Higher-Order and Symbolic Computation, 15, 91–131, 2002
© 2002 Kluwer Academic Publishers. Manufactured in The Netherlands.

Dependent Types for Program Termination Verification*

HONGWEI XI hwxi@cs.bu.edu
Computer Science Department, Boston University, Boston, MA 02215, USA

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.

Keywords: termination, dependent types

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 non-terminating 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.

*This work is partially supported by NSF grants no. CCR-0081316 and no. CCR-0092703 and a grant from the Ohio Board of Regents. An extended abstract of the paper has been previously published in the proceedings of the sixteenth IEEE Symposium on Logic in Computer Science (LICS 2001).

In order to design a termination checker for practical use, these issues must be properly addressed.

There is already a rich body of 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 [7], polynomial interpretation [2], etc.). While these approaches are mainly developed for first-order languages, some work in higher-order settings can also be found (e.g., [14]). When a program that should be terminating if implemented correctly, cannot be proved terminating, it is often difficult for the programmer to determine whether this is caused by a program error or by a 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 [5], Coq [8], Isabelle [19] and PVS [18]. 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 too 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 and then automatically verifies that the provided information indeed suffices. An analogy would be like a theorem prover that allows the user to provide induction hypotheses in inductive theorem proving and then proves 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 manner.

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 strictly decreasing when a recursive function call is made. In this paper, we present an approach that uses the dependent types developed in DML [24, 29], where general recursion is allowed, 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. In the current prototype implementation of DML, the termination of

```
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 the Ackermann function.

a function is verified only if the programmer indicates so (with some special syntax). More explanation can be found in Section 5.2. We now illustrate the basic idea with a concrete example before going into further details.

In Figure 1, an implementation of the Ackermann 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 $\{\texttt{i:nat,j:nat}\}$ 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 $\Sigma k : \mathit{nat}.\texttt{int}(k)$, 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 proved in this paper). Note that although this is a simple example, its termination cannot be proved with (lexicographic) structural ordering (as the semantic meaning of both addition $+$ and subtraction $-$ is needed, which is captured by the type system in our case).[1]

More realistic examples are to be presented in Section 5.1, involving dependent datatypes [26], 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 the Ackermann 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 5.1 to understand this point. We seek a method that can be readily adapted to handle various common programming features when they are added, including mutual recursion, datatypes, polymorphism, etc. This naturally leads us to the reducibility method [22]. We are to form a notion of reducibility for the dependent types extended with metrics, in which the novelty lies in combining the usual notion of reducibility with metrics. This formation, which makes it suitable to handle general recursion, 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, whose termination, unfortunately, is not proved 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 of great use for detecting program errors.

The presented work builds upon our previous work on the use of dependent types in practical programming [24, 29]. 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 well-typed 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 5.1, illustrating how our approach to program termination verification can be effectively applied in practice. Lastly, we mention some related works and potential future research questions and then conclude.

2. $\mathbf{ML}_0^{\Pi,\Sigma}$

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.

index constants
$c_I \ ::= \ \cdots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \cdots$

index expressions
$i \ ::= \ a \mid c_I \mid i_1 + i_2 \mid i_1 - i_2 \mid i_1 * i_2 \mid i_1/i_2 \mid i_1 \ mod \ i_2 \mid if(P, i_1, i_2)$

index propositions
$P \ ::= \ i_1 < i_2 \mid i_1 \leq i_2 \mid i_1 > i_2 \mid i_1 \geq i_2 \mid i_1 = i_2 \mid i_1 \neq i_2 \mid$
$\quad P_1 \wedge P_2 \mid P_1 \vee P_2$

index sorts
$\gamma \ ::= \ int \mid \{a : \gamma \mid P\}$

index variable contexts
$\phi \ ::= \ \cdot \mid \phi, a : \gamma \mid \phi, P$

index constraints
$\Phi \ ::= \ P \mid P \supset \Phi \mid \forall a : \gamma.\Phi$

types
$\tau \ ::= \ \delta(\vec{\imath}) \mid \mathbf{1} \mid \tau_1 * \tau_2 \mid \tau_1 \rightarrow \tau_2 \mid \Pi\vec{a} : \vec{\gamma}.\tau \mid \Sigma a : \gamma.\tau$

contexts
$\Gamma \ ::= \ \cdot \mid \Gamma, x : \tau \mid \Gamma, f : \tau$

constants
$c \ ::=$ (some primitive functions) $\mid$
$\quad true \mid false \mid 0 \mid 1 \mid -1 \mid 2 \mid -2 \mid \cdots$

expressions
$e \ ::= \ x \mid f \mid c \mid \mathbf{if}(e, e_1, e_2) \mid \lambda\vec{a} : \vec{\gamma}.v \mid e[\vec{\imath}] \mid$
$\quad \langle\rangle \mid \langle e_1, e_2\rangle \mid \mathbf{fst}(e) \mid \mathbf{snd}(e) \mid \mathbf{lam}\ x : \tau.e \mid e_1(e_2) \mid$
$\quad \mathbf{fun}\ f[\vec{a} : \vec{\gamma}] : \tau\ \mathbf{is}\ v \mid \langle\rangle \mid \langle i \mid e\rangle \mid \mathbf{open}\ e_1\ \mathbf{as}\ \langle a \mid x\rangle\ \mathbf{in}\ e_2$

values
$v \ ::= \ x \mid c[\vec{\imath}] \mid \lambda\vec{a} : \vec{\gamma}.v \mid \langle\rangle \mid \langle v_1, v_2\rangle \mid \mathbf{lam}\ x : \tau.e \mid \langle i \mid v\rangle$

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

2.1. *Syntax*

The syntax for $\mathrm{ML}_0^{\Pi,\Sigma}$ is given in Figure 2. We fix an integer domain and restrict 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* as an abbreviation for the subset sort $\{a : int \mid a \geq 0\}$. We use $if(P, i_1, i_2)$ for an index expression that equals i_1 if P holds, and equals i_2 otherwise. We use $\delta(\vec{\imath})$ for a base type indexed with a sequence of index expressions $\vec{\imath}$, which may be empty. For instance, `bool`(0) and `bool`(1) are types for boolean values *false* and *true*, respectively; for each integer i, `int`(i) is the singleton type for integer expressions with value equal to i.

We use $\phi \models P$ for a satisfaction relation, meaning P holds under ϕ, that is, the formula $(\phi)P$, defined below, holds in the integer domain.

$$(\cdot)\Phi = \Phi$$
$$(\phi, a : int)\Phi = (\phi)\forall a : int.\Phi$$

$$(\phi, \{a : \gamma \mid P\})\Phi = (\phi, a : \gamma)(P \supset \Phi)$$
$$(\phi, P)\Phi = (\phi)(P \supset \Phi)$$

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. Although the syntax in Figure 2 allows non-linear[2] integer constraints, in our implementation, non-linear constraints are immediately rejected. We currently translate the problem of determining whether a given (linear) constraint is satisfiable into some integer programming problem, for which there are various existing methods.

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 $a_1 : \gamma_1, \ldots, a_n : \gamma_n$. Note γ_k may contain free occurrences of a_j for $1 \leq j < k \leq n$. In practice, we also have types of form $\Sigma\vec{a} : \vec{\gamma}.\tau$, which we omit here for simplifying the presentation. In particular, given a type constructor δ that takes index expressions $\vec{\imath}$ of sorts $\vec{\gamma}$ to form a base type $\delta(\vec{\imath})$, we often use δ for $\Sigma\vec{a} : \vec{\gamma}.\delta(\vec{a})$. For instance, `bool` and `int` stand for $\Sigma a : bool.\texttt{bool}(a)$ and $\Sigma a : int.\texttt{int}(a)$, respectively, where the sort $bool = \{a : int \mid 0 \leq a \leq 1\}$.

We also introduce lam-variables and fix-variables in $\mathrm{ML}_0^{\Pi,\Sigma}$ and use x and f for them, respectively. A lambda-abstraction can only be formed over a lam-variable while recursion (via fixed point operator) must be formed over a fix-variable. A lam-variable is a value but a fix-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.

There are two forms of application. We write $e[\vec{\imath}]$ for applying e to a sequence of index expressions and $e_1(e_2)$ for applying e_1 to e_2. The meaning of such forms of application is to be clear after we present the dynamic semantics of $\mathrm{ML}_0^{\Pi,\Sigma}$.

There are various constants in $\mathrm{ML}_0^{\Pi,\Sigma}$, including booleans, integers, and some primitive functions. We use a signature Σ to assign types to constants. In general, we assume that a primitive function has a type of form $\Pi\vec{a} : \vec{\gamma}.\tau_1 \to \tau_2$, where τ_1 and τ_2 are ground types, that is, types that can be formed without using either $\to$ or Π. The types of some constants are given in Figure 3. Note that the type of / indicates that division can only be applied to a pair of integers $\langle x_1, x_2\rangle$ such that x_2 is not 0. This may seem restrictive, but it is not. For instance, we can define the usual division function on integers as follows (after the exception mechanism becomes available),

$$\lambda a_1 : int.\lambda a_2 : int.\mathbf{lam}\ x : \texttt{int}(a_1) * \texttt{int}(a_2).$$
$$\mathbf{if}(=[a_2, 0](\langle \mathbf{snd}(x), 0\rangle), \mathbf{raise}\ \mathit{DivisionByZero}, /[a_1, a_2](x))$$

$$\begin{aligned}
\Sigma(\mathit{true}) &= \mathtt{bool}(1) \\
\Sigma(\mathit{false}) &= \mathtt{bool}(0) \\
\Sigma(n) &= \mathtt{int}(n), \text{ for } n = 0, 1, -1, 2, -2, \ldots \\
\Sigma(+) &= \Pi\{a_1 : \mathit{int}, a_2 : \mathit{int}\}.\mathtt{int}(a_1) * \mathtt{int}(a_2) \to \mathtt{int}(a_1 + a_2) \\
\Sigma(-) &= \Pi\{a_1 : \mathit{int}, a_2 : \mathit{int}\}.\mathtt{int}(a_1) * \mathtt{int}(a_2) \to \mathtt{int}(a_1 - a_2) \\
\Sigma(*) &= \Pi\{a_1 : \mathit{int}, a_2 : \mathit{int}\}.\mathtt{int}(a_1) * \mathtt{int}(a_2) \to \mathtt{int}(a_1 * a_2) \\
\Sigma(/) &= \Pi\{a_1 : \mathit{int}, a_2 : \{a : \mathit{int} \mid a \neq 0\}\}. \\
&\quad\ \mathtt{int}(a_1) * \mathtt{int}(a_2) \to \mathtt{int}(a_1 / a_2) \\
\Sigma(\mathit{mod}) &= \Pi\{a_1 : \mathit{int}, a_2 : \{a : \mathit{int} \mid a \neq 0\}\}. \\
&\quad\ \mathtt{int}(a_1) * \mathtt{int}(a_2) \to \mathtt{int}(a_1\ \mathit{mod}\ a_2) \\
\Sigma(=) &= \Pi\{a_1 : \mathit{int}, a_2 : \mathit{int}\}. \\
&\quad\ \mathtt{int}(a_1) * \mathtt{int}(a_2) \to \mathtt{int}(\mathit{if}(a_1 = a_2, 1, 0)) \\
\Sigma(\neq) &= \Pi\{a_1 : \mathit{int}, a_2 : \mathit{int}\}. \\
&\quad\ \mathtt{int}(a_1) * \mathtt{int}(a_2) \to \mathtt{int}(\mathit{if}(a_1 \neq a_2, 1, 0)) \\
&\ \ \vdots
\end{aligned}$$

Figure 3. The types assigned to some constants in $\mathrm{ML}_0^{\Pi,\Sigma}$.

which raises an exception when the divisor is 0. Notice that we need the following sorting rule for typing $/[a_1, a_2](x)$ in the above expression.

$$\frac{\phi \vdash a_1 : \mathit{int} \quad \phi \vdash a_2 : \mathit{int} \quad \phi \models a_2 \neq 0}{\phi \vdash a_1/a_2 : \mathit{int}}$$

The precise reason why this works can be readily understood once the typing rules for $\mathrm{ML}_0^{\Pi,\Sigma}$ are introduced.

2.2. *Static semantics*

We write $\phi \vdash \tau$ **[well-formed]** to mean that τ is a legally formed type under ϕ. For instance, the following rules are for constructing types of forms $\mathtt{int}(i)$ and $\mathtt{bool}(i)$, respectively.

$$\frac{\phi \vdash i : \mathit{int}}{\phi \vdash \mathtt{int}(i)\ \textbf{[well-formed]}} \qquad \frac{\phi \vdash i : \{a : \mathit{int} \mid 0 \leq a \leq 1\}}{\phi \vdash \mathtt{bool}(i)\ \textbf{[well-formed]}}$$

Notice that the rule for constructing $\mathtt{bool}(i)$ indicates that we need to show that $0 \leq i \leq 1$ holds in order to form such a type. Therefore, $\mathtt{bool}(2)$ is ill-formed. We omit other standard rules for constructing well-formed types.

index substitutions	$\theta_I ::= [] \mid \theta_I[a \mapsto i]$
substitutions	$\theta ::= [] \mid \theta[x \mapsto v] \mid \theta[f \mapsto e]$

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 **dom**(θ_I) for the domain of θ_I. Note that $\theta_I[a \mapsto i]$ extends θ_I, whose domain does not contain a, with a mapping from a to i. Similar notations are used for substitutions on variables. We write $\bullet[\theta_I]$ ($\bullet[\theta]$) for the result of applying θ_I (θ) to $\bullet$, respectively, where $\bullet$ can be a type, a context, an expression, etc. The standard definition is omitted. In this paper, we substitute only values for lam-variables.

We use a judgment of form $\phi \vdash i : \gamma$ to mean that i can be assigned the sort γ under the index variable context ϕ. We omit the standard rules for deriving such judgments. The following rules are for deriving judgments of form $\phi \vdash \theta_I : \phi'$, which roughly means that θ_I has "type" ϕ'.

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

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

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

Lemma 2.1 (*Index substitution*). *If $\phi \vdash \theta_I : \phi'$ holds and $\phi, \phi' \vdash i : \gamma$ is derivable, then $\phi \vdash i[\theta_I] : \gamma[\theta_I]$ is also derivable.*

Proof: This directly follows from a structural induction on the derivation of $\phi, \phi' \vdash i : \gamma$. □

We write $\phi \models \tau \equiv \tau'$ for the congruent extension of $\phi \models i = j$ from index expressions to types, which is determined by the following rules. Note that we write $\phi \models \vec{\imath} = \vec{\imath}'$ to mean $\phi \models i_k = i'_k$ for $k = 1, \ldots, n$, assuming $\vec{\imath} = (i_1, \ldots, i_n)$ and $\vec{\imath}' = (i'_1, \ldots, i'_n)$. The application of these rules generates constraints during type-checking.

$$\frac{\delta \text{ has the kind } \vec{\gamma} \rightarrow * \quad \phi \vdash \vec{\imath} : \vec{\gamma} \quad \phi \vdash \vec{\imath}' : \vec{\gamma} \quad \phi \models \vec{\imath} = \vec{\imath}'}{\phi \models \delta(\vec{\imath}) \equiv \delta(\vec{\imath}')}$$

$$\frac{\phi \models \tau_1 \equiv \tau'_1 \quad \phi \models \tau_2 \equiv \tau'_2}{\phi \models \tau_1 * \tau_2 \equiv \tau'_1 * \tau'_2} \qquad \frac{\phi \models \tau'_1 \equiv \tau_1 \quad \phi \models \tau_2 \equiv \tau'_2}{\phi \models \tau_1 \rightarrow \tau_2 \equiv \tau'_1 \rightarrow \tau'_2}$$

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

We say that δ has the kind $\vec{\gamma} \rightarrow *$ if δ takes index expressions $\vec{\imath}$ of sorts $\vec{\gamma}$ to form a type. There are currently no formal judgments for sort equivalence or subsorting. They are not really needed at this point, since ultimately all sorts only describe various subsets of integers. Some details can be found in [24] as to how an expression of type $\Pi a : \gamma.\tau$ can still be implicitly coerced into one with equivalent dynamic semantics, but of type $\Pi a : \gamma'.\tau$, where γ and γ' are sorts such that $a : \gamma' \vdash a : \gamma$.

As could be expected, we have the following proposition.

Proposition 2.2. *Type conversion is an equivalence relation*:
1. *If* $\phi \vdash \tau$ **[well-formed]** *holds, then* $\phi \models \tau \equiv \tau$ *also holds.*
2. *If* $\phi \models \tau \equiv \tau'$ *holds, then* $\phi \models \tau' \equiv \tau$ *also holds.*
3. *If both* $\phi \models \tau \equiv \tau'$ *and* $\phi \models \tau' \equiv \tau''$ *hold, then* $\phi \models \tau \equiv \tau''$ *also holds.*

We present the typing rules for $\mathrm{ML}_0^{\Pi,\Sigma}$ in Figure 4. We use $\phi; \Gamma \vdash e : \tau$ for a typing judgment. The judgment basically means that e can be assigned type τ under $\phi; \Gamma$, which map free index variables and variables in e to sorts and types, respectively. We use $\mathcal{D}$ for typing derivations and $\mathcal{D} :: \phi; \Gamma \vdash e : \tau$ for a typing derivation $\mathcal{D}$ whose conclusion is $\phi; \Gamma \vdash e : \tau$.

$$\frac{\phi;\Gamma \vdash e : \tau_1 \quad \phi \models \tau_1 \equiv \tau_2}{\phi;\Gamma \vdash e : \tau_2}\ (\textbf{type-eq})$$

$$\frac{\Sigma(c) = \tau}{\phi;\Gamma \vdash c : \tau}\ (\textbf{type-constant})$$

$$\frac{\Gamma(x) = \tau}{\phi;\Gamma \vdash x : \tau}\ (\textbf{type-lam-var})$$

$$\frac{\Gamma(f) = \tau}{\phi;\Gamma \vdash f : \tau}\ (\textbf{type-fix-var})$$

$$\frac{\phi, \vec{a} : \vec{\gamma}; \Gamma \vdash v : \tau}{\phi;\Gamma \vdash \lambda\vec{a} : \vec{\gamma}.v : \Pi\vec{a} : \vec{\gamma}.\tau}\ (\textbf{type-ilam})$$

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

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

$$\frac{}{\phi;\Gamma \vdash \langle\rangle : \mathbf{1}}\ (\textbf{type-unit})$$

$$\frac{\phi;\Gamma \vdash e_1 : \tau_1 \quad \phi;\Gamma \vdash e_2 : \tau_2}{\phi;\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 * \tau_2}\ (\textbf{type-tuple})$$

$$\frac{\phi;\Gamma \vdash e : \tau_1 * \tau_2}{\phi;\Gamma \vdash \mathbf{fst}(e) : \tau_1}\ (\textbf{type-fst})$$

$$\frac{\phi;\Gamma \vdash e : \tau_1 * \tau_2}{\phi;\Gamma \vdash \mathbf{snd}(e) : \tau_2}\ (\textbf{type-snd})$$

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

$$\frac{\phi;\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \phi;\Gamma \vdash e_2 : \tau_1}{\phi;\Gamma \vdash e_1(e_2) : \tau_2}\ (\textbf{type-app})$$

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

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

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

Figure 4. Typing rules for $\mathrm{ML}_0^{\Pi,\Sigma}$.

Let $\vec{a}$ be a sequence of index variables $a_1, \ldots, a_n$ and $\vec{\imath}$ be a sequence of index expressions $i_1, \ldots, i_n$; we write $[\vec{a} \mapsto \vec{\imath}]$ for a substitution that maps a_k to i_k for $k = 1, \ldots, n$; given an index variable context $\vec{a} : \vec{\gamma}$, that is, $a_1 : \gamma_1, \ldots, a_n : \gamma_n$ for $\vec{\gamma} = (\gamma_1, \ldots, \gamma_n)$, we write $\phi \vdash \vec{\imath} : \vec{\gamma}$ to mean that $\phi \vdash [\vec{a} \mapsto \vec{\imath}] : (\vec{a} : \vec{\gamma})$. Notice that $\phi \vdash \vec{\imath} : \vec{\gamma}$ does not simply mean $\phi \vdash i_k : \gamma_k$ for $k = 1, \ldots, n$ as a_k may have a occurrence in $\gamma_{k'}$ for $1 \leq k < k' \leq n$. Some of the typing rules have obvious side conditions, which are omitted. For instance, in the rule **(type-ilam)**, $\vec{a}$ cannot have free occurrences in Γ. We write **dom**(Γ) 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 **dom**$(\theta) =$ **dom**(Γ') and $\phi; \Gamma[\theta_I] \vdash \theta(x) : \Gamma'(x)[\theta_I]$ for all $x \in$ **dom**(Γ').

The following lemma plays a pivotal rôle in proving the subject reduction theorem for $\mathrm{ML}_0^{\Pi,\Sigma}$. We omit its standard proof, which is available in [24].

Lemma 2.3. *Assume that $\phi, \phi'; \Gamma, \Gamma' \vdash e : \tau$ is derivable and $\phi; \Gamma \vdash (\theta_I; \theta) : (\phi'; \Gamma')$ holds. Then we can derive $\phi; \Gamma[\theta_I] \vdash e[\theta_I][\theta] : \tau[\theta_I]$.*

2.3. *Dynamic semantics*

We present the dynamic semantics of $\mathrm{ML}_0^{\Pi,\Sigma}$ through the use of evaluation contexts, which are defined below.

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

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 2.4. A redex is defined as follows.

- Let c be a primitive function such as $+, -, *, /$, etc., and $\Pi\vec{a} : \vec{\gamma}.\tau_1 \to \tau_2$ be the type of c. Then $c[\vec{\imath}](v_1)$ is a redex if v_1 is of type $\tau_1[\vec{a} \mapsto \vec{\imath}]$, which reduces to some value v_2 of type $\tau_2[\vec{a} \mapsto \vec{\imath}]$ according to the definition of c. For example, $+[1, 1](\langle 1, 1\rangle)$ reduces to 2. Note that we need to assume that the type of v_2 is the same as that of $c[\vec{\imath}](v_1)$, that is, the implementation of a primitive function is consistent with its assigned type.
- **if**(c, e_1, e_2) are redexes for $c =$ true, false; they reduce to e_1 and e_2, respectively.
- **fst**$(\langle v_1, v_2\rangle)$ is a redex, which reduces to v_1.
- **snd**$(\langle v_1, v_2\rangle)$ is a redex, which reduces to v_2.
- (**lam** $x : \tau.e)(v)$ is a redex, which reduces to $e[x \mapsto v]$.
- Let e be **fun** $f[\vec{a} : \vec{\gamma}] : \tau$ **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 $v[\vec{a} \mapsto \vec{\imath}]$.
- **open** $\langle i \mid v\rangle$ **as** $\langle a \mid x\rangle$ **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 also 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 are now ready to establish the the following subject reduction theorem for $\mathrm{ML}_0^{\Pi,\Sigma}$.

Theorem 2.5 (*Subject reduction*). *Assume* $\cdot;\cdot \vdash e : \tau$ *is derivable in* $\mathrm{ML}_0^{\Pi,\Sigma}$, *that is*, e *is a well-typed closed expression in* $\mathrm{ML}_0^{\Pi,\Sigma}$. *If* $e \hookrightarrow e'$, *then* $\cdot;\cdot \vdash e' : \tau$ *is also derivable in* $\mathrm{ML}_0^{\Pi,\Sigma}$.

Proof: Assume $e = C[r]$. The proof proceeds by induction on the structure of C. The only interesting case is where $C = []$. In this case, $e = r$ and we proceed by induction on the height of a typing derivation $\mathcal{D}$ of $\cdot;\cdot \vdash e : \tau$.

– $\mathcal{D}$ ends with an application of the rule **(type-eq)**, that is, $\mathcal{D}$ is of the following form.

$$\frac{\cdot \models \tau' \equiv \tau \quad \mathcal{D}_1 :: \cdot;\cdot \vdash e : \tau'}{\cdot;\cdot \vdash e : \tau}$$

By induction hypothesis on $\mathcal{D}_1$, $\cdot;\cdot \vdash e' : \tau'$ is derivable, which leads to the following.

$$\frac{\cdot \models \tau' \equiv \tau \quad \cdot;\cdot \vdash e' : \tau'}{\cdot;\cdot \vdash e' : \tau}$$

We now deal with the cases where the last rule applied in $\mathcal{D}$ is not **(type-eq)**.

– $e = \textbf{fun}\ f[\vec{a} : \vec{\gamma}] : \tau_1\ \textbf{is}\ v$ and $\tau = \Pi\vec{a} : \vec{\gamma}.\tau_1$. Then we have the following derivation, where the last applied rule is **(type-fun)**.

$$\frac{\mathcal{D}_1 :: \vec{a} : \vec{\gamma}; f : \Pi\vec{a} : \vec{\gamma}.\tau_1 \vdash v : \tau_1}{\cdot;\cdot \vdash \textbf{fun}\ f[\vec{a} : \vec{\gamma}] : \tau_1\ \textbf{is}\ v : \Pi\vec{a} : \vec{\gamma}.\tau_1}$$

By Lemma 2.3, we know that $\vec{a} : \vec{\gamma};\cdot \vdash v[f \mapsto e] : \tau_1$ is derivable. Note that $e' = \lambda\vec{a} : \vec{\gamma}.v[f \mapsto e]$. Therefore, we have that $\cdot;\cdot \vdash e' : \Pi\vec{a} : \vec{\gamma}.\tau_1$ is derivable.

– $e = \textbf{open}\ \langle i \mid v\rangle\ \textbf{as}\ \langle a \mid x\rangle\ \textbf{in}\ e_1$. Then we have the following derivation, where the last applied rule is **(type-open)**.

$$\frac{\mathcal{D}_1 :: \cdot;\cdot \vdash \langle i \mid v\rangle : \Sigma a : \gamma.\tau_1 \quad \mathcal{D}_2 :: a : \gamma; x : \tau_1 \vdash e_1 : \tau}{\cdot;\cdot \vdash \textbf{open}\ \langle i \mid v\rangle\ \textbf{as}\ \langle a \mid x\rangle\ \textbf{in}\ e_1 : \tau}$$

There are two possibilities.

- $\mathcal{D}_1$ is of the following form, ending with an application of the rule **(type-pack)**.

$$\frac{\cdot \vdash i : \gamma \quad \cdot;\cdot \vdash v : \tau_1[a \mapsto i]}{\cdot;\cdot \vdash \langle i \mid v\rangle : \Sigma a : \gamma.\tau_1}$$

Note that a has no free occurrence in τ. By Lemma 2.3, we know that $\cdot;\cdot \vdash e' = e_1[a \mapsto i][x \mapsto v] : \tau[a \mapsto i] = \tau$ is derivable.

- $\mathcal{D}_1$ is of the following form, ending with an application of the rule **(type-eq)**.

$$\frac{\mathcal{D}_1' :: \cdot;\cdot \vdash \langle i \mid v\rangle : \Sigma a:\gamma.\tau_1' \quad \cdot \models \Sigma a:\gamma.\tau_1' \equiv \Sigma a:\gamma.\tau_1}{\cdot;\cdot \vdash \langle i \mid v\rangle : \Sigma a:\gamma.\tau_1}$$

  Note that both $\cdot \vdash i : \gamma$ and $a : \gamma \models \tau_1' \equiv \tau_1$ are derivable and thus $\cdot \models \tau_1'[a \mapsto i] \equiv \tau_1[a \mapsto i]$ is derivable. We have two possibilities.

  - $\mathcal{D}_1'$ is of the following form, ending with an application of the rule **(type-pack)**.

$$\frac{\cdot \vdash i:\gamma \quad \cdot;\cdot \vdash v : \tau_1'[a \mapsto i]}{\cdot;\cdot \vdash \langle i \mid v\rangle : \Sigma a:\gamma.\tau_1'}$$

    Therefore we have the following derivation $\mathcal{D}_1^*$ of $\cdot;\cdot \vdash \langle i \mid v\rangle : \Sigma a:\gamma.\tau_1$ that ends with an application of the rule **(type-pack)**. Notice that the height of $\mathcal{D}_1^*$ is the same as that of $\mathcal{D}_1$.

$$\frac{\cdot \vdash i:\gamma \qquad \dfrac{\cdot;\cdot \vdash v : \tau_1'[a \mapsto i] \quad \cdot \models \tau_1'[a \mapsto i] \equiv \tau_1[a \mapsto i]}{\cdot;\cdot \vdash v : \tau_1'[a \mapsto i]}}{\cdot;\cdot \vdash \langle i \mid v\rangle : \Sigma a:\gamma.\tau_1}$$

    By the previous case analysis, we are done.

  - $\mathcal{D}_1'$ is of the following form, ending with an application of **(type-eq)**.

$$\frac{\mathcal{D}_1'' :: \cdot;\cdot \vdash \langle i \mid v\rangle : \Sigma a:\gamma.\tau_1'' \quad \cdot \models \Sigma a:\gamma.\tau_1'' \equiv \Sigma a:\gamma.\tau_1'}{\cdot;\cdot \vdash \langle i \mid v\rangle : \Sigma a:\gamma.\tau_1'}$$

    By Proposition 2.2, we have $\cdot \models \Sigma a:\gamma.\tau_1'' \equiv \Sigma a:\gamma.\tau_1$. Then we have the following derivation $\mathcal{D}_1^*$ of $\cdot;\cdot \vdash \langle i \mid v\rangle : \Sigma a:\gamma.\tau_1$,

$$\frac{\mathcal{D}_1'' :: \cdot;\cdot \vdash \langle i \mid v\rangle : \Sigma a:\gamma.\tau_1'' \quad \cdot \models \Sigma a:\gamma.\tau_1'' \equiv \Sigma a:\gamma.\tau_1}{\cdot;\cdot \vdash \langle i \mid v\rangle : \Sigma a:\gamma.\tau_1}$$

    which is shorter than $\mathcal{D}_1$. Clearly, $\mathcal{D}_1^*$ leads to a derivation $\mathcal{D}^*$ of $\cdot;\cdot \vdash e : \tau$, which is shorter than $\mathcal{D}$. By induction hypothesis on $\mathcal{D}^*$, we are done.

The rest of cases can be handled similarly. □

2.4. Erasure

We can simply transform $\mathrm{ML}_0^{\Pi,\Sigma}$ into a language ML_0 by erasing all syntax related to 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.[3]

Given a well-typed closed expression e in $\mathrm{ML}_0^{\Pi,\Sigma}$, it can be shown that if $|e|$ reduces to e_0 in ML_0 then e reduces to some e_1 such that $|e_1| = e_0$. For this, we need the observation that the erasure of a value in $\mathrm{ML}_0^{\Pi,\Sigma}$ is a value in ML_0, which can be readily verified. This is precisely the reason why we impose the requirement that the body of each λ be a value. Otherwise, the erasure of $\lambda a:\gamma.e$, which is $|e|$, may not necessarily be a value in ML_0. We will present an example at the end of Section 3 to illustrate this (subtle) point.

We can now show that if e is a well-typed closed expression in $\mathrm{ML}_0^{\Pi,\Sigma}$ that is terminating, 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 (likely) done through the evaluation of its erasure in ML_0. Please find more details on the issue of erasure in [24, 29].

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

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 $<$ for the usual *strict* lexicographic ordering on tuples of natural numbers, that is, 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 $1 \leq k \leq n, i_j = i'_j$ for $j = 1, \ldots, k-1$ and $i_k < i'_k$. Evidently, $<$ is a well-founded ordering. 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 $\phi \vdash i_k : nat$ are derivable for $k = 1, \ldots, n$. We write $\phi \vdash \mu : \mathbf{metric}$ to mean that 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 rule for forming such a type is given below.

$$\frac{\phi, \vec{a} : \vec{\gamma} \vdash \tau\ \mathbf{[well\text{-}formed]} \quad \phi, \vec{a} : \vec{\gamma} \vdash \mu : \mathbf{metric}}{\phi \vdash \Pi\vec{a} : \vec{\gamma}.\mu \Rightarrow \tau\ \mathbf{[well\text{-}formed]}}$$

We emphasize that a decorated type can only occur at top level. In other words, when a decorated type $\Pi\vec{a} : \vec{\gamma}.\mu \Rightarrow \tau$ is formed, τ must be a regular type as is defined in $\mathrm{ML}_0^{\Pi,\Sigma}$.

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 fix-variable f in its domain to a decorated type, and a recursive function in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ is of form $\mathbf{fun}\ f[\vec{a} : \vec{\gamma}] : \mu \Rightarrow \tau\ \mathbf{is}\ v$. As an example, we present in Figure 5 an expression in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ corresponding to the implementation of the Ackermann function in Figure 1. Note that we write $= [a_1, 0](\langle x_1, 0\rangle)$ for testing whether x_1 equals 0 as $=$ is previously assumed to have the following type.

$$\Pi\{a_1 : int, a_2 : int\}.\mathtt{int}(a_1) * \mathtt{int}(a_2) \to \mathtt{bool}(if(a_1 = a_2, 1, 0))$$

The process that translates the implementation into the expression in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ is what we call *elaboration*, which is explained in [24, 29]. Our approach to program termination

$$
\begin{array}{l}
\textbf{fun}\ \mathit{ack}[a_1 : \mathit{nat}, a_2 : \mathit{nat}] : \\
\quad \langle a_1, a_2 \rangle \Rightarrow \mathit{int}(a_1) \rightarrow \mathit{int}(a_2) \rightarrow \Sigma a : \mathit{nat}.\mathit{int}(a)\ \ \textbf{is} \\
\ \ \textbf{lam}\ x_1 : \mathtt{int}(a_1).\textbf{lam}\ x_2 : \mathtt{int}(a_2). \\
\quad \textbf{if}\ (= [a_1, 0](\langle x_1, 0 \rangle), \\
\qquad \langle a_2 + 1 \mid +[a_2, 1](\langle x_2, 1 \rangle) \rangle, \\
\qquad \textbf{if}\ (= [a_2, 0](\langle x_2, 0 \rangle), \\
\qquad\quad \mathit{ack}[a_1 - 1, 1](-[a_1, 1](\langle x_1, 1 \rangle))(1), \\
\qquad\quad \textbf{open}\ \mathit{ack}[a_1, a_2 - 1](x_1)(-[a_2, 1](\langle x_2, 1 \rangle)) \\
\qquad\qquad \textbf{as}\ \langle a_2' \mid x_2' \rangle\ \textbf{in}\ \mathit{ack}[a_1 - 1, a_2'](-[a_1, 1](\langle x_1, 1 \rangle))(x_2')))
\end{array}
$$

Figure 5. The Ackermann function in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$.

verification is intended to be applied to elaborated programs and we assume the availability of such an elaboration process when presenting realistic examples.

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

Suppose $\Gamma(f) = \Pi\vec{a} : \vec{\gamma}.\mu \Rightarrow \tau$; roughly speaking, $\phi; \Gamma \vdash e : \tau \ll_f \mu_0$ means that, for each free occurrence of f in e, f is followed by a sequence of index expressions $[\vec{\imath}]$, and $\mu[\vec{a} \mapsto \vec{\imath}]$, 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 $e = \textbf{fun}\ f[\vec{a} : \vec{\gamma}] : \mu \Rightarrow \tau\ \textbf{is}\ v$ in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ and $\vec{\imath}$ 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; with the metric attached to f, we can show 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 sheds 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 formally carried out directly in the presence of higher-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 for the following ones.

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

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

We present the rules for deriving metric typing judgments in Figures 6 and 7. 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 $1 \leq k \leq n$, we have that $\phi, i_1 = i_1', \ldots, i_{j-1} = i_{j-1}' \models i_j \leq i_j'$ hold for all $1 \leq j < k$ and $\phi, i_1 = i_1', \ldots, i_{k-1} = i_{k-1}' \models i_k < i_k'$ also holds. It should be clear that if $\phi \models \mu < \mu'$ holds and $\cdot \vdash \theta_I : \phi$ is derivable, then $\mu[\theta_I] < \mu'[\theta_I]$ is true.

$$\frac{\phi;\Gamma\vdash e:\tau_1 \ll_f \mu_0 \quad \phi\models\tau_1\equiv\tau_2}{\phi;\Gamma\vdash e:\tau_2 \ll_f \mu_0}\ (\ll\textbf{-eq})$$

$$\frac{\Sigma(c)=\tau}{\phi;\Gamma\vdash c:\tau \ll_f \mu_0}\ (\ll\textbf{-constant})$$

$$\frac{\Gamma(x)=\tau}{\phi;\Gamma\vdash x:\tau \ll_f \mu_0}\ (\ll\textbf{-lam-var})$$

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

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

$$\frac{\phi,\vec{a}:\vec{\gamma};\Gamma\vdash v:\tau \ll_f \mu_0}{\phi;\Gamma\vdash \lambda\vec{a}:\vec{\gamma}.v:\Pi\vec{a}:\vec{\gamma}.\tau \ll_f \mu_0}\ (\ll\textbf{-ilam})$$

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

$$\frac{\phi;\Gamma\vdash e_1:\tau_1 \ll_f \mu_0 \quad \phi;\Gamma\vdash e_2:\tau_2 \ll_f \mu_0}{\phi;\Gamma\vdash \langle e_1,e_2\rangle:\tau_1*\tau_2 \ll_f \mu_0}\ (\ll\textbf{-tuple})$$

$$\frac{\phi;\Gamma\vdash e:\tau_1*\tau_2 \ll_f \mu_0}{\phi;\Gamma\vdash \mathbf{fst}(e):\tau_1 \ll_f \mu_0}\ (\ll\textbf{-fst})$$

$$\frac{\phi;\Gamma\vdash e:\tau_1*\tau_2 \ll_f \mu_0}{\phi;\Gamma\vdash \mathbf{snd}(e):\tau_2 \ll_f \mu_0}\ (\ll\textbf{-snd})$$

Figure 6. Metric typing rules for $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ (1).

$$\frac{\phi;\Gamma,x:\tau_1\vdash e:\tau_2 \ll_f \mu_0}{\phi;\Gamma\vdash \mathbf{lam}\ x:\tau_1.e:\tau_1\rightarrow\tau_2 \ll_f \mu_0}\ (\ll\textbf{-lam})$$

$$\frac{\phi;\Gamma\vdash e_1:\tau_1\rightarrow\tau_2 \ll_f \mu_0 \quad \phi;\Gamma\vdash e_2:\tau_1 \ll_f \mu_0}{\phi;\Gamma\vdash e_1(e_2):\tau_2 \ll_f \mu_0}\ (\ll\textbf{-app})$$

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

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

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

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

Figure 7. Metric typing rules for $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ (2).

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 $\phi; \Gamma[\theta_I] \vdash e[\theta_I][\theta] : \tau[\theta_I]$.*
2. *Assume $\phi, \phi'; \Gamma, \Gamma' \vdash e : \tau \ll_f \mu_0$ is derivable and $\phi; \Gamma \vdash (\theta_I; \theta) : (\phi'; \Gamma')$ holds and $f \in \mathbf{dom}(\Gamma)$. Then we can derive $\phi; \Gamma[\theta_I] \vdash e[\theta_I][\theta] : \tau[\theta_I] \ll_f \mu_0[\theta_I]$.*

Proof: (1) and (2) are proved simultaneously by structural induction on the derivation $\mathcal{D}_1$ of $\phi, \phi'; \Gamma, \Gamma' \vdash e : \tau$ and the derivation $\mathcal{D}_2$ of $\phi, \phi'; \Gamma, \Gamma' \vdash e : \tau \ll_f \mu$, respectively. We present some cases.

– $\mathcal{D}_1$ is of the following form, where $e = \mathbf{lam}\ x : \tau_1.e_1$ and $\tau = \tau_1 \to \tau_2$.

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

By induction hypothesis, $\phi; \Gamma[\theta_I], x : \tau_1[\theta_I] \vdash e_1[\theta_I][\theta] : \tau_2[\theta_I]$ is derivable. Hence, $\phi; \Gamma \vdash \mathbf{lam}\ x : \tau_1[\theta_I].e_1[\theta_I][\theta] : \tau_1[\theta_I] \to \tau_2[\theta_I]$ is derivable. Note that $e[\theta_I][\theta] = \mathbf{lam}\ .x : \tau_1[\theta_I].e_1[\theta_I][\theta]$ (as we can choose x to make sure it has no free occurrences in θ) and $\tau[\theta_I] = \tau_1[\theta_I] \to \tau_2[\theta_I]$. Hence, we are done.

– $\mathcal{D}_2$ is of the following form, where $e = \langle i \mid e_1 \rangle$ and $\tau = \Sigma a : \gamma.\tau_1$.

$$\frac{\phi, \phi' \vdash i : \gamma \quad \phi, \phi'; \Gamma, \Gamma' \vdash e_1 : \tau_1[a \mapsto i] \ll_f \mu_0}{\phi, \phi'; \Gamma, \Gamma' \vdash \langle i \mid e_1 \rangle : \Sigma a : \gamma.\tau_1 \ll_f \mu_0}\ \textbf{($\ll$-pack)}$$

By induction hypothesis, $\phi; \Gamma[\theta_I] \vdash e_1[\theta_I][\theta] : \tau_1[a \mapsto i][\theta_I] \ll_f \mu_0[\theta_I]$ is derivable. By Lemma 2.1, $\phi \vdash i[\theta_I] : \gamma[\theta_I]$ is derivable. Clearly, we can choose a so that it has no free occurrences in θ_I and thus have $\tau_1[a \mapsto i][\theta_I] = \tau_1[\theta_I[a \mapsto i[\theta_I]]$. We then have the following derivation, where the last applied rule is **($\ll$-pack)**.

$$\frac{\phi \vdash i[\theta_I] : \gamma[\theta_I] \quad \phi; \Gamma[\theta_I] \vdash e_1[\theta_I][\theta] : \tau_1[\theta_I][a \mapsto i[\theta_I]] \ll_f \mu_0[\theta_I]}{\phi; \Gamma[\theta_I] \vdash \langle i[\theta_I] \mid e_1[\theta_I][\theta] \rangle : \Sigma a : \gamma[\theta_I].\tau_1[\theta_I] \ll_f \mu_0[\theta_I]}$$

Note that $e[\theta_I][\theta] = \langle i[\theta_I], e_1[\theta_I][\theta] \rangle$ and $\tau[\theta_I] = (\Sigma a : \gamma.\tau_1)[\theta_I] = \Sigma a : \gamma[\theta_I].\tau_1[\theta_I]$. We are done.

The rest of the cases can be handled similarly. □

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

Proof: The proof is essentially the same as the one for Theorem 2.5. □

As could be expected, we have the following.

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

Proof: The proof immediately follows from a structural induction on $\mathcal{D}$. □

3.3. Reducibility

We use $e\downarrow$ to mean that there is no infinite reduction sequence starting from e. Clearly, for every e such that $e\downarrow$ holds, there exists a smallest number l such that the number of steps in each reduction sequence from e is at most l; we use $l(e)$ for such a number.

In the current setting, $e\downarrow$ is equivalent to $e \hookrightarrow^* v$ for some v. However, this is to be changed after pattern matching is introduced, which may bring nondeterminism into evaluation. We define the notion of reducibility for well-typed closed expressions e in such a manner so that $e\downarrow$ hold for all reducible expressions e. The intention is to show that every well-typed closed expression e is reducible and thus there is no infinite reduction sequence starting from e.

Definition 3.5 (Reducibility). Suppose that e is a closed expression of type τ. We define that e is reducible of type τ by induction on the complexity of τ, namely, the number of type constructors $*$, $\rightarrow$, Π and Σ in τ.

1. τ is a base type. Then e is reducible of type τ if $e\downarrow$ holds.
2. $\tau = \tau_1 * \tau_2$. Then e is reducible of type τ if $e\downarrow$ holds and for every v such that $e \hookrightarrow^* v$ holds, $v = \langle v_1, v_2\rangle$ for some values v_1 and v_2 that are reducible of types τ_1 and τ_2, respectively.
3. $\tau = \tau_1 \rightarrow \tau_2$. Then e is reducible of type τ if $e\downarrow$ holds and $e(v)$ is reducible of type τ_2 for every value v reducible of type τ_1.
4. $\tau = \Pi\vec{a} : \vec{\gamma}.\tau_1$. Then e is reducible of type τ if $e\downarrow$ and $e[\vec{\imath}]$ is reducible of type $\tau_1[\vec{a} \mapsto \vec{\imath}]$ for each $\vec{\imath} : \vec{\gamma}$.
5. $\tau = \Sigma a : \gamma.\tau_1$. Then e is reducible of type τ if $e\downarrow$ holds and for all v satisfying $e \hookrightarrow^* v$, we have $v = \langle i \mid v_1\rangle$ for some i and v_1 such that v_1 is reducible of type $\tau_1[a \mapsto i]$.

We may use the phrase that e is reducible to mean that e is reducible of type τ for some τ.

Proposition 3.6. *Given a ground type τ, that is, there is no $\rightarrow$ or Π inside τ, then every value of type τ is reducible.*

Proof: This immediately follows from induction on the complexity of τ. □

Proposition 3.7. *Assume that e is a closed expression of type τ.*
1. *If e is reducible of type τ and $e \hookrightarrow e'$, then e' is reducible of type τ.*
2. *If for every e' such that $e \hookrightarrow e'$, e' is reducible of type τ, then e is reducible of type τ.*
3. *If $e\downarrow$ holds and for every v such that $e \hookrightarrow^* v$, v is reducible, then e is also reducible.*

Proof: (1) and (2) immediately follow from induction on the complexity of τ. We prove (3) by induction on $l(e)$. Assume that $e \hookrightarrow e'$. Then $l(e') < l(e)$. For each v such that $e' \hookrightarrow^* v$, we have $e \hookrightarrow^* v$ and thus v is reducible. By induction hypothesis, e' is reducible. By (2), we know that e is reducible. □

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

Definition 3.8 (μ-Reducibility). Let e be a well-typed closed recursive function **fun** $f[\vec{a} : \vec{\gamma}] : \mu \Rightarrow \tau$ **is** v and μ_0 be a closed metric. We define that e is μ_0-reducible if $e[\vec{\imath}]$ is reducible of type $\tau[\vec{a} \mapsto \vec{\imath}]$ for each $\vec{\imath} : \vec{\gamma}$ satisfying $\mu[\vec{a} \mapsto \vec{\imath}] < \mu_0$.

We are now ready to present the main technical result in this paper.

Lemma 3.9 (*Main lemma*). *Assume that* $\phi; \Gamma \vdash e : \tau$ *is derivable and* $\cdot; \cdot \vdash (\theta_I; \theta) : (\phi; \Gamma)$ *holds. Also assume that* (1) *for every* $x \in \mathbf{dom}(\Gamma)$, $\theta(x)$ *is reducible*, *and* (2) *for every* $f \in \mathbf{dom}(\Gamma)$, $\cdot; \Gamma[\theta_I] \vdash e[\theta_I] : \tau[\theta_I] \ll_f \mu_f$ *is derivable for some* μ_f *such that* $\theta(f)$ *is* 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 some interesting cases.

– The rule **(type-constant)** is last applied in $\mathcal{D}$,

$$\frac{\Sigma(c) = \tau}{\phi; \Gamma \vdash c : \tau} \ \textbf{(type-constant)}$$

where $e = c$. We have two possibilities.

- τ is a ground type. Then c is reducible by Proposition 3.6.
- c is a primitive function and $\tau = \tau_1 \to \tau_2$ for some ground types τ_1 and τ_2. Assume that v_1 is reducible of type τ_1 and $c(v_1) \hookrightarrow v_2$. Then v_2 is of type τ_2 and thus reducible by Proposition 3.6.

Hence, we conclude that e is reducible.

– The rule **(type-ilam)** is last applied in $\mathcal{D}$,

$$\frac{\mathcal{D}_1 :: \phi, \vec{a} : \vec{\gamma}; \Gamma \vdash v : \tau_1}{\phi; \Gamma \vdash \lambda\vec{a} : \vec{\gamma}.v : \Pi\vec{a} : \vec{\gamma}.\tau_1} \ \textbf{(type-ilam)}$$

where $e = \lambda\vec{a} : \vec{\gamma}.v$ and $\tau = \Pi\vec{a} : \vec{\gamma}.\tau_1$. Let $e^* = e[\theta_I][\theta]$. Assume that $\cdot \vdash \vec{\imath} : \vec{\gamma}[\theta_I]$. It is clear that $\cdot \vdash \theta'_I : (\phi, \vec{a} : \vec{\gamma})$ is derivable for $\theta'_I = \theta_I[\vec{a} \mapsto \vec{\imath}]$. Note that no index variables in $\vec{a}$ have free occurrences in Γ. By induction hypothesis on $\mathcal{D}_1$, $v[\theta'_I][\theta]$ is reducible. Hence $e^*[\vec{\imath}]$ is reducible since $e^*[\vec{\imath}] \hookrightarrow v[\theta'_I][\theta]$. Note that e^* is a value. Hence, e^* is reducible by definition.

– The rule **(type-if)** is last applied in $\mathcal{D}$,

$$\frac{\begin{array}{c}\mathcal{D}_0 :: \phi; \Gamma \vdash e_0 : \mathtt{bool}(i) \\ \mathcal{D}_1 :: \phi, i = 1; \Gamma \vdash e_1 : \tau \quad \mathcal{D}_2 :: \phi, i = 0; \Gamma \vdash e_2 : \tau\end{array}}{\phi; \Gamma \vdash \mathbf{if}(e_0, e_1, e_2) : \tau}$$

where $e = \mathbf{if}(e_0, e_1, e_2)$. Let $e^* = e[\theta_I][\theta]$, and $e_k^* = e_k[\theta_I][\theta]$ for $k = 0, 1, 2$. By induction hypothesis on $\mathcal{D}_0$, e_0^* is reducible. Hence, $e_0^* \downarrow$ holds. Assume that $e^* \hookrightarrow e'$. We prove that e' is reducible by induction on $l_0 = l(e_0^*)$. There are three possibilities.

- $e_0^* \hookrightarrow e_0'$ and $e' = \mathbf{if}(e_0', e_1^*, e_2^*)$. By induction hypothesis, e' is reducible since $l(e_0') < l(e_0^*)$.
- $e_0^* = \mathit{true}$ and $e' = e_1^*$. Then $\cdot \vdash i[\theta_I] = 1$ holds by Theorem 3.3. This implies that $\cdot \vdash \theta_I : (\phi, i = 1)$ is derivable. Hence, $e' = e_1^*$ is reducible by induction hypothesis on $\mathcal{D}_1$.
- $e_0^* = \mathit{false}$ and $e' = e_2^*$. Then $\cdot \vdash i[\theta_I] = 0$ holds by Theorem 3.3. This implies that $\cdot \vdash \theta_I : (\phi, i = 0)$ is derivable. Hence, $e' = e_2^*$ is reducible by induction hypothesis on $\mathcal{D}_2$.

By Proposition 3.7 (2), we conclude that e^* is reducible.

– The rule **(type-lam)** is last applied in $\mathcal{D}$,

$$\frac{\mathcal{D}_1 :: \phi; \Gamma, x : \tau_1 \vdash e_1 : \tau_2}{\phi; \Gamma \vdash \mathbf{lam}\ x : \tau_1.e_1 : \tau_1 \to \tau_2}\ \textbf{(type-lam)}$$

where $e = \mathbf{lam}\ x : \tau_1.e_1$ and $\tau = \tau_1 \to \tau_2$. Obviously, we can assume that $x \notin \mathbf{dom}(\theta)$. Let $e^* = e[\theta_I][\theta]$. Then e^* is of type $\tau_1^* \to \tau_2^*$ for $\tau_1^* = \tau_1[\theta_I]$ and $\tau_2^* = \tau_2[\theta_I]$. Assume that v is a reducible value of type τ_1^*. Then $e^*(v) \hookrightarrow e_1^* = e_1[\theta_I][\theta[x \mapsto v]]$. By induction hypothesis on $\mathcal{D}_1$, e_1^* is reducible, and thus $e^*(v)$ is reducible. By the definition of reducibility, e^* is reducible since e^* is a value and $e^*(v)$ is reducible for every reducible value of type τ_1^*.

– The rule **(type-app)** is last applied in $\mathcal{D}$,

$$\frac{\mathcal{D}_1 :: \phi; \Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \mathcal{D}_2 :: \phi; \Gamma \vdash e_2 : \tau_1}{\phi; \Gamma \vdash e_1(e_2) : \tau_2}\ \textbf{(type-app)}$$

where $e = e_1(e_2)$ and $\tau = \tau_2$. Let $e^* = e[\theta_I][\theta]$. By induction hypotheses on $\mathcal{D}_1$ and $\mathcal{D}_2$, respectively, both $e_1^* = e_1[\theta_I][\theta]$ and $e_2^* = e_2[\theta_I][\theta]$ are reducible. Hence $e_1^* \downarrow$ and $e_2^* \downarrow$ hold. Assume $e^* \hookrightarrow e'$. We show that e' is reducible by induction on $l(e_1^*) + l(e_2^*)$. There are three possibilities.

- $e_1^* \hookrightarrow e_1'$ and $e' = e_1'(e_2^*)$. By induction hypothesis, e' is reducible since $l(e_1') + l(e_2^*) < l(e_1^*) + l(e_2^*)$.
- e_1^* is a value and $e_2^* \hookrightarrow e_2'$ and $e' = e_1^*(e_2')$. By induction hypothesis, e' is reducible since $l(e_1^*) + l(e_2') < l(e_1^*) + l(e_2^*)$.
- e_1^* and e_2^* are some values of types $\tau_1^* \to \tau_2^*$ and τ_1^*, respectively, where $\tau_1^* = \tau_1[\theta_I]$ and $\tau_2^* = \tau_2[\theta_I]$. Then $e_1^*(e_2^*)$ is reducible of type τ_2^* by the definition of reducibility. Hence, by Proposition 3.7 (1), e' is reducible.

We conclude that e^* is reducible by Proposition 3.7 (2).

– The rule **(type-open)** is last applied in $\mathcal{D}$,

$$\frac{\mathcal{D}_1 :: \phi; \Gamma \vdash e_1 : \Sigma a : \gamma.\tau_1 \quad \mathcal{D}_2 :: \phi, a : \gamma; \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\phi; \Gamma \vdash \mathbf{open}\ e_1\ \mathbf{as}\ \langle a \mid x \rangle\ \mathbf{in}\ e_2 : \tau_2}$$

where $e = \textbf{open}\ e_1\ \textbf{as}\ \langle a \mid x\rangle\ \textbf{in}\ e_2$ and $\tau = \tau_2$. Obviously, we can assume that $a \notin \textbf{dom}(\theta_I)$ and $x \notin \textbf{dom}(\theta)$. Let $e^* = e[\theta_I][\theta]$. Then $e^* = \textbf{open}\ e_1^*\ \textbf{as}\ \langle a \mid x\rangle\ \textbf{in}\ e_2^*$, where $e_1^* = e_1[\theta_I][\theta]$ is of type $\Sigma a : \gamma^*.\tau_1^*$ for $\gamma^* = \gamma[\theta_I]$ and $\tau_1^* = \tau_1[\theta_I]$. By induction hypothesis on D_1, e_1^* is reducible. Hence $e_1^* \downarrow$ holds. Assume $e^* \hookrightarrow e'$. We prove that e' is reducible by induction on $l(e_1^*)$. There are two possibilities.

- $e_1^* \hookrightarrow e_1'$ and $e' = \textbf{open}\ e_1'\ \textbf{as}\ \langle a \mid x\rangle\ \textbf{in}\ e_2^*$. By induction hypothesis, e' is reducible since $l(e_1') < l(e_1^*)$.
- $e_1^* = \langle i \mid v\rangle$ for some i of sort γ^* and some reducible value v of type $\tau_1^*[a \mapsto i]$. Hence, we have $e' = e_2[\theta_I][\theta][a \mapsto i][x \mapsto v] = e_2[\theta_I[a \mapsto i]][\theta[x \mapsto v]]$, which is reducible by induction hypothesis on $\mathcal{D}_2$.

We conclude that e^* is reducible by Proposition 3.7 (2).

– The rule **(type-fun)** is last applied in $\mathcal{D}$,

$$\frac{\phi, \vec{a}_1 : \vec{\gamma}_1; \Gamma, f_1 : \Pi\vec{a}_1 : \vec{\gamma}_1.\mu_1 \Rightarrow \tau_1 \vdash v_1 : \tau_1 \ll_f \mu_1}{\phi; \Gamma \vdash \textbf{fun}\ f_1[\vec{a}_1 : \vec{\gamma}_1] : \mu_1 \Rightarrow \tau_1\ \textbf{is}\ v_1 : \Pi\vec{a}_1 : \vec{\gamma}_1.\tau_1}\ \textbf{(type-fun)}$$

where we have $e = \textbf{fun}\ f_1[\vec{a}_1 : \vec{\gamma}_1] : \mu_1 \Rightarrow \tau_1\ \textbf{is}\ v_1$ and $\tau = \Pi\vec{a}_1 : \vec{\gamma}_1.\tau_1$. Clearly, we can require that $\vec{a}_1$ have no free occurrences in θ_I. Suppose that $e^* = e[\theta_I][\theta]$ is not reducible. It is obvious that $e^* \downarrow$ holds. Hence, by the definition of reducibility at Π-types and the well-foundedness of the lexicographic ordering $<$, there exists $\vec{\imath}_0 : \vec{\gamma}_1^*$ 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 $\mu_1^* = \mu_1[\theta_I]$. Let $\mu_{f_1} = \mu_1^*[\vec{a}_1 \mapsto \vec{\imath}_0]$, and we have that e^* is μ_{f_1}-reducible. We can derive $\cdot; \Gamma[\theta_I], f_1 : \Pi\vec{a}_1 : \vec{\gamma}_1^*.\tau_1[\theta_I] \vdash v_1[\theta_I[\vec{a}_1 \mapsto \vec{\imath}_0]] : \tau_1[\theta_I[\vec{a}_1 \mapsto \vec{\imath}_0]] \ll_f \mu_{f_1}$ by Lemma 3.2. Note that 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 on $\mathcal{D}_1$, 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 v_1^* is the only value such that $e^*[\vec{\imath}_0] \hookrightarrow v_1^*$ holds. Hence, $e^*[\vec{\imath}_0]$ is reducible, contradicting the definition of $\vec{\imath}_0$. Therefore, e^* is reducible.

All other cases can be treated similarly. □

The following is the main result of the paper.

Theorem 3.10. *If $\cdot \vdash e : \tau$ is derivable in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$, then e is reducible and thus terminating as there is no infinite reduction sequence from e.*

Proof: The theorem immediately follows from Lemma 3.9. □

We now present an example to illustrate an interesting point. Let e be $\textbf{fun}\ f[a : \bot] : \langle 0\rangle \Rightarrow \mathtt{int}(a) \to \mathtt{int}(a)\ \textbf{is lam}\ x : \mathtt{int}(a).f[a](x)$, where $\bot$ is the empty sort $\{a : \mathit{int} \mid 1 \leq a \leq 0\}$. Note that we can derive

$$\phi; \Gamma \vdash \textbf{lam}\ x : \mathtt{int}(a).f[a](x) : \mathtt{int}(a) \to \mathtt{int}(a) \ll_f \langle 0\rangle$$

for $\phi = a : \bot$ and $\Gamma = f : \Pi a : \bot.\langle 0\rangle \Rightarrow \mathtt{int}(a) \rightarrow \mathtt{int}(a)$, since $a : \bot \models 0 < 0$ holds. Therefore, $\cdot; \cdot \vdash \lambda a : \bot.e[0](0) : \Pi a : \bot.\mathtt{int}(0)$ could have been derived in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ if we had not required that the body following a λ be a value. However, the erasure of $\lambda a : \bot.e[0](0)$, which is (**fun** $f : \mathtt{int} \rightarrow \mathtt{int}$ **is lam** $x : \mathtt{int}.f(x)$)(0), reduces to itself, leading to an infinite reduction sequence in ML_0. The restriction requiring that the body of a λ be a value is partly motivated by avoiding this undesirable consequence, that is, the erasure of a terminating expression in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ may not be terminating in ML_0.

4. Extensions

In this section, we 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 each extension.

4.1. *Mutual recursion*

We here describe the treatment of mutual recursion, which is slightly different from the standard treatment. The syntax for handling mutual recursion is given in Figure 8. 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. Note that this should not be confused with the product of types $\tau_1, \ldots, \tau_n$. The n in $e.n$ must be a fixed positive integer. The following typing rule **(type-funs)**

$$\frac{\begin{array}{c}\tau = (\Pi\vec{a}_1 : \vec{\gamma}_1.\tau_1, \ldots, \Pi\vec{a}_n : \vec{\gamma}_n.\tau_n) \\ \Gamma_0 = f_1 : \Pi\vec{a}_1 : \vec{\gamma}_1 : \mu_1 \Rightarrow \tau_1, \ldots, f_n : \Pi\vec{a}_n : \vec{\gamma}_n : \mu_n \Rightarrow \tau_n \\ \phi, \vec{a}_1 : \vec{\gamma}_1; \Gamma, \Gamma_0 \vdash v_1 : \tau_1 \ll_{\vec{f}} \mu_1 \\ \vdots \\ \phi, \vec{a}_n : \vec{\gamma}_n; \Gamma, \Gamma_0 \vdash v_n : \tau_n \ll_{\vec{f}} \mu_n\end{array}}{\phi; \Gamma \vdash e : \tau}$$

types
$\tau ::= \cdots \mid (\tau_1, \ldots, \tau_n)$

expressions
$e ::= \cdots \mid e.n \mid$
  funs $f_1[\vec{a}_1 : \vec{\gamma}_1] : \mu_1 \Rightarrow \tau_1$ **is** v_1 **and** $\ldots$
  and $f_n[\vec{a}_n : \vec{\gamma}_n] : \mu_n \Rightarrow \tau_n$ **is** v_n

values
$v ::= \cdots \mid$
  funs $f_1[\vec{a}_1 : \vec{\gamma}_1] : \mu_1 \Rightarrow \tau_1$ **is** v_1 **and** $\ldots$
  and $f_n[\vec{a}_n : \vec{\gamma}_n] : \mu_n \Rightarrow \tau_n$ **is** v_n

evaluation contexts
$E ::= \cdots \mid E.n$

Figure 8. The syntax and typing rules for mutual recursion.

is for expression e of the form:

$$\textbf{funs } f_1[\vec{a}_1 : \vec{\gamma}_1] : \mu_1 \Rightarrow \tau_1 \textbf{ is } v_1 \textbf{ and} \ldots \textbf{and } f_n[\vec{a}_n : \vec{\gamma}_n] : \mu_n \Rightarrow \tau_n \textbf{ is } v_n$$

We also have a the following typing rule **(type-choose)** for choosing a mutually recursively defined function.

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

Let v be the following expression.

$$\textbf{funs } f_1[\vec{a}_1 : \vec{\gamma}_1] : \mu_1 \Rightarrow \tau_1 \textbf{ is } v_1 \textbf{ and} \ldots \textbf{and } f_n[\vec{a}_n : \vec{\gamma}_n] : \mu_n \Rightarrow \tau_n \textbf{ is } v_n$$

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 $\vec{f} = (f_1, \ldots, f_n)$. 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 Figures 6 and 7 except **($\ll$-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 **($\ll$-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 $e \downarrow$ holds. Then e is a reducible expression of type $(\tau_1, \ldots, \tau_n)$ if $e.k$ is reducible of type τ_k for each $1 \leq k \leq n$.

4.2. *Currying*

A decorated type must so far be of the form $\Pi\vec{a} : \vec{\gamma}.\mu \Rightarrow \tau$ and this restriction has a rather unpleasant consequence. For instance, we may want to assign the following type τ to the implementation of the Ackermann function `ack` in Figure 1:

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

which is formally written as

$$\Pi a_1 : nat.\texttt{int}(a_1) \rightarrow \Pi a_2 : nat.\texttt{int}(a_2) \rightarrow \Sigma a : nat.\texttt{int}(a).$$

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. Note that such a type is more general than the following type when elaboration is concerned.

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

For instance, given the former type, `ack(1)` is elaborated into an expression of type `{j:nat} int(j) -> [k:nat] int(k)`; given the latter type, `ack(1)` is elaborated into an expression of type `int(J) -> [k:nat] int(k)`, where `J` is some unification variable that needs to be solved later; therefore, for the sake of elaboration, the former type is more general than the latter one.

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

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

We introduce the following form of expression e for representing a recursive function.

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

Note that e_0 is required to be a value if $n = 0$. In the following presentation, we only deal with the case where $n = 1$. For $n > 1$, the treatment is similar.

For $e = \mathbf{fun}\ f[\vec{a}_1 : \vec{\gamma}_1](x_1 : \tau_1)[\vec{a} : \vec{\gamma}] : \tau\ \mathbf{is}\ e_0$, we have $e \hookrightarrow \lambda\vec{a}_1 : \vec{\gamma}_1.\mathbf{lam}\ x_1 : \tau_1.\lambda\vec{a} : \vec{\gamma}.e_0$ and the following typing rule

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

and the following metric typing rule,

$$\frac{\begin{array}{c}\phi \models \vec{\imath}_1 : \vec{\gamma}_1 \quad \phi \models \vec{\imath} : \vec{\gamma}[\vec{a}_1 \mapsto \vec{\imath}_1] \quad \phi \models \mu[\vec{\imath}_1 \mapsto \vec{a}_1][\vec{a} \mapsto \vec{\imath}] < \mu_0 \\ \phi; \Gamma \vdash e_1 : \tau_1[\vec{a}_1 \mapsto \vec{\imath}_1] \ll_f \mu_0 \quad \phi; \Gamma \vdash f : \tau_0\end{array}}{\phi; \Gamma \vdash f[\vec{\imath}_1](e_1)[\vec{\imath}] : \tau[\vec{a}_1 \mapsto \vec{\imath}_1][\vec{a} \mapsto \vec{\imath}] \ll_f \mu_0}$$

where $\tau_0 = \Pi\vec{a}_1 : \vec{\gamma}_1.\tau_1 \rightarrow \Pi\vec{a} : \vec{\gamma}.\mu \Rightarrow \tau$.

Definition 4.2 (μ-reducibility). Let e be a closed recursive function $\mathbf{fun}\ f[\vec{a}_1 : \vec{\gamma}_1](x_1 : \tau_1)[\vec{a} : \vec{\gamma}] : \tau\ \mathbf{is}\ e_0$ and μ_0 be a closed metric. We say that e is μ_0-reducible if $e[\vec{\imath}_1](v)[\vec{\imath}]$ is 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 $\mu[\vec{a}_1 \mapsto \vec{\imath}_1][\vec{a} \mapsto \vec{\imath}] < \mu_0$.

4.3. *Pattern matching*

The mechanism for declaring datatypes in ML is of great use in practice. It can offer both convenience in programming and clarity in code. We extend $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ with pattern matching in this section. However, the handling of pattern matching is rather involved.

In the rest of this section, we fix δ_0 to be a user-defined type constructor that takes index expressions $\vec{\imath}$ of sorts $\vec{\gamma}$ to form a type $\delta_0(\vec{\imath})$. We assume that the constructors c_k ($k = 1, \ldots, m$) of types $\Pi\vec{a}_k : \vec{\gamma}_k.\tau^k \rightarrow \delta_0(\vec{\imath}_k)$ are associated with δ_0 for forming values of types of form $\delta_0(\vec{\imath})$, that is, for each closed value v of type $\delta_0(\vec{\imath})$, v is of form $c_k[\vec{\imath}'](v')$ for some $1 \leq k \leq m$.

The syntax for this extension is given below. Note that we now use c for both constants and constructors.

patterns	$p ::= x \mid \langle\rangle \mid \langle p_1, p_2\rangle \mid c[\vec{a}](p)$
clauses	$ms ::= (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n)$
expressions	$e ::= \cdots \mid c[\vec{\imath}](e) \mid \textbf{case } e \textbf{ of } ms$
values	$v ::= \cdots \mid c[\vec{\imath}](v)$
evaluation contexts	$E ::= \cdots \mid c[\vec{\imath}](E) \mid \textbf{case } E \textbf{ of } ms$

We require that c be a constructor if $c[\vec{\imath}](p)$ is a pattern or $c[\vec{\imath}](v)$ is a value. No index variables or lam-variables can occur more than once in a pattern. The typing and metric typing rules for pattern matching are presented in Figure 9. We use $\tau_1 \Rightarrow \tau_2$ for the type of a clause $p \Rightarrow e$, which basically means that e can be assigned type τ_2 if p is required to have type τ_1. A judgment of form $p \downarrow \tau \Rightarrow (\phi; \Gamma)$ means that if p is given the type τ, then the index variables and **lam**-variables in p must have the sorts and types declared in ϕ and

$$\frac{}{x \downarrow \tau \Rightarrow (\cdot; x : \tau)}\ \textbf{(pat-var)} \qquad \frac{}{\langle\rangle \downarrow \mathbf{1} \Rightarrow (\cdot; \cdot)}\ \textbf{(pat-unit)}$$

$$\frac{p_1 \downarrow \tau_1 \Rightarrow (\phi_1; \Gamma_1) \quad p_2 \downarrow \tau_2 \Rightarrow (\phi_2; \Gamma_2)}{\langle p_1, p_2\rangle \downarrow \tau_1 * \tau_2 \Rightarrow (\phi_1, \phi_2; \Gamma_1, \Gamma_2)}\ \textbf{(pat-prod)}$$

$$\frac{\Sigma(c) = \Pi\vec{a} : \vec{\gamma}.\tau \to \delta_0(\vec{\imath}) \quad p \downarrow \tau \Rightarrow (\phi; \Gamma)}{c[\vec{a}](p) \downarrow \delta_0(\vec{\imath}') \Rightarrow (\vec{a} : \vec{\gamma}, \vec{\imath} = \vec{\imath}', \phi; \Gamma)}\ \textbf{(pat-constructor)}$$

$$\frac{p \downarrow \tau_1 \Rightarrow (\phi'; \Gamma') \quad \phi, \phi'; \Gamma, \Gamma' \vdash e : \tau_2}{\phi; \Gamma \vdash p \Rightarrow e : \tau_1 \Rightarrow \tau_2}\ \textbf{(type-clause)}$$

$$\frac{\phi; \Gamma \vdash p_i \Rightarrow e_i : \tau_1 \Rightarrow \tau_2 \ \text{ for } i = 1, \ldots, n}{\phi; \Gamma \vdash (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) : \tau_1 \Rightarrow \tau_2}\ \textbf{(type-clauses)}$$

$$\frac{\begin{array}{c}\Sigma(c) = \Pi\vec{a} : \vec{\gamma}.\tau \to \delta_0(\vec{\imath}) \\ \phi \vdash \vec{\imath}' : \vec{\gamma} \quad \phi; \Gamma \vdash e : \tau[\vec{a} \mapsto \vec{\imath}']\end{array}}{\phi; \Gamma \vdash c[\vec{\imath}'](e) : \delta_0(\vec{\imath}[\vec{a} \mapsto \vec{\imath}'])}\ \textbf{(type-constructor)}$$

$$\frac{\phi; \Gamma \vdash e : \tau_1 \quad \phi; \Gamma \vdash ms : \tau_1 \Rightarrow \tau_2}{\phi; \Gamma \vdash \textbf{case } e \textbf{ of } ms : \tau_2}\ \textbf{(type-case)}$$

$$\frac{\begin{array}{c}\Sigma(c) = \Pi\vec{a} : \vec{\gamma}.\tau \to \delta_0(\vec{\imath}) \\ \phi \vdash \vec{\imath}' : \vec{\gamma} \quad \phi; \Gamma \vdash e : \tau[\vec{a} \mapsto \vec{\imath}'] \ll_f \mu_0\end{array}}{\phi; \Gamma \vdash c[\vec{\imath}'](e) : \delta_0(\vec{\imath}[\vec{a} \mapsto \vec{\imath}']) \ll_f \mu_0}\ \textbf{(}\ll\textbf{-constructor)}$$

$$\frac{p \downarrow \tau_1 \Rightarrow (\phi'; \Gamma') \quad \phi, \phi'; \Gamma, \Gamma' \vdash e : \tau_2 \ll_f \mu_0}{\phi; \Gamma \vdash p \Rightarrow e : \tau_1 \Rightarrow \tau_2 \ll_f \mu_0}\ \textbf{(}\ll\textbf{-clause)}$$

$$\frac{\phi; \Gamma \vdash p_i \Rightarrow e_i : \tau_1 \Rightarrow \tau_2 \ll_f \mu_0 \quad \text{for } i = 1, \ldots, n}{\phi; \Gamma \vdash (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) : \tau_1 \Rightarrow \tau_2 \ll_f \mu_0}\ \textbf{(}\ll\textbf{-clauses)}$$

$$\frac{\phi; \Gamma \vdash e : \tau_1 \ll_f \mu_0 \quad \phi; \Gamma \vdash ms : \tau_1 \Rightarrow \tau_2 \ll_f \mu_0}{\phi; \Gamma \vdash \textbf{case } e \textbf{ of } ms : \tau_2 \ll_f \mu_0}\ \textbf{(}\ll\textbf{-case)}$$

Figure 9. Typing rules for tuples and pattern matching in $\mathrm{ML}_0^{\Pi,\Sigma}$.

$$\frac{}{v \downarrow x \Rightarrow ([\,]; [x \mapsto v])}\ \textbf{(match-var)}$$

$$\frac{}{\langle\rangle \downarrow \langle\rangle \Rightarrow ([\,]; [\,])}\ \textbf{(match-unit)}$$

$$\frac{v_1 \downarrow p_1 \Rightarrow (\theta_I^1; \theta_1) \quad v_2 \downarrow p_2 \Rightarrow (\theta_I^2; \theta_2)}{\langle v_1, v_2\rangle \downarrow \langle p_1, p_2\rangle \Rightarrow (\theta_I^1 \cup \theta_I^2; \theta_1 \cup \theta_2)}\ \textbf{(match-prod)}$$

$$\frac{v \downarrow p \Rightarrow (\theta_I; \theta)}{c[\vec{\imath}](v) \downarrow c[\vec{a}](p) \Rightarrow ([\vec{a} \mapsto \vec{\imath}] \cup \theta_I; \theta)}\ \textbf{(pat-constructor)}$$

Figure 10. The rules for matching values against patterns in $\mathrm{ML}_0^{\Pi,\Sigma}$.

Γ, respectively; if we treat p as an expression, then $\phi_0, \phi; \Gamma \vdash p : \tau$ is derivable, where we assume that τ is a well-formed type under ϕ_0.

We write $v \downarrow p \Rightarrow (\theta_I; \theta)$ to mean that the value v matches the pattern p and this matching generates an index substitution θ_I and a substitution θ. The rules for deriving such a judgment are listed in Figure 10. If $v \downarrow p \Rightarrow (\theta_I; \theta)$ holds for some clause $p \Rightarrow e$ in *ms* then **case** v **of** *ms* is a redex, which reduces to $e[\theta_I][\theta]$. Note that there may be a certain amount of nondeterminism in the evaluation of a redex of form **case** v **of** *ms*: if there are several clauses $p_i \Rightarrow e_i$ in *ms* such that v matches p_i, then any of them (finitely many) may be chosen for evaluating **case** v **of** *ms*. Notice that we can still claim by König's Lemma that for every e such that $e \downarrow$ holds, there exists a smallest number l such that the number of steps in each reduction sequence from e is at most l; as before, we use $l(e)$ for such a number.

In the case where no clause in *ms* matches v, the evaluation of **case** v **of** *ms* becomes *stuck*. Given a well-typed closed expression e, we now have three possibilities for a reduction sequence from e: it reaches some value, or it becomes stuck, or it proceeds forever. In the current setting, $e \downarrow$ means that every reduction sequences from e either reaches a value or becomes stuck.

With the presence of pattern matching, we can readily construct a nonterminating function without using **fun**. The following is such an example written in Standard ML [16].

```
datatype omega = D of omega -> int
val f: omega -> int = fn (x) => case x of D g => g(x)
```

Clearly, the evaluation of `f(D(f))` is nonterminating. Later, it will be clear that this is mainly caused by the negative occurrence of `omega` in the type of the argument of `D`. We thus need to impose some restrictions on user-defined datatypes in order to use types to guarantee program termination.

Definition 4.3. We define ground, positive and negative occurrences of δ_0 in a given type τ by structural induction on τ.

- The occurrence of δ_0 in $\delta_0(\vec{\imath})$ is both ground and positive.
- An occurrence of δ_0 in $\tau = \tau_1 * \tau_2$ is ground (positive, negative) if it is ground (positive, negative) in τ_1 or ground (positive, negative) in τ_2.

- No occurrences of δ_0 in $\tau = \tau_1 \to \tau_2$ are ground. An occurrence of δ_0 in τ is positive (negative) if it is negative (positive) in τ_1 or positive (negative) in τ_2.
- No occurrences of δ_0 in $\tau = \Pi\vec{a} : \vec{\gamma}.\tau_1$ are ground. An occurrence of δ_0 in τ is positive (negative) if it is positive (negative) in τ_1.
- An occurrence of δ_0 is ground (positive, negative) in $\tau = \Sigma a : \gamma.\tau_1$ if it is ground (positive, negative) in τ_1.

For example, in $\tau = (\delta_0(\vec{\imath}_1) \to \delta_0(\vec{\imath}_2)) \to \delta_0(\vec{\imath}_3)$, the first and the third occurrences are positive, the second is negative, and no occurrences of δ_0 are ground. In $\delta_0(\vec{\imath}_1) * \delta_0(\vec{\imath}_2)$, both occurrences of δ_0 are ground and positive, and no occurrences are negative. We impose the following condition on δ_0 for the rest of this section.

Condition 1. All occurrences of δ_0 in τ are *positive* whenever a constructor c associated with δ_0 has a type of the form $\Pi\vec{a} : \vec{\gamma}.\tau \to \delta_0(\vec{\imath})$.

Definition 4.4. For each natural number n, we define $[\tau]_n$ to be a set of values of type τ as follows, where the definition is inductive on n and the complexity of τ, lexicographically ordered.

1. τ is a base type that is not of the form $\delta_0(\vec{\imath})$. Then $[\tau]_n$ is the entire set of values of type τ.
2. $\tau = \delta_0(\vec{\imath})$. $[\tau]_n = \emptyset$ if $n = 0$. For $n > 0$, $[\tau]_n$ is the set of values v of type τ such that $v = c_k[\vec{\imath}'](v')$ for some $1 \leq k \leq m$ and $v' \in [\tau^k[\vec{a}_k \mapsto \vec{\imath}']]_{n-1}$.
3. $\tau = \tau_1 * \tau_2$. Then $[\tau]_n = \{\langle v_1, v_2\rangle \mid v_1 \in [\tau_1]_n \text{ and } v_2 \in [\tau_2]_n\}$
4. $\tau = \tau_1 \to \tau_2$. Then $[\tau]_n$ is the set of values v of type τ such that for each $v_1 \in [\tau_1]_n$, $v(v_1) \downarrow$ holds and if $v(v_1) \hookrightarrow^* v_2$ then $v_2 \in [\tau_2]_n$.
5. $\tau = \Pi\vec{a} : \vec{\gamma}.\tau_1$. Then $[\tau]_n$ is the set of values v of type τ such that for each $\vec{\imath} : \vec{\gamma}$, $v[\vec{\imath}] \hookrightarrow^* v_1$ holds for some $v_1 \in [\tau_1[\vec{a} \mapsto \vec{\imath}]]_n$. Notice that the definition is given in such a manner because v must be of the form $\lambda\vec{a} : \vec{\gamma}.v'$ for some value v'.
6. $\tau = \Sigma a : \gamma.\tau_1$. Then $[\tau]_n$ is the set of values of form $\langle i \mid v_1\rangle$ such that $v_1 \in [\tau_1[a \mapsto i]]_n$.

We say that a type τ is positive (negative) if all occurrences of δ_0 in τ are positive (negative).

Lemma 4.5. *We have the following.*
1. *If τ is positive, then $[\tau]_n \subseteq [\tau]_{n+1}$ for all natural numbers n.*
2. *If τ is negative, then $[\tau]_n \supseteq [\tau]_{n+1}$ for all natural numbers n.*

Proof: (1) and (2) are proved simultaneously by induction on n and the complexity of τ, lexicographically ordered. We present one interesting case for (1).

- $\tau = \delta_0(\vec{\imath})$. If $n = 0$, then $[\tau]_n = \emptyset \subseteq [\tau]_{n+1}$. Assume $n > 0$ and $v \in [\tau]_n$. Then $v = c_k[\vec{\imath}']$ (v') for some $1 \leq k \leq m$ and $v' \in [\tau^k[\vec{a}_k \mapsto \vec{\imath}']]_{n-1}$. Note that τ^k is positive by Condition 1. Hence, $\tau^k[\vec{a}_k \mapsto \vec{\imath}']$ is also positive. By induction hypothesis, $[\tau^k[\vec{a}_k \mapsto \vec{\imath}']]_{n-1} \subseteq [\tau^k[\vec{a}_k \mapsto \vec{\imath}']]_n$, which implies that $v \in [\tau]_{n+1}$. Therefore, $[\tau]_n \subseteq [\tau]_{n+1}$.

The rest of the cases can be treated similarly. □

Definition 4.6 (Reducibility). Suppose that e is a closed expression of type $\delta_0(\vec{\imath})$. We say that e is reducible if $e \downarrow$ holds and for each v satisfying $e \hookrightarrow^* v$, $v \in [\delta_0(\vec{\imath})]_n$ holds for some $n \geq 0$.

The notion of reducibility can now be defined for types containing occurrences of δ_0 by following Definition 3.5. As can be expected, Proposition 3.7 still holds for this new definition of reducibility. Our main task is to establish the following lemma, which is precisely Lemma 3.9 for the new definition of reducibility.

Lemma 4.7. *Assume that* $\phi; \Gamma \vdash e : \tau$ *is derivable and* $\cdot; \cdot \vdash (\theta_I; \theta) : (\phi; \Gamma)$ *holds. Also assume that* (1) *for every* $x \in \mathbf{dom}(\Gamma)$, $\theta(x)$ *is reducible, and* (2) *for every* $f \in \mathbf{dom}(\Gamma)$, $\cdot; \Gamma[\theta_I] \vdash e[\theta_I] : \tau[\theta_I] \ll_f \mu_f$ *is derivable for some* μ_f *such that* $\theta(f)$ *is* 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 $\mathcal{D}$. The proof is of great similarity to that of Lemma 3.9. All of the existing cases from the proof of Lemma 3.9 can be reused here essentially without changes. This leaves just the cases dealing with rules **(type-case)** and **(type-constructor)** for datatype elimination and introduction.

First, we need the following lemmas for handing the typing rule **(type-case)**. □

Lemma 4.8. *We have the following.*
1. *Assume that* τ *is positive. Then for every* n, *all values* v *in* $[\tau]_n$ *are reducible.*
2. *Assume that* τ *is negative. Then for every* n, *all reducible values of type* τ *are in* $[\tau]_n$.

Proof: (1) and (2) are proved simultaneously by induction on n and the complexity of τ, lexicographically ordered. We first present some cases for (1).

- $\tau = \delta_0(\vec{\imath})$. By the definition of reducibility.
- $\tau = \tau_1 \rightarrow \tau_2$ and $v \in [\tau]_n$. Assume that $v_1 \in [\tau_1]_n$. Then $v(v_1) \downarrow$ holds by definition. Assume that $v(v_1) \hookrightarrow^* v_2$. Then $v_2 \in [\tau_2]_n$ holds by definition. By induction hypothesis, v_2 is reducible. Hence, we have that $v(v_1)$ is reducible by Proposition 3.7 (3). Since τ is positive, τ_1 must be negative. By induction hypothesis, all reducible values of type τ_1 are in $[\tau_1]_n$. Hence v is reducible.

We now present a case for (2).

- $\tau = \tau_1 \rightarrow \tau_2$. Assume that v is reducible of type τ. Also assume that $v_1 \in [\tau_1]_n$. By induction hypothesis, we have that all values in $[\tau_1]_n$ are reducible of type τ_1 since τ_1 is positive. Hence, v_1 is reducible. By the definition of reducibility, $v(v_1)$ is reducible of type τ_2. By induction hypothesis, $[\tau_2]_n$ contains all reducible values of type τ_2 since τ_2 is negative. Therefore, $v(v_1)$ is in $[\tau_2]_n$. By definition, v is in $[\tau]_n$. □

Lemma 4.9. *Let* c_k *be a constructor associated with* δ_0. *If* $c_k[\vec{\imath}](v)$ *is reducible, then* v *is also reducible.*

Proof: Note that the type of c_k is $\Pi\vec{a}_k : \vec{\gamma}_k.\tau^k \rightarrow \delta_0(\vec{\imath}_k)$. By the definition of reducibility, $c_k[\vec{\imath}](v)$ is in $[\delta_0(\vec{\imath}')]_{n+1}$ for $\vec{\imath}' = \vec{\imath}_k[\vec{a}_k \mapsto \vec{\imath}]$ and some n. This implies that $v \in [\tau^k[\vec{a}_k \mapsto \vec{\imath}]]_n$. Therefore, v is reducible by Lemma 4.8 (1) as all occurrences of δ_0 in τ^k are positive. □

Proof of Lemma 4.7 (continued) We are now ready to handle the typing rule **(type-case)** when proving Lemma 4.7 in the presence of pattern matching.

– Assume that $\mathcal{D}$ is of the following form, where $e = \textbf{case}\ e_0\ \textbf{of}\ ms$ and $\tau = \tau_2$.

$$\frac{\phi; \Gamma \vdash e_0 : \tau_1 \quad \phi; \Gamma \vdash ms : \tau_1 \Rightarrow \tau_2}{\phi; \Gamma \vdash \textbf{case}\ e_0\ \textbf{of}\ ms : \tau_2}\ \textbf{(type-case)}$$

Let $e^* = e[\theta_I][\theta]$, $e_0^* = e_0[\theta_I][\theta]$ and $ms^* = ms[\theta_I][\theta]$. Assume that $e^* \hookrightarrow e'$. By induction hypothesis, e_0^* is reducible. Hence, $e_0^* \downarrow$. We show that e' is reducible by induction on $l(e_0^*)$, that is, the number of steps in a longest reduction sequence from e_0^*. We have two possibilities.

- $e_0^* \hookrightarrow e_0'$ and $e' = \textbf{case}\ e_0'\ \textbf{of}\ ms^*$. Since $l(e_0') < l(e_0^*)$, e' is reducible by induction hypothesis.
- e_0^* is a value and $e' = e_1^*[\theta_I'][\theta']$, where $e_0^* \downarrow p_1 \Rightarrow (\theta_I'; \theta')$ and $e_1^* = e_1[\theta_I][\theta]$ for some clause $p_1 \Rightarrow e_1$ in ms. Then we have the following derivation inside $\mathcal{D}$.

$$\frac{p_1 \downarrow \tau_1 \Rightarrow (\phi'; \Gamma') \quad \mathcal{D}_2 :: \phi, \phi'; \Gamma, \Gamma' \vdash e_1 : \tau_2}{\phi; \Gamma \vdash p_1 \Rightarrow e_1 : \tau_1 \Rightarrow \tau_2}\ \textbf{(type-clause)}$$

As can be expected, we can show $\cdot; \cdot \vdash (\theta_I \cup \theta_I'; \theta \cup \theta') : (\phi, \phi'; \Gamma, \Gamma')$. By Lemma 4.9, we can also show that $\theta'(x)$ is reducible for each $x \in \textbf{dom}(\Gamma')$. By induction hypothesis on $\mathcal{D}_2$, we know that e' is reducible.

By Proposition 3.7 (2), e^* is reducible.

It is more complicated to handle the typing rule **(type-constructor)**, since the converse of Lemma 4.9 does not hold under the current definition of reducibility. We can, however, show the following, weaker lemmas:

Lemma 4.10. *Assume that all occurrences of δ_0 in τ are ground. Then for every reducible value v of type τ, $v \in [\tau]_n$ for some $n \geq 0$.*

Proof: The proof proceeds by induction on the complexity of τ. We present some cases.

– $\tau = \delta_0(\vec{\imath})$. By the definition of reducibility.
– $\tau = \tau_1 * \tau_2$. Then all occurrences of δ_0 in τ_1 and τ_2 are ground. Since v is of type τ, $v = \langle v_1, v_2 \rangle$ for some values v_1 and v_2 of types τ_1 and τ_2, respectively. By induction hypothesis, there are natural numbers n_1 and n_2 such that $v_1 \in [\tau_1]_{n_1}$ and $v_2 \in [\tau_2]_{n_2}$. Let $n = \max(n_1, n_2)$. By Lemma 4.5, we have $v \in [\tau]_n$.
– $\tau = \tau_1 \rightarrow \tau_2$. Then there are no occurrences of δ_0 in either τ_1 or τ_2. By definition, we know that $[\tau]_n$ is the entire set of reducible values of type τ for every n. Hence, we are done.

The rest of the cases can be handled similarly. □

Lemma 4.11. *Let c_k be a constructor associated with δ_0 and its type be $\Pi\vec{a}_k : \vec{\gamma}_k.\tau^k \to \delta_0(\vec{\imath}_k)$. Assume that all occurrences of δ_0 in τ^k are ground. If v is reducible and $c_k[\vec{\imath}](v)$ is well-typed, then $c_k[\vec{\imath}](v)$ is reducible.*

Proof: By Lemma 4.10, we have $v \in [\tau^k[\vec{a}_k \mapsto \vec{\imath}]]_n$ for some n. By definition, $c_k[\vec{\imath}](v) \in [\delta_0(\vec{\imath}_k[\vec{a}_k \mapsto \vec{\imath}])]_{n+1}$. Hence, $c_k[\vec{\imath}](v)$ is reducible. □

Therefore, we are motivated to impose the following condition when extending $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ with pattern matching:

Condition 2. All occurrences of δ_0 in τ are *ground* whenever a constructor c associated with δ_0 has a type of the form $\Pi\vec{a} : \vec{\gamma}.\tau \to \delta_0(\vec{\imath})$.

Proof of Lemma 4.7 (continued): We are now ready to handle the typing rule **(type-constructor)**.

– Assume that the derivation $\mathcal{D}$ ends with an application of the typing rule **(type-constructor)**, where $e = c[\vec{\imath}_1](e_1)$ and $\tau = \delta_0(\vec{\imath}[\vec{a} \mapsto \vec{\imath}_1])$.

$$\frac{\Sigma(c) = \Pi\vec{a} : \vec{\gamma}.\tau_1 \to \delta_0(\vec{\imath}) \quad \phi \vdash \vec{\imath}_1 : \vec{\gamma} \quad \mathcal{D}_1 :: \phi; \Gamma \vdash e_1 : \tau_1[\vec{a} \mapsto \vec{\imath}_1]}{\phi; \Gamma \vdash c[\vec{\imath}_1](e_1) : \delta_0(\vec{\imath}[\vec{a} \mapsto \vec{\imath}_1])}$$

By induction hypothesis on $\mathcal{D}_1$, we know that $e_1^* = e_1[\theta_I][\theta]$ is reducible. Therefore, $e[\theta_I][\theta] = c[\vec{\imath}_1[\theta_I]](e_1^*)$ is reducible by Lemma 4.11 if all occurrences of δ_0 in τ_1 are ground.

This concludes the proof of Lemma 4.7. □

We feel that imposing Condition 2 on datatypes is practically adequate. For instance, all the examples of datatypes we present in the paper satisfy Condition 2. However, there are also many realistic examples that do not satisfy this condition. For instance, it rules out the following datatype for lazy lists:

```
datatype 'a lazyList =
  NIL | CONS of unit -> 'a * 'a lazyList
```

We can actually drop Condition 2 if we employ transfinite induction,[4] which is amply explained in [3]. We sketch the idea as follows.

In Definition 4.4, we can make the definition inductive on ordinals instead of on natural numbers. Given a positive (negative) type τ; for a limit ordinal α, we define $[\tau]_\alpha$ as the union (intersection) of $[\tau]_\beta$ for all $\beta < \alpha$; for a successor ordinal α, $[\tau]_\alpha$ is defined in the same manner as is in the case of natural numbers. We define $[\tau]$ as the union (intersection) of $[\tau]_\alpha$ for all ordinals α. Note that $[\tau]$ is clearly a set, as it is a subset of all the closed values of type τ. We then modify the definition of reducibility at δ_0 (Definition 4.6) by now requiring that every value that e can evaluate to, belong to $[\delta_0(\vec{\imath})]_\alpha$ for some ordinal α. We can now prove the following lemma, using the observation that for every positive or negative type τ, $[\tau] = [\tau]_\alpha$ for some ordinal α.

Lemma 4.12. *Given a positive or negative type τ, $[\tau]$ is the set of all reducible values of type τ.*

Proof: This follows from induction on the complexity of τ. □

With Condition 1, we can now show that a well-typed value of the form $c[\vec{\imath}](v)$ is reducible if and only if v is reducible, and then prove Lemma 4.7.

We anticipate that the strategy for defining reducibility for a single recursively defined datatype can be extended to handle mutually recursively defined datatypes in a straightforward manner. Suppose that $\delta_1, \dots, \delta_n$ are mutually recursively defined. Then the requirement is that for $m = 1, \dots, n$, the type $\Pi\vec{a} : \vec{\gamma}.\tau \to \delta_m(\vec{\imath})$ of each constructor c associated with δ_m should satisfy the property that τ contains no negative of occurrences of either of $\delta_1, \dots, \delta_n$.

Lastly, we mention that the above strategy for defining reducibility for datatypes is incremental. Suppose that we now declare a new type constructor δ'_0. As the notion of reducibility has been defined for types of the form $\delta_0(\vec{\imath})$, we can treat these types as base types when we define the notion of reducibility for types of form $\delta'_0(\vec{\imath})$.

4.4. Polymorphism

We can readily extend $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ with first-order polymorphism, which encompasses let-polymorphism supported in DML.

type variables	α
types	$\tau ::= \cdots \mid \alpha$
type schemes	$\sigma ::= \tau \mid \forall\alpha.\sigma$
expressions	$e ::= \cdots \mid \Lambda\alpha.v \mid e[\tau]$
values	$v ::= \cdots \mid \Lambda\alpha.v$

A context Γ now maps variables to type schemes. A type judgment is of form $\vec{\alpha}; \phi; \Gamma \vdash e : \sigma$, where all free type variables in Γ and σ are declared in $\vec{\alpha}$. It should be clear how to modify each previous typing rule to accommodate type variables. We present the typing rules for handling type abstraction and application as follows.

$$\frac{\vec{\alpha}, \alpha; \phi; \Gamma \vdash v : \sigma}{\vec{\alpha}; \phi; \Gamma \vdash \Lambda\alpha.v : \forall\alpha.\sigma}\ \textbf{(type-tlam)}$$

$$\frac{\vec{\alpha}; \phi \vdash \tau\,\textbf{[well-formed]} \quad \vec{\alpha}; \phi; \Gamma \vdash e : \forall\alpha.\sigma}{\vec{\alpha}; \phi \vdash e[\tau] : \sigma[\alpha \mapsto \tau]}\ \textbf{(type-tapp)}$$

Definition 4.13 (Reducibility). For each type scheme σ, the complexity of σ is the number of $\forall$ occurring in σ. The notion of reducibility for σ can be defined inductively on the complexity of σ.

– $\sigma = \tau$. The notion of reducibility for type τ is defined as before.
– $\sigma = \forall\alpha.\sigma_1$. Assume that e is an expression of type scheme σ. Then e is reducible if $e \downarrow$ holds and $e[\tau]$ is reducible for every closed type τ. This is a valid definition since $e[\tau]$ is of type scheme $\sigma_1[\alpha \mapsto \tau]$, whose complexity is less than that of σ as τ does not contain any occurrences of $\forall$.

We see no difficulty in extending $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ to encompass second-order polymorphic lambda-calculus, that is, System $\mathcal{F}$ [10]. For such an extension, we expect to use the notion of reducibility candidates [9] for proving that every well-typed closed expression in this extended system is reducible. However, we have so far not carried out such an extension in either theory or practice.

4.5. Effects

When extending $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ with exceptions, all we essentially need is to claim that the value carried by an exception is reducible whenever the exception is captured. Therefore, we see no difficulty in extending $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ with exceptions that carry no values or values of ground types since all such values are reducible by Proposition 3.6. However, the following example indicates that a nonterminating function can be readily defined without using recursion when exceptions carrying functions are allowed.

```
exception D of (unit -> unit) -> unit
val f = fn x => x () handle D g => g x
(* the expression f (fn () => raise D f) loops forever *)
```

Similarly, there seems no difficulty in handling references to values of base types or user-defined datatypes by assuming that all memory locations for storing such values are reducible. The problem is with references to functions, which, if allowed, could easily lead to the construction of nonterminating programs (without using recursion) as is shown in the following example in Standard ML.

```
val r: (int -> int) ref = ref (fn x => x)

(* now f is a terminating function *)
fun f(x: int): int = !r (x)

(* now f becomes a nonterminating function *)
val _ = r := f
```

We currently do not know what sensible restrictions are needed for handling either exceptions carrying or references to values that are not of ground types. Thus, we disallow such exceptions or references in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$.

5. Practice

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

5.1. Examples

We use some examples to demonstrate how various programming features are handled in practice by our approach to program termination verification. All these examples have been verified in a prototype implementation of DML, which is available at [25].

5.1.1. Primitive recursion. The following is an implementation of the primitive recursion operator R in Gödel's $\mathcal{T}$, which is clearly typable in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$.

```
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 stands for the type [n:nat] Nat(n) *)
```

Note that Z and S are assigned the following types after the datatype declaration.

$$\Sigma(Z) = \mathit{Nat}(0)$$
$$\Sigma(S) = \Pi n : \mathit{nat}.\mathit{Nat}(n) \to \mathit{Nat}(n+1)$$

By Theorem 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 proved terminating with a structural ordering. The point we make is that though it seems "evident" that the use of R cannot cause nontermination, it is not trivial at all to prove that 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 [22]. Actually, every term in $\mathcal{T}$ is strongly normalizing, that is, there is no infinite reduction sequence from the term even if a redex is allowed to be reduced in any context (not necessarily in an evaluation context), but this obviously is untrue in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$.

5.1.2. Nested recursive function call. The program in Figure 11 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 $\phi \models \max(0, 101 - (i + 11)) < \max(0, 101 - i)$ holds for $\phi = (i : \mathit{int}, i \leq 100)$, 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 $(\phi, j : \mathit{int}, P)$

```
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 11. An implementation of McCarthy's "91" function.

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.

5.1.3. Mutual recursion. The program in Figure 12 implements quicksort on a list, where the functions *qs* and *par* are defined mutually recursively. We informally explain why this program is typable in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ and thus *qs* is a terminating function by Theorem 3.10. Note that the list constructors [] and :: have been given the following types, where $(\alpha)list(n)$ is a type for lists of length n in which each element has type α.

$$\Sigma([]) = \forall\alpha.(\alpha)list(0)$$
$$\Sigma(::) = \forall\alpha.\Pi a : nat.\alpha * (\alpha)list(a) \to (\alpha)list(a + 1)$$

```
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 12. An implementation of quicksort on a list.

For the call to *par* in the body of *qs*, the label is $\langle 0+0+a, a+1\rangle$, where a is the length of xs'. So we need to verify that $\phi \models \langle 0+0+a, a+1\rangle < \langle n, 0\rangle$ holds 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 $\phi \models \langle p, 0\rangle < \langle p+q+r, r+1\rangle$ and $\phi \models \langle q, 0\rangle < \langle p+q+r, r+1\rangle$ 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 $\phi \models \langle (p+1)+q+a, a+1\rangle < \langle p+q+r, r+1\rangle$ and $\phi \models \langle p+(q+1)+a, a+1\rangle < \langle p+q+r, r+1\rangle$ 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.

5.1.4. Higher-order function. The code in Figures 13 and 14 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.[5]

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 calls 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 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 $\Pi a' : \gamma.(char)list(a') \to$ `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 $i' < i$[6] 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 Theorem 3.10, *acc* is terminating, which implies that *accept* is

```
datatype pattern with nat =

  Empty(1) (* 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 *)
```

Figure 13. An implementation of pattern matching on strings (1).

```
(* 'length' computes the length of a list *)
(* empty tuple <> is used for proving that 'length' is
   terminating since 'length' is not recursive *)
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)

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)

   (* in this case, k is used for backtracking *)
  | Plus(p1, p2) =>
    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 14. An implementation of pattern matching on strings (2).

terminating (assuming *explode* is terminating). In every aspect, this is a nontrivial 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 [12], there is an elegant implementation of *accept* that does some processing on the pattern to be matched and then eliminates the test. We have also proved that this example is terminating [27].

5.1.5. Run-time check. Given the limitation of the type system of DML, there are certainly many cases where termination depends on a program invariant that cannot (or is difficult to) be captured in DML. For instance, the following program in DML implements the function that computes the greatest common divisor of two natural numbers, where *mod* is the usual modulo operator.

```
fun gcd (m, n) = if m = 0 then n else gcd (n mod m, m)
withtype {a:nat,b:nat} <a> => int(a) * int(b) -> int
```

Unfortunately, we currently cannot verify that the implementation is well-typed in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ because it requires that the following nonlinear constraints be solved.

$$a : nat, b : nat, a \neq 0 \models b\ mod\ a < a$$
$$a : nat, b : nat, a \neq 0 \models 0 \leq b\ mod\ a$$

However, we can insert some run-time checks as follows and then verify that the implementation is well-typed and thus terminating.

```
fun gcd' (m, n) =
  if m = 0 then n
  else let
           val m' = (n mod m: int)
       in
           if 0 <= m' andalso m' < m then gcd' (m', m)
           else raise Impossible
       end
withtype {a:nat,b:nat} <a> => int(a) * int(b) -> int
```

Though we can readily strengthen the constraint solver for DML to verify the termination of the `gcd` function, this example does indicate that we can insert run-time checks to verify program termination, sometimes, approximating a liveness property with a safety property. A particularly interesting example is an implementation of the Knuth-Morris-Pratt string matching algorithm, which is proven terminating with the insertion of some run-time checks [27].

Though using run-time checks to verify program termination may slow down program execution, it offers the programmer opportunities to capture program errors that would otherwise cause nonterminating program execution. In this respect, it is similar to using run-time array bound checks to capture program errors that would otherwise lead to meaningless or even harmful program execution such as security breaching.

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.

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, or more precisely, the reducibility, 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. In particular, the reader can find numerous realistic examples at [27] in support of our claim.

Currently, we have integrated $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ into DML. However, there are still many remaining problems. We have not implemented the notion of termination cast. It is completely up to the programmer as to whether a metric is to be provided; if it is provided, then the type-checker can use it to verify program termination and thus may report some potential program errors when the verification fails; otherwise, no termination verification is performed and the function is assumed to be terminating. In this respect, the integration of $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ into DML resembles imposing a soft type system onto an untyped language.

In the presence of nonterminating functions, there is clearly a need for distinguishing between functions that are not expected to always terminate, and those whose termination is expected but perhaps not (yet) proved; for instance, such a distinction can enable the programmer to identify a (suspicious) use of a (pontentially) nonterminating function in the definition of a terminating function. When run-time debugging is concerned, there is also clearly a need for distinguishing functions whose termination is proved, and those whose termination is expected but perhaps not (yet) proved; for instance, such a distinction can help the programmer identify the cause of (seemingly) nonterminating computation. We expect to study how to make such distinctions in a effective manner in a future implementation.

6. Potential applications

The main motivation behind our work is to provide a practical mechanism for facilitating program termination verification. As a consequence, it can help the programmer detect more program errors and thus enhance program robustness. In this section, we mention some other potential applications of our approach.

6.1. Resource bound inference

We feel that the types in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ can be of great use for inferring the time and/or space complexity of a (first-order) function. It is shown in [11] that cost recurrence equations can be automatically extracted from a first-order DML program. There, it is also suggested that such automatic extraction should benefit significantly from the presence of metrics in type annotations. Therefore, we expect that the types in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ are to be used in resource bound inference for functional programs.

6.2. Resource bound certification

The notion of proof-carrying code (PCC) provides a means to certifying a property of low-level code by attaching an independently verifiable proof proving that the code indeed possesses the property [17]. A type system for low-level code is presented in [6] for resource bound certification, and the programmer is required to write cost functions at source-level for generating such low-level code. Therefore, a (challenging) question is whether a compiler can be built to translate the types in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$ into a proof that certifies resource bounds for the code compiled from the DML programs. In particular, it is interesting to see whether the termination of low-level code can be thus certified.

7. 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 [23], there is also a large amount of work on proving termination of logic programs. In [21], 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 [20], where it checks whether a given structural ordering (possibly on higher-order terms) is decreasing in an inductive proof or a logic program. Clearly, approaches based on checking complement those based on synthesis.

Our approach also relates to the semantic labelling approach [30] 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.

A notion of sized types is introduced in [13] for proving the correctness of reactive systems. There, the type system is capable of guaranteeing the termination of well-typed programs. An approach is also presented in [4] to infer sized types for functional programs. The language presented in [13], which is designed for embedded functional programming, contains a significant restriction as it only supports (a minor variant of) primitive recursion on sized type parameters, 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 size of data in [13] is uniformly defined as the height of the tree representation of the data while we allow the user to define size for data of recursive datatypes. In addition, the notion of existential dependent types, which we deem indispensable in practical programming, does not exist in [13].

It is also observed in [11] that the dependent types in DML can be used to encode a notion of input size. There, an algorithm is presented to extract cost recurrence equations from the first-order programs in DML. With the availability of metrics in $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$, we expect that the extracted equations can be readily verified to be recurrences for general recursion.

When compared to various (interactive) theorem proving systems such as NuPrl [5], Coq [8], Isabelle [19] and PVS [18], 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.

8. 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 [28] that dependent types in DML can guarantee that every well-typed program in DML possesses such a safety property, effectively facilitating run-time 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 functional programming, such information may allow a compiler to decide whether a thunk should be formed. In future, we hope 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. In particular, the recent work on size-change analysis [15] presents a simple and general approach to automatic program termination verification. Therefore, it seems promising 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.

A probably more important research question, which, unfortunately, is often ignored in most studies on or related to termination, is to study how program termination verification can be performed effectively in the presence of nonterminating functions. For instance, when implementing an interpreter for a simple imperative programming language, one may want to verify that the actual implementation captures the invariant that a program terminates if all the loops in the program terminate. We believe that this is a rather difficult question that needs to be properly addressed in future with both theoretical justifications and practical concerns.

Acknowledgments

I thank Chad Brown for providing me with his comments on a draft of the paper and Thorsten Altenkirch for explaining to me how the notion of reducibility can be formed for user-defined datatypes. I also thank the anonymous referees for their voluminous constructive comments, which have undoubtedly raised the quality of the paper significantly.

Notes

1. There is an implementation of the Ackermann function that involves only primitive recursion (of higher types) and can thus be easily proved terminating, but the point we drive here is that this particular implementation can be proved terminating with our approach.
2. A constraint is non-linear if contains a non-linear term such as $a_1 * a_2$ for some index variables a_1 and a_2.
3. The erasure $|$**open** e_1 **as** $\langle a \mid x \rangle$ **in** $e_2|$ is $(\lambda x : \tau.|e_2|)|e_1|$, where τ is the type of $|e_1|$, or **let** $x = |e_1|$ **in** $|e_2|$ **end** when let-expressions are introduced.
4. It also seems highly likely that we can drop Condition 2 by adopting the strategy in [1], which leads to a predicative strong normalization proof for a λ-calculus with interleaving inductive types.
5. The author learned this implementation from Frank Pfenning.
6. Note that *length*(cs') and *length*(cs) have the types `int`(i') and `int`(i), respectively, and thus *length*$(cs') =$ *length*(cs) has the type `bool`$(\mathit{if}(i' = i, 1, 0))$. Thus, $i' < i$ can be inferred *in the type system* of $\mathrm{ML}_{0,\ll}^{\Pi,\Sigma}$.

References

1. Abel, A. and Altenkirch, T. A predicative strong normalisation proof for a λ-calculus with interleaving inductive types. In *Proceedings of International Workshop on Types for Proof and Programs (TYPES '99)*, T. Coquand, P. Dybjer, B. Nordström, and J. Smith (Eds.). LNCS, Vol. 1956, Springer-Verlag, 2000, pp. 21–40.
2. BenCherifa, A. and Lescanne, P. Termination of rewriting systems by polynomial interpretations and its implementation. *SCP*, **9**(2) (1987) 137–160.
3. Chang, C.C. and Keisler, H.J. *Model Theory*, Studies in Logic and Mathematical Foundations, Vol. 73. North-Holland, Amsterdam, The Netherlands, 1977.
4. Chin, W.-N. and Khoo, S.-C. Calculating sized types. *Higher-Order and Symbolic Computation*, **14**(2/3), to appear.
5. Constable, R.L. et al. *Implementing Mathematics with the NuPrl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
6. Crary, K. and Weirich, S. Resource bound certification. In *Proceedings of 27th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2000)*, Boston, 2000, pp. 184–198.

7. Dershowitz, N. Orderings for term rewriting systems. *Theoretical Computer Science*, **17**(3) (1982) 279–301.
8. Dowek, G., Felty, A., Herbelin, H., Huet, G., Murthy, C., Parent, C., Paulin-Mohring, C., and Werner, B. The Coq proof assistant user's guide. Rapport Technique 154, INRIA, Rocquencourt, France. Version 5.8. 1993.
9. Girard, J.-Y. Interprétation Fonctionnelle et Élimination des Coupures dans l'Arithmétique d'Ordre Supérieur. Thèse de doctorat d'état, Université de Paris VII, Paris, France, 1972.
10. Girard, J.-Y., Lafont, Y., and Taylor, P. *Proofs and Types*, Cambridge Tracts in Theoretical Computer Science, Vol. 7, Cambridge University Press, Cambridge, England, 1989.
11. Grobauer, B. Cost recurrences for DML programs. In *Proceedings of Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*, Florence, Italy, 2001, pp. 253–264.
12. Harper, R. Proof-directed debugging. *Journal of Functional Programming*, **9**(4) (1999) 471–477.
13. Hughes, J., Pareto, L., and Sabry, A. Proving the correctness of reactive systems using sized types. In *Conference Record of 23rd ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '96)*, 1996, pp. 410–423.
14. Jouannaud, J.-P. and Rubio, A. The higher-order recursive path ordering. In *Proceedings of 14th IEEE Symposium on Logic in Computer Science (LICS '99)*, Trento, Italy, 1999, pp. 402–411.
15. Lee, C.S., Jones, N.D., and Ben-Amram, A.M. The size-change principle for program termination. In *Proceeding of the 28th ACM Symposium on Principles of Programming Languages (POPL '01)*, London, UK, 2001, pp. 81–92.
16. Milner, R., Tofte, M., Harper, R.W., and MacQueen, D. *The Definition of Standard ML (Revised).* MIT Press, Cambridge, Massachusetts, 1997.
17. Necula, G. Proof-carrying code. In *Conference Record of 24th Annual ACM Symposium on Principles of Programming Languages*, ACM Press, Paris, France, 1997, pp. 106–119.
18. Owre, S., Rajan, S., Rushby, J., Shankar, N., and Srivas, M. PVS: Combining specification, proof checking, and model checking. In *Proceedings of the 8th International Conference on Computer-Aided Verification (CAV '96)*, R. Alur and T.A. Henzinger (Eds.). New Brunswick, NJ, LNCS, Vol. 1102, Springer-Verlag, 1996, pp. 411–414.
19. Paulson, L. *Isabelle: A Generic Theorem Prover*. LNCS, Vol. 828, Springer-Verlag, 1994.
20. Pientka, B. and Pfenning, F. Termination and reduction checking in the logical framework. In *Proceedings of Workshop on Automation of Proofs by Mathematical Induction*, Pittsburgh, PA, 2000.
21. Speirs, C., Somogyi, Z., and Søndergaard, H.Termination Analysis for Mercury. In *Proceedings of the 4th Static Analysis Symposium (SAS '97)*, P.V. Hentenryck (Ed.). Paris, France, LNCS, Vol. 1302, Springer-Verlag, 1997, pp. 157–171.
22. Tait, W.W. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, **32**(2) (1967) 198–212.
23. Ullman, J.D. and Van Gelder, A. Efficient tests for top-down termination of logic rules. *Journal of the ACM*, **35**(2) (1988) 345–373.
24. Xi, H. Dependent types in practical programming. Ph.D. Thesis, Carnegie Mellon University, 1998, pp. viii+189. Available as `http://www.cs.cmu.edu/~hwxi/DML/thesis.ps`.
25. Xi, H. Dependent ML, 1999. Available at `http://www.cs.bu.edu/~hwxi/DML/DML.html`.
26. Xi, H. Dependently typed data structures. In *Proceedings of Workshop on Algorithmic Aspects of Advanced Programming Languages*, Paris, France, 1999, pp. 17–33.
27. Xi, H. Dependent types for program termination verification, 2000. Available as `http://www.cs.bu.edu/~hwxi/DML/Term`.
28. Xi, H. and Pfenning, F. Eliminating array bound checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montréal, Canada, 1998, pp. 249–257.
29. Xi, H. and Pfenning, F. Dependent types in practical programming. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, San Antonio, Texas, 1999, pp. 214–227.
30. Zantema, H. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae*, **24** (1995) 89–105.