LambdaM: A Simple Language with Termination Checking based on Dependent Types

Wenhao Tang

Department of EECS
Peking University

May 5, 2020

Contents

1 Introduction

Termination checking means checking if the program will terminate before actually running the program. It is of vital significance because of the following reasons:

- Most programs are supposed to terminate.
- Termination checking can reduce run-time bugs which type checking cannot find.
- For interactive theorem provers like Agda and Coq which are based on the Curry-Howard isomorphism, only terminating functions correspond to valid proofs.

However, because of the undecidability of the Halting Problem, there is no method to check the termination of all programs. But like the type system, we can give some rules to guarantee the programm will terminate when passing these rules while some terminating program may not pass the rules.

Previous methods can be divided into two styles:

- Syntactical Method: based on type-less syntactical structures, such as Foetus termination checker which is currently used in Agda [Abel, 1998].
- Type-based Method: design a stronger type system to check the termination, such as the sized types [Abel, 2004, Barthe et al., 2004].

Syntactical methods have many drawbacks, for example it is hard to find a syntactical criterion for complex terms and it is sensitive to small changes in the code. The type-based method is a better choice. The main idea of sized types is equipping the types of recursive data structures with size information. A recursive function is only accepted if the sizes of arguments to recursive calls are bounded by the size of its inputs. Another type-based method is to use a restricted form of dependent types [Xi and Pfenning, 1999, Xi, 2001]. Similar to sized type, it uses a metric to compare between different types. The dependent types based method requires the user to provide more informations meanwhile provides a stronger expressiveness.

Our work λ_M, where the M stands for metric, is a simple language with termination checking based on dependent types. We implemente the complete dependent product types and dependent sum types in λ_M and introduce metrics into it. The main idea of λ_M is similar to the $ML_{0,\ll}^{\Pi,\Sigma}$ proposed by [Xi, 2001]. We assign recursive functions with metrics and try to check if the metrics in the recursive calls are less than the metrics in the definition. The metrics are closely related to the parameters of the functions. We introduce a type family of vector which dependents on natural numbers and use the varible of natural numbers used in the dependent vector type as the metric of functions. The details will be discussed in section 4.

The λ_M has some shortcomings compared to the $ML_{0,\ll}^{\Pi,\Sigma}$, for example, we don't allow linear constraints on metrics. But one of our advantages is λ_M supports other comparison functions for metrics. λ_M currently only supports recursive functions which take at least one parameter of type vector. It is easy to extend it to support other type families and other types of metrics in addition to natural numbers.

The main syntax and type systems of λ_M are discussed in section 2 and section 3, together with evaluation rules in section 5. In section 4, we discuss the metrics and termination checking of λ_M. Section 6 gives some examples which show the ability of termination checking of λ_M. The usage of the code is shown in section 7, before we conclude in section 8.

2 Syntax

The syntax of MetricML can be divided into two parts: one part for a simply typed lambda calculus with some extensions and dependent types; another part for metrics and recursive functions with metrics. The following subsections will discuss the two parts in detail separately.

2.1 λ-Calculus with Dependent Types

The syntax of MetricML is based on the simply typed lambda calculus with booleans, natural numbers and general recursion. We extend it with the dependent product types $\Pi x : S.T$ and dependent sum types $\Sigma x : S.T$, which are generalization of function types and pair types. When x doesn't appear freely in T, $\Pi x : S.T$ is the same as $S \to T$ and $\Sigma x : S.T$ is the same as (S, T).

In order to write some non-trivial examples, we need to add some type families (also called dependent types) into MetricML. A better way to introduce type families is to introduce the mechanism for defining new data types including type families.

But for simplicity, we use a built-in type family of vector: vector dependents on natural numbers. $Vector\ n$ (where n is of type Nat) is a type which represents all vectors with length n. It is enough for writing some examples to show the ability of termination checking of λ_M. And it is not hard to generalize it to other family types.

The full syntax except the metrics and recursive functions with metrics can be found in the following table.

t	::=		terms:
		x	variable
		$\lambda x : T.t$	abstraction
		$t\ t$	application
		$(t, t : \Sigma x : T.T)$	typed pair
		$t.1$	first projection
		$t.2$	second projection
		true	constant true
		false	constant false
		if t then t else t	conditional
		0	constant zero
		succ t	successor
		pred t	predecessor
		iszero t	zero test
		fix t	fix point of t
		nil	empty vector
		cons $t\ t\ t$	vector constructor
		isnil $t\ t$	test for empty vector
		head $t\ t$	head of a vector
		tail $t\ t$	tail of a vector
v	::=		values:
		$\lambda x : T.t$	abstraction value
		$(v, v : \Sigma x : T.T)$	pair value
		true	true value

		false	false value
		nv	numeric value
		nil	empty vector
		cons t v v	vector value
nv	$::=$		numeric values:
		0	zero value
		succ nv	successor value
T	$::=$		types:
		$\Pi x : T.T$	dependent product type
		T t	type family application
		$\Sigma x : T.T$	dependent sum type
		Bool	type of booleans
		Nat	type of natural numbers
		Vector t	type family of vectors
K	$::=$		kinds:
		$*$	kind of proper types
		$\Pi x : T.K$	kind of type families
Γ	$::=$		contexts:
		$\varnothing$	empty context
		$\Gamma, x : T$	term variable binding
		$\Gamma, x : K$	type variable binding

2.2 Metrics and Recursive Functions with Metrics

The metrics are just defined as tuples of terms. Actually we only use lists of natural numbers in the current system, but other terms can also be used with no additional difficulty (because we already have a complete depedent type mechanism).

Recursive functions can be assigned with a metric. And when it is assigned with a metric, the type checker will also perform the role of a termination checker which checks whether the function will terminate given any input.

The new syntax for metrics and recursive functions with metrics are shown below.

t	$::=$	$\dots$	terms:
		fun $f : [m]$ $T.t$	recursive functions with metrics
		f $[m]$	application of functions to metrics
m	$::=$		metrics:
		t	terms
		m, t	tuples of terms

Here are some explanations of the term fun $f : [m]$ $T.t$: suppose m is a metric $n_1, n_2, \dots, n_k$ with k natural numbers, then we have a recursive function named f. The actual type of f is $\Pi n_1 : Nat.\Pi n_2 : Nat.\dots.\Pi n_k : Nat.T$ and the actual function body of f is $\lambda n_1 : Nat.\lambda n_2 :$

$Nat. \ldots .\lambda n_k : Nat.t$. It indicates that we can use the derived forms of them for evaluation. Further discussions can be found in section 5.

3 Kinding and Typing

The typing rules for booleans, natrual numbers and fixpoint is exactly the same as the rules decribed in *Types and Programming Language*[Pierce, 2002] and I won't write down them anymore for simplicity. The following subsections show the kinding rules and typing rules for dependent product types, dependent sum types and the built-in type family of vectors dependented on natural numbers. The typing rules for recursive functions with metrics will be discussed in section 4.

The definition of type equivalence is needed when kinding and typing because of the application rules. The last subsection discusses the definition of equivalence used in this system.

3.1 Kinding Rules for Dependent Types

We follow the convention to use $*$ to represent the proper types, i.e. the sorts of type expressions that are actually used to classify terms, like Bool or Nat. To formalize the kinding, first we need to define what is a well-formed kind.
Well-formed kinds: $\boxed{\Gamma \vdash K}$

$$\Gamma \vdash * \qquad \text{(WF-STAR)}$$

$$\frac{\Gamma \vdash T :: * \quad \Gamma, x : T \vdash K}{\Gamma \vdash \Pi x : T.K} \qquad \text{(WF-PI)}$$

Kinding: $\boxed{\Gamma \vdash T :: K}$

$$\frac{X :: K \in \Gamma \quad \Gamma \vdash K}{\Gamma \vdash X :: K} \qquad \text{(K-VAR)}$$

$$\frac{\Gamma \vdash S :: * \quad \Gamma, x : S \vdash T :: *}{\Gamma \vdash \Pi x : S.T :: *} \qquad \text{(K-PI)}$$

$$\frac{\Gamma \vdash S :: \Pi x : T.K \quad \Gamma \vdash t : T}{\Gamma \vdash S\ t : [x \mapsto t]K} \qquad \text{(K-APP)}$$

$$\frac{\Gamma \vdash S :: * \quad \Gamma, x : S \vdash T :: *}{\Gamma \vdash \Sigma x : S.T :: *} \qquad \text{(K-SIGMA)}$$

$$\frac{\Gamma \vdash T :: K \quad \Gamma \vdash K \equiv K'}{\Gamma \vdash T :: K'} \qquad \text{(K-CONV)}$$

3.2 Typing Rules for Dependent Types

Typing: $\boxed{\Gamma \vdash t : T}$

$$\frac{x : T \in \Gamma \quad \Gamma \vdash T :: *}{\Gamma \vdash x : T} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash S :: * \quad \Gamma, x : S \vdash t : T}{\Gamma \vdash \lambda x : S.t \;\; : \Pi x : S.T} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : \Pi x : S.T \quad \Gamma \vdash t_2 : S}{\Gamma \vdash t_1 \; t_2 : [x \mapsto t_2]T} \qquad \text{(T-APP)}$$

$$\frac{\Gamma \vdash t_1 : S \quad \Gamma \vdash t_2 : [x \mapsto t_1]T}{\Gamma \vdash (t_1, t_2 : \Sigma x : S.T) \;\; : \Sigma x : S.T} \qquad \text{(T-PAIR)}$$

$$\frac{\Gamma \vdash t : \Sigma x : S.T}{\Gamma \vdash t.1 : S} \qquad \text{(T-PROJ1)}$$

$$\frac{\Gamma \vdash t : \Sigma x : S.T}{\Gamma \vdash t.2 : [x \mapsto t.1]T} \qquad \text{(T-PROJ2)}$$

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash T \equiv T' :: *}{\Gamma \vdash t : T'} \qquad \text{(T-CONV)}$$

The rules above are quite similar to the rules in the chapter 2 of *Advanced Topics in Types and Programming Languages* [Pierce, 2005]. To implement these rules, we need to reformulate them to be syntax-derived. The syntax-derived algorithmic version of typing and kinding rules can be found in the chapter 2 of *ATTAPL*. For simplicity, we don't repeat them here.

3.3 Kinding and Typing of Dependent Vectors

We introduce a built-in type family of vector into λ_M. The kinding and typing rules of the terms associated with it are shown below. To simplify we only allow the vector to hold natural numbers. It can be easily generalized to terms with other types.

Kinding $\boxed{\Gamma \vdash T :: K}$

$$\Gamma \vdash \text{Vector } n :: * \quad \text{(T-CONV)}$$

Notice that the first term n taken by *cons, isnil, head* and *tail* indicates the length of the vector it handles and we check if the length of the vector it takes is n when typing.

Typing $\boxed{\Gamma \vdash t : T}$

$$\Gamma \vdash \text{nil} : \text{Vector } 0 \qquad \text{(T-NIL)}$$

$$\frac{\Gamma \vdash n : Nat \quad \Gamma \vdash t_1 : Nat \quad \Gamma \vdash t_2 : \text{Vector } n' \quad n' \equiv n}{\Gamma \vdash \text{cons } n \; t_1 \; t_2 : \text{Vector (succ } n)} \qquad \text{(T-CONS)}$$

$$\frac{\Gamma \vdash n : Nat \quad \Gamma \vdash t : \text{Vector } n' \quad n' \equiv n}{\Gamma \vdash \text{isnil } n \; t : Bool} \qquad \text{(T-ISNIL)}$$

$$\frac{\Gamma \vdash n : Nat \quad \Gamma \vdash t : \text{Vector } n' \quad n' \equiv n}{\Gamma \vdash \text{head } n \; t : Nat} \qquad \text{(T-HEAD)}$$

$$\frac{\Gamma \vdash n : Nat \quad \Gamma \vdash t : \text{Vector } n' \quad n' \equiv n}{\Gamma \vdash \text{tail } n \; t : \text{Vector (pred } n)} \qquad \text{(T-TAIL)}$$

3.4 Equivalence of Types and Terms

Type equivalence is needed because there are kinds application and type application rules where we need to check if two types are equivalence. We consider the structural equivalence: two types are equivalence if and only if they have the same structure. Since we have introduced the dependent types, checking the equivalence of two types includes checking the equivalence of two terms.

Because the dependent types in λ_M are only used for metrics, we only need to check the equivalence of terms with the type Nat. In fact, I add a bit more restrictions on the metrics: we can only use terms which always terminate when evaluating in metrics. Since λ_M has the ability of termination checking, we can just use the terms which passed the termination checking of λ_M.

Thus the equivalence checking of terms can be simplified to evaluating the terms and checking if the structure of the results are the same. The equivalence rules for types and terms can all be derived to structural equivalence which are quite straightforward and their formalization is omitted here.

4 Termination Checking

4.1 Termination Checking using Metric

Before we start, it is worth noting that although λ_M provide the fixpoint operator, we won't check the terminability of recursive functions defined using the fixpoint. Every recursive function to be checked for termination must be defined using the $fun\ f : [m]T.t$ mentioned in section 2.2.

The main idea of the termination checking of λ_M is similar to the $ML_{0,\ll}^{\Pi,\Sigma}$ proposed by [Xi, 2001]. Every recursive function defined by $fun\ f : [m]T.t$ is assigned with a metric m, which is related to the parameters of function f. For example, a recursive function which takes a paremeter of type Vector n and return a term of type T may have a metric n and is defined as $fun\ f : [n]\Pi v : \text{Vector } n.T.t$. f is actually a function with type $\Pi n : Nat.\Pi v : \text{Vector } n.T$ If for every recursive call of itself $f\ [m]\ t'$ in t, m is less than n, then the recursive function f is garanteed to be terminated since the metric is of type Nat and cannot decrease forever.

For simplicity we only consider the recursive functions which has at least one parameter of type Vector n.

Definition 1 (Metric of function). *Suppose all parameters with type Vector n of function f is Vector n_1, Vector n_2, $\ldots$, Vector n_k, then a tuple of an arbitrary subsequence of $n_1, \ldots, n_k$ is a metric of function f.*

If we choose $n'_1, \ldots, n'_p$ to be the metric of function f, then the definition of f should be modified as $fun\ f : [n'_1, \ldots, n'_p]T.t$.

The following theorem guarantees the correctness of termination checking:

Theorem 1 (Termination of function). *Suppose function f has metric $n_1, \ldots, n_k$ and is defined as $fun\ f : [n_1, \ldots, n_k]T.t$, if every recursive call of the form $f[m_1, \ldots, m_k]t'$ in t satisfy $(n_1, \ldots, n_k) < (m_1, \ldots, m_k)$ where $<$ is the the lexicographical comparison operator between tuples, then f will terminate given any input.*

Proof. The proof of this theorem is straightforward. Whenever a recursive call to itself is evaluted, the metric will decrease. Since the metric is simply a tuple of natural numbers, it won't decrease forever. Thus it must terminate after a finite number of recursive calls. The only possibility for unterminating terms in λ_M comes from infinite recursive calls. Thus every function satisfies the condition of this theorem will terminate. $\square$

Notice that we can replace the lexicographical comparison with other comparison functions between metrics as long as the comparison guarantees the tuple of natural numbers cannot decrease forever. For example we can use $h((n_1, \ldots, n_k), (m_1, \ldots, m_k)) = \sum_{i=1}^{k} n_i < \sum_{i=1}^{k} m_i$ as the comparison function. The proof of correctness is almost the same.

Here is a simple example of a recursive function which has a decreasing metric because *pred* $n < n$ is always true. More examples can be found in the example section.

```
fun sum : [n] Πv:Vector n. Nat
  λv : Vector n.
    if isnil v then 0
    else head v + sum [pred n] (tail v)
```

4.2 Typing Rules for Recursive Functions with Metric

In this section we want to formulate the institution we discussed above into typing rules. To achieve this, we need to introduce another typing relation $\Gamma \vdash t : T <_f m_f$. The idea is similar to [Xi, 2001].

Definition 2 (Definitional Metric of Recursive Function). *Suppose f is a recursive function whose definition is $fun\ f : [m].T.t$, then $m_f := m$ is the definitional metric of function f.*

Definition 3 (Typing Relation with Metric). $\Gamma \vdash t : T <_f m_f$ *not only means that the term t has type T under context Γ, but also means that f is a recursive function with a definitional metric m_f and for every occurence of $f[m]$ in term t, $m < m_f$ is true, where $<$ is a comparison function of metrics.*

To implement it we need a context for metrics. Use Φ to denote the context for metrics, Φ is defined as:

Φ	::=		metric contexts:
		$\varnothing$	empty context
		$\Phi, m_f : m$	metric binding

The m_f means the definitional metric of a funtion f. The typing relation with metric is actually $\Phi; \Gamma \vdash t : T <_f m_f$. Notice that the Φ can be combined into the Γ, we only write the Γ context for simplicity.

The typing rules for the typing relation $\Gamma \vdash t : T <_f m_f$ is almost the same as the typing rules for $\Gamma \vdash t : T$ except the cases of definition and application of recursive functions with metrics. We give the formulation of typing rules for definition and application of recursive functions with metrics below.

Typing: $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash n_i : Nat \quad \Gamma \vdash T :: * \qquad \Gamma, m_f : (n_1, \ldots, n_k), f : \Pi n_1 : Nat \ldots \Pi n_k : Nat.T, n_i : Nat \vdash t : T <_f m_f}{\Gamma \vdash \text{fun } f : [n_1, n_2, \ldots, n_k]\ T.t : \Pi n_1 : Nat \ldots \Pi n_k : Nat.T} \quad \text{(T-MFUN)}$$

$$\frac{\Gamma \vdash f : \Pi n_1 : Nat \ldots \Pi n_k : Nat.T \quad m = (m_1, \ldots, m_k) \quad \Gamma \vdash m_i : Nat}{\Gamma \vdash f[m] : [n_1 \mapsto m_1, \ldots, n_k \mapsto m_k]T} \quad \text{(T-MAPP)}$$

Typing with metric: $\boxed{\Gamma \vdash t : T <_f m_f}$

$$\frac{\Gamma \vdash f : \Pi n_1 : Nat \ldots \Pi n_k : Nat.T \quad m = (m_1, \ldots, m_k) \quad \Gamma \vdash m_i : Nat \quad m < m_f}{\Gamma \vdash f[m] : [n_1 \mapsto m_1, \ldots, n_k \mapsto m_k]T <_f m_f} \quad \text{(T'-MAPP)}$$

Notice that in T-MFUN we use the typing relation with metric $\Gamma \vdash t : T <_f m_f$ and in T'-MAPP we check if the m in application $f[m]$ is less than m_f to guarantee the termination. Other typing rules for $\Gamma \vdash t : T <_f m_f$ is the same as $\Gamma \vdash t : T$ and we won't repeat here.

5 Operational Semantics

As in the case with typing rules, the evaluation rules for booleans, natrual numbers and fixpoint is exactly the same as the rules decribed in *TAPL*. And since the dependent product types and dependent sum typs are just generalizations of function types and pair types, the evaluation rules for $\lambda x : S.t$ and $(t_1, t_2 : T)$ are exactly the same as the rules for normal abstraction and pair terms. Thus we only emphasize the evaluation rules for the type family of vector and recursive functions with metrics.

5.1 Evaluation Rules for Dependent Vectors

As we have said that the first parameter of most terms for dependent vectors except nil is a natrual number n which indicates the length of the vector it handles. It is only used for typing and we can just omit it during evaluation. The evaluation rules are given below.

$$\frac{t_1 \to t_1'}{\text{cons } n\ t_1\ t_2 \to \text{cons } n\ t_1'\ t_2} \quad \text{(E-CONS1)}$$

$$\frac{t_2 \to t_2'}{\text{cons } n\ v_1\ t_2 \to \text{cons } n\ v_1\ t_2'} \quad \text{(E-CONS2)}$$

$$\text{isnil } n\ (\text{nil}) \to \text{true} \quad \text{(E-ISNILNIL)}$$

$$\text{isnil } n\ (\text{cons } n\ v_1\ v_2) \to \text{false} \quad \text{(E-ISNILCONS)}$$

$$\frac{t_1 \to t_1'}{\text{isnil } n\ t_1 \to \text{isnil } n\ t_1'} \quad \text{(E-ISNIL)}$$

$$\text{head } n\ (\text{cons } n\ v_1\ v_2) \to v_1 \quad \text{(E-HEADCONS)}$$

$$\frac{t_1 \to t_1'}{\text{head } n\ t_1 \to \text{head } n\ t_1'} \quad \text{(E-HEAD)}$$

$$\text{tail } n \text{ (cons } n\ v_1\ v_2) \to v_2 \qquad \text{(E-TAILCONS)}$$

$$\frac{t_1 \to t_1'}{\text{tail } n\ t_1 \to \text{tail } n\ t_1'} \qquad \text{(E-TAIL)}$$

5.2 Derived Froms of Recursive Functions with Metrics

We can just use derived forms for evaluation of $fun\ f : [m]T.t$ and $f[m]$ terms.

$$\text{fun } f : [n_1, \ldots, n_k]T.t \stackrel{\text{def}}{=}$$
$$\text{fix } (\lambda f : \Pi n_1 : \text{Nat}.\Pi n_2 : \text{Nat} \ldots \Pi n_k : \text{Nat}.T\ .\ \lambda n_1 : \text{Nat}.\lambda n_2 : \text{Nat} \ldots \lambda n_k : \text{Nat}.t)$$
$$f[n_1, n_2, \ldots, n_k] \stackrel{\text{def}}{=} (\ldots((f\ n_1)n_2)\ldots)n_k$$

6 Examples

In this section we give some examples to show the ability of termination checking of λ_M. These examples are all about functions operating on vectors. The codes of all examples can be found and test in the source codes of λ_M.

6.1 Lexicographical Comparison between Metrics

First we show some functions which lexicographical comparison between metrics is enough to declare they will terminate.

The first example is *lenless*, which compares if the length of the first vector is less than the length of the second vector using recursion. It is obvious that $(pred\ n_1, pred\ n_2) < (n_1, n_2)$.

```
fun lenless : [n₁, n₂] Π v₁:Vector n₁. Π v₂:Vector n₂. Bool
  λ v₁ : Vector n₁.
    λ v₂ : Vector n₂.
      if isnil n₁v₁ then true
      else if isnil n₂v₂ then false
      else lenless [pred n₁, pred n₂] (tail n₁v₁) (tail n₂v₂)
```

The second example is *evens*, which returns a vector of all elements at even positions of the original vector. Notice that since the length of the result vector is uncertain, we use a dependent sum type for the result. For neatness we omit the explicit annotation of the type T in $(t_1, t_2 : T)$.

```
fun evens : [n] Π v:Vector n. Π d:Nat. Σ p:Nat.Vector(p)
  λ v : Vector n. λ d : Nat.
    if isnil n v then (0, nil)
    else if iseven d then
      (succ (evens (pred n) (tail n v) (succ d)).1,
       cons
        (succ (evens (pred n) (tail n v) (succ d)).1
        (head n v) (evens (pred n) (tail n v) (succ d)).2
        )
      else
        ((evens (pred n) (tail n v) (succ d)).1,
         (evens (pred n) (tail n v) (succ d)).2)
```

The third example is *append*, which concatenates two vectors (i.e. append the second vector to the end of the first vector). To define it, we fitst define *snoc*, a reverse version of *cons* which insert an element at the end of a vector. Notice that in *append* we only use the length of the second vector as the metric because it will decrease every recursive call.

```
fun snoc : [n] Π v:Vector n. Π x:Nat. Vector (succ n)
  λ v : Vector n. λ x : Nat.
    if isnil n v then cons n x v
    else cons n (head n v) (snoc (pred n) (tail v) x)
fun append : [n2] Π n1 : Nat.Π v1 : Vector n1 . Π v2 : Vector n2 . Σ p : Vector p.
  λ n1 : Nat. λ v1 : Vector n1 . λ v2 : Vector n2.
    if isnil n2 v2 then (n1, v1)
    else ((append [pred n2] (succ n1) (snoc n1 v1 (head n2 v2) (tail n2 v2))).1,
           (append [pred n2] (succ n1) (snoc n1 v1 (head n2 v2) (tail n2 v2))).2)
```

6.2 Other Comparison Functions between Metrics

In addition to the lexicographical comparison, there are also some other comparison functions which guarantee the tuple of natural numbers cannotdecrease forever. We can also use them to compare two metrics in the typing rule T'-MAPP. One possible comparison is to compare the sum of all elements of the tuples, which we call the sum comparison. There are many examples which cannot pass the termination checking when using the lexicographical comparison but can be proved to be terminating using the sum comparison. For example, consider the following function g of two vectors v_1, v_2:

$$g(v_1, v_2) = \begin{cases} g(\mathit{tail}\ (\mathit{tail}\ v_1), \mathit{cons}\ 0\ v_2){+\!+}g(\mathit{cons}\ 0\ v_1 + \mathit{tail}\ (\mathit{tail}\ v_2)) & |v_1| \geq 2 \wedge |v_2| \geq 2 \\ v_1{+\!+}g(\mathit{cons}\ 0\ v_1 + \mathit{tail}\ (\mathit{tail}\ v_2)) & |v_1| < 2 \wedge |v_2| \geq 2 \\ g(\mathit{tail}\ (\mathit{tail}\ v_1), \mathit{cons}\ 0\ v_2){+\!+}v_2 & |v_1| \geq 2 \wedge |v_2| < 2 \\ v_1{+\!+}v_2 & |v_1| < 2 \wedge |v_2| < 2 \end{cases}$$

where ++ means the concatenation of two vectors.

Suppose the length of v_1 is n_1 and the length of v_2 is n_2, then a possible definitional metric of g is (n_1, n_2). But the metric of recursive calls to g are $(n_1 - 2, n_1 + 1)$ and $(n_1 + 1, n_2 - 2)$, which are neither comparable to (n_1, n_2) using the lexicographical comparison. The sum comparsion can solve this problem since the sum of $(n_1 - 2, n_1 + 1)$ and $(n_1 + 1, n_2 - 2)$ are both $n_1 + n_2 - 1$ and it is less than $n_1 + n_2$.

The code to compute function g in λ_M is algo given in the source code and we don't write it here for simplicity.

7 Implementation and Usage

7.1 Code Structure

The source code of λ_M is written in OCaml. It is based on the implemantation of simply typed λ-calculus from the code of *TAPL*.

Contents of each file:

- *syntax.ml* contains the definition of abstract syntax trees and associated support functions for substitution, context management, etc.
- *typecheck.ml* contains functions for kinding, typing and equivalence checking.
- *eval.ml* contains functions for evaluation.
- *metric.ml* contains functions for metric management and comparison.
- *print.ml* contains functions for printing and debugging.
- *examples.ml* contains all the examples discussed in section 6 together with some other examples and test functions.

I haven't wrote a lexer and parser for λ_M, thus all examples are written in the form of abstract syntax tree using OCaml. One conequence is that you need to handle the De Bruijn index by yourself when directly writing abstract syntax tree. But despite this, the syntax tree is a good choice for testing languages.

7.2 Build and Test

To build the λ_M:

- Run `make` to build λ_M.
- Run `make clean` to clean up the directory.

To test with the examples in *examples.ml*:

- Run `utop -init init.ml` or `ocaml -init init.ml` to test the code in the REPL of OCaml.
- Use `prty function_name_here` to print the type of functions.
- Use `prty test_term_here` to print the type of terms and use `prtm test_term_here` to print the results of terms.

You can also write abstract syntax trees by yourself and test them using `prty` and `prtm`.

8 Conclusion

In conclusion, we have proposed and implementated the λ_M, a simple language with termination checking based on dependent types. And we show that many non-trivial recursive functions on dependent vectors can be written and checkked for terminability in λ_M. The λ_M has a slight advantage over the $ML_{0,\ll}^{\Pi,\Sigma}$ proposed by [Xi, 2001]: λ_M supports different comparison functions between metrics. But since λ_M doesn't support linear constraints on metrics, it is still weaker than $ML_{0,\ll}^{\Pi,\Sigma}$ in other respects.

Here we list some possible improvements of λ_M:

- Support linear or even non-linear constraints on metrics.

- Introduce more complicated terms into metrics in addition to natural numbers, such as tree structures.
- Introduce a more powerful machanism of type inference to simplify the process of writing codes.

References

A. Abel. Foetus–termination checker for simple functional programs. *Programming Lab Report*, (474), 1998.

A. Abel. Termination checking with types. *Theoretical Informatics and Applications*, 38(4): 277–319, 2004.

G. Barthe, M. J. Frade, E. Gimenez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14(1):97–141, 2004.

B. Pierce. *Types and programming languages*. MIT Press, 2002.

B. Pierce. *Advanced topics in types and programming languages*. MIT Press, 2005.

H. Xi. Dependent types for program termination verification. 15(1):231–242, 2001.

H. Xi and F. Pfenning. Dependent types in practical programming. pages 214–227, 1999.