aware language, and provides typing rules that expose size information; Section 6 gives a number of examples, including higher-order functions and the use of alternative size metrics; Section 3 introduces dependent types; Section 4 describes the implementation of an interpreter for our resource-aware language,
including the use of de Bruijn indices to avoid the technical problems with renaming that have been encountered by previous approaches; Section 5 extends our language with list and tree types, which are used in the examples in section 6; Section 7 describes related work; and finally, Section 8 concludes.

2. \( \mathcal{RAFL} \): a Resource Aware Functional Language

Our objective is to define a resource-aware functional programming language, \( \mathcal{RAFL} \), that allows the expression of high level programs with guaranteed resource bounds, where all required information about resource usage is captured in the type system. The core of our language is a typed \( \lambda \)-calculus, extended with size annotations that may depend on size variables and which are used to represent resource usage in terms of a defined notion of size for the data structures used by the program. These annotations may express the input and output sizes of a function or program or indicate the size of some expression form. Types may also contain embedded propositions over these sizes, which allow the programmer to express precisely the desired relationships between input and output sizes, for example.

In addition to ensuring that \( \mathcal{RAFL} \) correctly captures resource usage information, we also require a verified implementation of this language, where a representation of an object language program provides a statically checkable and independently verifiable proof that the program conforms to the required size bounds.

2.1 Motivating Examples

We will use three simple examples to motivate our approach.

1. Simple list functions. At a minimum, we would like to write functions over high level data structures such as lists, encode size information in their types and specify the properties of those sizes. Previous work, such as [14, 25], allows this to some extent, although with limits on the properties which may be claimed. For example, we might want to write a list append function and guarantee that the resulting list’s length is the sum of the input lengths.

2. Higher order functions. Many functional programs make extensive use of higher order functions, particularly to introduce new abstractions and reusable code. We would like to be able to use higher order functions, to some extent, and retain some size information. In particular, we would like to express that map preserves list length and filter produces a list no longer than the input.

3. Structures with more than one size bound. Some previous approaches to size aware type systems are limited in that structures may only have one size metric. This is not always appropriate — for example, binary trees have a number of elements as one metric, and depth as another. We might want to write \( \text{flatten} \), preserving the length metric in the resulting list, or even an insertion function which maintains balance (by specifying a maximum depth).

We will develop \( \mathcal{RAFL} \) to allow all of the above functions to be expressed, and show implementations of these in section 6. In addition, our implementation in section 4 will express the programs in such a way as to give strong guarantees of size properties.

2.2 The \( \mathcal{RAFL} \) Type Language

Sizes in \( \mathcal{RAFL} \)

We begin by defining \( S \), a language of size expressions. Our sizes are represented by simple integers, and we define both addition and multiplication on sizes, and an operator for taking the maximum of two sizes.

\[
S ::= i \quad \text{Integer literal} \\
\alpha \quad \text{Variable} \\
S + S \quad \text{Addition} \\
S 	imes S \quad \text{Multiplication} \\
\text{MAX}(S, S) \quad \text{Maximum of two sizes}
\]

Propositions in \( \mathcal{RAFL} \)

We also define \( P \), a language of propositions, which specify the properties that size expressions are expected to satisfy. We define a number of comparison relations (equality, less than, etc.) plus the conjunction and disjunction of propositions.

\[
P ::= S = S \mid S \leq S \mid \ldots \mid P \land P \mid P \lor P
\]

We take \( p(\alpha) \in P \) to mean that \( p \) is a property depending on a size variable \( \alpha \). Then \( p(s) \), where \( s \in S \) instantiates that property with a specific size expression and \( \Gamma \vdash p(s) \), means that property \( p \) holds with respect to a specific context \( \Gamma \).

Types in \( \mathcal{RAFL} \)

\( \mathcal{RAFL} \) is essentially a simply-typed \( \lambda \)-calculus, with the addition of size abstraction and properties of size expressions. We initially introduce one sized data type, the integers, where the size simply represents the magnitude of the integer. Other types carry no size information. While the memory used by an integer, or the time taken to evaluate it, will usually not depend on its magnitude, integers are often used as induction parameters or in other ways that do impact resource usage. Later, in Section 5 we will introduce further sized types, namely lists and trees, where resource usage can be more immediately related to the size information we provide.

\[
T ::= \text{Int} \quad \text{Integer, with size} \\
T \to T \quad \text{Function space} \\
\forall \alpha. T \quad \text{Universal size variable binding} \\
\exists \alpha. (T, P) \quad \text{Proposition}
\]

The type of propositions is a combination of a value with its size, plus a proof that the size respects some property. This follows on from the Size construct in our previous work [1].

\[
t ::= x \quad \text{Variable} \\
\lambda t: T. t \quad \text{Lambda abstraction} \\
t t \quad \text{Application} \\
\xi \alpha. t \quad \text{Size abstraction} \\
t @ S \quad \text{Size application} \\
(S, t) \quad \text{Sized term requiring a proof} \\
\text{let}(\alpha, x) = t \in t \quad \text{Sized value binding} \\
\hat{x} \quad \text{Integer} \\
\text{intrec} t t t \quad \text{Primitive recursion}
\]

Figure 1. The \( \mathcal{RAFL} \) Term Language
2.3 The \( \mathcal{RAFL} \) Term Language

The syntax of \( \mathcal{RAFL} \) is given in figure 1. This is a typed \( \lambda \) calculus, with an explicit construct for primitive recursion over integers and additional constructs for managing size expressions and proofs of propositions. The \( \zeta \alpha. \ t \) construct binds a size variable \( x \) for use in size expressions in the scope of the binding \( t \). Correspondingly, there is a \( t \otimes S \) construct, which is the application of a size to such a binding. The \( (S, t)_p \) construct marks a proof obligation in the program — \( t \) is a term whose type may depend on the given size, \( S \). Typechecking such a term requires an explicit proof to be constructed, either by a proof search algorithm (e.g. Pugh’s Omega calculator [22]) or by a programmer supplied proof — a significant advantage of choosing a dependently typed implementation language for \( \mathcal{RAFL} \) is that it is possible for the user to provide an explicit proof when an external proof search algorithm is unable to find one for some proposition.

Correspondingly, we allow such propositions to be let-bound. Since the value’s type depends on a new size variable, the let-binding allows simultaneous introduction of the size variable and the value.

Primitive recursion (and therefore termination) is guaranteed by using an explicit recursion operator, \texttt{intrec}. This takes three arguments, and can be considered to be a fold operation over integers: the first argument is the integer to be examined (the scrutinee); the second describes the base case, i.e., the result when the scrutinee is zero; and the third argument describes the recursive case.

2.4 Typing rules for \( \mathcal{RAFL} \)

Terms in \( \mathcal{RAFL} \) are typed with respect to a context \( \Gamma \), which contains information about the variables in scope. A \( \mathcal{RAFL} \) context can be extended with a variable binding (with a valid type) or a size binding, and is defined inductively as follows:

\[
\begin{align*}
\Gamma \vdash \text{valid} & \quad \text{Empty context} \\
\Gamma, \alpha \vdash T & \quad \text{Extend with variable binding} \\
\Gamma, \alpha \vdash \text{valid} & \quad \text{Extend with size binding}
\end{align*}
\]

Figure 2 gives the typing rules. These rules make use of a conversion relation between types, \( \Gamma \vdash A \simeq B \). Two types are convertible, with respect to a context, if they are syntactically equal up to propositional equality of corresponding size expressions, i.e.:

\[
\begin{align*}
\Gamma \vdash \text{Int}_n \simeq \text{Int}_{n'}, & \quad \text{if } \Gamma \vdash x \equiv y \\
\Gamma \vdash A \to B \simeq C \to D, & \quad \text{if } \Gamma \vdash A \simeq C \text{ and } \Gamma \vdash B \simeq D \\
\Gamma \vdash \forall \alpha. \ A \simeq \forall \beta. \ B, & \quad \text{if } \Gamma; \alpha \vdash A \simeq B[\alpha/\beta], \alpha \not\in \text{FV}(B) \\
\Gamma \vdash \exists \alpha. \ (A, P) \simeq \exists \beta. \ (B, Q), & \quad \text{if } \Gamma \vdash A \simeq B, \ P \equiv Q
\end{align*}
\]

For example, if \( \Gamma \vdash x : \text{Int}_{n+2} \), then \( \Gamma \vdash x : \text{Int}_{n+z} \), because addition is commutative and so the size indices are propositionally equal, therefore the types are convertible.

The type system of \( \mathcal{RAFL} \) incorporates a restricted form of dependent types, with values depending on sizes and including propositions. The rules for size application, propositions, let binding and primitive recursion involve a certain amount of evaluation at the type level, to substitute sizes into types.

The rule for primitive recursion over natural numbers relies on a \textbf{motive}, \( \Phi \). This is a meta-level function which computes a return type based on the size of the input type — this allows the result of a recursive call to depend on the size of the input.
2.6 Example — plus

A simple example in the language is the addition function; the size of the output is the sum of the sizes of the inputs:

\[
\begin{align*}
\text{plus} & : \forall a, b. \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \\
\text{plus} & = \lambda \alpha. \beta. \lambda x. \lambda y. \text{Nat} \\
\quad & \text{intrec } x 0 (\text{\texttt{\#(gamma, lambda y. \text{Nat})}}) \\
& \text{intrec } y 0 (\text{\texttt{\#(gamma, lambda y. \text{Nat})}}) \\
& \text{. (1 + ih)}
\end{align*}
\]

In the recursive call, we take the motive to be \( \Phi(\delta) = \text{Nat} + \beta \), since the resulting size of the recursive call is the size of the first input \( \alpha \) added to the size of the second \( \beta \), and recursion is over the first input.

3. Dependently Typed Programming

We will use **EPiGRAM** [18, 16] as our implementation language; **EPiGRAM** is a platform for full spectrum dependently typed functional programming, based on a strongly normalising core type theory with **inductive families** [8], together with a sophisticated type-directed elaborator from source programs to the type theory.

3.1 Programming with 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 will define a “lists with length” (or vector) type below. We first, however, need to declare a type of natural numbers to represent such lengths:

\[
\begin{align*}
data & : \text{Nat} : \text{*} & \text{where } & \text{N} : \text{Nat} & \text{.} & n : \text{Nat} \\
& \text{when } & 0 : \text{Nat} & \text{.} & n : \text{Nat} \\
& \text{where } & \text{N} : \text{Nat} & \text{.} & n : \text{Nat}
\end{align*}
\]

Addition and multiplication can be easily defined by primitive recursion. We can now declare vectors as follows: \( \text{Vect} A n \) defines an inductive family of lists indexed over \( A \), the type of vector elements, and also over \( n \), the vector length. Note that, by construction, \( e \) only targets vectors of length zero, and \( x::xs \) only targets vectors of length greater than zero:

\[
\begin{align*}
data & : \text{Nat} : \text{*} & \text{where } & \text{N} : \text{Nat} & \text{.} & n : \text{Nat} \\
& \text{when } & 0 : \text{Nat} & \text{.} & n : \text{Nat} \\
& \text{where } & \text{N} : \text{Nat} & \text{.} & n : \text{Nat}
\end{align*}
\]

Note that \( A \) and \( k \) are implicit arguments to the infix constructor \( :: \) — their types can be inferred from the type of \( \text{Vect} \). When the type includes explicit length information in this way, it follows that any type-correct function over values in that type must express the invariant properties of the length. For example, we can write a bounds-safe list lookup function that gives a static guarantee that a value is never projected from an empty list. In order to do do this, we define a datatype of finite sets, which can be used to represent numbers with an upper bound:

\[
\begin{align*}
data & : \text{Nat} : \text{*} & \text{where } & \text{N} : \text{Nat} & \text{.} & n : \text{Nat} \\
& \text{where } & 0 : \text{Nat} & \text{.} & n : \text{Nat} \\
& \text{where } & \text{N} : \text{Nat} & \text{.} & n : \text{Nat}
\end{align*}
\]

Note that there are no elements of \( \text{Fin} 0 \) — this would be a finite set with zero elements. The type of \( \text{lookup} \) below expresses statically that the bound of the index and the size of the list are the same, so there can be no run-time error:

\[
\begin{align*}
\text{let } & i : \text{Fin } n & x : \text{Vect } A & n \\
\text{lookup } & i x s : A \\
\text{lookup } & f 0 (x::ys) : x \\
\text{lookup } & f (s j) (x::ys) : \text{lookup } j ys
\end{align*}
\]

The \texttt{elim} and \texttt{case} notation invoke the primitive recursion and case analysis operators respectively on \( i \) and \( xs \). Termination is guaranteed since these operators are the only means to inspect data. Unlike a simply typed language, we do not need to give error handling cases: the typechecker verifies that the empty vector cannot be a legal input. We can see this by observing that neither constructor of \( \text{Fin} \) targets the type \( \text{Fin } 0 \), therefore no well-typed application of \texttt{lookup} could accept a \texttt{Vect } \( A \).

By giving additional static information about the \texttt{lookup} function, we obtain a stronger guarantee of its behaviour from the typechecker. The definition itself, however, is written in the usual way — indeed, it is more concise since there is no need for error checking. When writing the type, we are really writing a specification. Then, in writing the program, we are at the same time providing a proof that the implementation meets this specification.

4. A Resource Bounded Interpreter

The interpreter we present here uses inductive families to represent the required correctness properties, namely well-typedness, including guarantees that sizes satisfy the specified properties, and synchronisation of type and value environments. By using inductive families, we can express explicit relationships between data structures, in a similar way to the relationship between a \texttt{Vect} and its length described in Section ??.

We have chosen **EPiGRAM** as an implementation language for \( \mathcal{RAFL} \), which we defined in Section 2, because of the strong static guarantees it gives. By representing the language as an inductive family, and taking care to choose appropriate invariants, we can guarantee that any program we represent conforms to the \( \mathcal{RAFL} \) typing rules. **EPiGRAM**, as we saw in section 3 allows us to represent proofs directly within programs — therefore if we choose the right representation for \( \mathcal{RAFL} \), we can include proofs of the necessary properties explicitly.

The representation we choose for the language makes extensive use of inductive families; in particular, the types we use express the relationship between values and their type and context membership. This means that, without having to prove any theorems externally, we have static guarantees that evaluation preserves type, and that context lookup will always succeed. In the presence of sized types for representing resource bounds, this provides even stronger guarantees: the representation of \( \mathcal{RAFL} \) terms includes explicit and independently checkable (e.g. by another proof assistant such as CoQ [4]) proofs that the size properties are satisfied.

The interpreter translates a representation of programs in the object language (\( \mathcal{RAFL} \)) into a program in the meta language (\( \mathcal{T} \)). We require this translation process to remove size information from the program — as size information is used only statically (i.e. in typechecking) and not dynamically (i.e. at run-time) we do not want this information to be present in the program which eventually is executed.

At a high level, we require representations of types, terms, sizes and propositions and the following interpretation functions:
Interpretation of types This gives the meta language type of an object language program. Interpretation of types removes sizes and propositions, leaving only simple types.

Interpretation of values This gives a meta language program which conforms to the semantics of the object language program. This removes sizes and proofs of propositions, leaving only a simply typed value.

Interpretation of sizes Although we do not require sizes at runtime, we need an interpretation function for types in order to build proofs of size properties, and to implement conversion between types.

Interpretation of propositions We require interpretation of propositions to build a meta language representation of the proposition, which requires an explicit proof in the meta language.

4.1 Representation of Types

Types may depend on size variables. We therefore choose a representation for types which allows us to express this dependency. To avoid technical difficulties with renaming, we choose to represent variables by de Bruijn indices [7]. Since this means that variables are simply represented by numbers, we index our representation of types by the number of bound size variables. We know exactly how many size variables are quantified over the type, statically, by looking at the index.

Figure 5 shows our representation of size expressions. Variables themselves are represented as elements of a finite set, Fin a, where a is the number of size variables bound. This ensures that all size expressions are well-scoped; it would be a compile-time error to attempt to refer to a variable which is out of scope. Integer literals are represented as natural numbers, N.

The representation of types is shown in figure 7. The integer type, TyInt, is coupled with a size expression, representing the object language type Int. Binding a size variable introduces a new size variable into the scope, and so the index on the argument of SizeBind is incremented. Likewise, building a proposition binds a size variable.

Figure 7. Types

4.2 Interpreting Sizes and Propositions

Before we come to the representation of RAFL itself, we need to define some operations on types, propositions and sizes. In particular, we need to translate size expressions into meta language values, and propositions into meta language types.

To translate a size expression into a value, we require an environment containing the values of each size variable in the expression. We use a vector for this purpose, and initialise it to ε:

\[
\begin{align*}
\text{let} & \quad \text{sizes} : \text{Vect} N a \\
& \quad x : \text{SizeExp} a \\
\text{sizeInterp} & \quad \text{sizes} x : N
\end{align*}
\]

sizeInterp is defined by primitive recursion over the size expression. Size variable lookup is performed by the lookup function defined in section 3 — variable lookup can never fail; its type, and the representation of size variables as elements of a finite set, ensure that we never try to project a value out of the environment which is out of scope.

The representation of types is shown in figure 7. The integer type, TyInt, is coupled with a size expression, representing the object language type Int. Binding a size variable introduces a new size variable into the scope, and so the index on the argument of SizeBind is incremented. Likewise, building a proposition binds a size variable.

\[
\begin{align*}
\text{data} & \quad a : N \\
& \quad \text{Ty} a : \star \\
\text{where} & \quad s : \text{SizeExp} a \\
& \quad \text{TyInt} x : \text{Ty} a \\
& \quad S T : \text{Ty} a \\
& \quad S \Rightarrow T : \text{Ty} a \\
& \quad T : \text{Ty} (\text{succ} a) \\
& \quad \text{SizeBind} T : \text{Ty} a \\
& \quad T : \text{Ty} (\text{succ} a) \\
& \quad P : \text{Prop}(\text{succ} a) \\
\text{TyProp} & \quad T P : \text{Ty} a
\end{align*}
\]

Figure 7. Types

\[
\begin{align*}
\text{let} & \quad \text{sizes} : \text{Vect} N a \\
& \quad x : \text{SizeExp} a \\
\text{sizeInterp} & \quad \text{sizes} x : N
\end{align*}
\]

sizeInterp is defined by primitive recursion over the size expression. Size variable lookup is performed by the lookup function defined in section 3 — variable lookup can never fail; its type, and the representation of size variables as elements of a finite set, ensure that we never try to project a value out of the environment which is out of scope.

\[
\begin{align*}
\text{sizeInterp} & \quad \text{sizes} x \quad \Leftarrow \text{elim} x \\
\text{sizeInterp} & \quad \text{sizes} (\text{SNum} n) \quad \Rightarrow n \\
\text{sizeInterp} & \quad \text{sizes} (\text{SVar} i) \quad \Rightarrow \text{lookup} i \text{ sizes} \\
\text{sizeInterp} & \quad \text{sizes} (\text{Add} x y) \\
& \quad \Rightarrow \text{sizeInterp} \text{ sizes} x + \text{sizeInterp} \text{ sizes} y \\
\text{sizeInterp} & \quad \text{sizes} (\text{Mult} x y) \\
& \quad \Rightarrow \text{sizeInterp} \text{ sizes} x \times \text{sizeInterp} \text{ sizes} y \\
\text{sizeInterp} & \quad \text{sizes} (\text{Max} x y) \\
& \quad \Rightarrow \text{max} (\text{sizeInterp} \text{ sizes} x) (\text{sizeInterp} \text{ sizes} y)
\end{align*}
\]

We can build a meta-language proposition (i.e. a type which is inhabited by a proof object) using the propInterp function, declared as follows and defined by recursion over x:

\[
\begin{align*}
\text{let} & \quad \text{sizes} : \text{Vect} N a \\
& \quad x : \text{TyProp} a \\
\text{propInterp} & \quad \text{sizes} x : \star
\end{align*}
\]
### 4.3 Interpreting Types

Interpreting a type is by primitive recursion over its representation. Since we want the translated code to evaluate only the computational parts of RAFL programs, ignoring the size, type interpretation removes all size and proposition information from the type, leaving only a simple type:

\[
\begin{align*}
\text{let } & \quad x : \text{Ty } a \\
\text{tyInterp } x & \quad \Rightarrow \text{elim } x \\
\text{tyInterp } (\text{TyInt } a) & \quad \Rightarrow N \\
\text{tyInterp } (S \Rightarrow T) & \quad \Rightarrow \text{tyInterp } S \Rightarrow \text{tyInterp } T \\
\text{tyInterp } (\text{SizeBind } T) & \quad \Rightarrow \text{tyInterp } T \\
\text{tyInterp } (\text{TyProp } T P) & \quad \Rightarrow \text{tyInterp } T
\end{align*}
\]

Erasure of the size information is particularly important in a resource aware setting, since we do not want any extraneous objects at runtime affecting the size information.

### 4.4 Substitution

A major difficulty in the implementation of a functional language is in the treatment of variable naming, particularly regarding the generation of fresh names and the requirement to check for name clashes in substitution. Our approach follows that of [17] — we use de Bruijn indices to avoid problems with renaming, and use a single well-defined interface to manipulate indices.

It can be challenging to implement the manipulation of de Bruijn indices correctly. Using a single interface for this manipulation undoubtedly helps, in that it minimises the points where failure can occur — only the basic operations need perform the arithmetic on the indices. Additionally, using Fin as a representation helps us further — we can use the type system to help ensure a correct implementation of the basic operations, as we can never construct an out of scope variable.

The basic operation we need in the implementation of RAFL is the substitution of a size expression into a type — in [17] this operation is called “instantiate”. The type of this operation expresses that substitution must be into a type with a non-zero number of variables, and that it reduces the number of variables by one.

\[
\begin{align*}
\text{let } & \quad \text{size } : \text{SizeExp } a \\
\text{tySubst } \text{size } : \text{ty } a
\end{align*}
\]

Substitution replaces the instances of the index zero with the size expression; under \( n \) binders it replaces the index \( n \), decrements indices greater than \( n \) and leaves indices below \( n \) untouched. The implementation needs to manage the indices in such a way that the size expressions remain wellscoped. Unlike [17], our use of a dependently typed language means that we make no assumptions about the number of bound variables in an expression — the type tells us exactly how many there are.

### 4.5 Contexts

Recall that the context, \( \Gamma \), contains information about the value and size variables which are in scope. The representation of the context is given in figure 8. Like our inductive definition of the context in section 2.4, this definition includes rules for the empty context, extending the context with a variable binding, and extending with a size binding. The type is indexed over both the number of value variables and size variables — the de Bruijn indices for each are counted independently. Each type in the context depends on the number of previously bound size variables.

\[
\begin{align*}
data & \quad n, a : N \\
\text{where} & \\
\text{empty} : & \quad \text{tyEnv } n a : \star \\
\text{sizeExtend } G : & \quad \text{tyEnv } n a \\
\text{typeExtend } t G : & \quad \text{tyEnv } n a
\end{align*}
\]

#### Figure 8. Context representation

Since the index of types in the context differs according to the number of size variables at that point, projection from the context needs some care. When we look up an entry, we are interested in the type at the point of use, not the point at which it is entered in the environment. The type lookup function has the following type:

\[
\begin{align*}
\text{let } & \quad i : \text{Fin } a \\
\text{tyLookup } i G & \quad \Rightarrow \text{ty } a
\end{align*}
\]

The type of \( \text{tyLookup} \) indicates that if we have a context with \( a \) size variables, we want a type which is usable in a context with \( a \) size variables. In practice, this means that a type which was bound with fewer variables in the context needs to be abstracted over the variables which were bound later. This requires that we increase the index for each size variable by the number of size variables bound after the type.

For this we define a function \( \text{weaken} \):

\[
\begin{align*}
\text{let } & \quad t : \text{Ty } a \\
\text{weaken } t : & \quad \text{ty } a
\end{align*}
\]

Using this, the definition of \( \text{tyLookup} \) is given in figure 9. This is similar to the ordinary vector lookup from section 3.1, skipping over the size bindings — whenever there is a size binding, we need to weaken the result of the recursive call, to allow for the extra size binding. Similarly to the definition of \( \text{lookup} \), since there are no elements of \( \text{Fin } 0 \), there are no cases for the empty context.

\[
\begin{align*}
\text{let } & \quad i : \text{Fin } a \\
\text{tyLookup } i G & \quad \Rightarrow \text{ty } a
\end{align*}
\]

#### Figure 9. Projection of a Type from the Context

### 4.6 Conversion

The conversion relation, \( \Gamma \vdash A \simeq B \), allows values to be converted between two types, provided there is a proof that any sizes in the
two types are propositionally equal. This allows, for example, a
value of type Int{a,0} to be used where a value of type Int{a,1} is
expected, since we can prove the commutativity of addition.

In the representation, we will need to represent the use of con-
version explicitly, so we define a relation to represent convertibility of
types. A partial definition is given in figure 10 — the rest is defined
structurally over types.

\[
\begin{align*}
\text{data} & \quad \begin{array}{ll}
x, y : \mathbf{T} & \quad \begin{array}{lll}
\text{Conv} & x & y : \\
& & \star
\end{array}
\end{array} \\
p & : (xs : \text{Vec} N a) \rightarrow \text{sizeInterp} \, xs \, i = \text{sizeInterp} \, xs \, j
\end{align*}
\]

\[
\text{where} \quad \begin{array}{ll}
l & : \text{Conv} \, A \, C \quad \begin{array}{lll}
\text{ConvFn} & i & r : \text{Conv} \, (A \Rightarrow B) \quad (C \Rightarrow D)
\end{array}
\end{array}
\]

\[\ldots\]

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{conversion_relation}
\caption{Conversion Relation}
\end{figure}

The most important constructor is \text{ConvInt} — this shows that
\text{TyInt} \, i \, is \, convertible \, with \, \text{TyInt} \, j, \, where \, i \, and \, j \, are \, size \, expressions,
provided that there is a proof \( p \) that the interpretation of both
size expressions are equal in \emph{any} context, given by the \( xs \) argument.

\section{Language Representation}

We are now able to define the representation of RAFL, given
in figure 11. This definition encodes only terms which are well-
typed according to the typing rules given in section 2.4 — since
it is not possible to express an incorrectly typed program with this
representation, there is no need for any type error checking in an
interpreter.

In order to express the well-typedness of a term, we index the
representation over a type environment and the type of the term.
Then the type of each constructor of \text{Expr} also encodes a type.

The representation requires in particular that a function must be
applied to an argument of exactly the required type — an expres-
sion of any other type will lead to a compile-time error. Where a
function is applied to a convertible type, this requires an explicit
conversion, using the \text{convert} constructor. This constructor ensures
that the conversion is valid by taking an explicit proof that the con-
version relation holds for the specific types.

Where a value requires a size to conform to some property, we use
the \text{getSize} constructor which takes an explicit proof that the property
holds in \emph{any} size context. This constructor is the only means to
build an expression with a propositional type — all properties
required of a program must be shown with an explicit proof.

The \text{let} constructor is used to extract a specific size and value
from an expression carrying a proof. In functions such as \text{filter},
which returns a size and a proof that the size is bounded by the
original list length, this binds the actual size in the scope. The
\text{getSize} function is a simple helper function which extracts the
size expression from a proposition:

\[
\begin{align*}
\text{let} & \quad T : \mathbf{Ty} (\text{succ} \, a) \quad e : \mathbf{Expr} \, G \, (\mathbf{TyProp} \, T \, P) \\
& \quad \begin{array}{ll}
\text{getSize} & e : \mathbf{SizeExp} \, a \\
\end{array} \\
& \quad \begin{array}{ll}
\text{getSize} & (\text{pf} \, s \, e \, P \, p) \Rightarrow s
\end{array}
\end{align*}
\]

The subscript \( T \) indicates that it is an implicit argument. Only
one case is necessary for this function — the \text{EPIGRAM} elaborator
identifies that only one constructor of \text{Expr} can have the declared
type.

Note that the \text{intrec} constructor makes the motive of the recursion
explicit. Recall that the motive of the recursion is a function of the
size of the input. We represent this by using a type with an extra
size variable — instantiating this size variable with the input size
gives the type of the result of the recursion. The type of the \( xe \)
argument instantiates \( \Phi \) with zero, since the input size is zero. For
\( se \) the situation is more complex; its type encodes the RAFL type
\( \forall \beta. \text{Int}_\beta \rightarrow \Phi(\beta) \rightarrow \Phi(1 + \beta) \).

\section{The Value Environment}

To implement the interpreter, we will require a value environment,
with entries corresponding to entries in the context. We define a
family \text{ValEnv}, indexed over the context, in figure 12. Indexing
over the context ensures that each entry in the value environment
gets its type from the corresponding entry in the context.

\[
\begin{align*}
\text{data} & \quad G : \mathbf{TyEnv} \, n \, a \\
\text{ValEnv} & \quad G : \star \\
\text{where} & \quad \begin{array}{ll}
\text{vEmpty} & : \text{ValEnv} \, \text{empty} \\
\text{vSizeExtend} & \quad \text{env} : \text{ValEnv} \, (\text{sizeExtend} \, G) \\
\text{vTypeExtend} & \quad \text{env} : \text{ValEnv} \, (\text{typeExtend} \, T \, G)
\end{array}
\end{align*}
\]

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{val_env}
\caption{Value environments}
\end{figure}

Projection from the value environment corresponds to projection
from the type environment, and is defined in figure 13. The type
system ensures that the value projected is of the correct type.

\[
\begin{align*}
\text{let} & \quad G : \mathbf{TyEnv} \, n \, a \\
& \quad i : \text{Fin} \, n \\
& \quad e : \text{ValEnv} \, G \\
& \quad \text{envLookup}, \, i \, \text{env} : \text{tyInterp} \, (\text{tyLookup} \, i \, G) \\
& \quad \text{envLookup} \, i \, \text{env} \quad \Rightarrow \text{elim} \, i \\
& \quad \text{envLookup} \, f_0 \, \text{env} \quad \Rightarrow \text{elim} \, e \, \text{env} \\
& \quad \text{envLookup} \, f_0 \, (\text{vSizeExtend} \, \text{env}) \quad \Rightarrow \text{envLookup} \, f_0 \, \text{env} \\
& \quad \text{envLookup} \, (f_s \, i) \, \text{env} \quad \Rightarrow \text{envLookup} \, (f_s \, i) \, \text{env} \\
& \quad \text{envLookup} \, (f_s \, i) \, (\text{vTypeExtend} \, t \, \text{env}) \quad \Rightarrow \text{envLookup} \, (f_s \, i) \, \text{env} \\
& \quad \text{envLookup} \, (f_s \, i) \, (\text{vSizeExtend} \, \text{env}) \quad \Rightarrow \text{envLookup} \, (f_s \, i) \, \text{env} \\
\end{align*}
\]

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{projection}
\caption{Projection from a value environment}
\end{figure}

\section{The Interpreter}

We are now able to define the interpreter for RAFL, by primitive
recursion over the input expression. The \text{interp} function is given
in figure 14. It returns a semantic representation, as a \text{TT} term,
of the input. For example, interpreting a lambda abstraction (\text{km})
builds a lambda abstraction in \text{TT}.

The interpreter must produce a value corresponding to the correct
type, given by \text{tyInterp}. This mean in particular that proofs and
sizes are dropped from the interpreted values. Furthermore, since TT is a language of total functions, the fact that interp is well-typed means that interpretation will always terminate without error.

4.10 Example — plus

Returning to the example in section 2.6 — let us see how this is represented and interpreted. For plus, plus can be defined as follows, leaving for the moment the motive of the recursion (Phi) and the conversion proofs (left as holes □₁ and □₂):

\[
\text{plus} = \text{zeta} (\text{zeta} (\lambda \text{lam} (\lambda (\text{intrec Phi} (\text{convert ConvInt} \, \text{□₁} \, 0)))
\text{(zeta} (\lambda \text{lam} (\lambda (\text{suc} (\text{convert □₂ (\text{var} (\text{fs f₀})))))))))))
\]

The motive is represented in an expression as a type with an extra free size variable, representing the value passed to the motive. We want to represent \(\Phi(\delta) = \text{In}_{2+\beta}\), where \(\beta\) is the second size variable. As de Bruijn indices, \(\delta = 0\) and \(\beta = 1\). Therefore Phi is defined as:

\[
\Phi = \text{Ty}(\text{SAdd}(\text{SVar} f₀) (\text{SVar} (f₀)))
\]

The conversion proofs we require have the following types:

\[
\begin{align*}
□₁ : (x, y : \text{N}) & \rightarrow 0 + y = y \\
□₂ : (x, y, z : \text{N}) & \rightarrow (\text{suc } y) + z = \text{suc}(y + z)
\end{align*}
\]

Both are proved trivially by reflexivity. Interpreting this expression, in the empty environment, gives a meta-language implementation of addition without any size annotations.

\[
\text{interpplusEmpty} = \lambda x, y : \text{N}. \text{primrec}: x 0 (\lambda k, i : \text{N}. \text{suc } \text{ιk})
\]

5. Lists and Trees

We extend the type language with lists (with one size index, its length) and binary trees (with two size indices; the number of elements and the maximum depth):

\[
T ::= \ldots | [T]_s | \text{Tree } T_s, s
\]

The typing rules for lists and trees are given in figure 15. The motive for recursion over lists computes the result type in the same way as recursion over natural numbers — lists are structured similarly. For trees, the motive is more complicated, since there are two size indices and two recursive arguments.

6. Examples

In this section we give some examples of programs written in RAFL with lists and trees. These are programs of the kind we discuss in section 2.1; here we show that the core language of RAFL is capable of capturing the desired size information.

6.1 Appending Lists

One class of function we would like to write is over lists, capturing the sizes of the lists in the type. A straightforward example is the append function whose type shows that total size is preserved. This follows the structure of the plus function in section 2.6.

In the following definition, we take \(A\) as the element type of lists. Since RAFL as it stands does not have polymorphism, we can consider this a definition of a family of functions which can be instantiated for any concrete \(A\). In future work we may consider adding polymorphic types to RAFL.
let \( \text{env} : \text{ValEnv} \) e \( \Rightarrow \text{Expr} G \) T
\[
\text{interp} \ [\text{env} e : \text{tyInterp} T]
\]
\[
\text{interp} \ [e] \ (\text{var} i) \mapsto \text{envLookup} \ i \ [\text{env}]
\]
\[
\text{interp} \ [e] \ (\text{lam} \_ \ a 
\) \mapsto \lambda x : \text{tyInterp} A . \text{interp} \ (\text{vTypeExtend} x \ [\text{env}]) e
\]
\[
\text{interp} \ [e] \ (\text{app} f \ s) \mapsto (\text{interp} e f) (\text{interp} e s)
\]
\[
\text{interp} \ [e] \ (\text{szapp} f \ size) \mapsto \text{interp} e f
\]
\[
\text{interp} \ [e] \ (\text{srec} e) \mapsto \text{interp} (\text{vSizeExtend} e) [\text{env}]
\]
\[
\text{interp} \ [e] \ (\text{let} e \ \text{scope}) \mapsto \text{interp} (\text{vTypeExtend} \ (\text{interp} e) [\text{env}]) \ [\text{scope}]
\]
\[
\text{interp} \ [e] \ (\text{convert} \ conv e) \mapsto \text{interp} e e
\]
\[
\text{interp} \ [e] \ (\text{zero}) \mapsto 0
\]
\[
\text{interp} \ [e] \ (\text{srec} j) \mapsto \text{succ} (\text{interp} e j)
\]
\[
\text{interp} \ [e] \ (\text{intrec} \Phi x z s) \mapsto \text{primrec} (\text{interp} e x) (\text{interp} e z) (\text{interp} e s)
\]
\[
\text{let} \ n : \text{N} \ z s : A \ s : \text{N} \rightarrow A \rightarrow A \\
\text{primrec} n z s \mapsto \text{elim} n
\]
\[
\text{primrec} 0 z s \mapsto z
\]
\[
\text{primrec} (\text{srec} k) z s \mapsto s k \ (\text{primrec} k z s)
\]

**Figure 14.** The interpreter

\[
\begin{align*}
\text{nil} : \text{Expr} G (\text{TyList} T (\text{SNum} 0)) &= x : \text{Expr} G T \quad xs : \text{Expr} G (\text{TyList} T xsn) \\
\text{cons} x xs : \text{Expr} G (\text{TyList} T (\text{SAdd} (\text{SNum} 1 xsn))) \\
\Phi : T (\text{succ} a) \\
x : \text{Expr} G (\text{TyList} T x\text{sn}) &= \text{nil} _\Phi : \text{Expr} G (\text{tySubst} (\text{SNum} 0) \Phi) \\
\text{cons} _\Phi : \text{Expr} G (\text{SizeBind} (T \Rightarrow \text{TyList} T (\text{SVar} f0) \Rightarrow x : \text{Expr} G (\text{tySubst} (\text{SAdd} (\text{SNum} 1) (\text{SVar} f0))) (\text{weakenTy} \ x))) \\
\text{listrec} _\Phi x : \text{Nil} _\Phi \ \text{cons} _\Phi : \text{Expr} G (\text{tySubst} x \Phi)
\end{align*}
\]

**Figure 16.** Representation of Lists

\[
\begin{align*}
\text{leaf} : \text{Expr} G (\text{TyTree} T (\text{SNum} 0) (\text{SNum} 0)) \\
\text{l} : \text{Expr} G (\text{TyTree} T \text{nl} \ \text{dl}) &= x : \text{Expr} G T \quad r : \text{Expr} G (\text{TyTree} T \text{nr} \ \text{dr}) \\
\text{node} l x r : \text{Expr} G (\text{TyTree} (\text{SAdd} (\text{SNum} 1) \ (\text{SAdd} \text{nl} \ \text{dr})) (\text{SAdd} (\text{SNum} 1) \ (\text{SMax} \text{dl} \ \text{dr}))) \\
\Phi : T (\text{4} + \text{sn}) \\
x : \text{Expr} G (\text{TyTree} T \text{n} \ \text{d}) &= \text{leaf} _\Phi : \text{Expr} G (\text{tySubst} (\text{SNum} 0) \Phi) \\
\text{node} _\Phi : \text{Expr} G (\text{SizeBind} (\text{SizeBind} (\text{SizeBind} (\text{TyTree} T (\text{SVar} 3) (\text{SVar} 2)) \Rightarrow \\
\text{tySubst} (\text{SVar} 3) (\text{weakenTy} (\text{tySubst} (\text{SVar} 2) (\text{weakenTy} \Phi))) \Rightarrow T) \\
\text{TyTree} T (\text{SVar} 1) (\text{SVar} 0) \Rightarrow \\
\text{tySubst} (\text{SVar} 1) (\text{weakenTy} (\text{tySubst} (\text{SVar} 0) (\text{weakenTy} \Phi))) \Rightarrow \\
\text{tySubst} (\text{SAdd} (\text{SNum} 1) (\text{SAdd} (\text{SVar} 3) (\text{SVar} 1))) (\text{weakenTy} (\text{tySubst} (\text{SAdd} (\text{SNum} 1) (\text{SMax} (\text{SVar} 2) (\text{SVar} 0))) (\text{weakenTy} \Phi))) \\
\text{treerrec} _\Phi x : \text{leaf} _\Phi \ \text{node} _\Phi : \text{Expr} G (\text{tySubst} \text{n} (\text{tySubst} d \Phi))
\end{align*}
\]

**Figure 17.** Representation of Trees

\[
\begin{align*}
\text{append} : \forall \alpha, \beta. [A]_\alpha \rightarrow [A]_\beta \rightarrow [A]_{\alpha + \beta} \\
\text{append} = \zeta \alpha, \beta. \text{Appr} x : [A]_\alpha, \text{Appr} y : [A]_\beta, \lambda x y : [A]_{\alpha + \beta}. \\
\text{sizeapp} f \ : \text{Expr} G T \\
(\zeta \gamma. \text{Appr} y : A, \text{Appr} x : [A]_\gamma, \text{Appr} y : [A]_{\gamma + \beta}, \\
\text{sizeapp} f \ : \text{Expr} G T)
\end{align*}
\]

As with \text{plus}, typechecking the recursion operators relies on inferring an appropriate motive, \( \Phi \), for the recursion. Knowing that the target of the recursion is \( x_\alpha \), and that the expected type of the whole listrec expression is \( [A]_{\alpha + \beta} \), the typechecker can easily identify that the resulting size is the sum of \( \beta \) and the input size. Therefore, for typechecking listrec, we take \( \Phi(\delta) = \delta + \beta \).
6.2 Filter — Higher Order Functions

For `append`, the resulting size is exact; it is computed from the sum of the input sizes. However, it is not always possible to get an exact size, as the size of the result may depend on the specific contents of the input. One example of this is `filter`, whose result size depends on how many elements conform to the predicate. This is also an example of a higher order function, the predicate being passed as an argument.

We assume the introduction of a boolean type `Bool` to RAFL and a corresponding `if` . . . then . . . else expression — this is trivial to add to the language, and has the expected typing rules.

Since we do not know the resulting size of `filter` statically, but merely an upper bound, we return a constrained type. The resulting list has some size $\beta$, and we constrain that size to be no larger than the size of the input list. This requires some management of the proof structure throughout the definition, but otherwise it is similar to a traditional simply typed `filter`:

\[
\begin{align*}
\text{filter} : & \forall \alpha. [A]_{\alpha} \rightarrow (A \rightarrow \text{Bool}) \rightarrow \exists \beta. ([A]_{\beta}, \beta \leq \alpha) \\
\text{filter} & = \zeta \alpha. \let\alpha. \lambda x : [A]_{\alpha}. \lambda p : (A \rightarrow \text{Bool}). \text{listrec} z s \lfloor (0, \text{nil}, 0) \rfloor \lfloor (\zeta \beta. \lambda y : A. \lambda y : [A]_{\beta}. \lambda y : H : \exists \gamma. ([A]_{\gamma}, \gamma \leq \beta). \let\gamma. \text{let}(\delta, \text{val} \, \text{in}) = \text{ys} \, H \, \text{in} \lfloor (p \, y) \rfloor \, \text{then} (1 + \gamma, \text{cons} \, y \, \text{val}) \, \text{else} (\gamma, \text{val}) \, \text{else} \rfloor \, \text{else} \rfloor
\end{align*}
\]

We have marked the values which require proofs as $p1$ , $p2$ and $p\beta$. This definition can only typecheck if the sizes of these values can be shown to satisfy the required properties; these properties are:

\[
\begin{align*}
p1 & : 0 \leq 0 \\
p2 & : \gamma \leq \beta \rightarrow 1 + \gamma \leq 1 + \beta \\
p\beta & : \gamma \leq \beta \rightarrow \gamma \leq 1 + \beta
\end{align*}
\]

In the EPigram representation, these correspond to the following propositions:

\[
\begin{align*}
\Box_1 & : 0 \leq 0 \\
\Box_2 & : (x : \text{N}) \rightarrow (y : \text{N}) \rightarrow x \leq y \rightarrow (\text{su} c \, x) \leq (\text{su} c \, y) \\
\Box_2 & : (x : \text{N}) \rightarrow (y : \text{N}) \rightarrow x \leq y \rightarrow x \leq (\text{su} c \, y)
\end{align*}
\]

Each of these are straightforward to prove with the Omega decision procedure, so can be shown by the $\mathcal{R}_{AF}L$ typechecker with no user interaction.

6.3 Flattening Trees

In RAFL, binary trees have two size metrics — number of elements, and maximum depth. Each metric is useful in different contexts; maximum depth can be used to specify a worst case execution time for a search algorithm, or to specify a maximum depth to preserve balance. We use the length metric here to guarantee that the flattening of a tree into a list maintains the correct number of elements.

\[
\begin{align*}
\text{flatten} & : \forall \alpha. \beta. \lambda x : A_{\alpha, \beta}. \lambda t : A_{\alpha, \beta}. \lambda \text{treerec} \, t \\
\text{flatten} & = \zeta \alpha. \beta. \lambda x : A_{\alpha, \beta}. \lambda y : H : \exists \gamma. ([A]_{\gamma}, \gamma \leq \beta). \lambda x : A. \lambda \text{treerec} \, x . A_{\alpha, \beta}. \text{treerec} \, t \, \text{nil} \\
\text{flatten} & \, \text{then} (1 + \gamma, \text{cons} \, y \, \text{val}) \, \text{else} (\gamma, \text{val})
\end{align*}
\]

Again, the typechecker needs to infer a motive for the recursion operator. Since the expected result is $[A]_{\alpha}$ and the input is $T_{\alpha, \beta}$, it is clear that the result does not depend on $\beta$ and preserves $\alpha$. Therefore for typechecking `treerec`, we take $\psi(\delta, \theta) = [A]_{\alpha}$.

In these examples, we have concentrated on the relationship between input and output sizes. One purpose this serves is in aiding algorithmic correctness — we can specify certain size related properties of an algorithm. More importantly, however, it allows us to make accurate advance predictions of resource requirements. Applied to Hume programs, if the relationship between the input and output sizes of a box are known, the run-time system can allocate enough space in advance without any need for dynamic checks for out of memory errors during evaluation of the box.

7. Related Work

KH: need to cover the following: Hughes/Pareto; Taha et al. including Tagless Staged Interpreters; Hofmann and Jost; DML; Cavénne, Xi; Poppe & Chin, Cray & Vanderwaart, Cray & Weirich. Mandelbaum, Walker and Harper looks like a possibly nice overview. What about proof-carrying code? Cheney on binders? DTAL (Dependently Typed Assembly Language); Maybe Cyclone??

References at the end of the file...

This paper focuses on the a-priori and rigorous construction of upper bounds on resource usage from program source, with proofs derived automatically from properties expressed in an extended dependendent type system. The close connection between types and proofs through the Curry-Howard isomorphism is, of course, well known, and several authors have consequently explored the use of types to expose formal information about resource usage (e.g. [15, 14, 19, 10, 6, 5, 12, 13]). Our work goes beyond this earlier work by exploiting a richer full-spectrum dependent typing mechanism to allow the automatic construction of correct formal proofs from programmer specified properties. Our approach allows both a much richer set of properties to be expressed, and much more complex proofs to be developed. It can be seen as a restricted form of proof-carrying code [20], where our proofs are associated with properties that are associated with program fragments, and whose
form is restricted by the dependently typed language that is used to construct them.

Hughes, Pareto and Sabry [14] originally developed the idea of sized types in a simply typed framework, describing a type checking algorithm for a simple higher-order, non-strict functional language, and considering how resource usage could be described for MML, [19], a strict functional language using memory regions to control memory usage. In their system, the notion of size is fixed for each data type (lists and integers), rather than being user-specified as here. Their system exposes sets of linear constraints in terms of Presburger formulae over sizes which are then solved by an external solver. Since these constraints are automatically derived from program source and must be solved using standard constraint solving technology, the system is comparatively restricted in the range and complexity of program properties and proofs that can be expressed.

Chin and Khoo [2] subsequently used this sized type system as the basis for a type inference algorithm that is capable of computing size information from program source; and we have independently developed automatic cost analyses that are capable of deriving time and space cost information from unannotated program source expressions [23, ?].

Grobauer [10] has similarly used singleton dependent types to extract cost recurrences from a Dependent ML (DML) program. Like $\mathcal{R}_A F L$, size-annotated types are used to capture size information. However, unlike $\mathcal{R}_4 F L$, DML is first-order, and imposes similar restrictions in terms of proofs to Hughes, Pareto and Sabry.

Tuha et al’s GeHB [24] is a two-level staged notation that builds on Hofmann’s linearly-typed LFPL [12]. GeHB automatically generates first-order, bounded LFPL programs from higher-order specifications.

Finally, types have also been used to expose properties other than the resource properties we consider here. For example, Flanagan and Abadi [9] have shown how types can be used to enforce safe locking in concurrent systems; Popeea and Chin [21] have introduced a type system for verifying protocols; Chin et al. [3] have shown how relational sizes can be used to verify safety properties for object-based languages; and Xi [26] has shown how dependent types can be exposed to verify program termination properties. We anticipate that the approach described here could, in due course, be extended to these and other domains.

8. Conclusions and Further Work

KH: more needed here...

This paper has developed a model of resource usage based on data structure sizes in a language notation that exposes program properties and proof obligations through dependent types. By using a dependent type framework, we have been able to develop a language interpreter that automatically verifies the specified resource properties through the construction of formal proof requirements.

Where in previous work, we have focused on the definition of type effect systems permitting the reconstruction of simple sized types for non-recursive [11] and primitive recursive definitions [2], considering both space usage and time usage [21], in this paper we focus on more powerful dependent type systems which permit the expression of more complex relationships between resource properties, and which will automatically expose proof obligations.

While our approach is described in the context of a simple standalone, functional programming notation, we anticipate using these ideas in future, more powerful, versions of Hume in due course.

8.1 Further Work

Acknowledgements

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

References


