

# Embedding a Language with Certified Size Constraints in a Dependently Typed Metalanguage

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. We use *full-spectrum dependent types* to embed a language of terms,  $\mathcal{R}\mathcal{A}\mathcal{F}\mathcal{L}$ , with resource usage properties and associated correctness proofs.

Since these properties and associated proofs are directly expressed in  $\mathcal{R}\mathcal{A}\mathcal{F}\mathcal{L}$  through a formal logic, it follows that correctly specified resource properties of programs written in  $\mathcal{R}\mathcal{A}\mathcal{F}\mathcal{L}$  can be formally verified simply by composing proofs according to the underlying program structure. We illustrate this by constructing a dependently typed interpreter for  $\mathcal{R}\mathcal{A}\mathcal{F}\mathcal{L}$  that ensures that the representation of  $\mathcal{R}\mathcal{A}\mathcal{F}\mathcal{L}$  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 bounds on resource usage that can be automatically checked.

Compared with other approaches to bounding resource usage, our work has the twin advantages of flexibility and generality, whilst retaining simplicity and automation. We demonstrate these advantages by considering some representative 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 verified and automatically checked. 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.

[copyright notice will appear here]

## 1.1 Contributions

The main contributions of this paper are:

1. we define a simple resource-aware purely functional programming language,  $\mathcal{R}\mathcal{A}\mathcal{F}\mathcal{L}$ , that gives strong *static guarantees* on resource usage;
2. we describe an implementation of a complex type system, i.e. one that goes beyond conventional Hindley-Milner types using a modern *dependently typed* framework to guarantee by construction that terms are well-typed (and hence size-safe, with respect to user defined constraints); and
3. we provide a practical example of programming with full-spectrum dependent types in EPIGRAM.

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. By *full-spectrum*, we mean that types may depend on types, types on values and values on types. We believe that such power is necessary to express the properties directly; with a weaker dependent type system (e.g. using GADTs [31]) expressing properties of a complex type system would be impossible, or at best intractable<sup>1</sup>.

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 prove automatically;

This paper goes beyond our previous work [2, 3, 15] 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 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* [21, 20]; however, our system permits more flexible definition both of resource properties and of the associated correctness proofs. Our work goes beyond that of Cray and Weirich [9], who have also applied dependent types in a resource-bounded setting, in that we allow an extensible algebra of propositions on size expressions and a flexible meaning of size.

The remainder of this paper is structured as follows: Section 2 introduces our size-aware language, and provides typing rules that expose size information; Section 3 introduces dependent types;

<sup>1</sup> However, we would be delighted to be shown to be wrong!

Section 4 describes the implementation of an interpreter for our 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. $\mathcal{RALFL}$ : a Resource Aware Functional Language

In this section, we define a resource-aware functional programming language,  $\mathcal{RALFL}$ , that allows the expression of high level programs with guaranteed resource bounds, such that all required resource usage information is captured in the type system. The core of our language is a typed  $\lambda$ -calculus, extended with size annotations that are used to represent resource usage. 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.  $\mathcal{RALFL}$  therefore has a weak form of type dependency, with values depending on sizes.

### 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 [21, 40], 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 length of the result 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, and still retain some size information. In particular, we would like to be able to express that **map** preserves list length and **filter** produces a result list that is no longer than the input.
3. **Structures with more than one size bound.** Some previous approaches to resource-aware type systems are limited in that structures may only have one size metric [21]. This is not always appropriate — for example, binary trees have the number of elements as one metric, and maximum depth as another. We might want to write **flatten**, preserving the length metric in the resulting list, or even an insertion function which maintains balance (by specifying a maximum depth).

In Section 6, we give  $\mathcal{RALFL}$  implementations of these functions, and in Section 4 we will show how these examples can be represented and interpreted so as to give the strong guarantees of size properties that are required.

### 2.2 The $\mathcal{RALFL}$ Type Language

We begin by defining  $S$ , a language of size expressions. Our sizes are represented by natural numbers, and we define both addition and multiplication on sizes, and an operator for taking the maximum of two sizes.

$$S ::= i \mid \sigma \mid S + S \mid S \times S \mid \text{MAX}(S, S)$$

$t ::=$	$x$	Variable
$\lambda x : T. t$		Lambda abstraction
$t t$		Application
$\zeta \sigma. t$		Size abstraction
$t@S$		Size application
$(S, t)$		Sized term requiring a proof
$\text{let } (\sigma, x) = t \text{ in } t$		Sized value binding
$0$		Zero
$\text{SUC}(t)$		Successor
$\text{intrec } t t t$		Primitive recursion

Figure 1. The  $\mathcal{RALFL}$  Term Language

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

$$P ::= S = S \mid S \leq S \mid \dots \mid P \wedge P \mid P \vee P$$

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

### Types in $\mathcal{RALFL}$

$\mathcal{RALFL}$  is essentially a simply-typed  $\lambda$ -calculus, with the addition of terms to represent size abstractions, applications and properties. We initially introduce one *sized* data type, the natural numbers, where the size simply represents the magnitude of the number. Other types carry no size information. While the memory used by a number, or the time taken to evaluate it, will usually not depend on its magnitude, numbers are often used as induction parameters or in other ways that do directly impact resource usage, and the magnitude is therefore a useful indirect metric. 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.

$T ::=$	$\text{Nat}_S$	Number, with size
$T \rightarrow T$		Function space
$\forall \sigma. T$		Universal size variable binding
$\exists \sigma. (T, P)$		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 [2], and is closely related to McKinna and Burstall's notion of Deliverables [28] in that it pairs a program with a proof of a user-specified property.

### 2.3 The $\mathcal{RALFL}$ Term Language

The syntax of  $\mathcal{RALFL}$  is shown in Figure 1. The  $\zeta \sigma. t$  construct binds a size variable  $\sigma$  for use in size expressions in the scope of the binding  $t$ . We also provide the corresponding  $t@S$  construct, which is the application of a size to such a size binding. The  $(S, t)$  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 that the property be proven. Such propositions may 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, `intrec`. This takes three arguments, and can be considered to be a *fold* operation over numbers: the first argument is the number 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 records which variables in scope. A  $\mathcal{RAFL}$  context can be extended with a variable binding (with a valid type) or a size binding. Contexts are defined inductively as follows:

$\overline{\Gamma \vdash \text{valid}}$	Empty context
$\frac{\Gamma \vdash \text{valid} \quad \Gamma \vdash T \text{type} \quad a \notin \Gamma}{\Gamma; a : T \vdash \text{valid}}$	Extend with var binding
$\frac{\Gamma \vdash \text{valid} \quad \sigma \notin \Gamma}{\Gamma; \sigma \vdash \text{valid}}$	Extend with size binding

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

$$\begin{aligned} \Gamma \vdash \text{Nat}_s &\simeq \text{Nat}_t, \text{ if } \Gamma \vdash s = t \\ \Gamma \vdash A \rightarrow B &\simeq C \rightarrow D, \text{ if } \Gamma \vdash A \simeq C \text{ and } \Gamma \vdash B \simeq D \\ \Gamma \vdash \forall \sigma. A &\simeq \forall \tau. B, \text{ if } \Gamma; \sigma \vdash A \simeq B[\sigma/\tau], \sigma \notin \text{FV}(B) \\ \Gamma \vdash \exists \sigma. (A, P) &\simeq \exists \tau. (B, Q), \text{ if } \Gamma \vdash A \simeq B, \Gamma \vdash P \Leftrightarrow Q \end{aligned}$$

For example, if  $\Gamma \vdash x : \text{Nat}_{x+y}$ , then  $\Gamma \vdash x : \text{Nat}_{y+x}$ , since addition is commutative; hence the size indices are propositionally equal, and therefore the types are convertible. However, since propositional equality of arbitrary expressions is not decidable, showing convertibility may, in general, require some programmer intervention.

The  $\mathcal{RAFL}$  type system incorporates a restricted form of dependent types, with values depending on sizes and including propositions. The rule for primitive recursion over natural numbers relies on a **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.

The `let`-binding rule indicates that the size of the value it binds affects the type of the scope,  $e_2$ . Since the context in which  $e_2$  is checked includes the size variable  $\sigma$ , and binding  $e_1$  instantiates  $\sigma$  with a size, we instantiate  $\sigma$  in the type  $T_2$  with the size of  $e_1$ . Typechecking this rule requires a small amount of evaluation at compile-time (as is often the case in a system with a form of dependent types), in order to extract the size. However, since we allow only primitive recursion and all functions are total, typechecking remains decidable. We will not discuss this typechecking algorithm further, although we note as an aside that the implementation of the algorithm requires similar techniques to those for checking dependent types [7].

## 2.5 Dynamic Semantics

The reduction rules for  $\mathcal{RAFL}$  are shown below. The  $\beta$ - and  $\delta$ -reduction rules are as normal, but with  $\delta$ -reduction substituting both a size and a value in the scope of the `let`-binding.  $\sigma$ -reduction is similar to  $\beta$ -reduction, substituting sizes into types.

$\overline{\Gamma \vdash 0 : \text{Nat}_0}$	
$\frac{\Gamma \vdash e : \text{Nat}_k}{\Gamma \vdash \text{SUC}(e) : \text{Nat}_{1+k}}$	
$\overline{x : T \in \Gamma \vdash x : T}$	
$\frac{\Gamma; x : A \vdash e : T}{\Gamma \vdash \lambda x : A : A \rightarrow T}$	
$\frac{\Gamma \vdash e_1 : A \rightarrow T \quad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1 e_2 : T}$	
$\frac{\Gamma; \sigma \vdash e : T}{\Gamma \vdash \zeta \sigma. e : \forall \sigma. T}$	
$\frac{\Gamma \vdash e : \forall \sigma. T \quad s \in S}{\Gamma \vdash e@s : T[s/\sigma]}$	
$\frac{\Gamma; \sigma \vdash e : T \quad \Gamma \vdash s \in S \quad \Gamma \vdash p(\sigma) \in P}{\Gamma \vdash (s, e) : \exists \sigma. (T, p)}$	if $\Gamma \vdash p(s)$
$\frac{\Gamma \vdash e_1 : \exists \sigma. (T_1, p) \quad \Gamma; \sigma; x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \text{let}(\sigma, x) = e_1 \text{ in } e_2 : T_2[\text{SIZE}(e_1)/\sigma]}$	
$\frac{\Gamma \vdash e_1 : \text{Nat}_s \quad \Gamma \vdash e_2 : \Phi(0) \quad \Gamma \vdash e_3 : \forall \tau. \text{Nat}_\tau \rightarrow \Phi(\tau) \rightarrow \Phi(1+\tau) \quad \Gamma; \delta \vdash \Phi(\delta) \in T, \text{ for any } \delta}{\Gamma \vdash \text{intrec } e_1 e_2 e_3 : \Phi(s)}$	
$\frac{\Gamma \vdash e : A \quad \Gamma \vdash A \simeq B}{\Gamma \vdash e : B}$	

Figure 2.  $\mathcal{RAFL}$  Typing Rules

$$\begin{aligned} \beta\text{-reduction} & \quad (\lambda x : T. e_1) e_2 \triangleright e_1[e_2/x] \\ \delta\text{-reduction} & \quad \text{let}(\tau, x) = (s, e_1) \text{ in } e_2 \triangleright e_2[s/\tau, e_1/x] \\ \sigma\text{-reduction} & \quad (\zeta \sigma. e)@s \triangleright e[s/\sigma] \end{aligned}$$

We also define computation rules for `intrec`, defining primitive recursion over naturals.

$$\begin{aligned} \text{intrec } 0 e_1 e_2 &\triangleright e_1 \\ \text{intrec } (n+1) e_1 e_2 &\triangleright e_2 n (\text{intrec } n e_1 e_2) \end{aligned}$$

Normalisation is defined by structural closure of  $\triangleright$ , using a strict semantics to avoid unnecessary complications with shared and suspended computations.

## 2.6 Example — plus

We first show the addition function as a simple example of using  $\mathcal{RAFL}$ . Here the size of the output is simply the sum of the sizes of the inputs:

$$\begin{aligned} \text{plus} & : \forall \sigma, \tau. \text{Nat}_\sigma \rightarrow \text{Nat}_\tau \rightarrow \text{Nat}_{\sigma+\tau} \\ \text{plus} & = \zeta \sigma, \tau. \lambda x : \text{Nat}_\sigma. \lambda y : \text{Nat}_\tau. \\ & \quad \text{intrec } x \ 0 \ (\zeta \gamma. \lambda k : \text{Nat}_\gamma. \lambda ih : \text{Nat}_{\gamma+\tau}. \text{SUC}(ih)) \end{aligned}$$

In the recursive call, we take the motive to be  $\Phi(\delta) = \text{Nat}_{\delta+\tau}$ , since the size resulting from the recursive call is the size of the first input ( $\sigma$ ) added to the size of the second input ( $\tau$ ), where the recursion is defined over the value of the first input.

### 3. Dependently Typed Programming

We will use EPIGRAM [25, 27] 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** [11], together with a sophisticated type-directed elaborator from source programs to the type theory.

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:

$$\text{data } \overline{\mathbb{N}} : \star \quad \text{where } \overline{0} : \overline{\mathbb{N}} \quad \frac{n : \mathbb{N}}{\text{suc } n : \overline{\mathbb{N}}}$$

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,  $\epsilon$  only targets vectors of length zero, and  $x::xs$  only targets vectors of length greater than zero:

$$\text{data } \frac{A : \star \quad n : \mathbb{N}}{\text{Vect } A \ n : \star} \quad \text{where } \frac{}{\epsilon : \text{Vect } A \ 0} \\ \frac{x : A \quad xs : \text{Vect } A \ k}{x::xs : \text{Vect } A \ (\text{suc } k)}$$

Here  $A$  and  $k$  are implicit arguments to the infix list 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 provides a static guarantee that a value will never be projected from an empty list. In order to do this, we define a datatype of finite sets, which can be used to represent numbers with an upper bound:

$$\text{data } \frac{n : \mathbb{N}}{\text{Fin } n : \star} \\ \text{where } \overline{\text{f0}} : \text{Fin } (\text{suc } n) \quad \frac{i : \text{Fin } n}{\text{fs } i : \text{Fin } (\text{suc } n)}$$

Note that there are no elements of  $\text{Fin } 0$  since this describes a finite set with zero elements. The type of **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 due to out-of-bounds accesses to the list:

$$\text{let } \frac{i : \text{Fin } n \quad xs : \text{Vect } A \ n}{\text{lookup } i \ xs : A} \\ \text{lookup } i \ xs \leftarrow \text{elim } i \leftarrow \text{case } xs \\ \text{lookup } \text{f0} \ (x :: ys) \mapsto x \\ \text{lookup } (\text{fs } j) \ (x :: ys) \mapsto \text{lookup } j \ ys$$

The **elim** and **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 **lookup** could accept a  $\text{Vect } A \ 0$ .

By giving additional static information about the **lookup** function, we obtain a stronger guarantee of its behaviour from the type-checker. The definition itself, however, is written similarly to the way it would be expressed in a simply-typed notation — indeed, it is more concise since there is no need for error checking. When writing the type of our program, we are really writing a specification for the program. Then, in writing the program, we are at the same time providing a proof that the implementation of the program meets this specification.

### 4. A Resource Bounded Interpreter

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.

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  $\text{Vect}$  and its length described in Section 3.

We have chosen EPIGRAM as an implementation language for  $\mathcal{RAFL}$  because of the strong static guarantees it provides. By representing the language as an inductive family, and by 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. It follows that if we choose the correct representation for  $\mathcal{RAFL}$ , we can explicitly include proofs of the necessary properties within our EPIGRAM-based implementation.

The representation we choose makes extensive use of inductive families; in particular, the types we use both express the relationship between values and their type and also the membership of specific contexts. This means that, without having to prove *any* theorems externally, we possess static guarantees both 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*<sup>2</sup> proofs that the required size properties are satisfied.

The interpreter translates a representation of programs in the object language ( $\mathcal{RAFL}$ ) into a program in the metalanguage (EPIGRAM). We require this translation process to remove size information from the program: since size information is used only statically (i.e. in typechecking) and not dynamically (i.e. at run-time) this information should not be present in the program which is eventually 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 metalanguage type of an object language program. Interpretation of types removes sizes and propositions, leaving only simple types.

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

<sup>2</sup>e.g. by another proof assistant such as COQ [6]

**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 metalanguage representation of the proposition, which requires an explicit proof in the metalanguage.

#### 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 [10]. Since this means that variables are simply represented by numbers, we index our representation of types by the number of bound size variables. We consequently know *exactly* how many size variables are quantified over the type, statically, simply by inspecting the index.

The EPIGRAM representation of size expressions is shown below. Variables themselves are represented as elements of a finite set,  $\text{Fin } v_s$ , where  $v_s$  is the number of size variables bound. This ensures that all size expressions are well-scoped; it would be a meta-level *compile-time* error to attempt to refer variable which is out of scope. Literals are simply represented as natural numbers,  $\mathbb{N}$ .

$$\begin{array}{l}
\text{data} \quad \frac{v_s : \mathbb{N}}{\text{SizeExp } v_s : \star} \\
\text{where} \quad \frac{n : \mathbb{N}}{\text{SNum } n : \text{SizeExp } v_s} \quad \frac{i : \text{Fin } v_s}{\text{SVar } i : \text{SizeExp } v_s} \\
\frac{x, y : \text{SizeExp } v_s}{\text{SAdd } x \ y : \text{SizeExp } v_s} \quad \frac{x, y : \text{SizeExp } v_s}{\text{SMult } x \ y : \text{SizeExp } v_s} \\
\frac{x, y : \text{SizeExp } v_s}{\text{SMax } x \ y : \text{SizeExp } v_s}
\end{array}$$

We now show the representation of propositions. Note that although this is limited here to equality and the less than or equal relation, the representation is easily extended to admit other properties.

$$\begin{array}{l}
\text{data} \quad \frac{v_s : \mathbb{N}}{\text{Prop } v_s : \star} \\
\text{where} \quad \frac{x, y : \text{SizeExp } v_s}{\text{PEq } x \ y : \text{Prop } v_s} \quad \frac{x, y : \text{SizeExp } v_s}{\text{PLE } x \ y : \text{Prop } v_s} \\
\cdots \\
\frac{x, y : \text{Prop } v_s}{\text{PAnd } x \ y : \text{Prop } v_s} \quad \frac{x, y : \text{Prop } v_s}{\text{POr } x \ y : \text{Prop } v_s}
\end{array}$$

Finally, the representation of types is shown below.

$$\begin{array}{l}
\text{data} \quad \frac{v_s : \mathbb{N}}{\text{Ty } v_s : \star} \\
\text{where} \quad \frac{s : \text{SizeExp } v_s}{\text{TyNat } s : \text{Ty } v_s} \quad \frac{S, T : \text{Ty } v_s}{S \Rightarrow T : \text{Ty } v_s} \\
\frac{T : \text{Ty } (\text{suc } v_s)}{\text{SizeBind } T : \text{Ty } v_s} \\
\frac{T : \text{Ty } (\text{suc } v_s) \quad P : \text{Prop } (\text{suc } v_s)}{\text{TyProp } T \ P : \text{Ty } v_s}
\end{array}$$

The natural number type,  $\text{TyNat}$ , is paired with a size expression, representing the object language type  $\text{Nat}_s$ . Binding a size variable introduces a new size variable into the scope, and so the index on the argument of  $\text{SizeBind}$  is incremented. Likewise, building a proposition binds a size variable, incrementing the index in the same way.

#### 4.2 Interpreting Sizes and Propositions

Before we come to the representation of  $\mathcal{RAFL}$  itself, we need to define some operations on types, propositions and sizes. In particular, we need to translate size expressions into metalanguage values, and propositions into metalanguage 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  $\epsilon$ .

$$\begin{array}{l}
\text{SizeEnv} = \text{Vect } \mathbb{N} \\
\text{let} \quad \frac{\text{sizes} : \text{SizeEnv } v_s \quad x : \text{SizeExp } v_s}{\text{sizeInterp } \text{sizes } x : \mathbb{N}}
\end{array}$$

$\text{sizeInterp}$  is defined by primitive recursion over the size expression. Size variable lookup is performed by the bounds safe  $\text{lookup}$  function defined in section 3.

$$\begin{array}{l}
\text{sizeInterp } \text{sizes} \quad x \quad \Leftarrow \text{elim } x \\
\text{sizeInterp } \text{sizes} \quad (\text{SNum } n) \quad \mapsto n \\
\text{sizeInterp } \text{sizes} \quad (\text{SVar } i) \quad \mapsto \text{lookup } i \ \text{sizes} \\
\text{sizeInterp } \text{sizes} \quad (\text{SAdd } x \ y) \\
\quad \mapsto \text{sizeInterp } \text{sizes} \ x \ + \ \text{sizeInterp } \text{sizes} \ y \\
\text{sizeInterp } \text{sizes} \quad (\text{SMult } x \ y) \\
\quad \mapsto \text{sizeInterp } \text{sizes} \ x \ \times \ \text{sizeInterp } \text{sizes} \ y \\
\text{sizeInterp } \text{sizes} \quad (\text{SMax } x \ y) \\
\quad \mapsto \max (\text{sizeInterp } \text{sizes} \ x) \ (\text{sizeInterp } \text{sizes} \ y)
\end{array}$$

We can build a metalanguage proposition (i.e. a type which is inhabited by a proof object) using the  $\text{propInterp}$  function, declared as follows and defined by recursion over  $x$ :

$$\begin{array}{l}
\text{let} \quad \frac{\text{sizes} : \text{SizeEnv } v_s \quad x : \text{TyProp } v_s}{\text{propInterp } \text{sizes} \ x : \star}
\end{array}$$

#### 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  $\mathcal{RAFL}$  programs, ignoring the size, type interpretation removes all size and proposition information from the type, leaving only a simple type:

$$\begin{array}{l}
\text{let} \quad \frac{x : \text{Ty } v_s}{\text{tyInterp } x : \star} \\
\text{tyInterp} \quad x \quad \Leftarrow \text{elim } x \\
\text{tyInterp} \quad (\text{TyNat } s) \quad \mapsto \mathbb{N} \\
\text{tyInterp} \quad (S \Rightarrow T) \quad \mapsto \text{tyInterp } S \ \rightarrow \ \text{tyInterp } T \\
\text{tyInterp} \quad (\text{SizeBind } T) \quad \mapsto \text{tyInterp } T \\
\text{tyInterp} \quad (\text{TyProp } T \ P) \quad \mapsto \text{tyInterp } T
\end{array}$$

Erasure of the size information is particularly important in a resource aware setting, since we do not want any extraneous objects at run-time 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 [26] — we use de Bruijn indices to avoid problems with renaming, and use a single well defined interface to manipulate indices.

The basic operation we need in the implementation of  $\mathcal{RAFL}$  is the substitution of a size expression into a type — in [26] 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.

$$\text{let } \frac{\text{size} : \text{SizeExp } v_s \quad t : \text{Ty } (\text{suc } v_s)}{\text{tySubst size } t : \text{Ty } v_s}$$

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 well-scoped. Unlike [26], 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 below, and mirrors the rules for contexts given in Section 2.4. The type is indexed over both the number of value variables ( $v_n$ ) and size variables ( $v_s$ ), with the de Bruijn indices for each counted independently. Each type in the context depends on the number of previously bound size variables.

$$\begin{array}{l} \text{data } \frac{v_n, v_s : \mathbb{N}}{\text{TyEnv } v_n v_s : \star} \\ \text{where } \frac{}{\text{empty} : \text{TyEnv } 0 0} \\ \frac{G : \text{TyEnv } v_n v_s}{\text{sizeExtend } G : \text{TyEnv } v_n (\text{suc } v_s)} \\ \frac{t : \text{Ty } v_s \quad G : \text{TyEnv } v_n v_s}{\text{typeExtend } t G : \text{TyEnv } (\text{suc } v_n) v_s} \end{array}$$

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:

$$\text{let } \frac{i : \text{Fin } v_n \quad G : \text{TyEnv } v_n v_s}{\text{tyLookup } i G : \text{Ty } v_s}$$

The type of **tyLookup** indicates that if we have a context with  $v_s$  size variables, we require a type which is usable in a context with  $v_s$  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 us to increase the index for each size variable by the number of size variables bound after the type. In order to achieve this, we define a function **weaken**:

$$\text{let } \frac{t : \text{Ty } v_s}{\text{weaken } t : \text{Ty } (\text{suc } v_s)}$$

We can use **weaken** to produce the definition of **tyLookup** below. This is similar to the normal vector lookup operation of Section 3, 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. Since there are no elements of  $\text{Fin } 0$ , as for **lookup**, there are no cases to deal with the empty context.

$$\begin{array}{l} \text{let } \frac{i : \text{Fin } v_n \quad G : \text{TyEnv } v_n v_s}{\text{tyLookup } i G : \text{Ty } v_s} \\ \text{tyLookup } i \quad G \quad \Leftarrow \text{elim } i \\ \text{tyLookup } \text{f0} \quad G \quad \Leftarrow \text{elim } G \\ \text{tyLookup } \text{f0} \quad (\text{sizeExtend } G) \\ \quad \mapsto \text{weaken } (\text{tyLookup } \text{f0 } G) \\ \text{tyLookup } \text{f0} \quad (\text{typeExtend } t G) \mapsto t \\ \text{tyLookup } (\text{fs } i) \quad G \quad \Leftarrow \text{elim } G \\ \text{tyLookup } (\text{fs } i) \quad (\text{sizeExtend } G) \\ \quad \mapsto \text{weaken } (\text{tyLookup } (\text{fs } i) G) \\ \text{tyLookup } (\text{fs } i) \quad (\text{typeExtend } t G) \mapsto \text{tyLookup } i G \end{array}$$

#### 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  $\text{Nat}_{\delta+\theta}$  to be used where a value of type  $\text{Nat}_{\theta+\delta}$  is expected, since we can prove the commutativity of addition.

In the EPIGRAM representation, we will need to represent the use of conversion explicitly, and we therefore define a relation to represent convertibility of types. A partial definition is given below, the remaining forms are defined structurally over types.

$$\begin{array}{l} \text{data } \frac{x, y : \text{Ty } a}{\text{Conv } x y : \star} \\ \text{where } \frac{p : (xs : \text{SizeEnv } v_s) \rightarrow \text{sizeInterp } xs i = \text{sizeInterp } xs j}{\text{ConvNat } p : \text{Conv } (\text{TyNat } i) (\text{TyNat } j)} \\ \frac{l : \text{Conv } A C \quad r : \text{Conv } B D}{\text{ConvFn } l r : \text{Conv } (A \Rightarrow B) (C \Rightarrow D)} \\ \dots \end{array}$$

Here, the most important constructor is **ConvNat** — this shows that  $\text{TyNat } i$  is convertible with  $\text{TyNat } j$ , where  $i$  and  $j$  are size expressions, provided that there is a proof  $p$  that the interpretation of both size expressions are equal in *any* context, as given by the  $xs$  argument.

#### 4.7 Language Representation

The representation of  $\mathcal{RAFL}$  terms is shown in Figure 3. 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 **Expr** also encodes the type corre-

$$\begin{array}{c}
\text{data} \quad \frac{G : \text{TyEnv } v_n v_s \quad T : \text{Ty } v_s}{\text{Expr } G T : \star} \\
\\
\text{where} \quad \frac{i : \text{Fin } v_n}{\text{var } i : \text{Expr } G (\text{tyLookup } i G)} \quad \frac{e : \text{Expr } (\text{typeExtend } A G) T}{\text{lam } e : \text{Expr } G (A \Rightarrow T)} \quad \frac{f : \text{Expr } G (A \Rightarrow T) \quad s : \text{Expr } G A}{\text{app } f s : \text{Expr } G T} \\
\\
\frac{e : \text{Expr } (\text{sizeExtend } G) T}{\text{zeta } e : \text{Expr } G (\text{SizeBind } T)} \quad \frac{f : \text{Expr } G (\text{SizeBind } T) \quad \text{size} : \text{SizeExp } v_s}{\text{sizeapp } f \text{ size} : \text{Expr } G (\text{tySubst } \text{size } T)} \\
\\
\frac{s : \text{SizeExp } v_s \quad e : \text{Expr } (\text{sizeExtend } G) T \quad P : \text{Prop } v_s \quad p : (xs : \text{SizeEnv } v_s) \rightarrow \mathbf{propInterp } xs P}{\text{prf } s e P p : \text{Expr } G (\text{TyProp } T P)} \\
\\
\frac{e : \text{Expr } G (\text{TyProp } A P) \quad \text{scope} : \text{Expr } (\text{sizeExtend } G) T}{\text{let } e \text{ scope} : \text{Expr } G (\text{tySubst } (\text{getSize } e) T)} \quad \frac{\text{conv} : \text{Conv } S T \quad e : \text{Expr } G S}{\text{convert } \text{conv } e : \text{Expr } G T} \\
\\
\frac{}{\text{zero} : \text{Expr } G (\text{TyNat } (\text{SNum } 0))} \quad \frac{j : \text{Expr } G (\text{TyNat } jn)}{\text{suc } j : \text{Expr } G (\text{TyNat } (\text{SAdd } (\text{SNum } 1) jn))} \\
\\
\frac{\Phi : \text{Ty } (\text{suc } v_s) \quad xe : \text{Expr } G (\text{TyNat } xn) \quad ze : \text{Expr } G (\text{tySubst } (\text{SNum } 0) \Phi)}{se : \text{Expr } G (\text{SizeBind } (\text{TyNat } (\text{SVar } f0) \Rightarrow \Phi \Rightarrow (\text{tySubst } (\text{SAdd } (\text{SNum } 1) (\text{SVar } f0)) (\text{weakenTy } \Phi))))} \\
\text{intrec } \Phi xe ze se : \text{Expr } G (\text{tySubst } xn \Phi)
\end{array}$$

Figure 3. Representation of  $\mathcal{RAFL}$  in EPIGRAM

$$\begin{array}{c}
\text{let} \quad \frac{\text{env} : \text{ValEnv } G \quad e : \text{Expr } G T}{\mathbf{interp } \text{env } e : \mathbf{tyInterp } T} \\
\\
\mathbf{interp } \text{env } e \quad \leftarrow \mathbf{elim } e \\
\mathbf{interp } \text{env } (\text{var } i) \quad \mapsto \mathbf{envLookup } i \text{ env} \\
\mathbf{interp } \text{env } (\text{lam}_A e) \quad \mapsto \lambda x : \mathbf{tyInterp } A. \mathbf{interp } (\text{vTypeExtend } x \text{ env}) e \\
\mathbf{interp } \text{env } (\text{app } f s) \quad \mapsto (\mathbf{interp } \text{env } f) (\mathbf{interp } \text{env } s) \\
\mathbf{interp } \text{env } (\text{zeta } e) \quad \mapsto \mathbf{interp } (\text{vSizeExtend } \text{env}) e \\
\mathbf{interp } \text{env } (\text{sizeapp } f \text{ size}) \quad \mapsto \mathbf{interp } \text{env } f \\
\mathbf{interp } \text{env } (\text{prf } s e P p) \quad \mapsto \mathbf{interp } \text{env } e \\
\mathbf{interp } \text{env } (\text{let } e \text{ scope}) \quad \mapsto \mathbf{interp } (\text{vTypeExtend } (\mathbf{interp } \text{env } e) \text{ env}) \text{ scope} \\
\mathbf{interp } \text{env } (\text{convert } \text{conv } e) \quad \mapsto \mathbf{interp } \text{env } e \\
\mathbf{interp } \text{env } \text{ zero} \quad \mapsto 0 \\
\mathbf{interp } \text{env } (\text{suc } j) \quad \mapsto \text{suc } (\mathbf{interp } \text{env } j) \\
\mathbf{interp } \text{env } (\text{intrec } \Phi x z s) \quad \mapsto \mathbf{primrec } (\mathbf{interp } \text{env } x) (\mathbf{interp } \text{env } z) (\mathbf{interp } \text{env } s) \\
\\
\text{let} \quad \frac{n : \mathbb{N} \quad z : A \quad s : \mathbb{N} \rightarrow A \rightarrow A}{\mathbf{primrec } n z s : A} \\
\mathbf{primrec } n z s \quad \leftarrow \mathbf{elim } n \\
\mathbf{primrec } 0 z s \quad \mapsto z \\
\mathbf{primrec } (\text{suc } k) z s \quad \mapsto s k (\mathbf{primrec } k z s)
\end{array}$$

Figure 4. The interpreter

sponding to the inference rules in Figure 2, where  $e : \text{Expr } G T$  means  $G \vdash e : T$ , in the usual way.

The representation requires in particular that a function must be applied to an argument of exactly the required type — an expression 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 `convert` constructor.

Where a value requires a size to conform to some property, we use the `prf` constructor which takes an explicit proof that the property holds in *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 `let` constructor is used to extract a specific size and value from an expression carrying a proof. In functions such as `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 `getSize` function is a helper function, defined simultaneously with `Expr`, which extracts the size expression from a proposition. The subscript  $T$  indicates that it is an implicit argument:

$$\text{let} \quad \frac{T : \text{Ty } (\text{suc } a) \quad e : \text{Expr } G (\text{TyProp } T P)}{\mathbf{getSize}_T e : \text{SizeExp } a}$$

Note that the `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  $ze$  argument instantiates  $\Phi$  with zero, since the input size is zero. For  $se$  the situation is more complex; its type encodes the  $\mathcal{RAL}$  type  $\forall\beta. \text{Nat}_\beta \rightarrow \Phi(\beta) \rightarrow \Phi(1 + \beta)$ .

#### 4.8 The Value Environment

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

$$\begin{array}{l} \text{data } \frac{G : \text{TyEnv } v_n v_s}{\text{ValEnv } G : \star} \\ \text{where } \frac{}{\text{vEmpty} : \text{ValEnv empty}} \\ \frac{env : \text{ValEnv } G}{\text{vSizeExtend } env : \text{ValEnv (sizeExtend } G)} \\ \frac{t : \mathbf{tyInterp } T \quad env : \text{ValEnv } G}{\text{vTypeExtend } t \ env : \text{ValEnv (typeExtend } T \ G)} \end{array}$$

Projection from the value environment corresponds to projection from the type environment. The type system ensures that the value projected is of the correct type.

$$\begin{array}{l} \text{let } \frac{G : \text{TyEnv } v_n v_s \quad i : \text{Fin } v_n \quad env : \text{ValEnv } G}{\text{envLookup}_G i \ env : \mathbf{tyInterp } (\text{tyLookup } i \ G)} \\ \text{envLookup } i \quad env \quad \leftarrow \underline{\text{elim}} \ i \\ \text{envLookup } f0 \quad env \quad \leftarrow \underline{\text{elim}} \ env \\ \text{envLookup } f0 \quad (\text{vSizeExtend } env) \quad \mapsto \text{envLookup } f0 \ env \\ \text{envLookup } f0 \quad (\text{vTypeExtend } t \ env) \quad \mapsto t \\ \text{envLookup } (fs \ i) \quad env \quad \leftarrow \underline{\text{elim}} \ env \\ \text{envLookup } (fs \ i) \quad (\text{vSizeExtend } env) \quad \mapsto \text{envLookup } (fs \ i) \ env \\ \text{envLookup } (fs \ i) \quad (\text{vTypeExtend } t \ env) \quad \mapsto \text{envLookup } i \ env \end{array}$$

#### 4.9 The Interpreter

We are now able to define the interpreter for  $\mathcal{RAL}$ , by primitive recursion over the input expression. The  $\mathbf{interp}$  function is given in Figure 4. It returns a semantic representation, as an EPIGRAM term, of the input. For example, interpreting a lambda abstraction ( $\text{lam}$ ) builds a lambda abstraction in EPIGRAM.

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

#### 4.10 Example — plus

We now return to the addition example from Section 2.6 and show how this can be represented and interpreted. An expression for  $plus$  can be defined as follows, leaving open for the moment the motive of the recursion ( $\text{Phi}$ ) and the conversion proofs (left as holes  $\square_1$  and  $\square_2$ ):

$$\begin{array}{l} plus = \text{zeta } (\text{zeta } (\text{lam } (\text{lam } \\ \quad (\text{intrec } \text{Phi } (\text{convert } (\text{ConvNat } \square_1 \ \text{zero}))) \\ \quad (\text{zeta } (\text{lam } (\text{lam } \\ \quad (\text{suc } (\text{convert } \\ \quad (\text{ConvNat } \square_2 \ (\text{var } (\text{fs } f0)))))))))) \end{array}$$

The motive is represented in an expression as a type with an extra free size variable, representing the value passed to the motive. Since we want to represent  $\Phi(\delta) = \text{Nat}_{\delta+\beta}$ , where  $\beta$  is the second size variable,  $\text{Phi}$  is defined as:

$$\text{Phi} = \text{TyNat } (\text{SAdd } (\text{SVar } f0) (\text{SVar } (\text{fs } f0)))$$

Note that as de Bruijn indices,  $\delta = 0$  and  $\beta = 1$

The conversion proofs we require then have the following types:

$$\begin{array}{l} \square_1 : (x, y : \mathbb{N}) \rightarrow 0 + y = y \\ \square_2 : (x, y, z : \mathbb{N}) \rightarrow (\text{suc } y) + z = \text{suc } (y + z) \end{array}$$

Both may be proved trivially by reflexivity. Interpreting this expression, in the empty environment, then gives a metalanguage implementation of addition without any size annotations.

$$\mathbf{interp} \ plus \ \text{vEmpty} = \lambda x, y : \mathbb{N}. \mathbf{primrec} \ x \ 0 \ (\lambda k, ih : \mathbb{N}. \text{suc } ih)$$

## 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):

$$T ::= \dots \mid [T]_s \mid \text{Tree } T_{s,s}$$

The typing rules for lists and trees are given below. The motive ( $\text{vMotive}$ ) 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 ( $\text{vMotiveT}$ ) is more complicated, since there are two size indices and two recursive arguments. To simplify the representation, we use numerals to represent elements of  $\text{Fin}$ . The representations of lists and trees as EPIGRAM terms are then shown in Figures 5—6.

$$\begin{array}{c} \frac{}{\Gamma \vdash \text{nil} : [T]_0} \\ \frac{\Gamma \vdash x : T \quad \Gamma \vdash xs : [T]_k}{\Gamma \vdash \text{cons } x \ xs : [T]_{k+1}} \\ \frac{\Gamma \vdash x : [T]_s \quad \Gamma \vdash \text{nil} : \Phi(0) \quad \Gamma \vdash \text{cons} : \forall\tau. T \rightarrow [T]_\tau \rightarrow \Phi(\tau) \rightarrow \Phi(1 + \tau)}{\Gamma \vdash \text{listrec } x \ \text{nil} \ \text{cons} : \Phi(s)} \\ \frac{}{\Gamma \vdash \text{leaf} : \text{Tree } T_{0,0}} \\ \frac{\Gamma \vdash l : \text{Tree } T_{n,i} \quad \Gamma \vdash x : T \quad \Gamma \vdash r : \text{Tree } T_{m,j}}{\Gamma \vdash \text{node } l \ x \ r : \text{Tree } T_{(n+m+1), (\text{MAX}(i,j)+1)}} \\ \frac{\Gamma \vdash x : \text{Tree } T_{s,r} \quad \Gamma \vdash \text{leaf} : \Psi(0,0) \quad \Gamma \vdash \text{node} : \forall\delta_l, \theta_l, \delta_r, \theta_r. \text{Tree } T_{\delta_l, \theta_l} \rightarrow \Psi(\delta_l, \theta_l) \rightarrow T \rightarrow \text{Tree } T_{\delta_r, \theta_r} \rightarrow \Psi(\delta_r, \theta_r) \rightarrow \Psi(\delta_l + \delta_r + 1, \text{MAX}(\theta_l, \theta_r) + 1)}{\text{treerec } x \ \text{leaf } \ \text{node} : \Psi(s, r)} \end{array}$$

$$\begin{array}{c}
\frac{}{\text{nil} : \text{Expr } G \text{ (TyList } T \text{ (SNum } 0))} \quad \frac{x : \text{Expr } G \text{ } T \quad xs : \text{Expr } G \text{ (TyList } T \text{ } xsn)}{\text{cons } x \text{ } xs : \text{Expr } G \text{ (TyList } T \text{ (SAdd (SNum } 1 \text{ } xsn))} \\
\Phi : \text{Ty (suc } v_s) \\
\frac{xe : \text{Expr } G \text{ (TyList } T \text{ } xn) \quad \text{nil\_e} : \text{Expr } G \text{ (tySubst (SNum } 0)\Phi)}{\text{cons\_e} : \text{Expr } G \text{ (SizeBind (} T \Rightarrow \text{TyList } T \text{ (SVar } f0) \Rightarrow \Phi \Rightarrow (\text{tySubst (SAdd (SNum } 1) \text{ (SVar } f0))(\text{weakenTy } \Phi))))} \\
\frac{}{\text{listrec } \Phi \text{ } xe \text{ } \text{nil\_e} \text{ } \text{cons\_e} : \text{Expr } G \text{ (tySubst } xn \text{ } \Phi)}
\end{array}$$

Figure 5. Representation of Lists

$$\begin{array}{c}
\text{leaf} : \text{Expr } G \text{ (TyTree } T \text{ (SNum } 0) \text{ (SNum } 0)) \\
\frac{l : \text{Expr } G \text{ (TyTree } T \text{ } nl \text{ } dl) \quad x : \text{Expr } G \text{ } T \quad r : \text{Expr } G \text{ (TyTree } T \text{ } nr \text{ } dr)}{\text{node } l \text{ } x \text{ } r : \text{Expr } G \text{ (TyTree (SAdd (SNum } 1) \text{ (SAdd } nl \text{ } nr)) (SAdd (SNum } 1) \text{ (SMax } dl \text{ } dr)))} \\
\text{phi} = \lambda x : \text{SizeExp } v_s. \lambda y : \text{SizeExp } v_s. \text{tySubst } x \text{ (weaken } y) \\
\Phi : \text{Ty (} 4 + sn) \\
\frac{xe : \text{Expr } G \text{ (Tree } T \text{ } n \text{ } d) \quad \text{leaf\_e} : \text{Expr } G \text{ (tySubst (SNum } 0)\Phi)}{\text{node\_e} : \text{Expr } G \text{ (SizeBind (SizeBind (SizeBind (SizeBind} \\
\text{(TyTree } T \text{ (SVar } 3) \text{ (SVar } 2) \Rightarrow \text{psi (SVar } 3) \text{ (SVar } 2) \Rightarrow} \\
\text{ } T \Rightarrow \text{TyTree } T \text{ (SVar } 1) \text{ (SVar } 0) \Rightarrow \text{psi (SVar } 1) \text{ (SVar } 0) \Rightarrow} \\
\text{psi (SAdd (SNum } 1) \text{ (SAdd (SVar } 3) \text{ (SVar } 1)))} \\
\text{(SAdd (SNum } 1) \text{ (SMax (SVar } 2) \text{ (SVar } 0))))} \\
\text{treerec } \Phi \text{ } xe \text{ } \text{leaf\_e} \text{ } \text{node\_e} : \text{Expr } G \text{ (tySubst } n \text{ (tySubst } d \text{ } \Phi))}
\end{array}$$

Figure 6. Representation of Trees

## 6. Examples

In this section we give some examples of programs written in  $\mathcal{R}\mathcal{A}\mathcal{F}\mathcal{L}$  that use lists and trees. These are programs of the kind we discussed as motivating examples in Section 2.1; here we show that the core language of  $\mathcal{R}\mathcal{A}\mathcal{F}\mathcal{L}$  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  $\mathcal{R}\mathcal{A}\mathcal{F}\mathcal{L}$  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  $\mathcal{R}\mathcal{A}\mathcal{F}\mathcal{L}$ .

$$\begin{array}{l}
\text{append} : \forall \alpha, \beta. [A]_\alpha \rightarrow [A]_\beta \rightarrow [A]_{\alpha+\beta} \\
\text{append} = \zeta \alpha, \beta. \lambda xs : [A]_\alpha. \lambda ys : [A]_\beta. \text{listrec } xs \\
\quad \text{ys} \\
\quad (\zeta \gamma. \lambda y : A. \lambda ys : [A]_\gamma. \lambda ysH : [A]_{\gamma+\beta}. \\
\quad \quad \text{cons } y \text{ } ysH)
\end{array}$$

As with **plus**, typechecking the recursion operators relies on inferring an appropriate motive,  $\Phi$ , for the recursion. Knowing that the target of the recursion is  $xs : [A]_\alpha$ , 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, in order to typecheck **listrec**, we take  $\Phi(\delta) = \delta + \beta$ .

### 6.2 Filter — Higher Order Functions

For **append**, the size of the result is exact, since it is computed directly as the sum of the input sizes. However, it is not always possible to obtain an exact size, since the size of the result may depend on the specific contents of the input, for example. One example of this is **filter**, whose result size depends on the number of elements in the list that conform to the (higher-order) filtering predicate.

We assume the introduction of a boolean type **Bool** to  $\mathcal{R}\mathcal{A}\mathcal{F}\mathcal{L}$  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 size of the result of **filter** statically, but merely have an upper bound on the size, we need to return a constrained type. The result 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**:

$$\begin{array}{l}
\text{filter} : \forall \alpha. [A]_\alpha \rightarrow (A \rightarrow \text{Bool}) \rightarrow \exists \beta. ([A]_\beta, \beta \leq \alpha) \\
\text{filter} = \zeta \alpha. \lambda xs : [A]_\alpha. \lambda p : (A \rightarrow \text{Bool}). \text{listrec } xs \\
\quad (0, \text{nil})_{p1} \\
\quad (\zeta \beta. \lambda y : A. \lambda ys : [A]_\beta. \lambda ysH : \exists \gamma. ([A]_\gamma, \gamma \leq \beta). \\
\quad \quad \text{let } (\delta, \text{val}) = \text{ysH in} \\
\quad \quad \text{if } (p \text{ } y) \\
\quad \quad \quad \text{then } (1 + \delta, \text{cons } y \text{ } \text{val})_{p2} \\
\quad \quad \quad \text{else } (\delta, \text{val})_{p3})
\end{array}$$

We have marked the values which require proofs as  $p1$ ,  $p2$  and  $p3$ . This definition will only typecheck if the sizes of these values can be shown to satisfy the required properties; these properties are:

$p1 : 0 \leq 0$   
 $p2 : \delta \leq \beta \rightarrow 1 + \delta \leq 1 + \beta$   
 $p3 : \delta \leq \beta \rightarrow \delta \leq 1 + \beta$

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

$\square_1 : 0 \leq 0$   
 $\square_2 : (x : \mathbb{N}) \rightarrow (y : \mathbb{N}) \rightarrow x \leq y \rightarrow (\text{succ } x) \leq (\text{succ } y)$   
 $\square_2 : (x : \mathbb{N}) \rightarrow (y : \mathbb{N}) \rightarrow x \leq y \rightarrow x \leq (\text{succ } y)$

Each of these are straightforward to prove with the Omega decision procedure [33], and can therefore be shown by the  $\mathcal{R}\mathcal{A}\mathcal{F}\mathcal{L}$  typechecker with no user interaction.

### 6.3 Flattening Trees

In  $\mathcal{R}\mathcal{A}\mathcal{F}\mathcal{L}$ , we have defined binary trees to have two size metrics: the number of elements in the tree; and the maximum depth of that tree. Each of these metrics is useful in different contexts. For example, the maximum depth can be used to specify a worst-case execution time for a search algorithm, or to specify a maximum depth to preserve the balance of the tree; or the number of elements in the tree can be used to bound a traversal of the tree structure. We here use this length metric to guarantee that the flattening of a tree into a list maintains the correct number of elements.

**flatten** :  $\forall \alpha, \beta. \text{Tree } A_{\alpha, \beta} \rightarrow [A]_{\alpha}$   
**flatten** =  $\zeta \alpha, \beta. \lambda t : \text{Tree } A_{\alpha, \beta}. \text{treerec } t$   
 $\text{nil}$   
 $(\zeta \delta_l, \theta_l, \delta_r, \theta_r. \lambda l : \text{Tree } A_{\delta_l, \theta_l}. \lambda r : \text{Tree } A_{\delta_r, \theta_r}. \lambda x : A. \lambda r : \text{Tree } A_{\delta_r, \theta_r}. \lambda rH : [A]_{\delta_r}. (\text{append}@_{\delta_l}@_{\theta_l}(1 + \delta_r) lH rH))$

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

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.

## 7. Related Work

This paper focuses on the *a priori* and rigorous construction of bounds on resource usage from program source, with proofs derived from properties expressed in an extended dependent 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 the costs of resource usage (e.g. [8, 9, 13, 16, 18, 21, 23, 20]). Our work goes beyond this earlier work by exploiting full-spectrum dependent types to support the construction and automatic checking of formal proofs from programmer specified properties. Our approach allows a much richer set of properties to be expressed and proved — it can be seen as a restricted form of proof-carrying code [30], 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.

Types have also been used to expose properties other than the resource cost properties we consider here. For example, Igarashi and

Kobayashi [22] describe a type system that ensures that file operations are correctly sequenced; Flanagan and Abadi [12] have shown how types can be used to enforce safe locking to avoid race conditions in concurrent systems; Popeea and Chin [32] have introduced a type system for verifying protocols; Chin et al. [5] have shown how relational sizes can be used to verify safety properties for object-based languages; Mandelbaum et al. have developed a type system for reasoning about the behaviour of effectful programs [24]; and Xi [39] has shown how dependent types can be exposed to verify program termination properties. We anticipate that the dependently typed approach described here could, in due course, be extended to a number of these domains.

### 7.1 Dependent Types and Resource Bounding

In their LXRes system, Crary and Weirich [9] add an explicit clock to a simple form of dependent type system. This clock is used to record differences in time usage, and is related to concrete time costs derived from TAL (Typed Assembly Language) [29]. A well-typed program in their setting does not “expire”, i.e. it does not take more time than the clock allows. In contrast, our metrics are primarily concerned with space, although their meaning is flexible and we could consider similarly extending the type system of  $\mathcal{R}\mathcal{A}\mathcal{F}\mathcal{L}$ . Unlike Crary and Weirich, we have not yet considered how the size information in our type system maps to low level assembly code or memory allocation.

Grobauer [13] has similarly used *singleton* dependent types to extract cost recurrences from a first-order Dependent ML (DML) program. Like  $\mathcal{R}\mathcal{A}\mathcal{F}\mathcal{L}$ , size-annotated types are used to capture size information. However, DML imposes similar restrictions in terms of proofs to the sized type system of Hughes, Pareto and Sabry [21]. Singleton types alone impose some limitations — our  $\text{Nat}_S$  type is a singleton (only one element of size  $S$ ), but coupled with the ability to express arbitrary propositions, we can express the type of a range of integers, e.g.  $\exists \sigma. (\text{Nat}_{\sigma}, 10 \leq \sigma \leq 20)$ .

Finally, in our own previous work [2], we have explored the use of dependent types to represent resource bounds, where we write resource-aware programs directly in the meta-language. In this paper we take the idea further by implementing a full, domain specific resource aware language. The advantage of this new approach is that we can be sure of maintaining resource safety through the restricted type system, although at the cost of some flexibility.

### 7.2 Amortized Analysis

LFPL [16] uses linear types to determine resource usage patterns. A special resource type called “*diamond*” is used to count constructors. First-order LFPL definitions can be computed in linearly bounded space, even in the presence of general recursion. More recently, Hofmann and Jost have introduced automatic inference of these resource types [18, 19], and thus of heap-space consumption, using an amortized cost model built on a difference metric similar to that of Crary and Weirich [9] and avoiding linearity restrictions. Our system uses explicit annotations which yields powerful tools to the programmer, while the goal in [18, 19] was to minimise programmer intervention, hence limiting control. Extensions of LFPL to higher-order functions have been studied in [17] where it was shown that such programs can be evaluated using dynamic programming in time  $O(2^{p(n)})$  where  $n$  is the size of the input and  $p$  is a fixed polynomial. It has been shown that this is equivalent to polynomial space plus an unbounded stack.

### 7.3 Sized Types

Hughes, Pareto and Sabry [21] originally developed *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 [20], a strict functional language using regions to control memory usage. The notion of size is fixed and limited to one metric for each data type (lists and integers). 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 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 [4] 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 [34, 37].

### 7.4 Resource-Bounded Staged Programming

Taha et al's GeHB [36] is a two-level staged notation that builds on LFPL. GeHB automatically generates first-order, bounded LFPL programs from higher-order specifications. Taha's work [35] emphasises the distinction between computation on the development and deployment platforms; on the development platform (the first stage) computations can involve arbitrary resource consumption, but in the second stage no new heap allocations are allowed.

We have previously considered the combination of a dependently typed meta-language with multi-stage programming for the generation of verified translators from a verified interpreter [3]. By adding staging annotations to the interpreter we implement in this paper, we would get a translator from  $\mathcal{RAFL}$  to EPIGRAM which is sound with respect to the typing rules of  $\mathcal{RAFL}$ .

## 8. Conclusions and Further Work

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 full-spectrum dependently typed metalanguage, we have been able to develop a language interpreter that embeds automatically checkable guarantees of the specified resource properties through the construction of formal proofs.

Where in previous work, we have focused on the definition of type effect systems permitting the *reconstruction* of simple sized types for non-recursive [34] and primitive recursive definitions [38], considering both space usage and time usage [37], 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. We believe, but have not yet proved, that there exists a formal translation between all terms in this earlier system and  $\mathcal{RAFL}$ , i.e. that  $\mathcal{RAFL}$  is strictly more powerful. However, unlike earlier work,  $\mathcal{RAFL}$  requires explicit proofs and propositions on resource usage. The representation of a  $\mathcal{RAFL}$  program is therefore not only resource safe but also externally verifiable, for example by another proof assistant such as COQ.

Like our own earlier work and much other work on resource costing, the approach we have described here is limited to primitive recursion. We believe this to be a sensible limitation, since it en-

ures termination in the type system and decidability of type and proof checking. Including general recursion would increase the expressivity of the language, at the cost of losing guarantees of total correctness.

### 8.1 Further Work

While our approach is described here in the context of a simple standalone, functional programming notation, our ultimate objective is to apply these ideas in Hume [14]: a domain-specific functionally-based language targeting real-time embedded systems, eventually producing a multi-stage language for embedded systems that is capable of enforcing strong but flexible resource bounds. To achieve this, we must incorporate a sized type system like that of  $\mathcal{RAFL}$  into Hume, extend the work described here to cover the entire Hume language, and derive additional concrete resource cost metrics.

In this paper, we have described resource usage costs in terms of absolute data structure sizes, where these sizes may be used either directly as a cost metric, or used indirectly to yield bounds on induction parameters in recursive definitions. It is straightforward to move from these abstract size metrics to concrete metrics in terms of heap allocations or time usage simply by instantiating constants appropriately [37]; in the case of stack or heap deallocations, however, where costs do not increase monotonically, we must use a size-difference metric such as that of Crary and Weirich [9] or Hofmann and Jost [19]. We have prototyped such a metric for time usage [1] in Hume, based on the Hofmann and Jost amortized model, targeted to a concrete machine architecture, the PowerPC G4, and using a formally-derived automatic program analysis. We must now adapt this metric to the work described here in order to obtain a complete time model in our dependently typed metalanguage.

### Acknowledgements

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

We would like to thank Steffen Jost and Walid Taha for their comments on earlier versions of this paper.

### References

- [1] A. Bonenfant, Z. Chen, K. Hammond, G. Michaelson, A. Wallace, and I. Wallace. Towards Resource-Certified Image Processing Software. Submitted to The Fourth ASIAN Symposium on Programming Languages and Systems (APLAS 2006), 2006.
- [2] E. Brady and K. Hammond. A Dependently Typed Framework for Static Analysis of Program Execution Costs. In *Proc. IFL 2005 — International Workshop on Implementation and Applications of Functional Programming, Dublin, Ireland*. Springer-Verlag Lecture Notes in Computer Science, 2006.
- [3] E. Brady and K. Hammond. A Verified Staged Interpreter is a Verified Compiler: Multi-stage Programming with Dependent Types. In *Proc. 2006 ACM Conf. on Generative Programming and Component Architecture, Portland, Oregon*, 2006.
- [4] W.-N. Chin and S.-C. Khoo. Calculating Sized Types. *Higher-Order and Symbolic Computing*, 14(2,3):261–300, 2001.
- [5] W.-N. Chin, S.-C. Khoo, S. Qin, C. Popeea, and H. H. Nguyen. Verifying Safety Policies with Size Properties and Alias Controls. In *Proc. ICSE '05: 27th International Conference on Software Engineering*, pages 186–195, New York, NY, USA, 2005. ACM Press.
- [6] Coq Development Team. The Coq proof assistant — reference manual. <http://coq.inria.fr/>, 2001.

- [7] T. Coquand. An Algorithm for Type-Checking Dependent Types. *Science of Computer Programming*, 26(1-3):167–177, 1996.
- [8] K. Cray and J. C. Vanderwaart. An Expressive, Scalable Type Theory for Certified Code. In *Proc. ICFP '02: Seventh ACM SIGPLAN International Conference on Functional Programming*, pages 191–205, New York, NY, USA, 2002. ACM Press.
- [9] K. Cray and S. Weirich. Resource Bound Certification. In *Proc. POPL '00: 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 184–198, New York, NY, USA, 2000. ACM Press.
- [10] N. de Bruijn. Lambda Calculus Notation with Nameless Dummies. *Indagationes Mathematicae*, 34:381–392, 1972.
- [11] P. Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4):440–465, 1994.
- [12] C. Flanagan and M. Abadi. Types for Safe Locking. In *Proc. ESOP '99: 8th European Symposium on Programming Languages and Systems*, pages 91–108, London, UK, 1999. Springer-Verlag.
- [13] B. Grobauer. Cost recurrences for DML programs. In *Proc. ICFP '01: Sixth ACM SIGPLAN International Conference on Functional Programming*, pages 253–264, New York, NY, USA, 2001. ACM Press.
- [14] K. Hammond and G. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *Proc. GPCE '03: Intl. Conf. Generative Programming and Component Engineering*. Springer-Verlag, Lecture Notes in Computer Science, 2003.
- [15] K. Hammond and G. Michaelson. Predictable Space Behaviour in FSM-Hume. In *Proc. IFL '02: International Workshop on Implementation of Functional Langs., Madrid, Spain*. Springer-Verlag, Lecture Notes in Computer Science 2670, 2003.
- [16] M. Hofmann. A Type System for Bounded Space and Functional In-Place Update. *Nordic Journal of Computing*, 7(4), Winter 2000.
- [17] M. Hofmann. The strength of non size-increasing computation. In *Proc. POPL '02: ACM Symposium on Principles of Programming Languages*. ACM Press, 2002.
- [18] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *Proc. POPL '03: ACM Symposium on Principles of Programming Languages*, pages 185–197, New Orleans, LA, USA, Jan. 2003. ACM Press.
- [19] M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis. In *Proc. ESOP 2006: 2006 European Symposium on Programming*, pages 22–37, 2006.
- [20] R. Hughes and L. Pareto. Recursion and Dynamic Data Structures in Bounded Space: Towards Embedded ML Programming. In *Proc. ICFP '99: 1999 ACM Intl. Conf. on Functional Programming*, pages 70–81, 1999.
- [21] R. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems using Sized Types. In *Proc. POPL '96: ACM Symposium on Principles of Programming Languages*, St Petersburg, FL, January 1996.
- [22] A. Igarashi and N. Kobayashi. Resource usage analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 331–342, New York, NY, USA, 2002. ACM Press.
- [23] N. Kobayashi. Time Regions and Effects for Resource Usage Analysis. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in Language Design and Implementation*, pages 50–61, New York, NY, USA, 2003. ACM Press.
- [24] Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 213–225, New York, NY, USA, 2003. ACM Press.
- [25] C. McBride. Epigram: Practical Programming with Dependent Types. Lecture Notes, International Summer School on Advanced Functional Programming, 2004.
- [26] C. McBride and J. McKinna. I am not a Number, I am a Free Variable. In *Proceedings of the ACM SIGPLAN Haskell Workshop*, 2004.
- [27] C. McBride and J. McKinna. The View From The Left. *Journal of Functional Programming*, 14(1):69–111, 2004.
- [28] J. McKinna. *Deliverables: a Categorical Approach to Program Development in Type Theory*. PhD thesis, University of Edinburgh, 1991.
- [29] G. Morrisett, D. Walker, K. Cray, and N. Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):528–569, May 1999.
- [30] G. C. Necula. Proof-Carrying Code. In *Proc. POPL '97: 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, New York, NY, USA, 1997. ACM Press.
- [31] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple Unification-Based Type Inference for GADTs. In *Proc. ICFP '06: 2006 International Conf. on Functional Programming*, 2006.
- [32] C. Popea and W.-N. Chin. A Type System for Resource Protocol Verification and its Correctness Proof. In *Proc. PEPM '04: 2004 ACM SIGPLAN symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 135–146, New York, NY, USA, 2004. ACM Press.
- [33] W. Pugh. The Omega Test: a Fast and Practical Integer Programming Algorithm for Dependence Analysis. *Communications of the ACM*, pages 102–114, 1992.
- [34] 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.
- [35] W. Taha. Resource-Aware Programming. In *Proc. First International Conference on Embedded Software and Systems*, pages 38–43, 2004.
- [36] W. Taha, S. Ellner, and H. Xi. Generating Heap-Bounded Programs in a Functional Setting. In *Proc. EMSOFT '03: 2003 Intl. Conf. on Embedded Software*, number 2855 in LNCS, pages 340–355. Springer-Verlag, 2003.
- [37] P. Vasconcelos. *Cost Inference and Analysis for Recursive Functional Programs*. PhD thesis, University of St Andrews, 2006. in preparation.
- [38] P. Vasconcelos and K. Hammond. Inferring Costs for Recursive, Polymorphic and Higher-Order Functional Programs. In *Proc. IFL '03: International Workshop on Implementation of Functional Languages*, pages 86–101. Springer-Verlag LNCS, 2004.
- [39] H. Xi. Dependent Types for Program Termination Verification. In *Proc. LICS '01: 16th IEEE Symposium on Logic in Computer Science, Boston, June 2000*, 2000.
- [40] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proc. POPL '99: 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 214–227, New York, NY, 1999.