Dependent Types for Program Termination Verification *

Hongwei Xi
University of Cincinnati
hwxi@ececs.uc.edu

Abstract

Program termination verification is a challenging research subject of significant practical importance. While there is already a rich body of literature on this subject, it is still undeniably a difficult task to design a termination checker for a realistic programming language that supports general recursion. In this paper, we present an approach to program termination verification that makes use of a form of dependent types developed in Dependent ML (DML), demonstrating a novel application of such dependent types to establishing a liveness property. We design a type system that enables the programmer to supply metrics for verifying program termination and prove that every well-typed program in this type system is terminating. We also provide realistic examples, which are all verified in a prototype implementation, to support the effectiveness of our approach to program termination verification as well as its unobtrusiveness to programming. The main contribution of the paper lies in the design of an approach to program termination verification that smoothly combines types with metrics, yielding a type system capable of guaranteeing program termination that supports a general form of recursion (including mutual recursion), higher-order functions, algebraic datatypes, and polymorphism.

1 Introduction

Programming is notoriously error-prone. As a consequence, a great number of approaches have been developed to facilitate program error detection. In practice, the programmer often knows certain program properties that must hold in a *correct* implementation; it is therefore an indication of program errors if the actual implementation violates some of these properties. For instance, various type systems have been designed to detect program errors that cause violations of the supported type disciplines.

It is common in practice that the programmer often knows for some reasons that a particular program should terminate if implemented correctly. This immediately implies that a termination checker can be of great value for detecting program errors that cause nonterminating program execution. However, termination checking in a realistic programming language that supports general recursion is often prohibitively expensive given that (a) program termination in such a language is in general undecidable, (b) termination checking often requires interactive theorem proving that can be too involved for the programmer, (c) a minor change in a program can readily demand a renewed effort in termination checking, and (d) a large number of changes are likely to be made in a program development cycle. In order to design a termination checker for practical use, these issues must be properly addressed.

There is already a rich literature on termination verification. Most approaches to automated termination proofs for either programs or term rewriting systems (TRSs) use various heuristics, some of which can be highly involved, to synthesize well-founded orderings (e.g., various path orderings [3], polynomial interpretation [1], etc.). While these approaches are mainly developed for first-order languages, the work in higher-order settings can also be found (e.g., [7]). When a program, which should be terminating if implemented correctly, cannot be proven terminating, it is often difficult for the programmer to determine whether this is caused by a program error or by the limitation of the heuristics involved. Therefore, such automated approaches are likely to offer little help in detecting program errors that cause nonterminating program execution. In addition, automated approaches often have difficulty handling realistic (not necessarily large) programs.

The programmer can also prove program termination in various (interactive) theorem proving systems such as NuPrl [2], Coq [4], Isabelle [8] and PVS [9]. This is a viable practice and various successes have been reported. However, the main problem with this practice is that the programmer may often need to spend so much time on proving the termination of a program compared with the time spent on simply implementing the program. In addition, a renewed effort may be required each time when some changes, which are likely in a program development cycle, are made to the program. Therefore, the programmer can often feel hesitant to adopt (interactive) theorem proving for detecting program errors in general programming.

We are primarily interested in finding a middle ground. In particular, we are interested in forming a mechanism in a programming language that allows the programmer to provide *key* information needed for establishing program termination

* Partially supported by NSF grant no. CCR-0092703

0-7695-1281-X/01 $10.00 © 2001 IEEE

```
fun ack m n =
  if m = 0 then n+1
  else if n = 0 then ack (m-1) 1 else ack (m-1) (ack m (n-1))
withtype {i:nat,j:nat} <i,j> => int(i) -> int(j) -> [k:nat] int(k)
```

Figure 1. An implementation of Ackerman function

and then automatically verifies that the provided information indeed suffices. An analogy would be like allowing the user to provide induction hypotheses in inductive theorem proving and then proving theorems with the provided induction hypotheses. Clearly, the challenging question is how such key information for establishing program termination can be formalized and then expressed. The main contribution of this paper lies in our attempt to address the question by presenting a design that allows the programmer to provide through dependent types such key information in a (relatively) simple and clean way.

It is common in practice to prove the termination of recursive functions with metrics. Roughly speaking, we attach a metric in a well-founded ordering to a recursive function and verify that the metric is always decreasing when a recursive function call is made. In this paper, we present an approach that uses the dependent types developed in DML [18, 14] to carry metrics for proving program termination. We form a type system in which metrics can be encoded into types and prove that every well-typed program is terminating. It should be emphasized that we are not here advocating the design of a programming language in which only terminating programs can be written. Instead, we are interested in designing a mechanism in a programming language, which, if the programmer chooses to use it, can facilitate program termination verification. This is to be manifested in that the type system we form can be smoothly embedded into the type system of DML. We now illustrate the basic idea with a concrete example before going into further details.

In Figure 1, an implementation of Ackerman function is given. The `withtype` clause is a type annotation, which states that for natural numbers i and j, this function takes an argument of type $\texttt{int}(i)$ and another argument of type $\texttt{int}(j)$ and returns a natural number as a result. Note that we have refined the usual integer type `int` into infinitely many singleton types $\texttt{int}(a)$ for $a = 0, 1, -1, 2, -2, \ldots$ such that $\texttt{int}(a)$ is precisely the type for integer expressions with value equal to a. We write `{i:nat,j:nat}` for universally quantifying over index variables i and j of sort *nat*, that is, the sort for index expressions with values being natural numbers. Also, we write `[k:nat] int(k)` for $\Sigma k : nat.\texttt{int}(k)$, which represents the sum of all types $\texttt{int}(k)$ for $k = 0, 1, 2, \ldots$. The novelty here is the pair $\langle i, j\rangle$ in the type annotation, which indicates that this is the metric to be used for termination checking. We now informally explain how termination checking is performed in this case; assume that i and j are two natural numbers and m and n have types $\texttt{int}(i)$ and $\texttt{int}(j)$, respectively, and attach the metric $\langle i, j\rangle$ to *ack* m n; note that there are three recursive function calls to *ack* in the body of *ack*; we attach the metric $\langle i-1, 1\rangle$ to the first *ack* since $m-1$ and 1 have types $\texttt{int}(i-1)$ and $\texttt{int}(1)$, respectively; similarly, we attach the metric $\langle i-1, k\rangle$ to the second *ack*, where k is assumed to be some natural number, and the metric $\langle i, j-1\rangle$ to the third *ack*; it is obvious that $\langle i-1, 1\rangle < \langle i, j\rangle$, $\langle i-1, k\rangle < \langle i, j\rangle$ and $\langle i, j-1\rangle < \langle i, j\rangle$ hold, where $<$ is the usual lexicographic ordering on pairs of natural numbers; we thus claim that the function *ack* is terminating (by a theorem proven in this paper). Note that although this is a simple example, its termination cannot be proven with (lexicographical) structural ordering (as the semantic meaning of both addition $+$ and subtraction $-$ is needed).[1]

More realistic examples are to be presented in Section 5, involving dependent datatypes [15], mutual recursion, higher-order functions and polymorphism. The reader may read some of these examples before studying the sections on technical development so as to get a feel as to what can actually be handled by our approach.

Combining metrics with the dependent types in DML poses a number of theoretical and pragmatic questions. We briefly outline our results and design choices.

The first question that arises is to decide what metrics we should support. Clearly, the variety of metrics for establishing program termination is endless in practice. In this paper, we only consider metrics that are tuples of index expressions of sort *nat* and use the usual lexicographic ordering to compare metrics. The main reasons for this decision are that (a) such metrics are commonly used in practice to establish termination proofs for a large variety of programs and (b) constraints generated from comparing such metrics can be readily handled by the constraint solver *already* built for type-checking DML programs. Note that the usual structural ordering on *first-order* terms can be obtained by attaching to the term the number of constructors in the term, which can be readily accomplished by using the dependent datatype mechanism in DML. However, we are currently unable to capture structural ordering on higher-order terms.

The second question is about establishing the soundness of our approach, that is, proving every well-typed program in the type system we design is terminating. Though the idea mentioned in the example of Ackerman function seems intuitive, this task is far from being trivial because of the presence of higher-order functions. The reader may take a look at the higher-order example in Section 5 to understand this. We seek a method that can be readily adapted to handle various common programming features when they are added,

[1] There is an implementation of Ackerman function that involves only primitive recursion and can thus be easily proven terminating, but the point we drive here is that this particular implementation can be proven terminating with our approach.

including mutual recursion, datatypes, polymorphism, etc. This naturally leads us to the reducibility method [12]. We are to form a notion of reducibility for the dependent types extended with metrics, in which the novelty lies in the treatment of general recursion. This formation, which is novel to our knowledge, constitutes the main technical contribution of the paper.

The third question is about integrating our termination checking mechanism with DML. In practice, it is common to encounter a case where the termination of a function f depends on the termination of another function g, which, unfortunately, is not proven for various reasons, e.g., it is beyond the reach of the adopted mechanism for termination checking or the programmer is simply unwilling to spend the effort proving it. Our approach is designed in a way that allows the programmer to provide a metric in this case for verifying the termination of f conditional on the termination of g, which can still be useful for detecting program errors.

The presented work builds upon our previous work on the use of dependent types in practical programming [18, 14]. While the work has its roots in DML, it is largely unclear, *a priori*, how dependent types in DML can be used for establishing program termination. We thus believe that it is a significant effort to actually design a type system that combines types with metrics and then prove that the type system guarantees program termination. This effort is further strengthened with a prototype implementation and a variety of verified examples.

The rest of the paper is organized as follows. We form a language $\mathrm{ML}_0^{\Pi,\Sigma}$ in Section 2, which essentially extends the simply typed call-by-value λ-calculus with a form of dependent types, developed in DML, and recursion. We then extend $\mathrm{ML}_0^{\Pi,\Sigma}$ to $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ in Section 3, combining metrics with types, and prove that every program in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ is terminating. In Section 4, we enrich $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ with some significant programming features such as datatypes, mutual recursion and polymorphism. We present some examples in Section 5, illustrating how our approach to program termination verification is applied in practice. We then mention some related work and conclude.

There is a full paper available on-line [16] in which the reader can find details omitted here.

2 $\mathrm{ML}_0^{\Pi,\Sigma}$

We start with a language $\mathrm{ML}_0^{\Pi,\Sigma}$, which essentially extends the simply typed call-by-value λ-calculus with a form of dependent types and (general) recursion. The syntax for $\mathrm{ML}_0^{\Pi,\Sigma}$ is given in Figure 2.

2.1 Syntax

We fix an integer domain and restrict type index expressions, namely, the expressions that can be used to index a type, to this domain. This is a sorted domain and subset sorts can be formed. For instance, we use nat for the subset sort $\{a : int \mid a \geq 0\}$. We use $\delta(\vec{\imath})$ for a base type indexed with a sequence of index expressions $\vec{\imath}$, which may be empty. For instance, $\mathtt{bool}(0)$ and $\mathtt{bool}(1)$ are types for boolean values *false* and *true*, respectively; for each integer i, $\mathtt{int}(i)$ is the singleton type for integer expressions with value equal to i.

We use $\phi \models P$ for a satisfaction relation, which means P holds under ϕ, that is, the formula $(\phi)P$, defined below, is satisfied in the domain of integers.

$$\begin{gathered}(\cdot)\Phi = \Phi \qquad (\phi, a : int)\Phi = (\phi)\forall a : int.\Phi \\ (\phi, a : \{a : \gamma \mid P\})\Phi = (\phi, a : \gamma)(P \supset \Phi) \\ (\phi, P)\Phi = (\phi)(P \supset \Phi)\end{gathered}$$

For instance, the satisfaction relation

$$a : nat, a \neq 0 \models a - 1 \geq 0$$

holds since the following formula is true in the integer domain.

$$\forall a : int.a \geq 0 \supset (a \neq 0 \supset a - 1 \geq 0)$$

Note that the decidability of the satisfaction relation depends on the constraint domain. For the integer constraint domain we use here, the satisfaction relation is decidable (as we do not accept nonlinear integer constraints).

We use $\Pi a : \gamma.\tau$ and $\Sigma a : \gamma.\tau$ for the usual dependent function and sum types, respectively. A type of form $\Pi\vec{a} : \vec{\gamma}.\tau$ is essentially equivalent to $\Pi a_1 : \gamma_1 \ldots \Pi a_n : \gamma_n.\tau$, where we use $\vec{a} : \vec{\gamma}$ for $a_1 : \gamma_1, \ldots, a_n : \gamma_n$.[2] We also introduce λ-variables and ρ-variables in $\mathrm{ML}_0^{\Pi,\Sigma}$ and use x and f for them, respectively. A lambda-abstraction can only be formed over a λ-variable while recursion (via fixed point operator) must be formed over a ρ-variable. A λ-variable is a value but a ρ-variable is not.

We use λ for abstracting over index variables, **lam** for abstracting over variables, and **fun** for forming recursive functions. Note that the body after either λ or **fun** must be a value. We use $\langle i \mid e\rangle$ for packing an index i with an expression e to form an expression of a dependent sum type, and **open** for unpacking an expression of a dependent sum type.

2.2 Static Semantics

We write $\phi \vdash \tau : *$ to mean that τ is a legally formed type under ϕ and omit the standard rules for such judgments.

$$\begin{array}{lcl}\text{index substitutions} & \theta_I & ::= \; [] \mid \theta_I[a \mapsto i] \\ \text{substitutions} & \theta & ::= \; [] \mid \theta[x \mapsto e] \mid \theta[f \mapsto e]\end{array}$$

A substitution is a finite mapping and [] represents an empty mapping. We use θ_I for a substitution mapping index variables to index expressions and $\mathbf{dom}(\theta_I)$ for the domain of θ_I. Similar notations are used for substitutions on variables. We write $\bullet[\theta_I]$ ($\bullet[\theta]$) for the result from applying θ_I (θ) to $\bullet$, where $\bullet$ can be a type, an expression, etc. The standard

[2] In practice, we also have types of form $\Sigma\vec{a} : \vec{\gamma}.\tau$, which we omit here for simplifying the presentation.

$$\begin{array}{llll}
\text{index constants} & c_I & ::= & \cdots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \cdots \\
\text{index expressions} & i & ::= & a \mid c_I \mid i_1 + i_2 \mid i_1 - i_2 \mid i_1 * i_2 \mid i_1 / i_2 \\
\text{index propositions} & P & ::= & i_1 < i_2 \mid i_1 \leq i_2 \mid i_1 > i_2 \mid i_1 \geq i_2 \mid i_1 = i_2 \mid i_1 \neq i_2 \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \\
\text{index sorts} & \gamma & ::= & \mathit{int} \mid \{a : \gamma \mid P\} \\
\text{index variable contexts} & \phi & ::= & \cdot \mid \phi, a : \gamma \mid \phi, P \\
\text{index constraints} & \Phi & ::= & P \mid P \supset \Phi \cdot\mid \forall a : \gamma.\Phi \\
\text{types} & \tau & ::= & \delta(\vec{\imath}) \mid \Pi\vec{a} : \vec{\gamma}.\tau \mid \Sigma a : \gamma.\tau \\
\text{contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : \tau \mid \Gamma, f : \tau \\
\text{constants} & c & ::= & \mathit{true} \mid \mathit{false} \mid 0 \mid 1 \mid -1 \mid 2 \mid -2 \mid \cdots \\
\text{expressions} & e & ::= & c \mid x \mid f \mid \mathbf{if}(e, e_1, e_2) \mid \lambda\vec{a} : \vec{\gamma}.v \mid \mathbf{lam}\ x : \tau.e \mid e_1(e_2) \mid \\
 & & & \mathbf{fun}\ f[\vec{a} : \vec{\gamma}] : \tau\ \mathbf{is}\ v \mid e[\vec{\imath}] \mid \langle i \mid e \rangle \mid \mathbf{open}\ e_1\ \mathbf{as}\ \langle a \mid x \rangle\ \mathbf{in}\ e_2 \\
\text{values} & v & ::= & c \mid x \mid \lambda\vec{a} : \vec{\gamma}.v \mid \mathbf{lam}\ x : \tau.e \mid \langle i \mid v \rangle
\end{array}$$

Figure 2. The syntax for $\mathrm{ML}_0^{\Pi,\Sigma}$

$$\frac{\phi; \Gamma \vdash e : \tau_1 \quad \phi \vdash \tau_1 \equiv \tau_2}{\phi; \Gamma \vdash e : \tau_2}\ \textbf{(type-eq)} \qquad \frac{\Gamma(x) = \tau}{\phi; \Gamma \vdash x : \tau}\ \textbf{(type-}\boldsymbol{\lambda}\textbf{-var)} \qquad \frac{\Gamma(f) = \tau}{\phi; \Gamma \vdash f : \tau}\ \textbf{(type-}\boldsymbol{\rho}\textbf{-var)}$$

$$\frac{\phi, \vec{a} : \vec{\gamma}; \Gamma \vdash v : \tau}{\phi; \Gamma \vdash \lambda\vec{a} : \vec{\gamma}.v : \Pi\vec{a} : \vec{\gamma}.\tau}\ \textbf{(type-ilam)} \qquad \frac{\phi; \Gamma \vdash e : \Pi\vec{a} : \vec{\gamma}.\tau \quad \phi \vdash \vec{\imath} : \vec{\gamma}}{\phi; \Gamma \vdash e[\vec{\imath}] : \tau[\vec{a} \mapsto \vec{\imath}]}\ \textbf{(type-iapp)}$$

$$\frac{\phi, \vec{a} : \vec{\gamma}; \Gamma, f : \Pi\vec{a} : \vec{\gamma}.\tau \vdash v : \tau}{\phi; \Gamma \vdash \mathbf{fun}\ f[\vec{a} : \vec{\gamma}] : \tau\ \mathbf{is}\ v : \Pi\vec{a} : \vec{\gamma}.\tau}\ \textbf{(type-fun)}$$

$$\frac{\phi; \Gamma \vdash e : \mathtt{bool}(i) \quad \phi, i = 1; \Gamma \vdash e_1 : \tau \quad \phi, i = 0; \Gamma \vdash e_2 : \tau}{\phi; \Gamma \vdash \mathbf{if}(e, e_1, e_2) : \tau}\ \textbf{(type-if)}$$

$$\frac{\phi; \Gamma, x : \tau_1 \vdash e : \tau_2}{\phi; \Gamma \vdash \mathbf{lam}\ x : \tau_1.e : \tau_1 \to \tau_2}\ \textbf{(type-lam)} \qquad \frac{\phi; \Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \phi; \Gamma \vdash e_2 : \tau_1}{\phi; \Gamma \vdash e_1(e_2) : \tau_2}\ \textbf{(type-app)}$$

$$\frac{\phi; \Gamma \vdash e_1 : \Sigma a : \gamma.\tau_1 \quad \phi, a : \gamma; \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\phi; \Gamma \vdash \mathbf{open}\ e_1\ \mathbf{as}\ \langle a \mid x \rangle\ \mathbf{in}\ e_2 : \tau_2}\ \textbf{(type-open)} \qquad \frac{\phi \vdash i : \gamma \quad \phi; \Gamma \vdash e : \tau[a \mapsto i]}{\phi; \Gamma \vdash \langle i \mid e \rangle : \Sigma a : \gamma.\tau}\ \textbf{(type-pack)}$$

Figure 3. Typing Rules for $\mathrm{ML}_0^{\Pi,\Sigma}$

definition is omitted. The following rules are for judgments of form $\phi \vdash \theta_I : \phi'$, which roughly means that θ_I has "type" ϕ'.

$$\frac{}{\phi \vdash [] : \cdot}\ \textbf{(sub-i-empty)}$$

$$\frac{\phi \vdash \theta_I : \phi' \quad \phi \vdash i : \gamma[\theta_I]}{\phi \vdash \theta_I[a \mapsto i] : \phi', a : \gamma}\ \textbf{(sub-i-var)}$$

$$\frac{\phi \vdash \theta_I : \phi' \quad \phi \models P[\theta_I]}{\phi \vdash \theta_I : \phi', P}\ \textbf{(sub-i-prop)}$$

We write $\mathbf{dom}(\Gamma)$ for the domain of Γ, that is, the set of variables declared in Γ. Given substitutions θ_I and θ, we say $\phi; \Gamma \vdash (\theta_I; \theta) : (\phi'; \Gamma')$ holds if $\phi \vdash \theta_I : \phi'$ and $\mathbf{dom}(\theta) = \mathbf{dom}(\Gamma')$ and $\phi; \Gamma \vdash \theta(x) : \Gamma'(x)[\theta_I]$ for all $x \in \mathbf{dom}(\Gamma')$.

We write $\phi \models \tau \equiv \tau'$ for the congruent extension of $\phi \models i = j$ from index expressions to types, determined by the following rules. It is the application of these rules that generates constraints during type-checking.

$$\frac{\phi \models i = j}{\phi \models \delta(i) \equiv \delta(j)} \qquad \frac{\phi \models \tau_1' \equiv \tau_1 \quad \phi \models \tau_2 \equiv \tau_2'}{\phi \models \tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'}$$

$$\frac{\phi, \vec{a} : \vec{\gamma} \models \tau \equiv \tau'}{\phi \models \Pi\vec{a} : \vec{\gamma}.\tau \equiv \Pi\vec{a} : \vec{\gamma}.\tau'} \qquad \frac{\phi, a : \gamma \models \tau \equiv \tau'}{\phi \models \Sigma a : \gamma.\tau \equiv \Sigma a : \gamma.\tau'}$$

We present the typing rules for $\mathrm{ML}_0^{\Pi,\Sigma}$ in Figure 3. Some of these rules have obvious side conditions, which are omitted. For instance, in the rule **(type-ilam)**, $\vec{a}$ cannot have free occurrences in Γ. The following lemma plays a pivotal rôle in proving the subject reduction theorem for $\mathrm{ML}_0^{\Pi,\Sigma}$, whose standard proof is available in [14].

Lemma 2.1 *Assume $\phi, \phi'; \Gamma, \Gamma' \vdash e : \tau$ is derivable and $\phi; \Gamma \vdash (\theta_I; \theta) : (\phi'; \Gamma')$ holds. Then we can derive $\phi; \Gamma \vdash e[\theta_I][\theta] : \tau[\theta_I]$.*

2.3 Dynamic Semantics

We present the dynamic semantics of $\mathrm{ML}_0^{\Pi,\Sigma}$ through the use of evaluation contexts defined below. Certainly, there are other possibilities for this purpose, which we do not explore here. [3]

$$\begin{array}{ll}\text{evaluation contexts } E \ ::= & \\ \quad [] \mid \mathbf{if}(E, e_1, e_2) \mid E[\vec{\imath}] \mid E(e) \mid v(E) \mid & \\ \quad \langle i \mid E \rangle \mid \mathbf{open}\ E\ \mathbf{as}\ \langle a \mid x \rangle\ \mathbf{in}\ e & \end{array}$$

We write $E[e]$ for the expression resulting from replacing the hole $[]$ in E with e. Note that this replacement can *never* result in capturing free variables.

Definition 2.2 *A redex is defined below.*

- $\mathbf{if}(c, e_1, e_2)$ *are redexes for* $c = \mathit{true}, \mathit{false}$, *which reduce to* e_1 *and* e_2, *respectively.*
- $(\mathbf{lam}\ x : \tau.e)(v)$ *is a redex, which reduces to* $e[x \mapsto v]$.
- *Let* e *be* $\mathbf{fun}\ f[\vec{a} : \vec{\gamma}] : \tau\ \mathbf{is}\ v$ *Then* e *is a redex, which reduces to* $\lambda\vec{a} : \vec{\gamma}.v[f \mapsto e]$.
- $(\lambda\vec{a} : \vec{\gamma}.v)[\vec{\imath}]$ *is a redex, which reduces to* $v[\vec{a} \mapsto \vec{\imath}]$.
- $\mathbf{open}\ \langle i \mid v \rangle\ \mathbf{as}\ \langle a \mid x \rangle\ \mathbf{in}\ e$ *is a redex, which reduces to* $e[a \mapsto i][x \mapsto v]$.

We use r *for a redex and write* $r \hookrightarrow e$ *if* r *reduces to* e. *If* $e_1 = E[r]$, $e_2 = E[e]$ *and* $r \hookrightarrow e$, *we write* $e_1 \hookrightarrow e_2$ *and say* e_1 *reduces to* e_2 *in one step.*

Let $\hookrightarrow^*$ be the reflexive and transitive closure of $\hookrightarrow$. We say e_1 reduces to e_2 (in many steps) if $e_1 \hookrightarrow^* e_2$. We omit the standard proof for the following subject reduction theorem, which uses Lemma 2.1.

Theorem 2.3 *(Subject Reduction) Assume* $\cdot; \cdot \vdash e : \tau$ *is derivable in* $\mathrm{ML}_0^{\Pi,\Sigma}$. *If* $e \hookrightarrow^* e'$, *then* $\cdot; \cdot \vdash e' : \tau$ *is also derivable in* $\mathrm{ML}_0^{\Pi,\Sigma}$.

2.4 Erasure

We can simply transform $\mathrm{ML}_0^{\Pi,\Sigma}$ into a language ML_0 by erasing all syntax related to type index expressions in $\mathrm{ML}_0^{\Pi,\Sigma}$. Then ML_0 basically extends simply typed λ-calculus with recursion. Let $|e|$ be the erasure of expression e. We have e_1 reducing to e_2 in $\mathrm{ML}_0^{\Pi,\Sigma}$ implies $|e_1|$ reducing to $|e_2|$ in ML_0. Therefore, if e is terminating in $\mathrm{ML}_0^{\Pi,\Sigma}$ then $|e|$ is terminating in ML_0. This is a crucial point since the evaluation of a program in $\mathrm{ML}_0^{\Pi,\Sigma}$ is (most likely) done through the evaluation of its erasure in ML_0. Please find more details on this issue in [18, 14].

3 $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$

We combine metrics with the dependent types in $\mathrm{ML}_0^{\Pi,\Sigma}$, forming a language $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$. We then prove that every well-typed program in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ is terminating, which is the main technical contribution of the paper.

3.1 Metrics

We use $\leq$ for the usual lexicographic ordering on tuples of natural numbers and $<$ for the strict part of $\leq$. Given two tuples of natural numbers $\langle i_1, \ldots, i_n \rangle$ and $\langle i'_1, \ldots, i'_{n'} \rangle$, $\langle i_1, \ldots, i_n \rangle < \langle i'_1, \ldots, i'_{n'} \rangle$ holds if $n = n'$ and for some $0 \leq k \leq n$, $i_j = i'_j$ for $j = 1, \ldots, k-1$ and $i_k < i'_k$. Evidently, $<$ is a well-founded. We stress that (in theory) there is no difficulty supporting various other well-founded orderings on natural numbers such as the usual multiset ordering. We fix an ordering solely for easing the presentation.

Definition 3.1 *(Metric) Let* $\mu = \langle i_1, \ldots, i_n \rangle$ *be a tuple of index expressions and* ϕ *be an index variable context. We say* μ *is a metric under* ϕ *if* $\phi \vdash i_j : \mathit{nat}$ *are derivable for* $j = 1, \ldots, n$. *We write* $\phi \vdash \mu : \mathit{metric}$ *to mean* μ *is a metric under* ϕ.

A decorated type in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ is of form $\Pi\vec{a} : \vec{\gamma}.\mu \Rightarrow \tau$, and the following rule is for forming such types.

$$\frac{\phi, \vec{a} : \vec{\gamma} \vdash \tau : * \qquad \phi, \vec{a} : \vec{\gamma} \vdash \mu : \mathit{metric}}{\phi \vdash \Pi\vec{a} : \vec{\gamma}.\mu \Rightarrow \tau : *}$$

The syntax of $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ is the same as that of $\mathrm{ML}_0^{\Pi,\Sigma}$ except that a context Γ in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ maps every ρ-variable f in its domain to a decorated type and a recursive function in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ is of form $\mathbf{fun}\ f[\vec{a} : \vec{\gamma}] : \mu \Rightarrow \tau\ \mathbf{is}\ v$. The process of translating a source program into an expression in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ is what we call *elaboration*, which is thoroughly explained in [18, 14]. Our approach to program termination verification is to be applied to elaborated programs.

3.2 Dynamic and Static Semantics

The dynamic semantics of $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ is formed in precisely the same manner as that of $\mathrm{ML}_0^{\Pi,\Sigma}$ and we thus omit all the details.

The difference between $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ and $\mathrm{ML}_0^{\Pi,\Sigma}$ lies in static semantics. There are two kinds of typing judgments in $\mathrm{ML}_0^{\Pi,\Sigma}$, which are of forms $\phi; \Gamma \vdash e : \tau$ and $\phi; \Gamma \vdash e : \tau \ll_f \mu_0$. We call the latter a metric typing judgment, for which we give some explanation. Suppose $\phi; \Gamma \vdash e : \tau \ll_f \mu_0$ and $\Gamma(f) = \Pi\vec{a} : \vec{\gamma}.\mu \Rightarrow \tau$; roughly speaking, for each free occurrence of f in e, f is followed by a sequence of index expressions $[\vec{\imath}]$ such that $\mu[\vec{a} \mapsto \vec{\imath}]$, which we call the label of this occurrence of f, is less than μ_0 under ϕ. Now suppose we have a well-typed closed recursive function

[3] For instance, it is suggested that one present the dynamic semantics in the style of natural semantics and then later form the notion of reducibility for evaluation rules.

$e = \textbf{fun } f[\vec{a} : \vec{\gamma}] : \mu \Rightarrow \tau \textbf{ is } v$ in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ and $\vec{\imath}$ are of sorts $\vec{\gamma}$; then $f[\vec{\imath}][f \mapsto e] = e[\vec{\imath}] \hookrightarrow^* v[\vec{a} \mapsto \vec{\imath}][f \mapsto e]$ holds; by the rule **(type-fun)**, we know that all labels of f in v are less than $\mu[\vec{a} \mapsto \vec{\imath}]$, which is the label of f in $f[\vec{\imath}]$; since labels cannot decrease forever, this yields some basic intuition on why all recursive functions in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ are terminating. However, this intuitive argument is difficult to be formalized directly in the presence of high-order functions.

The typing rules in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ for a judgment of form $\phi; \Gamma \vdash e : \tau$ are essentially the same as those in $\mathrm{ML}_{0}^{\Pi,\Sigma}$ except the following ones.

$$\frac{\Gamma(f) = \Pi\vec{a} : \vec{\gamma}.\mu \Rightarrow \tau}{\phi; \Gamma \vdash f : \Pi\vec{a} : \vec{\gamma}.\tau} \textbf{ (type-}\rho\textbf{-var)}$$

$$\frac{\phi, \vec{a} : \vec{\gamma}; \Gamma, f : \Pi\vec{a} : \vec{\gamma}.\mu \Rightarrow \tau \vdash v : \tau \ll_f \mu}{\phi; \Gamma \vdash \textbf{fun } f[\vec{a} : \vec{\gamma}] : \mu \Rightarrow \tau \textbf{ is } v : \Pi\vec{a} : \vec{\gamma}.\tau} \textbf{ (type-fun)}$$

We present the rules for deriving metric typing judgments in Figure 4. Given $\mu = \langle i_1, \ldots, i_n \rangle$ and $\mu' = \langle i'_1, \ldots, i'_n \rangle$, $\phi \models \mu < \mu'$ means that for some $1 \leq k < n$, $\phi, i_1 = i'_1, \ldots, i_{j-1} = i'_{j-1} \models i_j \leq i'_j$ are satisfied for all $1 \leq j < k$ and $\phi, i_1 = i'_1, \ldots, i_{k-1} = i'_{k-1} \models i_k < i'_k$ is also satisfied.

Lemma 3.2 *We have the following.*

1. *Assume $\phi, \phi'; \Gamma, \Gamma' \vdash e : \tau$ is derivable and $\phi; \Gamma \vdash (\theta_I; \theta) : (\phi'; \Gamma')$ holds. Then we can derive $\phi; \Gamma \vdash e[\theta_I][\theta] : \tau[\theta_I]$.*
2. *Assume $\phi, \phi'; \Gamma, \Gamma' \vdash e : \tau \ll_f \mu$ is derivable and $\phi; \Gamma \vdash (\theta_I; \theta) : (\phi'; \Gamma')$ holds and $f \in \mathbf{dom}(\Gamma)$. Then we can derive $\phi; \Gamma \vdash e[\theta_I][\theta] : \tau[\theta_I] \ll_f \mu[\theta_I]$.*

Proof (1) and (2) are proven simultaneously by structural induction on derivations of $\phi, \phi'; \Gamma, \Gamma' \vdash e : \tau$ and $\phi, \phi'; \Gamma, \Gamma' \vdash e : \tau \ll_f \mu$, respectively. ■

Theorem 3.3 *(Subject Reduction) Assume $\cdot; \cdot \vdash e : \tau$ is derivable in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$. If $e \hookrightarrow^* e'$, then $\cdot; \cdot \vdash e' : \tau$ is also derivable in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$.*

Obviously, we have the following.

Proposition 3.4 *Assume that $\mathcal{D}$ is a derivation $\phi; \Gamma \vdash e : \tau \ll_f \mu_0$. Then then there is a derivation of $\phi; \Gamma \vdash e : \tau$ with the same height[4] as $\mathcal{D}$.*

3.3 Reducibility

We define the notion of reducibility for well-typed closed expressions.

Definition 3.5 *(Reducibility) Suppose that e is a closed expression of type τ and $e \hookrightarrow^* v$ holds for some value v. The reducibility of e is defined by induction on the complexity of τ.*

1. τ is a base type. Then e is reducible.
2. $\tau = \tau_1 \to \tau_2$. Then e is reducible if $e(v_1)$ are reducible for all reducible values v_1 of type τ.
3. $\tau = \Pi\vec{a} : \vec{\gamma}.\tau_1$. Then e is reducible if $e[\vec{\imath}]$ are reducible for all $\vec{\imath} : \vec{\gamma}$.
4. $\tau = \Sigma a : \gamma.\tau_1$. Then e is reducible if $v = \langle i \mid v_1 \rangle$ for some i and v_1 such that v_1 is a reducible value of type $\tau_1[a \mapsto i]$.

Note that reducibility is *only* defined for closed expressions that reduce to values.

Proposition 3.6 *Assume that e is a closed expression of type τ and $e \hookrightarrow e'$ holds. Then e is reducible if and only if e' is reducible.*

Proof By induction on the complexity of τ. ■

The following is a key notion for handling recursion, which, though natural, requires some technical insights.

Definition 3.7 *(μ-Reducibility). Let e be a well-typed closed recursive function $\textbf{fun } f[\vec{a} : \vec{\gamma}] : \mu \Rightarrow \tau \textbf{ is } v$ and μ_0 be a closed metric. e is μ_0-reducible if $e[\vec{\imath}]$ are reducible for all $\vec{\imath} : \vec{\gamma}$ satisfying $\mu[\vec{a} \mapsto \vec{\imath}] < \mu_0$.*

Definition 3.8 *Let θ be a substitution that maps variables to expressions; for every $x \in \mathbf{dom}(\theta)$, θ is x-reducible if $\theta(x)$ is reducible; for every $f \in \mathbf{dom}(\theta)$, θ is (f, μ_f)-reducible if $\theta(f)$ is μ_f-reducible.*

In some sense, the following lemma verifies whether the notion of reducibility is formed correctly, where the difficulty probably lies in its formulation rather than in its proof.

Lemma 3.9 *(Main Lemma) Assume that $\phi; \Gamma \vdash e : \tau$ and $\cdot \vdash (\theta_I; \theta) : (\phi; \Gamma)$ are derivable. Also assume that θ is x-reducible for every $x \in \mathbf{dom}(\Gamma)$ and for every $f \in \mathbf{dom}(\Gamma)$, $\cdot; \Gamma[\theta_I] \vdash e[\theta_I] : \tau[\theta_I] \ll_f \mu_f$ is derivable and θ is (f, μ_f)-reducible. Then $e[\theta_I][\theta]$ is reducible.*

Proof Let $\mathcal{D}$ be a derivation of $\phi; \Gamma \vdash e : \tau$ and we proceed by induction on the height of $\mathcal{D}$. We present the most interesting case below. All other cases can be found in [16]. Assume that the following rule **(type-fun)** is last applied in $\mathcal{D}$,

$$\frac{\phi, \vec{a}_1 : \vec{\gamma}_1; \Gamma, f_1 : \Pi\vec{a}_1 : \vec{\gamma}_1.\mu_1 \Rightarrow \tau_1 \vdash v_1 : \tau_1 \ll_{f_1} \mu_1}{\phi; \Gamma \vdash \textbf{fun } f_1[\vec{a_1} : \vec{\gamma}_1] : \mu_1 \Rightarrow \tau_1 \textbf{ is } v_1 : \Pi\vec{a}_1 : \vec{\gamma}_1.\tau_1}$$

where we have $e = \textbf{fun } f_1[\vec{a}_1 : \vec{\gamma}_1] : \mu_1 \Rightarrow \tau_1 \textbf{ is } v_1$ and $\tau = \Pi\vec{a}_1 : \vec{\gamma}_1.\tau_1$. Suppose that $e^* = e[\theta_I][\theta]$ is not reducible. Then by definition there exist $\vec{\imath}_0 : \vec{\gamma}_1^*$ such that $e^*[\vec{\imath}_0]$ is not reducible but $e^*[\vec{\imath}]$ are reducible for all $\vec{\imath} : \vec{\gamma}_1^*$ satisfying $\mu_1^*[\vec{a}_1 \mapsto \vec{\imath}] < \mu_1^*[\vec{a}_1 \mapsto \vec{\imath}_0]$, where $\vec{\gamma}_1^* = \vec{\gamma}_1[\theta_I]$ and $\mu_1^* = \mu_1[\theta_I]$. In other words, e^* is μ_{f_1}-reducible for $\mu_{f_1} = \mu_1^*[\vec{a}_1 \mapsto \vec{\imath}_0]$. Note that we can derive $\cdot; \Gamma[\theta_I], f_1 : \Pi\vec{a}_1 : \vec{\gamma}_1^*.\tau_1[\theta_I] \vdash v_1[\theta_I[\vec{a}_1 \mapsto \vec{\imath}_0]] : \tau_1[\theta_I[\vec{a}_1 \mapsto$

[4] For a minor technicality reason, we count neither of the rules **(type-ρ-var)** and **($\ll$-ρ-var)** when calculating the height of a derivation.

$$\frac{\Gamma(x) = \tau}{\phi; \Gamma \vdash x : \tau \ll_f \mu_0}\ (\ll\textbf{-}\lambda\textbf{-var}) \qquad \frac{\Gamma(f_1) = \tau \quad f_1 \neq f}{\phi; \Gamma \vdash f_1 : \tau \ll_f \mu_0}\ (\ll\textbf{-}\rho\textbf{-var})$$

$$\frac{\phi; \Gamma \vdash e : \mathtt{bool}(i) \ll_f \mu_0 \quad \phi, i = 1; \Gamma \vdash e_1 : \tau \ll_f \mu_0 \quad \phi, i = 0; \Gamma \vdash e_2 : \tau \ll_f \mu_0}{\phi; \Gamma \vdash \mathbf{if}(e, e_1, e_2) : \tau \ll_f \mu_0}\ (\ll\textbf{-if})$$

$$\frac{\phi, \vec{a} : \vec{\gamma}; \Gamma \vdash v : \tau \ll_f \mu_0}{\phi; \Gamma \vdash \lambda\vec{a} : \vec{\gamma}.v : \Pi\vec{a} : \vec{\gamma}.\tau \ll_f \mu_0}\ (\ll\textbf{-ilam}) \qquad \frac{\phi; \Gamma \vdash e : \Pi\vec{a} : \vec{\gamma}.\tau \ll_f \mu_0 \quad \phi \vdash \vec{\imath} : \vec{\gamma}}{\phi; \Gamma \vdash e[\vec{\imath}] : \tau[\vec{a} \mapsto \vec{\imath}] \ll_f \mu_0}\ (\ll\textbf{-iapp})$$

$$\frac{\phi; \Gamma, x : \tau_1 \vdash e : \tau_2 \ll_f \mu_0}{\phi; \Gamma \vdash \mathbf{lam}\, x : \tau_1.e : \tau_1 \to \tau_2 \ll_f \mu_0}\ (\ll\textbf{-lam}) \qquad \frac{\phi; \Gamma \vdash e_1 : \tau_1 \to \tau_2 \ll_f \mu_0 \quad \phi; \Gamma \vdash e_2 : \tau_1 \ll_f \mu_0}{\phi; \Gamma \vdash e_1(e_2) : \tau_2 \ll_f \mu_0}\ (\ll\textbf{-app})$$

$$\frac{\begin{array}{c}\phi, \vec{a}_1 : \vec{\gamma}_1; \Gamma, f_1 : \Pi\vec{a}_1 : \mu_1 \Rightarrow \vec{\gamma}_1.\tau_1 \vdash v_1 : \tau_1 \ll_{f_1} \mu_1 \\ \phi, \vec{a}_1 : \vec{\gamma}_1; \Gamma, f_1 : \Pi\vec{a}_1 : \vec{\gamma}_1.\tau_1 \vdash e_1 : \tau_1 \ll_f \mu_0\end{array}}{\phi; \Gamma \vdash \mathbf{fun}\ f_1[\vec{a}_1 : \vec{\gamma}_1] : \mu_1 \Rightarrow \tau_1\ \mathbf{is}\ v_1 : \Pi\vec{a}_1 : \vec{\gamma}_1.\tau_1 \ll_f \mu_0}\ (\ll\textbf{-fun})$$

$$\frac{\phi \vdash \vec{\imath} : \vec{\gamma} \quad \phi \models \mu[\vec{a} \mapsto \vec{\imath}] < \mu_0 \quad \Gamma(f) = \Pi\vec{a} : \vec{\gamma}.\mu \Rightarrow \tau}{\phi; \Gamma \vdash f[\vec{\imath}] : \tau[\vec{a} \mapsto \vec{\imath}] \ll_f \mu_0}\ (\ll\textbf{-lab})$$

$$\frac{\phi \vdash i : \gamma \quad \phi; \Gamma \vdash e : \tau[a \mapsto i] \ll_f \mu_0}{\phi; \Gamma \vdash \langle i \mid e \rangle : \Sigma a : \gamma.\tau \ll_f \mu_0}\ (\ll\textbf{-pack})$$

$$\frac{\phi; \Gamma \vdash e_1 : \Sigma a : \gamma.\tau_1 \ll_f \mu_0 \quad \phi, a : \gamma; \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \ll_f \mu_0}{\phi; \Gamma \vdash \mathbf{open}\ e_1\ \mathbf{as}\ \langle a \mid x \rangle\ \mathbf{in}\ e_2 : \tau_2 \ll_f \mu_0}\ (\ll\textbf{-open})$$

Figure 4. Metric Typing Rules for $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$

$\vec{\imath}_0]] \ll_f \mu_{f_1}$. By Proposition 3.4, there is a derivation $\mathcal{D}_1$ of $\phi, \vec{a}_1 : \vec{\gamma}_1; \Gamma, f_1 : \Pi\vec{a}_1 : \vec{\gamma}_1.\mu_1 \Rightarrow \tau_1 \vdash v_1 : \tau_1$ such that the height of $\mathcal{D}_1$ is less than that of $\mathcal{D}$. By induction hypothesis, we have that $v_1^* = v_1[\theta_I[\vec{a}_1 \mapsto \vec{\imath}_0]][\theta[f_1 \mapsto e^*]]$ is reducible. Note that $e^*[\vec{\imath}_0] \hookrightarrow^* v_1^*$ and thus $e^*[\vec{\imath}_0]$ is reducible, contradicting the definition of $\vec{\imath}_0$. Therefore, e^* is reducible. ■

The following is the main result of the paper.

Corollary 3.10 *If $\cdot; \cdot \vdash e : \tau$ is derivable in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$, then e in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ is reducible and thus reduces to a value.*

Proof The corollary follows from Lemma 3.9. ■

4 Extensions

We can extend $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ with some significant programming features such as mutual recursion, datatypes and polymorphism, defining the notion of reducibility for each extension and thus making it clear that Lemma 3.9 still holds after the extension. We present in this section the treatment of mutual recursion and currying, leaving the details in [16].

4.1 Mutual Recursion

The treatment of mutual recursion is slightly different from the standard one. The syntax and typing rules for handling mutual recursion are given in Figure 5. We use $(\tau_1, \ldots, \tau_n)$ for the type of an expression representing n mutually recursive functions of types $\tau_1, \ldots, \tau_n$, respectively, which should not be confused with the product of types $\tau_1, \ldots, \tau_n$. Also, the n in $e.n$ must be a positive (constant) integer. Let v be the following expression.

$$\mathbf{funs}\ f_1[\vec{a}_1 : \vec{\gamma}_1] : \tau_1\ \mathbf{is}\ v_1\ \mathbf{and} \ldots \mathbf{and}\ f_n[\vec{a}_n : \vec{\gamma}_n] : \tau_n\ \mathbf{is}\ v_n$$

Then for every $1 \leq k \leq n$, $v.k$ is a redex, which reduces to $\lambda\vec{a}_k : \vec{\gamma}_k.v_k[f_1 \mapsto v.1, \ldots, f_n \mapsto v.n]$. Let $\vec{f} = f_1, \ldots, f_n$ and we form a metric typing judgment $\phi; \Gamma \vdash e \ll_{\vec{f}} \mu_0$ for verifying that all labels of $f_1, \ldots, f_n$ in e are less than μ_0 under ϕ. The rules for deriving such a judgment are essentially the same as those in Figure 4 except **(≪-lab)**, which is given below.

$$\frac{f \text{ in } \vec{f} \quad \Gamma(f) = \Pi\vec{a} : \vec{\gamma}.\mu \Rightarrow \tau \quad \phi \models \mu[\vec{a} \mapsto \vec{\imath}] < \mu_0}{\phi; \Gamma \vdash f[\vec{\imath}] : \tau[\vec{a} \mapsto \vec{\imath}] \ll_{\vec{f}} \mu_0}$$

The rule **(≪-funs)** for handling mutual recursion is straightforward and thus omitted.

Definition 4.1 *(Reducibility) Let e be a closed expression of type $(\tau_1, \ldots, \tau_n)$ and e reduces to v. e is reducible if $e.k$ are reducible for $k = 1, \ldots, n$.*

4.2 Currying

A decorated type must so far be of form $\Pi\vec{a} : \vec{\gamma}.\mu \Rightarrow \tau$ and this restriction has a rather unpleasant consequence. For

$$\begin{array}{llll} \text{types} & \tau & ::= & \cdots \mid (\Pi\vec{a}_1 : \vec{\gamma}_1.\tau_1, \ldots, \Pi\vec{a}_n : \vec{\gamma}_n.\tau_n) \\ \text{expressions} & e & ::= & \cdots \mid e.n \mid \textbf{funs } f_1[\vec{a}_1 : \vec{\gamma}_1] : \tau_1 \textbf{ is } v_1 \textbf{ and} \ldots \textbf{and } f_n[\vec{a}_n : \vec{\gamma}_n] : \tau_n \textbf{ is } v_n \\ \text{values} & v & ::= & \cdots \mid \textbf{funs } f_1[\vec{a}_1 : \vec{\gamma}_1] : \tau_1 \textbf{ is } v_1 \textbf{ and} \ldots \textbf{and } f_n[\vec{a}_n : \vec{\gamma}_n] : \tau_n \textbf{ is } v_n \end{array}$$

$$\frac{\begin{array}{c} \vec{f} = f_1, \ldots, f_n \qquad \tau = (\Pi\vec{a}_1 : \vec{\gamma}_1.\tau_1, \ldots, \Pi\vec{a}_n : \vec{\gamma}_n.\tau_n) \\ \phi, \vec{a}_1 : \vec{\gamma}_1; \Gamma, f_1 : \Pi\vec{a}_1 : \vec{\gamma}_1 : \mu_1 \Rightarrow \tau_1, \ldots, f_n : \Pi\vec{a}_n : \vec{\gamma}_n : \mu_n \Rightarrow \tau_n \vdash v_1 : \tau_1 \ll_{\vec{f}} \mu_1 \\ \cdots \quad \cdots \\ \phi, \vec{a}_n : \vec{\gamma}_n; \Gamma, f_1 : \Pi\vec{a}_1 : \vec{\gamma}_1 : \mu_1 \Rightarrow \tau_1, \ldots, f_n : \Pi\vec{a}_n : \vec{\gamma}_n : \mu_n \Rightarrow \tau_n \vdash v_n : \tau_n \ll_{\vec{f}} \mu_n \end{array}}{\phi; \Gamma \vdash \textbf{funs } f_1[\vec{a}_1 : \vec{\gamma}_1] : \mu_1 \Rightarrow \tau_1 \textbf{ is } v_1 \textbf{ and} \ldots \textbf{and } f_n[\vec{a}_n : \vec{\gamma}_n] : \mu_n \Rightarrow \tau_n \textbf{ is } v_n : \tau} \textbf{(type-funs)}$$

$$\frac{\phi; \Gamma \vdash e : (\tau_1, \ldots, \tau_n) \quad 1 \leq k \leq n}{\phi; \Gamma \vdash e.k : \tau_k} \textbf{(type-choose)}$$

Figure 5. The Syntax and Typing Rules for Mutual Recursion

instance, we may want to assign the following type τ to the implementation of Ackerman function in Figure 1:

```
{i:nat} int(i) -> {j:nat} int(j) -> int,
```

which is formally written as

$$\Pi a_1 : nat.\mathtt{int}(a_1) \to \Pi a_2 : nat.\mathtt{int}(a_2) \to \Sigma a : nat.\mathtt{int}(a).$$

If we decorate τ with a metric μ, then μ can only involve the index variable a_1, making it impossible to verify that the implementation is terminating.

We generalize the form of decorated types to the following so as to address the problem.

$$\Pi\vec{a}_1 : \vec{\gamma}_1.\tau_1 \to \cdots \to \Pi\vec{a}_n : \vec{\gamma}_n.\tau_n \to \Pi\vec{a} : \vec{\gamma}.\mu \Rightarrow \tau.$$

Also, we introduce the following form of expression e for representing a recursive function.

$$\textbf{fun } f[\vec{a}_1 : \vec{\gamma}_1](x_1 : \tau_1) \cdots [\vec{a}_n : \vec{\gamma}_n](x_n : \tau_n)[\vec{a} : \vec{\gamma}] : \tau \textbf{ is } e_0$$

We require that e_0 be a value if $n = 0$. In the following, we only deal with the case $n = 1$. For $n > 1$, the treatment is similar. For $e = \textbf{fun } f[\vec{a}_1 : \vec{\gamma}_1](x_1 : \tau_1)[\vec{a} : \vec{\gamma}] : \tau \textbf{ is } e_0$, we have $e \hookrightarrow \lambda\vec{a}_1 : \vec{\gamma}_1.\textbf{lam } x_1 : \tau_1.\lambda\vec{a} : \vec{\gamma}.e_0$ and the following typing rule

$$\frac{\phi, \vec{a}_1 : \vec{\gamma}_1, \vec{a} : \vec{\gamma}; \Gamma, f : \tau_0, x_1 : \tau_1 \vdash e : \tau \ll_f \mu}{\phi; \Gamma \vdash \textbf{fun } f[\vec{a}_1 : \vec{\gamma}_1](x_1 : \tau_1)[\vec{a} : \vec{\gamma}] : \mu \Rightarrow \tau \textbf{ is } e : \tau_0}$$

where $\tau_0 = \Pi\vec{a}_1 : \vec{\gamma}_1.\tau_1 \to \Pi\vec{a} : \vec{\gamma}.\tau$, and the following metric typing rule

$$\frac{\begin{array}{c} \phi \models \vec{\imath}_1 : \vec{\gamma}_1 \quad \phi \models \vec{\imath} : \vec{\gamma}[\vec{a}_1 \mapsto \vec{\imath}_1] \\ \phi \models \mu[\vec{\imath}_1 \mapsto \vec{a}_1][\vec{a} \mapsto \vec{\imath}] < \mu_0 \\ \phi; \Gamma \vdash e_1 : \tau_1[\vec{a}_1 \mapsto \vec{\imath}_1] \ll_f \mu_0 \\ \Gamma(f) = \Pi\vec{a}_1 : \vec{\gamma}_1.\tau_1 \to \Pi\vec{a} : \vec{\gamma}.\mu \Rightarrow \tau \end{array}}{\phi; \Gamma \vdash f[\vec{\imath}_1](e_1)[\vec{\imath}] : \tau[\vec{a}_1 \mapsto \vec{\imath}_1][\vec{a} \mapsto \vec{\imath}] \ll_f \mu_0}$$

Definition 4.2 *(μ-reducibility) Let e be a closed recursive function* **fun** $f[\vec{a_1} : \vec{\gamma_1}](x_1 : \tau_1)[\vec{a} : \vec{\gamma}] : \tau$ **is** *e and μ_0 be a closed metric. e is μ_0-reducible if $e[\vec{\imath}_1](v)[\vec{\imath}]$ are reducible for all reducible values $v : \tau_1[\vec{a}_1 \mapsto \vec{\imath}_1]$ and $\vec{\imath}_1 : \vec{\gamma}_1$ and $\vec{\imath} : \vec{\gamma}[\vec{a}_1 \mapsto \vec{\imath}_1]$ satisfying $\mu[\vec{a}_1 \mapsto \vec{\imath}_1][\vec{a} \mapsto \vec{\imath}] < \mu_0$.*

5 Practice

We have implemented a type-checker for $\mathrm{ML}^{\Pi,\Sigma}_{0,\ll}$ in a prototype implementation of DML and experimented with various examples, some of which are presented below. We also address the practicality issue at the end of this section.

5.1 Examples

We demonstrate how various programming features are handled in practice by our approach to program termination verification.

Primitive Recursion The following is an implementation of the primitive recursion operator R in Gödel's $\mathcal{T}$, which is clearly typable in $\mathrm{ML}^{\Pi,\Sigma}_{0,\ll}$. Note that Z and S are assigned the types $Nat(0)$ and $\Pi n : nat.Nat(n) \to Nat(n+1)$, respectively.

```
datatype Nat with nat =
  Z(0) | {n:nat} S(n+1) of Nat(n)

fun('a) R Z u v =
  u | R (S n) u v = v n (R n u v)
withtype
 {n:nat} <n> =>
 Nat(n) -> 'a -> (Nat -> 'a -> 'a) -> 'a
(* Nat is for [n:nat] Nat(n) in a type *)
```

By Corollary 3.10, it is clear that every term in $\mathcal{T}$ is terminating (or weakly normalizing). This is the only example in this paper that can be proven terminating with a structural ordering. The point we make is that though it seems "evident" that the use of R cannot cause non-termination, it is not trivial at all to prove every term in $\mathcal{T}$ is terminating. Notice that such a proof cannot be obtained in Peano arithmetic. The notion of reducibility is precisely invented for overcoming the difficulty [12]. Actually, every term in $\mathcal{T}$ is strongly normalizing, but this obviously is untrue in

$\mathrm{ML}^{\Pi,\Sigma}_{0,\ll}$.

Nested Recursive Function Call The program in Figure 6 involving a nested recursive function call implements McCarthy's "91" function. The withtype clause indicates that for every integer x, $f91(x)$ returns integer 91 if $x \leq 100$ and $x - 10$ if $x \geq 101$. We informally explain why the metric in the type annotation suffices to establish the termination of $f91$; for the inner call to $f91$, we need to prove that $\phi \models \max(0, 101 - (i + 11)) < \max(0, 101 - i)$ is satisfied for $\phi = i : int, i \leq 100$, which is obvious; for the outer call to $f91$, we need to verify that $\phi_1 \models \max(0, 101 - j) < \max(0, 101 - i)$, where ϕ_1 is $\phi, j : int, P$ and P is

$$(i + 11 \leq 100 \wedge j = 91) \vee (i + 11 \geq 101 \wedge j = i + 11 - 10)$$

If $i + 11 \leq 100$, then $j = 91$ and $\max(0, 101 - j) = 10 < 12 \leq 101 - i$; if $i + 11 \geq 101$, then $j = i + 11 - 10 = i + 1$ and $\max(0, 101 - j) < 101 - i$ (since $i \leq 100$ is assumed in ϕ). Clearly, this example can not be handled with a structural ordering.

Mutual Recursion The program in Figure 7 implements quicksort on a list, where the functions qs and par are defined mutually recursively. We informally explain why this program is typable in $\mathrm{ML}^{\Pi,\Sigma}_{0,\ll}$ and thus qs is a terminating function by Corollary 3.10.

For the call to par in the body of qs, the label is $(0 + 0 + a, a + 1)$, where a is the length of xs'. So we need to verify that $\phi \models (0 + 0 + a, a + 1) < (n, 0)$ is satisfied for $\phi = n : nat, a : nat, a + 1 = n$, which is obvious.

For the two calls to qs in the body of par, we need to verify that $\phi \models (p, 0) < (p + q + r, r + 1)$ and $\phi \models (q, 0) < (p + q + r, r + 1)$ for $\phi = p : nat, q : nat, r : nat, r = 0$, both of which hold since $\phi \models p \leq p + q$ and $\phi \models q \leq p + q$ and $\phi \models 0 < 1$. This also indicates why we need $r + 1$ instead of r in the metric for par.

For the two calls to par in the body of par, we need to verify that $\phi \models ((p + 1) + q + a, a) < (p + q + r, r)$ and $\phi \models (p + (q + 1) + a, a) < (p + q + r, r)$ for $\phi = p : nat, q : nat, r : nat, a : nat, r = a + 1$, both of which hold since $\phi \models (p + 1) + q + a = p + q + r$ and $\phi \models p + (q + 1) + a = p + q + r$ and $\phi \models a < r$. Clearly, this example can not be handled with a structural ordering.

Higher-order Function The program in Figure 8 implements a function $accept$ that takes a pattern p and a string s and checks whether s matches p, where the meaning of a pattern is given in the comments.

The auxiliary function acc is implemented in continuation passing style, which takes a pattern p, a list of characters cs and a continuation k and matches a prefix of cs against p and call k on the rest of characters. Note that k is given a type that allows k to be applied only to a character list not longer than cs. The metric used for proving the termination of acc is $\langle n, i \rangle$, where n is the size of p, that is the number constructors in p (excluding $Empty$) and i is the length of cs. Notice the call $acc\ p\ cs'\ k$ in the last pattern matching clause; the label attached to this call is $\langle n, i' \rangle$, where i' is the length of cs'; we have $i' \leq i$ since the continuation has the type $\Pi a' : \gamma.(char)list(a') \rightarrow \mathtt{bool}$, where γ is $\{a : nat \mid a \leq i\}$; we have $i \neq i'$ since $length(cs') = length(cs)$ must be false when this call happens; therefore we have $i' < i$ [5] and then $\langle n, i' \rangle < \langle n, i \rangle$. It is straightforward to see that the labels attached to other calls to acc are less than $\langle n, i \rangle$. By Corollary 3.10, acc is terminating, which implies that $accept$ is terminating (assuming $explode$ is terminating). In every aspect, this is a non-trivial example even for interactive theorem proving systems.

Notice that the test $length(cs') = length(cs)$ in the body of acc can be time-consuming. This can be resolved by using a continuation that accepts as its arguments both a character list and its length. In [5], there is an elegant implementation of $accept$ that does some processing on the pattern to be matched and then eliminates the test.

Run-time Check There are also realistic cases where termination depends on a program invariant that cannot (or is difficult to) be captured in the type system of DML. For instance, the following example is adopted from an implementation of bit reversing, which is a part of an implementation of fast Fourier transform (FFT).

```
fun loop (j, k) =
  if (k<j) then loop (j-k, k/2) else j+k
withtype
  {a:nat,b:nat} int(a) * int(b) -> int
```

Obviously, $loop(1, 0)$ is not terminating. However, we may know for some reason that the second argument of $loop$ can never be 0 during execution. This leads to the following implementation, in which we need to check that $k > 1$ holds before calling $loop(j - k, k/2)$ so as to guarantee that $k/2$ is a positive integer.

```
fun loop (j, k) =
  if (k < j) then
    if (k > 1) then loop (j - k, k / 2)
    else raise Impossible
  else j + k
withtype {a:nat,b:pos} <max(0, a-b)> =>
         int(a) * int(b) -> int
```

It can now be readily verified that $loop$ is a terminating function. This example indicates that we can insert run-time checks to verify program termination, sometimes, approximating a liveness property with a safety property.

5.2 Practicality

There are two separate issues concerning the practicality of our approach to program termination verification, which are (a) the practicality of the termination verification process and (b) the applicability of the approach to realistic programs.

[5] Note that $length(cs')$ and $length(cs)$ have the types $\mathtt{int}(i')$ and $\mathtt{int}(i)$, respectively, and thus $length(cs') = length(cs)$ has the type $\mathtt{bool}(i' = i)$, where $i' = i$ equals 1 or 0 depending on whether i' equals i. Thus, $i' < i$ can be inferred *in the type system*.

```
fun f91 (x) = if (x <= 100) then f91 (f91 (x + 11)) else x - 10
withtype
  {i:int} <max(0, 101-i)> =>
  int(i) -> [j:int | (i<=100 /\ j=91) \/ (i>=101 /\ j=i-10)] int(j)
```

Figure 6. An implementation of McCarthy's "91" function

```
fun('a) qs cmp xs =
  case xs of [] => [] | x :: xs' => par cmp (x, [], [], xs')
withtype ('a * 'a -> bool) -> {n:nat} <n,0> => 'a list(n) -> 'a list(n)

and('a) par cmp (x, l, r, xs) =
  case xs of
    [] => qs cmp l @ (x :: qs cmp r)
  | x' :: xs' => if cmp(x', x) then par cmp (x, x' :: l, r, xs')
                 else par cmp (x, l, x' :: r, xs')
withtype ('a * 'a -> bool) -> {p:nat,q:nat,r:nat} <p+q+r,r+1> =>
         'a * 'a list(p) * 'a list(q) * 'a list(r) -> 'a list(p+q+r+1)
```

Figure 7. An implementation of quicksort on a list

It is easy to observe that the complexity of type-checking in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ is basically the same as in $\mathrm{ML}_{0}^{\Pi,\Sigma}$ since the only added work is to verify that metrics (provided by the programmer) are decreasing, which requires solving some extra constraints. The number of extra constraints generated from type-checking a function is proportional to the number of recursive calls in the body of the function and therefore is likely small. Based on our experience with DML, we thus feel that type-checking in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ is suitable for practical use.

As for the applicability of our approach to realistic programs, we use the type system of the programming language C as an example to illustrate a design decision. Obviously, the type system of C is unsound because of (unsafe) type casts, which are often needed in C for typing programs that would otherwise not be possible. In spite of this practice, the type system of C is still of great help for capturing program errors. Clearly, a similar design is to allow the programmer to assert the termination of a function in DML if it cannot be verified, which we may call *termination cast.* Combining termination verification, run-time checks and termination cast, we feel that our approach is promising to be put into practice.

6 Related Work

The amount of research work related to program termination is simply vast. In this section, we mainly mention some related work with which our work shares some similarity either in design or in technique.

Most approaches to automated termination proofs for either programs or term rewriting systems (TRSs) use various heuristics to synthesize well-founded orderings. Such approaches, however, often have difficulty reporting comprehensible information when a program cannot be proven terminating. Following [13], there is also a large amount of work on proving termination of logic programs. In [11], it is reported that the Mercury compiler can perform automated termination checking on realistic logic programs.

However, we address a different question here. We are interested in checking whether a given metric suffices to establish the termination of a program and not in synthesizing such a metric. This design is essentially the same as the one adopted in [10], where it checks whether a given structural ordering (possibly on high-order terms) is decreasing in an inductive proof or a logic program. Clearly, approaches based on checking complements those based on synthesis.

Our approach also relates to the semantic labelling approach [19] designed to prove termination for term rewriting systems (TRSs). The essential idea is to differentiate function calls with labels and show that labels are always decreasing when a function call unfolds. The semantic labelling approach requires constructing a model for a TRS to verify whether labelling is done correctly while our approach does this by type-checking.

The notion of sized types is introduced in [6] for proving the correctness of reactive systems. There, the type system is capable of guaranteeing the termination of well-typed programs. The language presented in [6], which is designed for embedded functional programming, contains a significant restriction as it only supports (a minor variant) of primitive recursion, which can cause inconvenience in programming. For instance, it seems difficult to implement quicksort by using only primitive recursion. From our experience, general recursion is really a major programming feature that greatly complicates program termination verification. Also, the notion of existential dependent types, which we deem indispensable in practical programming, does not exist in [6].

When compared to various (interactive) theorem proving

```
datatype pattern with nat =
  Empty(0) (* empty string matches Empty *)
| Char(1) of char (* "c" matches Char (c) *)
| {i:nat,j:nat} Plus(i+j+1) of pattern(i) * pattern(j)
  (* cs matches Plus(p1, p2) if cs matches either p1 or p2 *)
| {i:nat,j:nat} Times(i+j+1) of pattern(i) * pattern(j)
  (* cs matches Times(p1, p2) if a prefix of cs matches p1 and
     the rest matches p2 *)
| {i:nat} Star(i+1) of pattern(i)
  (* cs matches Star(p) if cs matches some, possibly 0, copies of p *)

(* 'length' computes the length of a list *)
fun('a)
  length (xs) = let
    fun len ([], n) = n
      | len (x :: xs, n) = len (xs, n+1)
    withtype
      {i:nat,j:nat} <i> => 'a list(i) * int(j) -> int(i+j)
   in
      len (xs, 0)
   end
withtype {i:nat} <> => 'a list(i) -> int(i)
(* empty tuple <> is used since 'length' is not recursive *)

fun acc p cs k =
  case p of
    Empty => k (cs)
  | Char(c) =>
    (case cs of
       [] => false
     | c' :: cs' => if (c = c') then k (cs') else false)
  | Plus(p1, p2) => (* in this case, k is used for backtracking *)
    if acc p1 cs k then true else acc p2 cs k
  | Times(p1, p2) => acc p1 cs (fn cs' => acc p2 cs' k)
  | Star(p0) =>
    if k (cs) then true
    else acc p0 cs (fn cs' =>
                      if length(cs') = length(cs) then false
                      else acc p cs' k)
withtype {n:nat} pattern(n) ->
         {i:nat} <n, i> => char list(i) ->
         ({i':nat | i' <= i} char list(i') -> bool) -> bool

(* 'explode' turns a string into a list of characters *)
fun accept p s =
    acc p (explode s) (fn [] => true | _ :: _ => false)
withtype <> => pattern -> string -> bool
```

Figure 8. An implementation of pattern matching on strings

systems such as NuPrl [2], Coq [4], Isabelle [8] and PVS [9], our approach to program termination is weaker (in the sense that [many] fewer programs can be verified terminating) but more automatic and less obtrusive to programming. We have essentially designed a mechanism for program termination verification with a language interface that is to be used during program development cycle. We consider this as the main contribution of the paper. When applied, the designed mechanism intends to facilitate program error detection, leading to the construction of more robust programs.

7 Conclusion and Future Work

We have presented an approach based on dependent types in DML that allows the programmer to supply metrics for verifying program termination and proven its correctness. We have also applied this approach to various examples that involve significant programming features such as a general form of recursion (including mutual recursion), higher-order functions, algebraic datatypes and polymorphism, supporting its usefulness in practice.

A program property is often classified as either a safety property or a liveness property. That a program never performs out-of-bounds array subscripting at run-time is a safety property. It is demonstrated in [17] that dependent types in DML can guarantee that every well-typed program in DML possesses such a safety property, effectively facilitating runtime array bound check elimination. It is, however, unclear (*a priori*) whether dependent types in DML can also be used for establishing liveness properties. In this paper, we have formally addressed the question, demonstrating that dependent types in DML can be combined with metrics to establish program termination, one of the most significant liveness properties.

Termination checking is also useful for compiler optimization. For instance, if one decides to change the execution order of two programs, it may be required to prove that the first program always terminates. Also, it seems feasible to use metrics for estimating the time complexity of programs. In lazy function programming, such information may allow a compiler to decide whether a thunk should be formed. In future, we expect to explore along these lines of research.

Although we have presented many interesting examples that cannot be proven terminating with structural orderings, we emphasize that structural orderings are often effective in practice for establishing program termination. Therefore, it seems fruitful to study a combination of our approach with structural orderings that handles simple cases with either automatically synthesized or manually provided structural orderings and verifies more difficult cases with metrics supplied by the programmer.

References

[1] A. BenCherifa and P. Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *SCP*, 9(2):137–160, 1987.

[2] R. L. Constable et al. *Implementing Mathematics with the NuPrl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[3] N. Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.

[4] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.

[5] R. Harper. Proof-Directed Debugging. *Journal of Functional Programming*, 9(4):471–477, 1999.

[6] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *Conference Record of 23rd ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 410–423, 1996.

[7] J.-P. Jouannaud and A. Rubio. The higher-order recursive path ordering. In *Proceedings of 14th IEEE Symposium on Logic in Computer Science*, pages 402–411, July 1999.

[8] P. Lawrence. *Isabelle: A Generic Theorem Prover.* Springer-Verlag LNCS 828, 1994.

[9] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Proceedings of the 8th International Conference on Computer-Aided Verification, CAV '96*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag LNCS 1102.

[10] B. Pientka and F. Pfenning. Termination and Reduction Checking in the Logical Framework. In *Workshop on Automation of Proofs by Mathematical Induction*, June 2000.

[11] C. Speirs, Z. Somogyi, and H. Søndergarrd. Termination Analysis for Mercury. In *Proceedings of the 4th Static Analysis Symposium*, pages 157–171, September 1997.

[12] W. W. Tait. Intensional Interpretations of Functionals of Finite Type I. *Journal of Symbolic Logic*, 32(2):198–212, June 1967.

[13] J. D. Ullman and A. V. Gelder. Efficient tests for top-down termination of logic rules. *Journal of the ACM*, 35(2):345–373, 1988.

[14] H. Xi. *Dependent Types in Practical Programming.* PhD thesis, Carnegie Mellon University, 1998. pp. viii+189. Available as http://www.cs.cmu.edu/~hwxi/DML/thesis.ps.

[15] H. Xi. Dependently Typed Data Structures. In *Proceedings of Workshop on Algorithmic Aspects of Advanced Programming Languages*, pages 17–33, September 1999.

[16] H. Xi. Dependent Types for Program Termination Verification, July 2000. Available as http://www.ececs.uc.edu/~hwxi/DML/Term.

[17] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, June 1998.

[18] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, January 1999.

[19] H. Zantema. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae*, 24:89–105, 1995.