Dependent Types in Practical Programming*

(Extended Abstract)

Hongwei Xi
Department of Computer Science and Engineering
Oregon Graduate Institute of Science and Technology
hongwei@cse.ogi.edu

Frank Pfenning
Department of Computer Science
Carnegie Mellon University
fp@cs.cmu.edu

Abstract

We present an approach to enriching the type system of ML with a restricted form of dependent types, where type index objects are drawn from a constraint domain C, leading to the DML(C) language schema. This allows specification and inference of significantly more precise type information, facilitating program error detection and compiler optimization. A major complication resulting from introducing dependent types is that pure type inference for the enriched system is no longer possible, but we show that type-checking a sufficiently annotated program in DML(C) can be reduced to constraint satisfaction in the constraint domain C. We exhibit the unobtrusiveness of our approach through practical examples and prove that DML(C) is conservative over ML. The main contribution of the paper lies in our language design, including the formulation of type-checking rules which makes the approach practical. To our knowledge, no previous type system for a general purpose programming language such as ML has combined dependent types with features including datatype declarations, higher-order functions, general recursions, let-polymorphism, mutable references, and exceptions. In addition, we have finished a prototype implementation of DML(C) for an integer constraint domain C, where constraints are linear inequalities (Xi and Pfenning 1998).

1 Introduction

Type systems for functional languages can be broadly classified into those for rich, realistic languages such as Standard ML (Milner, Tofte, and Harper 1990), Caml (Weis and Leroy 1993), or Haskell!(Hudak, Peyton Jones, and Wadler 1992), and those for small, pure languages such as the ones underlying Coq (Dowek, Felty, Herbelin, Huet, Murthy, Parent, Paulin-Mohring, and Werner 1993), NuPrl (Constable et al. 1986), or PX (Hayashi and Nakano 1988). Type-checking in realistic languages should be theoretically decidable and must be practically feasible without requiring large amounts of type annotations. In order to achieve this, the type systems are relatively simple and only elementary properties of programs can be expressed and thus checked by a compiler. For instance, the error of taking the first element out of an empty list cannot be detected by the type system of ML since it does not distinguish an empty list from a non-empty one. Richer type theories such as the Calculus of Inductive Constructions (underlying Coq) or Martin-Löf type theories (underlying NuPrl) allow full specifications to be formulated, which means that type-checking becomes undecidable or requires excessively verbose annotations. It also constrains the underlying functional language to remain relatively pure, so that it is possible to effectively reason about program properties within a type theory.

Some progress has been made towards bridging this gap, for example, by extracting Caml programs from Coq proofs, by synthesizing proof skeletons from Caml programs (Parent 1995), or by embedding fragments of ML into NuPrl (Kreitz, Hayden, and Hickey 1998). In this paper, we address the issue of designing a type system for practical programming in which a restricted form of dependent types is available, allowing more program invariants to be captured by types. We conservatively refine the type system of ML by allowing some dependencies, without destroying desirable properties of ML such as practical and unintrusive type-checking.

We now present a short example from our implementation before going into further details. A correct implementation of the append function on lists should return a list of length $m+n$ when given two lists of lengths m and n, respectively. This property, however, cannot be captured by the type system of ML. This inadequacy can be remedied if we introduce a restricted form of dependent types.

The code in Figure 1 is written in the style of ML with a type annotation, which will be explained shortly. We assume that we are working over the domain of natural numbers with constants 0 and 1 and the addition operation $+$. The datatype `'a list` is defined and then indexed by a natural number, which stands for the length of a list in this case. The constructors of `'a list` are then assigned dependent types:

- `nil <| 'a list(0)` states that `nil` is an `'a list` of length 0.
- `cons <| {n:nat} 'a * 'a list(n) -> 'a list(n+1)` states that `cons` yields an `'a list` of length $n+1$ when given a pair consisting of an element of type `'a` and an `'a list` of length n. We write `{n:nat}` for the dependent function type constructor, usually written as

*This research was sponsored in part by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533.

To appear in the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 1999, San Antonio, Texas.

$\Pi n : nat$, which can also be seen as a universal quantifier.

The `where` clause in the declaration of `append` is a type annotation, which precisely states that `append` returns a list of length $m + n$ when given a pair of lists of lengths m and n, respectively. Generally speaking, the programmer is responsible for refining a datatype and programs are then automatically checked against type annotations with respect to the refinement made.

Let us consider another short example. Suppose that we intend to specify that an implementation of the evaluation function for the pure call-by-value λ-calculus returns a closure (if it terminates) when given a *closed* λ-expression. It seems difficult in ML, if not impossible, to construct a type for closed lambda expressions. With dependent types, this can be done elegantly.

The datatype `lamexp` in Figure 2 is a representation of λ-expressions in de Bruijn's notation. For instance, $\lambda x \lambda y.x(y)$ is represented as `Abs(Abs(App(Shift(One),One)).` `lamexp` is indexed with a natural number n, which roughly means that there are *at most* n free variables in a λ-expression of type `lamexp(n)`. Therefore, `lamexp(0)` is the type for closed λ-expressions. A complete implementation of the evaluation function can be found in (Xi 1997).

Adding dependent types to ML raises a number of theoretical and pragmatic questions. We briefly summarize our results and design choices.

The first question that arises is the meaning of expressions with effects when they occur as index objects to type families. In order to avoid these difficulties we require index objects to be pure. In fact, our type system is parameterized over a domain of constraints from which type index objects are drawn. We can maintain this purity and still make the connection to run-time values by using *singleton types*, such as `int(n)` which contains just the integer n. This is critical for practical applications such as static elimination of array bound checking (Xi and Pfenning 1998).

The second question is the decidability and practicality of type-checking. We address this in two steps: the first step is to define an explicitly typed (and unacceptably verbose) language for which type-checking is easily reduced to constraint satisfaction in C. The second step is to define an elaboration from $\mathrm{DML}(C)$, a slightly extended fragment of ML, to the fully explicitly typed language which preserves the standard operational semantics. The correctness of elaboration and decidability of type-checking modulo constraint satisfiability constitute the main technical contribution of this paper.

The third question is the interface between dependently annotated and other parts of a program or a library. For this we use existential dependent types, although they introduce non-trivial technical complications into the elaboration procedure. Our experience shows that existential dependent types are indispensable in practice. For instance, they are involved in almost all the realistic examples in our experiments.

We have so far finished developing a theoretical foundation for combining dependent types with all the major features in the core of ML, including datatype declarations, higher-order functions, general recursion, let-polymorphism, mutable references and exceptions. We have also implemented our design for a fragment of ML which encompasses all these features. The only main feature in the core of ML which we have not implemented is records. In addition, we have experimented with different constraint domains and applications. Many non-trivial examples are available at (Xi 1997). For the domain of linear inequalities on integers, they include quicksort on arrays, mergesort on lists, a red/black tree implementation, a highly optimized byte copy function, an implementation of Knuth-Morris-Pratt's algorithm for string matching and others in which array bound checks can be statically eliminated without excessive annotations. On symbolic domains we have verified the type preservation property for an *implementation* of the evaluation function for the pure simply typed call-by-value λ-calculus. Also a different red/black tree implementation is verified using a finite domain, where the constraint solver is based on model-checking.

In our experience, $\mathrm{DML}(C)$ is acceptable from the pragmatic point of view: programs can often be annotated with very little internal change, annotations are usually to the point and roughly comparable to what one would find in a typical ML program (including signatures). The resulting constraint simplification problems can be solved efficiently in practice. Also the annotations are mechanically verified, and therefore can be fully trusted as program documentation.

Due to length restrictions, it is impossible to present here all aspects of $\mathrm{DML}(C)$. Instead, we concentrate on its main features. In contrast to (Xi and Pfenning 1998), this extended abstract emphasizes the theoretical foundation of $\mathrm{DML}(C)$, showing that the type system of $\mathrm{DML}(C)$ is sound and type-checking in $\mathrm{DML}(C)$ can be made practical. We refer the interested reader to (Xi 1998) for the details.

The remainder of the paper is organized as follows. We present a monomorphic language ML_0 in Section 2, which is a simply typed λ-calculus with general pattern matching. We start with a monomorphic language simply because the development of dependent types is largely orthogonal to polymorphism. We then introduce the notion of constraint domain in Section 3. We proceed to extend ML_0 with *universal* dependent types in Section 4, leading to the language $\mathrm{ML}_0^{\Pi}(C)$ parameterized over a constraint domain C. We give the typing rules and operational semantics of $\mathrm{ML}_0^{\Pi}(C)$ and show why the type system of $\mathrm{ML}_0^{\Pi}(C)$ can be regarded as a restricted form of dependent types. In Section 5, we present the rules for elaboration from $\mathrm{DML}_0(C)$, an external language, into $\mathrm{ML}_0^{\Pi}(C)$ and prove the correctness of the elaboration. We explain the need for *existential* dependent types in Section 6 and extend $\mathrm{ML}_0^{\Pi}(C)$ to $\mathrm{ML}_0^{\Pi,\Sigma}(C)$. We then briefly mention in Section 7 how dependent types can be combined with let-polymorphism and effects. In Section 8, we sketch some interesting applications. The rest of the paper is concerned with some related work, current status and future research directions.

2 Mini-ML with Pattern Matching

We start with a monomorphic programming language (ML_0) along the lines of Mini-ML, including general pattern matching which is critical in practice and whose theory in this setting is non-trivial. Polymorphism, on the other hand, is largely orthogonal and therefore postponed until Section 7. There we also discuss how to extend the language with effects such as mutable references and exceptions. The syntax of ML_0 is given in Figure 3. We assume throughout that variables are declared at most once in a context and that bound variables may be renamed tacitly.

We omit the typing rules and the call-by-value natural semantics of this language, which are completely standard. Given e, v in ML_0, we write $e \rightarrow_0 v$ if e evaluates to v.

```
datatype 'a list = nil | cons of 'a * 'a list
typeref 'a list of nat with (* indexing the datatype 'a list with nat *)
    nil <| 'a list(0)
  | cons <| {n:nat} 'a * 'a list(n) -> 'a list(n+1)

fun('a)
    append(nil, ys) = ys
  | append(cons(x, xs), ys) = cons(x, append(xs, ys))
where append <| {m:nat}{n:nat} 'a list(m) * 'a list(n) -> 'a list(m+n)
```

Figure 1: An introductory example: append

```
datatype lamexp = One | Shift of lamexp | Abs of lamexp | App of lamexp * lamexp

typeref lamexp of nat with
    One <| {n:nat} lamexp(n+1)
  | Shift <| {n:nat} lamexp(n) -> lamexp(n+1)
  | Abs <| {n:nat} lamexp(n+1) -> lamexp(n)
  | App <| {n:nat} lamexp(n) * lamexp(n) -> lamexp(n)
```

Figure 2: Another introductory example: closed lambda expressions

3 Constraint Domains

Our enriched language will be parameterized over a domain of constraints from which the type index objects are drawn. Typical examples include linear inequalities over integers, boolean constraints, or finite sets. Due to space limitations, we only briefly sketch the interface to constraints as they are used in our type system.

First we note that constraints themselves are typed. In order to avoid confusion we call the types of the constraint language *index sorts*. We use b for base index sorts such as *bool* for booleans and *int* for integers. We use f for interpreted functions symbols, p for atomic predicates (that is, functions of sort $\gamma \rightarrow bool$) and we assume to have constants such as equality $\doteq$ for every sort, truth values $\top$ and $\bot$, negation $\neg$, conjunction $\wedge$, and disjunction $\vee$, all of which are interpreted as usual.

$$
\begin{array}{lrcl}
\text{index sorts} & \gamma & ::= & b \mid \mathbf{1} \mid \gamma_1 * \gamma_2 \mid \{a : \gamma \mid P\} \\
\text{index propositions} & P & ::= & \top \mid \bot \mid \mathrm{p}(i) \mid \\
 & & & P_1 \wedge P_2 \mid P_1 \vee P_2
\end{array}
$$

Here $\{a : \gamma \mid P\}$ is the subset index sort for those elements of γ satisfying proposition P. For instance, *nat* is an abbreviation for $\{a : int \mid a \geq 0\}$. We use a for index variables, and formulate index objects as follows.

$$
\begin{array}{lrcl}
\text{index objects} & i, j & ::= & a \mid () \mid (i, j) \mid \mathrm{f}(i) \\
\text{index contexts} & \phi & ::= & \cdot \mid \phi, a : \gamma \mid \phi, P \\
\\
\text{index constraints} & \Phi & ::= & P \mid \Phi_1 \wedge \Phi_1 \mid P \supset \Phi \mid \\
 & & & \forall a : \gamma.\Phi \mid \exists a : \gamma.\Phi \\
\\
\text{satisfaction relation} & & & \phi \models \Phi
\end{array}
$$

The satisfaction relation $\phi \models \Phi$ means that Φ is satisfied in the constraint domain under index context ϕ. The method for verifying such a relation depends on the constraint domain. For instance, model-checking can be chosen for finite domains.

We omit the standard sorting rules for this index language and the standard definition of constraint satisfaction.

The index constraints listed here are the ones which result from elaboration and should therefore be practically solvable for C in order to obtain a usable type-checker for DML(C). This is the case, for example, for integer inequalities, which our implementation solves by a variant of the Fourier variable elimination method. Empirical results and further references can be found in (Xi and Pfenning 1998).

4 Universal Dependent Types

We now present $\mathrm{ML}_0^{\Pi}(C)$, which extends ML_0 with universal dependent types. Given a domain C of constraints, the syntax of $\mathrm{ML}_0^{\Pi}(C)$ is given in Figure 4. Note that only the syntax different from ML_0 is present. We use δ for base type families, where we use $\delta()$ for unindexed types.

We do not specify here how new type families or constructor types are actually declared, but assume only that they can be processed into the form given above. Our implementation provides both built-in and user-declared refinement of types as shown in the examples.

The typing rules for $\mathrm{ML}_0^{\Pi}(C)$ should be familiar from a dependently typed λ-calculus (such as the ones underlying Coq or NuPrl), except that we separate index variables, abstractions, and applications from term variables, abstractions, and applications. The critical rule of *type conversion* uses the judgment $\phi \vdash \tau_1 \equiv \tau_2$ which is the congruent extension of equality on index objects to arbitrary types:

$$
\frac{\phi \models i \doteq i'}{\phi \vdash \delta(i) \equiv \delta(i')} \qquad \frac{\phi \vdash \tau_1 \equiv \tau_1' \quad \phi \vdash \tau_2 \equiv \tau_2'}{\phi \vdash \tau_1 * \tau_2 \equiv \tau_1' * \tau_2'}
$$

$$
\frac{\phi \vdash \tau_1' \equiv \tau_1 \quad \phi \vdash \tau_2 \equiv \tau_2'}{\phi \vdash \tau_1 \rightarrow \tau_2 \equiv \tau_1' \rightarrow \tau_2'} \qquad \frac{\phi, a : \gamma \vdash \tau \equiv \tau'}{\phi \vdash \Pi a : \gamma.\tau \equiv \Pi a : \gamma.\tau'}
$$

Notice that it is the application of these rules which generates constraints. For instance, the constraint $\phi \models (a + n) + 1 \doteq m + n$ is generated in order to derive $\phi \vdash intlist((a +$

base types	β	$::=$	$\texttt{bool} \mid \texttt{int} \mid$ (other user defined datatypes)
types	σ, τ	$::=$	$\beta \mid \mathbf{1} \mid \tau * \sigma \mid \sigma \to \tau$
patterns	p	$::=$	$x \mid c(p) \mid \langle\rangle \mid \langle p_1, p_2\rangle$
matches	ms	$::=$	$(p \Rightarrow e) \mid (p \Rightarrow e \mid ms)$
expressions	e	$::=$	$x \mid \langle\rangle \mid \langle e_1, e_2\rangle \mid c(e) \mid (\mathbf{case}\ e\ \mathbf{of}\ ms) \mid (\mathbf{lam}\ x : \tau.e) \mid e_1(e_2)$
			$\mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} \mid (\mathbf{fix}\ f : \tau.v)$
values	v	$::=$	$x \mid c(v) \mid \langle\rangle \mid \langle v_1, v_2\rangle \mid (\mathbf{lam}\ x : \tau.e)$
contexts	Γ	$::=$	$\cdot \mid \Gamma, x : \tau$

Figure 3: The syntax for ML_0

families	δ	$::=$	(family of built-in or user-declared refined types)
constructor signature	$\mathcal{S}$	$::=$	$\cdot \mid \mathcal{S}, c : \Pi a_1 : \gamma_1 \ldots \Pi a_n : \gamma_m.\tau \to \delta(i)$
major types	μ	$::=$	$\delta(i) \mid \mathbf{1} \mid (\tau_1 * \tau_2) \mid (\tau_1 \to \tau_2)$
types	τ	$::=$	$\mu \mid (\Pi a : \gamma.\tau)$
patterns	p	$::=$	$\cdots \mid c[a_1] \ldots [a_n](p)$
expressions	e	$::=$	$\cdots \mid c[i_1] \ldots [i_n](e) \mid (\lambda a : \gamma.e) \mid e[i]$
values	v	$::=$	$\cdots \mid c[i_1] \ldots [i_n](v) \mid (\lambda a : \gamma.v)$
substitutions	θ	$::=$	$\cdots \mid \theta[a \mapsto i]$

Figure 4: The syntax for $\mathrm{ML}_0^{\Pi}(C)$

$n) + 1) \equiv \mathit{intlist}(m + n)$ in an example below, where ϕ is $m : \mathit{nat}, n : \mathit{nat}, a : \mathit{nat}, a + 1 \doteq m$.

The only significant complication arises from pattern matching, where new index propositions P are generated. We restrict the index arguments to constructors appearing in patterns to index *variables* so that pattern matches fail or succeed independently of the indices. This is essential to proving the conservativity of $\mathrm{ML}_0^{\Pi}(C)$ over ML_0.

The judgment $p \downarrow \tau \rhd (\phi; \Gamma)$ expresses that the index and ordinary variables in pattern p have the sorts and types declared in ϕ and Γ, respectively, if we know that p must have type τ. It is defined by the following rules.

$$\frac{}{x \downarrow \tau \rhd (\cdot; x : \tau)} \qquad \frac{}{\langle\rangle \downarrow \mathbf{1} \rhd (\cdot; \cdot)}$$

$$\frac{p_1 \downarrow \tau_1 \rhd (\phi_1; \Gamma_1) \quad p_2 \downarrow \tau_2 \rhd (\phi_2; \Gamma_2)}{\langle p_1, p_2\rangle \downarrow \tau_1 * \tau_2 \rhd (\phi_1, \phi_2; \Gamma_1, \Gamma_2)}$$

$$\frac{\mathcal{S}(c) = \Pi\vec{a} : \vec{\gamma}.(\tau \to \delta(i)) \quad p \downarrow \tau \rhd (\phi_1; \Gamma)}{c[\vec{a}](p) \downarrow \delta(j) \rhd (\vec{a} : \vec{\gamma}, i \doteq j, \phi_1; \Gamma)}$$

Assume that *cons* is of type

$$\Pi a : \mathit{nat}.\mathit{int} * \mathit{intlist}(a) \to \mathit{intlist}(a + 1)$$

since polymorphism is not available at this moment. Then the following is derivable.

$$\begin{array}{ll} & \langle \mathit{cons}[a](\langle x, xs\rangle), ys\rangle \downarrow \mathit{intlist}(m) * \mathit{intlist}(n) \\ \rhd & a : \mathit{nat}, a + 1 \doteq m; \\ & x : \mathit{int}, xs : \mathit{intlist}(a), ys : \mathit{intlist}(n) \end{array}$$

The judgment for match expressions $\phi; \Gamma \vdash ms : \tau_1 \Rightarrow \tau_2$ checks independently for each case that given a subject of type τ_1 the case branch will have type τ_2. In other words, $\tau_1 \Rightarrow \tau_2$ is the type of match ms. The following rules are for the derivation of such a judgment.

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

$$\frac{\phi; \Gamma \vdash (p \Rightarrow e) : \tau_1 \Rightarrow \tau_2 \quad \phi; \Gamma \vdash ms : \tau_1 \Rightarrow \tau_2}{\phi; \Gamma \vdash (p \Rightarrow e \mid ms) : \tau_1 \Rightarrow \tau_2}$$

For instance, also using the rules for expressions from Figure 6, it can be readily verified that the following is derivable.

$$\begin{array}{ll} & m : \mathit{nat}, n : \mathit{nat}; \mathit{append} : \tau \\ \vdash & \langle \mathit{cons}[a](\langle x, xs\rangle), ys\rangle \Rightarrow \\ & \mathit{cons}[a + n](\langle x, \mathit{append}[a][n](\langle xs, ys\rangle)\rangle) : \\ & \mathit{intlist}(m) * \mathit{intlist}(n) \Rightarrow \mathit{intlist}(m + n) \end{array}$$

where τ is

$$\begin{array}{l} \Pi m : \mathit{nat}.\Pi n : \mathit{nat}. \\ \quad \mathit{intlist}(m) * \mathit{intlist}(n) \to \mathit{intlist}(m + n) \end{array}$$

Notice that this involves deriving the following, which is obviously true in an integer domain.

$$\begin{array}{ll} & m : \mathit{nat}, n : \mathit{nat}, a : \mathit{nat}, a + 1 \doteq m \\ \models & (a + n) + 1 \doteq m + n \end{array}$$

The remaining typing rules for $\mathrm{ML}_0^{\Pi}(C)$ are in Figure 6.

We now present an example in Figure 5, which is basically an expression in $\mathrm{ML}_0^{\Pi}(C)$ corresponding to a monomorphic version of the code in Figure 1. We also present a sugared version of the expression to enhance readability, but we emphasize that there is no sugared syntax for $\mathrm{ML}_0^{\Pi}(C)$ in our implementation (see Section 5).

Next we turn to the operational semantics. The critical design decisions are that (a) indices are never evaluated, (b) indices are never used to select branches during pattern matches, and (c) we evaluate underneath index abstractions $\lambda a : \gamma.e$. We do, however, substitute for index variables

$$
\begin{array}{l}
\textbf{fix } \mathit{append} : \Pi m : \mathit{nat}.\Pi n : \mathit{nat}.\mathit{intlist}(m) * \mathit{intlist}(n) \to \mathit{intlist}(m+n). \\
\quad \lambda m : \mathit{nat}.\lambda n : \mathit{nat}.\textbf{lam } l : \mathit{intlist}(m) * \mathit{intlist}(n). \\
\quad\quad \textbf{case } l \textbf{ of} \\
\quad\quad\quad \langle \mathit{nil}, \mathit{ys} \rangle \Rightarrow \mathit{ys} \\
\quad\quad\quad \langle \mathit{cons}[a](\langle x, \mathit{xs} \rangle), \mathit{ys} \rangle \Rightarrow \mathit{cons}[a+n](\langle x, \mathit{append}[a][n](\langle \mathit{xs}, \mathit{ys} \rangle)\rangle)
\end{array}
$$

```
fun append[0][n](nil, ys) = ys
  | append[a+1][n](cons[a](x, xs), ys) = cons[a+n](x, append[a][n](xs, ys))
where append <| {m:nat}{n:nat} intlist(m) * intlist(n) -> intlist(m+n)
```

Figure 5: An expression in $\mathrm{ML}_0^{\Pi}(C)$ and its sugared version

$$
\frac{\phi;\Gamma \vdash e : \tau_1 \quad \phi \models \tau_1 \equiv \tau_2}{\phi;\Gamma \vdash e : \tau_2}\ \textbf{(ty-eq)}
\qquad
\frac{\Gamma(x) = \tau \quad \phi \vdash \Gamma[\textbf{ctx}]}{\phi;\Gamma \vdash x : \tau}\ \textbf{(ty-var)}
$$

$$
\frac{\mathcal{S}(c) = \delta(i)}{\phi;\Gamma \vdash c : \delta(i)}\ \textbf{(ty-cons-wo)}
\qquad
\frac{\mathcal{S}(c) = \Pi\vec{a} : \vec{\gamma}.\tau \to \delta(i) \quad \phi \vdash \vec{i} : \vec{\gamma} \quad \phi;\Gamma \vdash v : \tau[\vec{a} := \vec{i}]}{\phi;\Gamma \vdash c[\vec{i}](v) : \delta(i[\vec{a} := \vec{i}])}\ \textbf{(ty-cons-w)}
$$

$$
\frac{}{\phi;\Gamma \vdash \langle\rangle : \mathbf{1}}\ \textbf{(ty-unit)}
\qquad
\frac{\phi;\Gamma \vdash e_1 : \tau_1 \quad \phi;\Gamma \vdash e_2 : \tau_2}{\phi;\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 * \tau_2}\ \textbf{(ty-prod)}
$$

$$
\frac{\phi \vdash \tau_1 : * \quad p \downarrow \tau_1 \rhd (\phi';\Gamma') \quad \phi,\phi';\Gamma,\Gamma' \vdash e : \tau_2 \quad \phi \vdash \tau_2 : *}{\phi;\Gamma \vdash p \Rightarrow e : \tau_1 \Rightarrow \tau_2}\ \textbf{(ty-match)}
$$

$$
\frac{\phi;\Gamma \vdash (p \Rightarrow e) : \tau_1 \Rightarrow \tau \quad \phi;\Gamma \vdash ms : \tau_1 \Rightarrow \tau}{\phi;\Gamma \vdash (p \Rightarrow e \mid ms) : \tau_1 \Rightarrow \tau}\ \textbf{(ty-matches)}
$$

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

$$
\frac{\phi, a : \gamma;\Gamma \vdash e : \tau}{\phi;\Gamma \vdash (\lambda a : \gamma.e) : (\Pi a : \gamma.\tau)}\ \textbf{(ty-ilam)}
\qquad
\frac{\phi;\Gamma \vdash e : \Pi a : \gamma.\tau \quad \phi \vdash i : \gamma}{\phi;\Gamma \vdash e[i] : \tau[a := i]}\ \textbf{(ty-iapp)}
$$

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

$$
\frac{\phi;\Gamma \vdash e_1 : \tau_1 \quad \phi;\Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\phi;\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} : \tau_2}\ \textbf{(ty-let)}
\qquad
\frac{\phi;\Gamma, f : \tau \vdash v : \tau}{\phi;\Gamma \vdash (\textbf{fix } f : \tau.v) : \tau}\ \textbf{(ty-fix)}
$$

Figure 6: Typing Rules for $\mathrm{ML}_0^{\Pi}(C)$

when a branch in a pattern match has been selected, or when a dependently typed function is applied to an index argument as in $e[i]$. These points together guarantee type preservation for $\mathrm{ML}_0^{\Pi}(C)$ and conservativity over ML_0.

There is no obstacle preventing the evaluation of type indices at run-time—there is simply no need to do so. Clearly, this would change immediately if run-time type-checking became necessary, but we do not intend to run DML programs which could not be type-checked at compile-time.

For the sake of brevity, we omit the operational semantics in this extended abstract. Given the remarks above, it is straightforward to define $e \to_d v$ in the style of natural semantics, which means e evaluates to v in $\mathrm{ML}_0^{\Pi}(C)$.

Next we prove the central properties of $\mathrm{ML}_0^{\Pi}(C)$. The first basic property states that dependent types are preserved under the operational semantics.

Theorem 4.1 *(Type preservation) Given e, v in $\mathrm{ML}_0^{\Pi}(C)$ such that $e \to_d v$ is derivable. If $\phi;\Gamma \vdash e : \tau$ is derivable, then $\phi;\Gamma \vdash v : \tau$ is derivable.*

Proof By a structural induction on the derivations of $e \to_d v$ and $\phi;\Gamma \vdash e : \tau$. ■

The following definition and theorems detail the relationship between $\mathrm{ML}_0^{\Pi}(C)$ and ML_0. Basically, $\mathrm{ML}_0^{\Pi}(C)$ is a refinement of the type system of ML_0 which allows us to express more properties, but neither affects the operational semantics nor the typing judgments already expressible in ML_0.

Definition 4.2 *We define the index erasure function $\|\cdot\|$*

as follows:

$$\begin{array}{lcl}
\|\delta(i_1,\ldots,i_n)\| & = & \delta \\
\|\tau_1 * \tau_2\| & = & \|\tau_1\| * \|\tau_2\| \\
\|\tau_1 \rightarrow \tau_2\| & = & \|\tau_1\| \rightarrow \|\tau_2\| \\
\|\Pi a:\gamma.\tau\| & = & \|\tau\| \\
\|c[i_1]\ldots[i_n](e)\| & = & c(\|e\|) \\
\|(\mathbf{lam}\ x:\tau.e)\| & = & (\mathbf{lam}\ x:\|\tau\|.\|e\|) \\
\|(\lambda a:\gamma.e)\| & = & \|e\| \\
\|e[i]\| & = & \|e\| \\
\|\mathbf{fix}\ x:\tau.v\| & = & \mathbf{fix}\ x:\|\tau\|.\|v\| \\
\|\Gamma, x:\tau\| & = & \|\Gamma\|, x:\|\tau\|
\end{array}$$

It maps an expression in $\mathrm{ML}_0^\Pi(C)$ *into one in* ML_0. *Note that a few trivial cases are omitted for the sake of brevity.*

The erasure of a program written in $\mathrm{ML}_0^\Pi(C)$ is executed in ML_0. In general, the erasure of a DML program is executed in ML. The next theorem guarantees that the index erasure of a well-typed program in $\mathrm{ML}_0^\Pi(C)$ is also well-typed in ML_0.

Theorem 4.3 *If* $\phi;\Gamma \vdash e:\tau$ *is derivable in* $\mathrm{ML}_0^\Pi(C)$, *then* $\|\Gamma\| \vdash \|e\| : \|\tau\|$ *is derivable in* ML_0.

Proof By a structural induction on the derivation of $\phi;\Gamma \vdash e:\tau$. ∎

A significant consequence of Theorem 4.3 is that if an untyped program is typable in $\mathrm{ML}_0^\Pi(C)$ then it is already typable in ML_0. This distinguishes our design from those which aim at making *more* programs typable by extending the type system of ML. Instead, our objective is to assign more accurate types to programs.

Also we must guarantee that the operational semantics of a program in $\mathrm{ML}_0^\Pi(C)$ is preserved when it is evaluated in ML_0. This is done by the following two theorems.

Theorem 4.4 *(Soundness) If* $e \rightarrow_d v$ *derivable in* $\mathrm{ML}_0^\Pi(C)$, *then* $\|e\| \rightarrow_0 \|v\|$ *is derivable.*

Proof By a structural induction on the derivation of $e \rightarrow_d v$. ∎

The corresponding completeness property relies on the restrictions on the form of constructor types and the index arguments to constructors in patterns.

Theorem 4.5 *(Completeness) Given* $\phi;\Gamma \vdash e:\tau$ *derivable in* $\mathrm{ML}_0^\Pi(C)$. *If* $\|e\| \rightarrow_0 v_0$ *is derivable for some* v_0 *in* ML_0, *then there exists* v *in* $\mathrm{ML}_0^\Pi(C)$ *such that* $e \rightarrow_d v$ *and* $\|v\| = v_0$.

Proof By a structural induction on the derivations of $\|e\| \rightarrow_0 v_0$ and $\phi;\Gamma \vdash e:\tau$. ∎

It is a straightforward observation on the typing rules for $\mathrm{ML}_0^\Pi(C)$ that the following theorem holds. Therefore, if the user does not index any types, then his code is valid in $\mathrm{ML}_0^\Pi(C)$ iff it is valid in ML_0.

Theorem 4.6 $\mathrm{ML}_0^\Pi(C)$ *is a conservative extension of* ML_0.

We call the type system of $\mathrm{ML}_0^\Pi(C)$ a *restricted form* of dependent types, since we view both index objects and expressions in $\mathrm{ML}_0^\Pi(C)$ as *terms*. In this view, the *type* of one term can depend on the *value* of other terms. For instance, the type of $append[m][n](\langle xs, ys\rangle)$ depends on m and n. An alternative is to view index objects as types, and therefore to regard the type system of $\mathrm{ML}_0^\Pi(C)$ as a polymorphic type system. However, this alternative leads some (unnecessary) complications. For instance, it is unclear which expressions are of type i if i is an index object. A more serious problem is how subset sorts should be treated under this alternative view.

In a *fully* dependent type system such as the one which underlies LF (Harper, Honsell, and Plotkin 1993) or Coq, there is no differentiation between type index objects and language expressions. In other words, the constraint domain is the same as the language. Therefore, constraint satisfaction is as difficult as program verification, which seems to be intractable in practice. The novelty of $\mathrm{ML}_0^\Pi(C)$ is precisely the differentiation between type index objects and language expressions, which makes our approach practical and scalable.

5 Elaboration

We have so far presented an *explicitly typed* language $\mathrm{ML}_0^\Pi(C)$. This presentation has a serious drawback from a programmer's point view: *one would quickly be overwhelmed with types when programming in such a setting.* It then becomes apparent that it is necessary to provide an *external language* $\mathrm{DML}_0(C)$ together with a mapping to the *internal language* $\mathrm{ML}_0^\Pi(C)$. This mapping is called *elaboration.* For instance, the declaration of (a monomorphic version of) `append` in Figure 1 is to be elaborated into the $\mathrm{ML}_0^\Pi(C)$-expression in Figure 5.

5.1 The External Language $\mathrm{DML}_0(C)$

The syntax for $\mathrm{DML}_0(C)$ is given as follows.

$$\begin{array}{lcl}
\text{patterns} & p & ::= x \mid c(p) \mid \langle\rangle \mid \langle p_1, p_2\rangle \\
\text{matches} & ms & ::= (p \Rightarrow e) \mid (p \Rightarrow e \mid ms) \\
\text{expressions} & e & ::= x \mid c(e) \mid \langle\rangle \mid \langle e_1, e_2\rangle \mid \\
\multicolumn{3}{l}{\quad \mathbf{case}\ e\ \mathbf{of}\ ms \mid e_1(e_2) \mid \mathbf{lam}\ x.e \mid \mathbf{lam}\ x:\tau.e \mid} \\
\multicolumn{3}{l}{\quad \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} \mid \mathbf{fix}\ f.v \mid \mathbf{fix}\ f:\tau.v \mid e:\tau}
\end{array}$$

Note that this is basically the syntax for ML_0 though types here could be dependent types. This partially attests to the unobtrusiveness of our enrichment.

5.2 Elaboration as Static Semantics

We illustrate the intuitions behind some elaboration rules while presenting them. Elaboration, which incorporates type-checking, is defined via two mutually recursive judgments: one to synthesize a type where this can be done in a most general way, and one to check a term against a type where synthesis is not possible. A synthesizing judgment has the form $\phi;\Gamma \vdash e \uparrow \tau \Rightarrow e^*$ and means that e elaborates into e^* *with* type τ. A checking judgment has the form $\phi;\Gamma \vdash e \downarrow \tau \Rightarrow e^*$ and means that e elaborates into e^* *against* type τ. In general, we use e, p, ms for external expressions, patterns and matches, and e^*, p^*, ms^* for their internal counterparts.

The purpose of the first two rules is to eliminate Π quantifiers. For instance, let us assume that $e_1(e_2)$ is in the code and a type of form $\Pi a:\gamma.\tau$ is synthesized for e_1; then we must apply the rule (**elab-pi-elim**) to remove the Π quantifier in the type. We continue doing so until a major type is reached, which must be of form $\tau_1 \rightarrow \tau_2$ (if the code is type-correct). Note that the actual index i is not locally determined, but becomes an existential variable for the constraint solver. The rule (**elab-pi-intro**) is simpler since we check against a given dependent function type.

$$\frac{\phi; \Gamma \vdash e \uparrow \Pi a : \gamma.\tau \Rightarrow e^* \quad \phi \vdash i : \gamma}{\phi; \Gamma \vdash e \uparrow \tau[a := i] \Rightarrow e^*[i]} \ \textbf{(elab-pi-elim)}$$

$$\frac{\phi, a : \gamma; \Gamma \vdash e \downarrow \tau \Rightarrow e^*}{\phi; \Gamma \vdash e \downarrow \Pi a : \gamma.\tau \Rightarrow (\lambda a : \gamma.e^*)} \ \textbf{(elab-pi-intro)}$$

The next rule (**elab-lam**) is for lambda abstraction, which checks a **lam** expression against a type. The rule for the fixed point operator is similar. We emphasize that we never synthesize types for either **lam** or **fix** expressions (for which principal types do not exist in general).

$$\frac{\phi; \Gamma, x : \tau_1 \vdash e \downarrow \tau_2 \Rightarrow e^*}{\phi; \Gamma \vdash (\mathbf{lam}\ x.e) \downarrow \tau_1 \rightarrow \tau_2 \Rightarrow (\mathbf{lam}\ x : \tau_1.e_1^*)}$$

The next rule (**elab-app-up**) is for function application, where the interaction between the two kinds of judgments takes place. After synthesizing a major type $\tau_1 \rightarrow \tau_2$ for e_1, we simply check e_2 against τ_1—synthesis for e_2 is unnecessary.

$$\frac{\phi; \Gamma \vdash e_1 \uparrow \tau_1 \rightarrow \tau_2 \Rightarrow e_1^* \quad \phi; \Gamma \vdash e_2 \downarrow \tau_1 \Rightarrow e_2^*}{\phi; \Gamma \vdash e_1(e_2) \uparrow \tau_2 \Rightarrow e_1^*(e_2^*)}$$

We maintain the invariant that the shape of types of variables in the context is always determined, modulo possible index constraints which may need to be solved. This means that with the rules above we can already check all normal forms. A term which is not in normal form most often will be a let-expression, but in any case will require a type annotation, as illustrated in the rule (**elab-let-down**) below.

$$\frac{\phi; \Gamma \vdash e_1 \uparrow \tau_1 \Rightarrow e_1^* \quad \phi; \Gamma, x : \tau_1 \vdash e_2 \downarrow \tau_2 \Rightarrow e_2^*}{\phi; \Gamma, x : \tau_1 \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} \downarrow \tau_2 \Rightarrow e^*}$$

where $e^* = \mathbf{let}\ x = e_1^*\ \mathbf{in}\ e_2^*\ \mathbf{end}$. Even if we are checking against a type, we must synthesize the type of e_1. If e_1 is a function or fixpoint, its type must be given, in practice mostly by writing $\mathbf{let}\ x : \tau = e_1\ \mathbf{in}\ e_2\ \mathbf{end}$ which abbreviates $\mathbf{let}\ x = (e_1 : \tau)\ \mathbf{in}\ e_2\ \mathbf{end}$. The following rule allows us to take advantage of such annotations.

$$\frac{\phi; \Gamma \vdash e \downarrow \tau \Rightarrow e^*}{\phi; \Gamma \vdash (e : \tau) \uparrow \tau \Rightarrow e^*} \ \textbf{(elab-anno-up)}$$

As a result, the only types which are required in realistic programs are due to declarations of functions and a few cases of polymorphic instantiation.

Moreover, in the presence of existential dependent types, which will be introduced in Section 6, a pure ML type without dependencies obtained in the first phase of type-checking is assumed if no explicit type annotation is given. This makes our extension truly conservative in the sense that pure ML programs will work exactly as before, not requiring any annotations.

Elaboration rules for patterns are particularly simple, due to the constraint nature of the types for constructors. We elaborate a pattern p against a type τ, yielding an internal pattern p^* and index and term contexts ϕ and Γ, respectively. This is written as $p \downarrow \tau \Rightarrow (p^*; \phi; \Gamma)$ in Figure 7. This judgment is used in the following rule (**elab-match**) for pattern matching: the generated context ϕ' is assumed into the context ϕ while elaborating e. For constraint satisfaction, the declarations in ϕ' are treated as hypotheses.

$$\frac{}{x \downarrow \tau \Rightarrow (x; \cdot; x : \tau)} \qquad \frac{}{\langle\rangle \downarrow \mathbf{1} \Rightarrow (\langle\rangle; \cdot; \cdot)}$$

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

$$\frac{\mathcal{S}(c) = \Pi\vec{a} : \vec{\gamma}.\tau \rightarrow \delta(i) \quad p \downarrow \tau \Rightarrow (p^*; \phi; \Gamma)}{c(p) \downarrow \delta(j) \Rightarrow (c[\vec{a}](p^*); \vec{a} : \vec{\gamma}, i \doteq j, \phi; \Gamma)}$$

Figure 7: The elaboration rules for patterns

$$\frac{p \downarrow \tau_1 \Rightarrow (p^*; \phi'; \Gamma') \quad \phi, \phi'; \Gamma, \Gamma' \vdash e \downarrow \tau_2 \Rightarrow e^*}{\phi; \Gamma \vdash (p \Rightarrow e) \downarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow (p^* \Rightarrow e_1^*)}$$

The complete elaboration rules for $\mathrm{DML}_0(C)$ are listed in Appendix B, and they are justified by the following theorem.

Theorem 5.1 *Let $\cong$ be the operational equivalence relation.*

1. *If $\phi; \Gamma \vdash e \uparrow \tau \Rightarrow e^*$ is derivable, then $\phi; \Gamma \vdash e^* : \tau$ is derivable and $e \cong e^*$.*
2. *If $\phi; \Gamma \vdash e \downarrow \tau \Rightarrow e^*$ is derivable, then $\phi; \Gamma \vdash e^* : \tau$ is derivable and $e \cong e^*$.*

Proof (1) and (2) follow straightforwardly from a simultaneous structural induction on the derivations of $\Gamma \vdash e \uparrow \tau \Rightarrow e^*$ and $\phi; \Gamma \vdash e \downarrow \tau \Rightarrow e^*$. ∎

The description of type reconstruction as static semantics is intuitively appealing, but there is still a gap between the description and its implementation. There, elaboration rules explicitly generate constraints, thus reduce dependent type-checking to constraint satisfaction. This kind of transformation is standard and therefore omitted here. For instance, when elaborating the first and second clauses in the function declaration in Figure 1, we generate the following two constraints, which are obviously true.

$$\forall n : nat.0 + n \doteq n$$
$$\forall m : nat.\forall n : nat.\forall a : nat.\ a + 1 \doteq m \supset (a + n) + 1 \doteq m + n$$

Therefore the code is well-typed in $\mathrm{ML}_0^{\Pi}(C)$. A thoroughly explained example on elaboration and constraint generation is available in (Xi 1998).

6 Existential Dependent Types

In practice, the constraint domain must be relatively simple to permit the implementation of an effective constraint solver. Therefore there remain many properties of indices which cannot be expressed. For instance, if we apply the following `filter` function to a list of length n, we cannot express the length of the resulting list since it depends on the predicate `p`.

```
fun filter p nil = nil
  | filter p (x::xs) =
    if p(x) then x::(filter p xs)
    else (filter p xs)
```

Nonetheless, we know that there exists some $m \leq n$ such that the length of the resulting list is m, which can be expressed using an existential dependent type, also called weak dependent sum. Also, existential types can mediate between dependent and ordinary ML types. For instance, given a function of ML type `'a list -> 'a list`, we can assign to it a dependent type which states that the function returns a list of unknown length when applied to a list of unknown length. This yields an approach to handling existing functions such as those in a library, whose definitions may not be available. Notice that this is crucial to support separate compilation. However, the use of existential types to represent ML types leads to a major loss of information at module boundaries. We would like soon to address this issue by exporting dependent types in signatures, extending DML to full SML. This approach closely relates to Extended ML (Sannella and Tarlecki 1989).

We now extend $\mathrm{ML}_0^{\Pi}(C)$ to $\mathrm{ML}_0^{\Pi,\Sigma}(C)$.

$$\begin{array}{llll}
\text{types} & \tau & ::= & \ldots \mid (\Sigma a : \gamma.\tau) \\
\text{expressions} & e & ::= & \ldots \mid \langle i \mid e \rangle \mid \\
 & & & \textbf{let } \langle a \mid x \rangle = e_1 \textbf{ in } e_2 \textbf{ end} \\
\text{values} & v & ::= & \ldots \mid \langle i \mid v \rangle
\end{array}$$

We need the additional typing rules **(t-sig-intro)** and **(t-sig-elim)**

$$\frac{\phi; \Gamma \vdash e : \tau[a := i] \qquad \phi \vdash i : \gamma}{\phi; \Gamma \vdash \langle i \mid e \rangle : (\Sigma a : \gamma.\tau)} \ \textbf{(t-sig-intro)}$$

$$\frac{\phi; \Gamma \vdash e_1 : \Sigma a : \gamma.\tau_1 \quad \phi, a : \gamma; \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\phi; \Gamma \vdash \textbf{let } \langle a \mid x \rangle = e_1 \textbf{ in } e_2 \textbf{ end} : \tau_2} \ \textbf{(t-sig-elim)}$$

where a may not occur in τ_2 in the latter (in addition to the general assumption that a and x are not already declared in ϕ and Γ, respectively). For instance, we can assign the following type to the function `filter`:

```
('a -> bool) ->
{n:nat} 'a list(n) -> [m:nat | m<=n] 'a list(m),
```

where `[m:nat|m<=n]` stands for

$$\Sigma m : \{a : nat \mid a \leq n\}.$$

Also, we can assign the type

```
([n:nat] 'a list(n)) -> [n:nat] 'a list(n),
```

to any function of ML type `'a list -> 'a list`.

We can then prove all the theorems in Section 4 for $\mathrm{ML}_0^{\Pi,\Sigma}(C)$. It is also straightforward to give a sound elaboration procedure from $\mathrm{DML}_0(C)$ to $\mathrm{ML}_0^{\Pi,\Sigma}(C)$. The following is a significant rule for this purpose.

$$\frac{\begin{array}{c}\phi; \Gamma \vdash e_1 \uparrow \Sigma a : \gamma.\tau_1 \Rightarrow e_1^* \\ \phi, a : \gamma; \Gamma, x : \tau_1 \vdash e_2 \downarrow \tau_2 \Rightarrow e_2^*\end{array}}{\phi; \Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} \downarrow \tau_2 \Rightarrow e^*},$$

where e^* is $\textbf{let } \langle a \mid x \rangle = e_1^* \textbf{ in } e_2^* \textbf{ end}$.

Unfortunately, this elaboration rule requires a **let** to be present in the source when eliminating an existential dependent type, which will not be the case in many typical ML programs. We therefore apply the A-translation (Moggi 1989; Sabry and Felleisen 1993) before elaboration. For example, suppose that the synthesized types of l and l' are $\Sigma a : nat.intlist(a)$, that is, the lengths of l and l' are unknown. If we check $append(\langle l, l' \rangle)$ against $\Sigma a : nat.intlist(a)$, then $append(\langle l, l' \rangle)$ is translated to

$$\textbf{let } x = l \textbf{ in let } x' = l' \textbf{ in } append(\langle x, x' \rangle) \textbf{ end end},$$

which is then elaborated into

$$\begin{array}{l}
\textbf{let } \langle a \mid x \rangle = l \textbf{ in} \\
\quad \textbf{let } \langle a' \mid x' \rangle = l' \textbf{ in} \\
\qquad \langle a + a' \mid append[a][a'](\langle x, x' \rangle) \rangle \textbf{ end end}
\end{array}$$

There are some pragmatic issues on whether A-translation should be controlled by the programmer or applied automatically. We have chosen the latter in our current implementation. Please see (Xi 1998) for details.

7 Polymorphism and Effects

It is straightforward to extend $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ with polymorphism. We have designed a two-phase elaboration algorithm which elaborates a program as follows.

Phase one It verifies that the index erasure of the program is a well-typed ML program.

Phase two It then applies the elaboration algorithm for $\mathrm{ML}_0^{\Pi,\Sigma}(C)$ to the result obtained in phase one. If a needed type annotation is unavailable, the type inferred in phase one is supplied. This guarantees that a valid ML program is always valid in $\mathrm{DML}(C)$.

The type system becomes unsound if dependent types are combined directly with effects. The symptom is the same as that of combining polymorphism with effects. Soundness can be recovered if we adopt a *value restriction*, that is, we replace the rule **(ty-ilam)** with the following.

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

After this, the development is standard, which is thoroughly explained in (Xi 1998).

8 Applications

8.1 Program Error Detection

A red/black tree is a balanced binary tree which satisfies the following conditions: (a) all leaves are marked black and all other node are marked either red or black; (b) for every node there are the same number of black nodes on every path connecting the node to a leaf, and this number is called the *black height* of the node; (c) the two sons of every red node are black.

In Figure 8, we define a polymorphic datatype `'a dict`, which is essentially a binary tree with colored nodes. We then refine the datatype with type index objects (c, bh) drawn from the sort $bool * nat$, where c and bh are the color and the black height of the root of the binary tree. The node is black if and only if c is *true*. Therefore, the properties of a red/black tree are naturally captured with this datatype refinement. This enables the programmer to catch program errors which lead to violations of these properties when implementing an insertion or deletion operation on red/black trees. We have indeed encountered errors caught in this way in practice.

The types of other balanced trees such as AVL trees can be declared similarly (see Appendix A).

```
type 'a entry = int * 'a

datatype 'a dict = Empty (* considered black *)
                 | Black of 'a entry * 'a dict * 'a dict
                 | Red of 'a entry * 'a dict * 'a dict

typeref 'a dict of bool * nat with
  Empty <| 'a dict(true, 0)
| Black <| {cl:bool}{cr:bool}{bh:nat}
           'a entry * 'a dict(cl, bh) * 'a dict(cr, bh) -> 'a dict(true, bh+1)
| Red <| {bh:nat}
         'a entry * 'a dict(true, bh) * 'a dict(true, bh) -> 'a dict(false, bh)
```

Figure 8: The red/black tree data structure

8.2 Array Bound Check Elimination

We refine the built-in types: (a) for every integer n, `int`(n) is a singleton type which contains only n, and (b) for every natural number n, `'a array`(n) is the type of arrays of size n. We then assume that the array operations `sub` and `update` have been assigned the following types.

```
sub <| {n:nat} {i:nat | i < n}
       'a array(n) * int(i) -> 'a
update <| {n:nat} {i:nat | i < n}
          'a array(n) * int(i) * 'a -> unit
```

Clearly, the array accesses through `sub` or `update` cannot result in array bounds violations, and therefore it is unnecessary to insert array bound checks when we compile the code. Please see (Xi and Pfenning 1998) for the details.

8.3 Dead Code Elimination

The following function *zip* zips two lists together. If the clause `zip(_, _) = raise zipException` is missing, then ML compilers will issue a warning message stating that *zip* may result in a match exception. This happens if two arguments of *zip* are of different lengths.

```
exception zipException
fun('a, 'b)
    zip(nil, nil) = nil
  | zip(cons(x, xs), cons(y, ys)) =
    cons((x,y), zip(xs, ys))
  | zip(_, _) = raise zipException
```

However, this function is meant to zip two lists of *equal* length. If we declare *zip* to be of the following dependent type,

```
{n:nat} 'a list(n) * 'b list(n) ->
('a * 'b) list(n)
```

the clause `zip(_, _) = raise zipException` in the definition of *zip* can never be reached, and therefore can be safely removed.

This leads to not only more compact but also potentially more efficient code. For instance, if it has been verified that the first argument of *zip* is non-empty, then the second argument must also be non-empty. Therefore, tag-checking can be reduced significantly when this example is implemented. Such examples are abundant in practice.

It will not be straightforward to extend the usual pattern compilation algorithms to take advantage of such additional information, and we have not yet tried this idea in a compiler. However, the benefit of such dead code elimination for error detection can be readily realized. We refer the interested reader to (Xi 1999) for further explanation.

8.4 Other Applications

There are many other potential applications of dependent types which can be found in (Xi 1998), including facilitating partial evaluation, performing loop-unrolling, passing dependent types to an assembly language, etc.

9 Related Work

Our work falls in between full program verification, either in type theory or systems such as PVS (Owre, Rajan, Rushby, Shankar, and Srivas 1996), and traditional type systems for programming languages. When compared to verification, our system is less expressive but more automatic when constraint domains with practical constraint satisfaction problems are chosen. Our work can be viewed as providing a systematic and uniform language interface for a verifier intended to be used as a type system during the program development cycle. Since it extends ML conservatively, it can be used sparingly as existing ML programs will work as before (if there is no keyword conflict).

Most closely related to our work is the system of *indexed types* developed independently by Zenger in his forthcoming Ph.D. Thesis (Zenger 1998) (an earlier version of which is described in (Zenger 1997)). He works in the context of of lazy functional programming. His language is clean and elegant and his applications (which significantly overlap with ours) are compelling. In general, his approach seems to require more changes to a given Haskell program to make it amenable to checking indexed types than is the case for our system and ML. This is particularly apparent in the case of existential dependent types, which are tied to data constructors. This has the advantage of a simpler algorithm for elaboration and type-checking than ours, but the program (and not just the type) has to be more explicit. Also, since his language is pure, he does not consider a value restriction.

When compared to traditional type systems for programming languages, perhaps the closest related work is refinement types (Freeman and Pfenning 1991), which also aims at expressing and checking more properties of programs that are already well-typed in ML, rather than admitting more programs as type correct, which is the goal of most other research on extending type systems. However, the mechanism of refinement types is quite different and incomparable in expressive power: while refinement types incorporate intersection and can thus ascribe multiple types to terms in a uniform way, dependent types can express properties such as "*these two argument lists have the same length*" which are not recognizable by tree automata (the basis for type refinements). We plan to consider a combination of these ideas in future work.

Parent (Parent 1995) proposed to reverse the process of extracting programs from constructive proofs in Coq (Dowek, Felty, Herbelin, Huet, Murthy, Parent, Paulin-Mohring, and Werner 1993), synthesizing proof skeletons from annotated programs. Such proof skeletons contain "holes" corresponding to logical propositions not unlike our constraint formulas. In order to limit the verbosity of the required annotations, she also developed heuristics to reconstruct proofs using higher-order unification. Our aims and methods are similar, but much less general in the kind of specifications we can express. On the other hand, this allows a richer source language with fewer annotations and, in practice, avoids interaction with a theorem prover.

Extended ML (Sannella and Tarlecki 1989) is proposed as a framework for the formal development of programs in a pure fragment of Standard ML. The module system of Extended ML can not only declare the type of a function but also the axioms it satisfies. This requires theorem proving during extended type checking. We specify and check less information about functions which avoids general theorem proving. On the other hand, we currently do not address module-level issues, although we believe that our approach should extend naturally to signatures and functors without much additional machinery.

Cayenne (Augustsson 1998) is a Haskell-like language in which fully dependent types are available, that is, language expressions can be used as type index objects. The steep price for this is undecidable type-checking in Cayenne. We feel that Cayenne pays greater attention to making more programs typable than assigning programs more accurate types. In Cayenne, the `printf` in C, which is not typable in ML (see (Danvy 1998) for further details), can be made typable, and modules can be replaced with records, but the notion of datatype refinement does not exist. This clearly separates our language design from that of Cayenne.

The notion of sized types is introduced in (Hughes, Pareto, and Sabry 1996) for proving the correctness of reactive systems. Though there exist some similarities between sized types and datatype refinement in DML(C) for some domain C of natural numbers, the differences are also substantial. We feel that the language presented in (Hughes, Pareto, and Sabry 1996) is too restrictive for general programming since the type system there can only handle (a minor variation) of primitive recursion. On the other hand, the use of sized types in the correctness proofs of reactive systems cannot be achieved in DML at this moment.

Jay and Sekanina (Jay and Sekanina 1996) have introduced a technique for array bounds checking based on the notion of shape types. Shape checking is a kind of partial evaluation and has very different characteristics and source language when compared to DML(C), where constraints are linear inequalities on integers. We feel that their design is more restrictive and seems more promising for languages based on iteration schemas rather than general recursion.

A key feature in DML(C) which does not exist in either of the above two systems is *existential dependent types*, which is indispensable in our experiment.

Finally, recent work by Pierce and Turner (Pierce and Turner 1998) which includes some empirical studies, is based on a similar bi-directional strategy for elaboration, although they are concerned with the interaction of polymorphism and subtyping, while we are concerned with dependent types. The use of constraints for index domains is quite different from the use of constraints to model subtyping constraints (see, for example, (Sulzmann, Odersky, and Wehr 1997)).

10 Conclusion

We have extended the entire core of ML with a restricted form of dependent types, yielding the DML(C) language schema. This includes proving the soundness of the type system of DML(C) and designing a type-checking algorithm. Type annotations are required, but not overly verbose. The algorithm has shown itself to be practical for typical programs and constraint domains, such as linear inequalities over integers for array bounds checking (Xi and Pfenning 1998). In addition, we have finished a prototype implementation of DML(C) in which all the major features in the core of ML except records are available. The only reason for omitting records is that we already have tuples and we would like to simplify the implementation. We have also experimented with integer, symbolic and finite domains. We are currently writing a frontend for Caml-light.

In future work, we plan to enrich DML with module-level constructs, that is, extend DML to full Standard ML. Since our design explicitly separates indices from ML expressions, we expect the extension to be mostly straightforward. Another practically important extension may be the introduction of limited forms of intersection types (Freeman and Pfenning 1991), so that more than one dependent type can be assigned to a function without code duplication.

Our primary motivation is to allow the programmer to express more program properties through types and thus catch more errors at compile time. We are also interested in using this as a front-end for a certifying compiler (Necula and Lee 1998) which propagates program properties through a compiler where they can be used for optimizations or be packaged with the binaries in the form of proof-carrying code (Necula 1997).

11 Acknowledgements

We are grateful to Rowan Davies for many technical discussions regarding the subject of this extended abstract. We also would like to thank Chad Brown for proofreading a draft and providing us with many helpful comments, and the referees for their highly constructive suggestions.

References

Augustsson, L. (1998). Cayenne – a language with dependent types. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, pp. 239–250.

Constable, R. L. et al. (1986). *Implementing Mathematics with the Nuprl Proof Development System*. Englewood Cliffs, New Jersey: Prentice-Hall.

Danvy, O. (1998, May). Functional unparsing. Technical Report RS-98-12, University of Aarhus.

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

Freeman, T. and F. Pfenning (1991). Refinement types for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Toronto, Ontario, pp. 268–277.

Harper, R. W., F. Honsell, and G. D. Plotkin (1993, January). A framework for defining logics. *Journal of the ACM 40*(1), 143–184.

Hayashi, S. and H. Nakano (1988). *PX: A Computational Logic*. The MIT Press.

Hudak, P., S. L. Peyton Jones, and P. Wadler (1992, May). Report on the programming language Haskell, a non-strict purely-functional programming language, Version 1.2. *SIGPLAN Notices 27*(5).

Hughes, J., L. Pareto, and A. Sabry (1996). Proving the correctness of reactive systems using sized types. In *Conference Record of 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 410–423.

Jay, C. and M. Sekanina (1996). Shape checking of array programs. Technical Report 96.09, University of Technology, Sydney, Australia.

Kreitz, C., M. Hayden, and J. Hickey (1998, July). A proof environment for the development of group communication systems. In H. Kirchner and C. Kirchner (Eds.), *15th International Conference on Automated Deduction*, LNAI 1421, Lindau, Germany, pp. 317–332. Springer-Verlag.

Milner, R., M. Tofte, and R. W. Harper (1990). *The Definition of Standard ML*. Cambridge, Massachusetts: MIT Press.

Moggi, E. (1989, June). Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, Pacific Grove, California, pp. 14–23.

Necula, G. (1997). Proof-carrying code. In *Conference Record of 24th Annual ACM Symposium on Principles of Programming Languages*, pp. 106–119. ACM press.

Necula, G. and P. Lee (1998, June). The design and implementation of a certifying compiler. In *ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pp. 333–344. ACM press.

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

Parent, C. (1995). Synthesizing proofs from programs in the calculus of inductive constructions. In *Proceedings of the International Conference on Mathematics for Programs Constructions*. Springer-Verlag LNCS 947.

Pierce, B. and D. Turner (1998). Local type inference. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 252–265.

Sabry, A. and M. Felleisen (1993). Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation 6*(3/4), 289–360.

Sannella, D. and A. Tarlecki (1989, February). Toward formal development of ML programs: Foundations and methodology. Technical Report ECS-LFCS-89-71, Laboratory for Foundations of Computer Science, Depatment of Computer Science, University of Edinburgh.

Sulzmann, M., M. Odersky, and M. Wehr (1997). Type inference with constrained types. In *Proceedings of 4th International Workshop on Foundations of Object-Oriented Languages*.

Weis, P. and X. Leroy (1993). *Le langage Caml*. Paris: InterEditions.

Xi, H. (1997, November). Some examples of DML programming. Available at
`http://www.cs.cmu.edu/~hwxi/DML/examples/`.

Xi, H. (1998). *Dependent Types in Practical Programming*. Ph. D. thesis, Carnegie Mellon University. pp. viii+189. Forthcoming. The current version is available as
`http://www.cs.cmu.edu/~hwxi/DML/thesis.ps`.

Xi, H. (1999, January). Dead code elimination through dependent types. In *The First International Workshop on Practical Aspects of Declarative Languages*, San Antonio, Texas. To appear.

Xi, H. and F. Pfenning (1998, June). Eliminating array bounds checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 249–257.

Zenger, C. (1997). Indexed types. *Theoretical Computer Science 187*, 147–165.

Zenger, C. (1998). *Indizierte Typen*. Ph. D. thesis, Fakultät für Informatik, Universität Karlsruhe. Forthcoming.

A Further Examples

We present some additional examples for those who may have difficulty accessing (Xi 1997), where complete these and other larger examples are available.

An AVL tree is a balanced binary tree such that for every interior node the difference between the heights of its two sons is at most one. The data structure in Figure 9 precisely declares the type of AVL trees.

Untyped λ-expressions in de Bruijn form and an implementation of evaluation are given in Figure 10. The dependent type checker verifies that no dangling de Bruijn references can occur during evaluation of a closed expression.

This can be extended to verify type-safety statically, but requires a symbolic constraint domain. Assume that we have sorts *type* and *context* and the following constants.

$$\begin{array}{rcl} unit & : & type \\ arrow & : & type * type \to type \\ empty & : & context \\ :: & : & type * context \to context \end{array}$$

In Figure 11, the datatype *lamexp*, declared in Figure 2, is refined to formulate the type of simply typed λ-expressions. Note that *lamexp* is indexed with a pair (t, ctx), where t stands for the simple type of a λ-expression and ctx records the types of free variables in the λ-expression. Therefore the DML type of closed well-typed λ-expressions is $\Sigma t : type.lamexp(t, empty)$.

Lastly, we present a short implementation of quicksort on lists in Figure 12, where the type guarantees that this implementation always returns a list of length n when given one of length n. Note that we use :: as an infix operator for *cons*.

B Elaboration Rules for $\mathrm{ML}_0^{\Pi}(C)$

We present the elaboration rules in Figures 13 and 14. Note that some rules have (obvious) side conditions, which can be found in (Xi 1998).

```
datatype 'a tree = empty | branch of int * 'a * 'a tree * 'a tree
                                (* height, key, left son, right son *)

typeref 'a tree of nat with (* the index stands for the height *)
  empty <| 'a tree(0)
| branch <| {lh:nat}{rh:nat | rh - 1 <= lh <= rh + 1}
               int(1+max(lh,rh)) * 'a * 'a tree(lh) * 'a tree(rh) ->
               'a tree(1+max(lh,rh))
```

Figure 9: AVL trees

```
datatype lamexp =
     One | Shift of lamexp | Abs of lamexp | App of lamexp * lamexp

typeref lamexp of int
with One <| {n:nat} lamexp(n+1)
   | Shift <| {n:nat} lamexp(n) -> lamexp(n+1)
   | Abs <| {n:nat} lamexp(n+1) -> lamexp(n)
   | App <| {n:nat} lamexp(n) * lamexp(n) -> lamexp(n)

datatype closure = Closure of lamexp * env
     and env = Nil | Cons of closure * env

typeref env of int
with Nil <| env(0)
   | Cons <| {n:nat} closure * env(n) -> env(n+1)
   | Closure <| {n:nat} lamexp(n) * env(n) -> closure

fun callbyvalue(exp) =
    let
        fun cbv(One, Cons(clo, env)) = clo
          | cbv(Shift(exp), Cons(clo, env)) = cbv(exp, env)
          | cbv(Abs(exp), env) = Closure(Abs(exp), env)
          | cbv(App(fexp, exp), env) =
            let
                val Closure(Abs(body), env1) = cbv(fexp, env)
                val clo = cbv(exp, env)
            in
                cbv(body, Cons(clo, env1))
            end
          (* exhaustiveness of these cases follows from the dependent types *)
        where cbv <| {n:nat} lamexp(n) * env(n) -> closure
    in
        cbv(exp, Nil)
    end
where callbyvalue <| lamexp(0) -> closure
```

Figure 10: Closed λ-expressions and evaluation

```
datatype lamexp = One | Shift of lamexp | Abs of lamexp | App of lamexp * lamexp

typeref lamexp of type * context with (* index lamexp with a pair (t, ctx) *)
  One <| {t:type}{ctx:context} lamexp(t,t::ctx)
| Shift <| {t1:type}{t2:type}{ctx:context} lamexp(t1,ctx) -> lamexp(t1,t2::ctx)
| Abs <| {t1:type}{t2:type}{ctx:context}
         lamexp(t2,t1::ctx) -> lamexp(arrow(t1,t2),ctx)
| App <| {t1:type}{t2:type}{ctx:context}
         lamexp(arrow(t1,t2),ctx) * lamexp(t1,ctx) -> lamexp(t2,ctx)
```

Figure 11: Simply-typed λ-expressions

```
fun('a) quickSort cmp [] = []
      | quickSort cmp (x::xs) = par cmp (x, [], [], xs)
where quickSort <| {n:nat} ('a * 'a -> bool) -> 'a list(n) -> 'a list(n)

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

Figure 12: Quicksort on lists

$$\frac{\phi;\Gamma\vdash e\uparrow\Pi a:\gamma.\tau\Rightarrow e^* \quad \phi\vdash i:\gamma}{\phi;\Gamma\vdash e\uparrow\tau[a:=i]\Rightarrow e^*[i]}\ \textbf{(elab-pi-elim)} \qquad \frac{\phi,a:\gamma;\Gamma\vdash e\downarrow\tau\Rightarrow e^*}{\phi;\Gamma\vdash e\downarrow\Pi a:\gamma.\tau\Rightarrow(\lambda a:\gamma.e^*)}\ \textbf{(elab-pi-intro)}$$

$$\frac{\Gamma(x)=\tau \quad \phi\vdash\Gamma[\mathbf{ctx}]}{\phi;\Gamma\vdash x\uparrow\tau\Rightarrow x}\ \textbf{(elab-var-up)} \qquad \frac{\phi;\Gamma\vdash x\uparrow\mu_1\Rightarrow e^* \quad \phi\models\mu_1\equiv\mu_2}{\phi;\Gamma\vdash x\downarrow\mu_2\Rightarrow e^*}\ \textbf{(elab-var-down)}$$

$$\frac{\mathcal{S}(c)=\Pi a_1:\gamma_1\ldots\Pi a_n:\gamma_n.\delta(i) \quad \phi\vdash i_1:\gamma_1\ \cdots\ \phi\vdash i_n:\gamma_n}{\phi;\Gamma\vdash c\uparrow\delta(i[a_1,\ldots,a_n:=i_1,\ldots i_n])\Rightarrow c[i_1]\ldots[i_n]}\ \textbf{(elab-cons-wo-up)}$$

$$\frac{\phi;\Gamma\vdash c\uparrow\mu_1\Rightarrow e^* \quad \phi\models\mu_1\equiv\mu_2}{\phi;\Gamma\vdash c\downarrow\mu_2\Rightarrow e^*}\ \textbf{(elab-cons-wo-down)}$$

$$\frac{\begin{array}{c}\mathcal{S}(c)=\Pi a_1:\gamma_1\ldots\Pi a_n:\gamma_n.\tau\to\delta(i)\\ \phi;\Gamma\vdash e\downarrow\tau[a_1,\ldots,a_n:=i_1,\ldots i_n]\Rightarrow e^*\\ \phi\vdash i_1:\gamma_1\ \cdots\ \phi\vdash i_n:\gamma_n\end{array}}{\phi;\Gamma\vdash c(e)\uparrow\delta(i[a_1,\ldots,a_n:=i_1,\ldots i_n])\Rightarrow c[i_1]\ldots[i_n](e^*)}\ \textbf{(elab-cons-w-up)}$$

$$\frac{\phi;\Gamma\vdash c(e)\uparrow\mu_1\Rightarrow e^* \quad \phi\models\mu_1\equiv\mu_2}{\phi;\Gamma\vdash c(e)\downarrow\mu_2\Rightarrow e^*}\ \textbf{(elab-cons-w-down)}$$

Figure 13: Elaboration rules for $\mathrm{ML}_0^{\Pi}(C)$, part I

$$\frac{}{\phi; \Gamma \vdash \langle\rangle \uparrow \mathbf{1} \Rightarrow \langle\rangle}\ \textbf{(elab-unit-up)} \qquad \frac{}{\phi; \Gamma \vdash \langle\rangle \downarrow \mathbf{1} \Rightarrow \langle\rangle}\ \textbf{(elab-unit-down)}$$

$$\frac{\phi; \Gamma \vdash e_1 \uparrow \mu_1 \Rightarrow e_1^* \quad \phi; \Gamma \vdash e_2 \uparrow \mu_2 \Rightarrow e_2^*}{\phi; \Gamma \vdash \langle e_1, e_2\rangle \uparrow \mu_1 * \mu_2 \Rightarrow \langle e_1^*, e_2^*\rangle}\ \textbf{(elab-prod-up)}$$

$$\frac{\phi; \Gamma \vdash e_1 \downarrow \tau_1 \Rightarrow e_1^* \quad \phi; \Gamma \vdash e_2 \downarrow \tau_2 \Rightarrow e_2^*}{\phi; \Gamma \vdash \langle e_1, e_2\rangle \downarrow \tau_1 * \tau_2 \Rightarrow \langle e_1^*, e_2^*\rangle}\ \textbf{(elab-prod-down)}$$

$$\frac{p \downarrow \tau_1 \Rightarrow (p^*; \phi'; \Gamma') \quad \phi, \phi'; \Gamma, \Gamma' \vdash e \downarrow \tau_2 \Rightarrow e^* \quad \phi \vdash \tau_2 : *}{\phi; \Gamma \vdash (p \Rightarrow e) \downarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow (p^* \Rightarrow e^*)}\ \textbf{(elab-match)}$$

$$\frac{\phi; \Gamma \vdash (p \Rightarrow e) \downarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow (p^* \Rightarrow e^*) \quad \phi; \Gamma \vdash ms \downarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow ms^*}{\phi; \Gamma \vdash (p \Rightarrow e \mid ms) \downarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow (p^* \Rightarrow e^* \mid ms^*)}\ \textbf{(elab-matches)}$$

$$\frac{\phi; \Gamma \vdash e \uparrow \tau_1 \Rightarrow e^* \qquad \phi; \Gamma \vdash ms \downarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow ms^*}{\phi; \Gamma \vdash (\mathbf{case}\ e\ \mathbf{of}\ ms) \downarrow \tau_2 \Rightarrow (\mathbf{case}\ e^*\ \mathbf{of}\ ms^*)}\ \textbf{(elab-case)}$$

$$\frac{\phi; \Gamma, x : \tau_1 \vdash e \downarrow \tau_2 \Rightarrow e^*}{\phi; \Gamma \vdash (\mathbf{lam}\ x.e) \downarrow \tau_1 \to \tau_2 \Rightarrow (\mathbf{lam}\ x : \tau_1.e^*)}\ \textbf{(elab-lam)}$$

$$\frac{\phi; \Gamma, x_1 : \tau_1, x : \tau \vdash e \downarrow \tau_2 \Rightarrow e^* \quad \phi; \Gamma, x_1 : \tau_1 \vdash x_1 \downarrow \tau \Rightarrow e_1^*}{\phi; \Gamma \vdash (\mathbf{lam}\ x : \tau.e) \downarrow \tau_1 \to \tau_2 \Rightarrow (\mathbf{lam}\ x_1 : \tau_1.\mathbf{let}\ x = e_1^*\ \mathbf{in}\ e^*\ \mathbf{end})}\ \textbf{(elab-lam-anno)}$$

$$\frac{\phi; \Gamma \vdash e_1 \uparrow \tau_1 \to \tau_2 \Rightarrow e_1^* \quad \phi; \Gamma \vdash e_2 \downarrow \tau_1 \Rightarrow e_2^*}{\phi; \Gamma \vdash e_1(e_2) \uparrow \tau_2 \Rightarrow e_1^*(e_2^*)}\ \textbf{(elab-app-up)}$$

$$\frac{\phi; \Gamma \vdash e_1(e_2) \uparrow \mu_1 \Rightarrow e^* \quad \phi \models \mu_1 \equiv \mu_2}{\phi; \Gamma \vdash e_1(e_2) \downarrow \mu_2 \Rightarrow e^*}\ \textbf{(elab-app-down)}$$

$$\frac{\phi; \Gamma \vdash e_1 \uparrow \tau_1 \Rightarrow e_1^* \quad \phi; \Gamma, x : \tau_1 \vdash e_2 \uparrow \tau_2 \Rightarrow e_2^*}{\phi; \Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} \uparrow \tau_2 \Rightarrow \mathbf{let}\ x = e_1^*\ \mathbf{in}\ e_2^*\ \mathbf{end}}\ \textbf{(elab-let-up)}$$

$$\frac{\phi; \Gamma \vdash e_1 \uparrow \tau_1 \Rightarrow e_1^* \quad \phi; \Gamma, x : \tau_1 \vdash e_2 \downarrow \tau_2 \Rightarrow e_2^*}{\phi; \Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} \downarrow \tau_2 \Rightarrow \mathbf{let}\ x = e_1^*\ \mathbf{in}\ e_2^*\ \mathbf{end}}\ \textbf{(elab-let-down)}$$

$$\frac{\phi; \Gamma, f : \tau \vdash v \downarrow \tau \Rightarrow v^*}{\phi; \Gamma \vdash (\mathbf{fix}\ f : \tau.v) \uparrow \tau \Rightarrow (\mathbf{fix}\ f : \tau.v^*)}\ \textbf{(elab-fix-up)}$$

$$\frac{\phi; \Gamma, f : \tau \vdash v \downarrow \tau \Rightarrow v^* \quad \phi; \Gamma, x : \tau \vdash x \downarrow \tau' \Rightarrow e^*}{\phi; \Gamma \vdash (\mathbf{fix}\ f : \tau.v) \downarrow \tau' \Rightarrow \mathbf{let}\ x = (\mathbf{fix}\ f : \tau.v^*)\ \mathbf{in}\ e^*\ \mathbf{end}}\ \textbf{(elab-fix-down)}$$

$$\frac{\phi; \Gamma \vdash e \downarrow \tau \Rightarrow e^*}{\phi; \Gamma \vdash (e : \tau) \uparrow \tau \Rightarrow e^*}\ \textbf{(elab-anno-up)} \qquad \frac{\phi; \Gamma \vdash (e : \tau) \uparrow \mu_1 \Rightarrow e^* \quad \phi \models \mu_1 \equiv \mu_2}{\phi; \Gamma \vdash (e : \tau) \downarrow \mu_2 \Rightarrow e^*}\ \textbf{(elab-anno-down)}$$

Figure 14: Elaboration rules for $\mathrm{ML}_0^{\Pi}(C)$, part II