A Schema for Adding Dependent Types to ML*

Hongwei Xi
Department of Mathematical Sciences
Carnegie Mellon University
hwxi@cs.cmu.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 form of dependent types, where index objects are restricted to constraint domains C, leading to the DML(C) language schema. Pure inference for the resulting system is no longer possible, but we show that type-checking a sufficiently annotated program can be reduced to constraint satisfaction. We prove that DML(C) is conservative over ML, but the main technical contribution of the paper lies in our language design, including its elaboration and type-checking rules which make the approach practical. This has been demonstrated in related experiments where we obtained significant speedups in many examples by statically eliminating array bound checks. There constraints are linear equalities and inequalities over integers, solved by a variant of Fourier's method.

1 Introduction

Type systems for functional languages can be broadly classified into those for rich, realistic languages such as Standard ML[10], CAML[19], or Haskell[6], and those for small, pure languages such as the ones underlying Coq[2], NuPrl[1], or PX[5]. Type checking and inference in realistic languages is theoretically decidable and 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. 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 proofs from CAML-like programs [15], or by embedding fragments of ML into NuPrl [8]. In this paper we take a different approach, conservatively refining the type system of ML by allowing some dependencies, without destroying the desirable properties of ML such as practical and unintrusive type checking. Note that this is quite different from the use of dependent types to analyze modular structure (as, for example, in [9]).

We now present a brief example from our implementation before going into further details. A correct implementation of a reverse function on lists should return a list of the same length as its argument. Unfortunately, this property cannot be captured by the ML's type system. The inadequacy can be remedied if we introduce *dependent types*.

The code in Figure 1 is written in the style of Standard ML with some type annotations, which will be explained shortly. We assume that we are working over the domain of natural numbers with constants 0 and 1 and 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:

*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.

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

fun('a) reverse(l) = let
    fun aux(nil, ys) = ys
      | aux(cons(x, xs), ys) = aux(xs, cons(x, ys))
    where aux <| {m:nat,n:nat} 'a list(m) * 'a list(n) -> 'a list(m+n)
in aux(l, []) end
where reverse <| {n:nat} 'a list(n) -> 'a list(n)
```

Figure 1: An introductory example: reverse

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

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 which may be used as type indices. 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 array bound checking.

The second question is the decidability and practicality of type-checking. We address this in two steps: the first step is to define an explicit (and unacceptably verbose) language $\mathrm{ML}_0^{\Pi}(C)$ for which type checking is easily reduced to the satisfiability problem for the constraint domain C. The second step is to define an elaboration from $\mathrm{DML}(C)$, a slightly extended fragment of ML, to the fully explicit 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 types, although they introduce non-trivial technical complications into the elaboration procedure. For the practical treatment of existential types, see [21]. The theoretical analysis of existential types and their properties is beyond the scope of this abstract.

We have implemented our design for a fragment of ML including definitions, recursion, datatypes, pattern matching and polymorphism, and experimented with different constraint domains and applications. Many of the examples are available at `http://www.cs.cmu.edu/~hwxi/mini_examples/`. For the domain of integer equalities and inequalities, they include quicksort, byte copy, Knuth-Morris-Pratt string matching and others in which array bound checks can be statically eliminated without requiring excessive annotation. On symbolic domains we verify that the red/black invariant for binary trees is preserved in a dictionary library, and verify type preservation for an interpreter of the simply-typed λ-calculus by checking dependent types.

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 less than 20% of the entire code, and the resulting constraint simplification problems can be solved in practice. Also the annotations are mechanically verified, and therefore can be fully trusted as program documentation.

base types	β	::=	bool | int | (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 2: The syntax for ML_0

The remainder of this extended abstract is organized as follows. We introduce the notion of constraint domain in Section 3. We then present in Section 4 the language $\mathrm{ML}_0^{\Pi}(C)$ parameterized over a constraint domain C, its typing rules and its operational semantics. In Section 5, we give the rules for elaboration from $\mathrm{DML}(C)$ to $\mathrm{ML}_0^{\Pi}(C)$, and prove its correctness. We explain the need for existential dependent types in Section 6 and extend $\mathrm{ML}_0^{\Pi}(C)$ to $\mathrm{ML}_0^{\Pi,\Sigma}(C)$. In the rest of the paper we discuss some related work and conclude.

2 Mini-ML with Pattern Matching

We start with a 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 nontrivial. Polymorphism, on the other hand, is largely orthogonal and therefore omitted here, although it is supported in the implementation with a value restriction in order to obtain soundness as in SML'97 [11] or CAML [19]. The syntax of ML_0 is given in Figure 2. Our own source language $\mathrm{DML}(C)$ will have essentially the same syntax, except that there is a richer language for types.

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

3 Constraints

Our enriched language will be parameterized over a domain of constraints from which the index objects for types are drawn. Typical examples include linear equalities and 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 propositions and *int* for integers. We use f for interpreted functions symbols, p for atomic predicates (that is, functions of sort $\gamma \to \mathit{bool}$) and we assume to have constants such as equality, truth values $\top$ and $\bot$, negation $\neg$, conjunction $\wedge$, and disjunction $\vee$, all of which are interpreted as usual.

$$\text{index sorts} \quad \gamma \quad ::= \quad b \mid \mathbf{1} \mid \gamma_1 * \gamma_2 \mid \{a : \gamma \mid \mathrm{p}(a)\}$$

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

index objects	i, j	::=	$a \mid () \mid (i, j) \mid \mathrm{f}(i)$
index contexts	ϕ	::=	$\cdot \mid \phi, a : \gamma \mid \phi, \mathrm{p}(i)$
index constraints	Φ	::=	$\mathrm{p}(i) \mid \Phi_1 \wedge \Phi_1 \mid \mathrm{p}(i) \supset \Phi \mid \forall a : \gamma.\Phi \mid \exists a : \gamma.\Phi$

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 equalities and inequalities, which our implementation solves by a variant of Fourier's method. Empirical results and further references can be found in [20].

4 ML_0 with Dependent Types

We now present $\mathrm{ML}_0^\Pi(C)$ with dependent types, which is an extension of ML_0. Given a domain C of constraints, the syntax of $\mathrm{ML}_0^\Pi(C)$ is given as follows. We use δ for base types or base type families, where we use $\delta()$ for an unindexed type.

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	σ	::=	$\delta(i) \mid \mathbf{1} \mid (\tau_1 * \tau_2) \mid (\tau_1 \to \tau_2)$
types	τ	::=	$\sigma \mid (\Pi a : \gamma.\tau)$
patterns	p	::=	$x \mid c[a_1] \ldots [a_n](p) \mid \langle\rangle \mid \langle p_1, p_2\rangle$
matches	ms	::=	$(p \Rightarrow e) \mid (p \Rightarrow e \mid ms)$
expressions	e	::=	$x \mid \langle\rangle \mid \langle e_1, e_2\rangle \mid c[i_1] \ldots [i_n](e) \mid (\mathbf{case}\ e\ \mathbf{of}\ ms)$
			$\mid (\lambda a : \gamma.e) \mid e[i] \mid (\mathbf{lam}\ x : \tau.e) \mid e_1(e_2)$
			$\mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} \mid (\mathbf{fix}\ f : \tau.v)$
values	v	::=	$x \mid c[i_1] \ldots [i_n](v) \mid \langle\rangle \mid \langle v_1, v_2\rangle \mid (\mathbf{lam}\ x : \tau.e) \mid (\lambda a : \gamma.v)$
contexts	Γ, Δ	::=	$\cdot \mid \Gamma, x : \tau$
substitutions	θ	::=	$[] \mid \theta[x \mapsto v] \mid \theta[a \mapsto i]$

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 $\phi \vdash \delta(i) \equiv \delta(j)$, which is the congruent extension of equality on index objects to arbitrary types.

The only significant complication arises from pattern matching where new index assumptions φ 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.

The judgment $p \downarrow \sigma \rhd (\varphi; \Delta)$ expresses that the index and ordinary variables in pattern p have the types in φ and Δ, respectively, if we know that p must have type σ.

$$\frac{}{x \downarrow \tau \rhd (\cdot; x : \tau)} \qquad \frac{}{\langle\rangle \downarrow \mathbf{1} \rhd (\cdot; \cdot)} \qquad \frac{p_1 \downarrow \tau_1 \rhd (\varphi_1; \Delta_1) \quad p_2 \downarrow \tau_2 \rhd (\varphi_2; \Delta_2)}{\langle p_1, p_2\rangle \downarrow \tau_1 * \tau_2 \rhd (\varphi_1, \varphi_2; \Delta_1, \Delta_2)}$$

$$\frac{\mathcal{S}(c) = \Pi a_1 : \gamma_1 \ldots \Pi a_n : \gamma_n.(\tau \to \delta(i)) \quad p \downarrow \tau \rhd (\varphi_1; \Delta)}{c[a_1] \ldots [a_n](p) \downarrow \delta(j) \rhd (a_1 : \gamma_1, \ldots, a_n : \gamma_n, \varphi_1, i \doteq j; \Delta)}$$

The judgment for match expressions, $\phi; \Gamma \vdash ms : \sigma \Rightarrow \tau$ checks independently for each case that, given a subject of type σ the case branch will have type τ. This will always be satisfied for branches which can be seen as impossible based purely on the type of the case subject,but it is not precise enough for some programs which are correct only when patterns are matched sequentially. This has been a minor problem in our experiments to date since one can always decompose patterns into disjoint ones by hand if necessary, but a refinement of the present strategy is likely to be necessary on symbolic index domains and will require further research.

$$\frac{p \downarrow \sigma \rhd (\varphi; \Delta) \quad \phi, \varphi; \Gamma, \Delta \vdash e : \tau}{\phi; \Gamma \vdash p \Rightarrow e : \sigma \Rightarrow \tau} \qquad \frac{\phi; \Gamma \vdash (p \Rightarrow e) : \sigma \Rightarrow \tau \quad \phi; \Gamma \vdash ms : \sigma \Rightarrow \tau}{\phi; \Gamma \vdash (p \Rightarrow e \mid ms) : \sigma \Rightarrow \tau}$$

$$\frac{\phi;\Gamma \vdash e : \tau_1 \quad \phi \vdash \tau_1 \equiv \tau_2}{\phi;\Gamma \vdash e : \tau_2} \qquad \frac{\Gamma(x) = \tau}{\phi;\Gamma \vdash x : \tau} \qquad \frac{\mathcal{S}(c) = \tau}{\phi;\Gamma \vdash c : \tau}$$

$$\frac{}{\phi;\Gamma \vdash \langle\rangle : \mathbf{1}} \qquad \frac{\phi;\Gamma \vdash e_1 : \tau_1 \quad \phi;\Gamma \vdash e_2 : \tau_2}{\phi;\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 * \tau_2}$$

$$\frac{\phi;\Gamma \vdash e : \sigma \quad \phi;\Gamma \vdash ms : \sigma \Rightarrow \tau}{\phi;\Gamma \vdash (\mathbf{case}\ e\ \mathbf{of}\ ms) : \tau}$$

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

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

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

Figure 3: Typing Rules for $\mathrm{ML}_0^\Pi(C)$

The remaining typing rules for $\mathrm{ML}_0^\Pi(C)$ are in Figure 3.

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

For the sake of brevity, we omit the operational semantics in this extended abstract. Given the remarks above, it is straightforward to define $e \overset{c}{\hookrightarrow} v$ 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 in $\mathrm{ML}_0^\Pi(C)$) Given e, v in $\mathrm{ML}_0^\Pi(C)$ such that $e \overset{c}{\hookrightarrow} v$ 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 $e \overset{c}{\hookrightarrow} v$ 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 *The erasure function $|\cdot|$ is defined in Figure 4, which 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.*

A program written in $\mathrm{ML}_0^\Pi(C)$ is executed in ML. The next theorem guarantees that the 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 $\phi;\Gamma \vdash e : \tau$. ∎

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 $e \overset{c}{\hookrightarrow} v$ derivable in $\mathrm{ML}_0^\Pi(C)$, then $|e| \overset{m}{\hookrightarrow} |v|$ is derivable.*

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

Figure 4: The definition of erasure function $|\cdot|$

Proof By a structural induction on the derivation of $e \overset{c}{\hookrightarrow} v$. ∎

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 $|e| \overset{m}{\hookrightarrow} v^*$ is derivable for some v^* in ML_0, then there exists v in $\mathrm{ML}_0^{\Pi}(C)$ such that $e \overset{c}{\hookrightarrow} v$ and $|v| = v^*$.*

Proof By a structural induction on the derivations of $|e| \overset{m}{\hookrightarrow} v^*$ 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.*

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* $\mathrm{DML}(C)$ together with a mapping to the *internal language* $\mathrm{ML}_0^{\Pi}(C)$. This mapping is called *elaboration.*

5.1 The External Language $\mathrm{DML}(C)$ for $\mathrm{ML}_0^{\Pi}(C)$

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

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

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 some intuition behind the 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. The synthesizing judgement then has the form $\phi;\Gamma \vdash e \uparrow \tau \Rightarrow e^*$ and means that e elaborates into e^* *with*

type τ. The checking judgement 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 first two rules is to eliminate universal 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 **pi-elim** to remove the quantifier in the type; we continue doing so until a major type is reached, which must be of form $\tau_1 \to \tau_2$ (if the code is type-correct). Note that the actually index i is not locally determined, but becomes an existential variable for the constraint solver. The rule **pi-intro** is simpler since we check against a given dependent functional type.

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

The next rule 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).

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

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

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

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, but in any case will require a type annotation, as illustrated in one of the two rules for let-expressions below.

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

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 be writing **let** $x : \tau = e_1$ **in** e_2 **end** which abbreviates **let** $x = (e_1 : \tau)$ **in** e_2 **end**. The following rule allows us to take advantage of such annotations.

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

As a result, the only types appearing 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 environments φ and Δ, respectively. This is written as $p \downarrow \tau \Rightarrow (p^*; \varphi; \Delta)$ in Figure 5. This judgment is used in the rule for pattern matching: The generated constraints φ are assumed into the context while elaborating e. For constraint satisfaction, these are treated as hypotheses.

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

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

$$\frac{p_1 \downarrow \tau_1 \Rightarrow (p_1^*; \varphi_1; \Delta_1) \quad p_2 \downarrow \tau_2 \Rightarrow (p_2^*; \varphi_2; \Delta_2)}{\langle p_1, p_2\rangle \downarrow \tau_1 * \tau_2 \Rightarrow (\langle p_1^*, p_2^*\rangle; \varphi_1, \varphi_2; \Delta_1, \Delta_2)}$$

$$\frac{\mathcal{S}(c) = \Pi a_1 : \gamma_1 \ldots \Pi a_n : \gamma_n.\tau \to \delta(i) \quad p \downarrow \tau \Rightarrow (p^*; \varphi; \Delta)}{c(p) \downarrow \delta(j) \Rightarrow (c[a_1]\ldots[a_n](p^*); a_1 : \gamma_1, \ldots, a_n : \gamma_n, \varphi, i \doteq j; \Delta)}$$

Figure 5: The elaboration rules for patterns

Lemma 5.1 *If $p \downarrow \sigma \Rightarrow (p^*; \varphi; \Delta)$ is derivable, then $p = |p^*|$ and $p^* \downarrow \sigma \triangleright (\varphi; \Delta)$ is derivable.*

The complete elaboration rules for DML(C) are listed in Appendix A can be justified by the following theorem.

Theorem 5.2 *We have the following.*

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

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. We refer the interested reader to [20] for an example on elaboration and constraint generation.

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 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 with some length when applied to a list of some length. This yields an approach to handling existing functions such as those in a library, whose definitions may not available.

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

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

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

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

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 easy to give a sound elaboration procedure for for DML(C) for this richer type language (which is also part of our implementation), but there is no clear and canonical choice for where existential types are to be unpacked. The theoretical analysis of elaboration for the extended type language is therefore beyond the scope of this extended abstract. We refer the interested reader to [21].

7 Related work

Our work falls in between full program verification, either in type theory or systems such as PVS [14], 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, existing ML programs will work as before (if there is no keyword conflict).

When compared to traditional type systems for programming languages, perhaps the closest related work is refinement types [3], 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[15] proposed to reverse the process of extracting programs from constructive proofs in Coq[2], 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 then in some cases 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.

Hayashi proposed a type system ATTT [4], which allows a notion of refinement types as in the type system for ML[3], plus intersection and union of refinement types and singleton refinement types. He demonstrated the value of singleton, union and intersection types in extracting realistic programs, which is similar to our use of corresponding logical operators on constraints. However, he does not address the practical problem of type checking or partial inference.

Sannella and Tarlecki proposed *Extended ML* [17] 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 leads to the need of theorem proving during 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.

Kreitz [8] embedded a subset of the OCaml programming language into NuPrl for reasoning about programs written in this subset. The objective is to develop automated tools for the verification and optimization of group communication systems. This embedding translates programing language constructs into type-theoretic expressions, which is a rather involved task requiring complex reasoning about the resulting expressions. We believe that our approach could be used to check some of these properties.

Jay and Sekanina [7] 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 C consists of linear integer equality and inequality constraints. We feel that their design is more restrictive and seems more promising for languages based on iteration schemas rather than general recursion.

Finally, recent work by Pierce and Turner [16] which includes some empirical studies, is based on a similar bi-directional strategy, 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, [18]).

8 Conclusion

We have designed a practical class of refinements of ML by allowing dependencies based on constraints. Type annotations are required, but not overly verbose, and the resulting type-checking algorithm has shown itself to be practical for typical programs and constraint domains, such as linear equalities and inequalities over integers for array-bounds checking.

In future work, we plan to extend $\mathrm{ML}_0^{\Pi}(C)$ to full Standard ML. The critical issues are exceptions, mutable references, and module-level constructs. Since our design explicitly separates indices from ML expressions, we expect these extensions to be mostly straightforward. Another practically important extension may be the introduction of limited forms of intersection types [3], 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 [12] 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 [13].

9 Acknowledgement

We thank Peter Lee and George Necula for providing us with many interesting examples We also gratefully acknowledge our discussion with Rowan Davies.

References

[1] Constable, R. L. et al. **Implementing Mathematics with the Nuprl Proof Development System**. Prentice-Hall, Englewood Cliffs, New Jersey, 1986, x+299 pp.

[2] 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 Techniques, no. 154, INRIA, Rocquencourt, France, 1993. *Version 5.8*.

[3] Freeman, T. and Pfenning, F. *Refinement types for ML*. in: **Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation**, Toronto, Ontario, June. 1991, pp. 268–277.

[4] Hayashi, S. *Singleton, union and intersection types for program extraction.* in: **Proceedings of the International Conference on Theoretical Aspects of Computer Software**, Sendai, Japan, September, edited by A. R. Meyer. 1991.

[5] Hayashi, S. and Nakano, H. **PX: A Computational Logic**. The MIT Press, 1988.

[6] Hudak, P., Peyton Jones, S. L., Wadler, P., et al. *A Report on the Functional Language Haskell.* **SIGPLAN Notices**, 1992.

[7] Jay, C. and Sekanina, M. *Shape checking of array programs.* no. 96.09, School of Computer Sciences, University of Technology, Sydney, Australia, 1996.

[8] Kreitz, C. *Formal Reasoning about Communication Systems I: Embedding ML into Type Theory.* Cornell University, Department of Computer Science, June 1997.

[9] MacQueen, D. B., Plotkin, G. D., and Sethi, R. *An Ideal Model for Recursive Polymorphic Types.* **Information and Control**, vol. 71 (1986), pp. 95–130.

[10] Milner, R., Tofte, M., and Harper, R. W. **The Definition of Standard ML**. MIT Press, Cambridge, Massachusetts, 1990, xi+101 pp.

[11] Milner, R., Tofte, M., Harper, R. W., and MacQueen, D. **The Definition of Standard ML**. MIT Press, Cambridge, Massachusetts, 1997.

[12] Necula, G. *Compiling with Proofs.* Thesis Proposal, Carnegie Mellon University, Department of Computer Science, April 1997.

[13] Necula, G. *Proof-Carrying Code.* in: **Conference Record of 24th Annual ACM Symposium on Principles of Programming Languages**, Paris, France, January, 15 – 17. 1997.

[14] Owre, S., Rajan, S., Rushby, J., Shankar, N., and Srivas, M. *PVS: Combining Specification, Proof Checking, and Model Checking.* in: **Computer-Aided Verification, CAV '96**, edited by R. Alur and T. A. Henzinger. **Lecture Notes in Computer Science**, vol. 1102, Springer-Verlag, New Brunswick, NJ, 1996, pp. 411–414.

[15] Parent, C. *Synthesizing proofs from programs in the Calculus of Inductive Constructions.* in: **Proceedings of Mathematics for Programs Constructions**. **Lecture Notes in Computer Science**, vol. 947, 1995.

[16] Pierce, B. and Turner, D. *Local Type Inference.* in: **Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**, San Diego, January 19–21. 1998.

[17] Sannella, D. and Tarlecki, A. *Toward Formal Development of ML Programs: Foundations and Methodology.* no. ECS-LFCS-89-71, Laboratory for Foundations of Computer Science, Depatment of Computer Science, University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, February 1989.

[18] Sulzmann, M., Odersky, M., and Wehr, M. *Type Inference with Constrained Types.* in: **Proceedings of 4th International Workshop on Foundations of Object-Oriented Languages**, Paris. 1997.

[19] Weis, P. and Leroy, X. **Le langage Caml**. InterEditions, Paris, 1993.

[20] Xi, H. and Pfenning, F. *Array Bounds Checking through Dependent Types.* November 1997. *Available as* `http://www.cs.cmu.edu/~hwxi/mini_papers/bounds.ps`.

[21] Xi, H. and Pfenning, F. *A Schema for Adding Dependent Types to ML.* July 1997. *Available as* `http://www.cs.cmu.edu/~hwxi/mini_papers/full.ps`.

A Elaboration rules for DML(C)

We list the elaboration rules for DML(C) as follows.

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

$$\frac{\Gamma(x) = \tau}{\phi;\Gamma \vdash x \uparrow \tau \Rightarrow x} \qquad \frac{\phi;\Gamma \vdash x \uparrow \sigma_1 \Rightarrow e^* \quad \phi \vdash \sigma_1 \equiv \sigma_2}{\phi;\Gamma \vdash x \downarrow \sigma_2 \Rightarrow e^*}$$

$$\frac{\mathcal{S}(c) = \tau}{\phi;\Gamma \vdash c \uparrow \tau \Rightarrow c} \qquad \frac{\phi;\Gamma \vdash c \uparrow \sigma_1 \Rightarrow e^* \quad \phi \vdash \sigma_1 \equiv \sigma_2}{\phi;\Gamma \vdash c \downarrow \sigma_2 \Rightarrow e^*}$$

$$\frac{}{\phi;\Gamma \vdash \langle\rangle \uparrow \mathbf{1} \Rightarrow \langle\rangle} \qquad \frac{}{\phi;\Gamma \vdash \langle\rangle \downarrow \mathbf{1} \Rightarrow \langle\rangle}$$

$$\frac{\phi;\Gamma \vdash e_1 \uparrow \sigma_1 \Rightarrow e_1^* \quad \phi;\Gamma \vdash e_2 \uparrow \sigma_2 \Rightarrow e_2^*}{\phi;\Gamma \vdash \langle e_1, e_2\rangle \uparrow \sigma_1 * \sigma_2 \Rightarrow \langle e_1^*, e_2^*\rangle}$$

$$\frac{\phi;\Gamma \vdash e_1 \downarrow \tau_1 \Rightarrow e_1^* \quad \phi;\Gamma \vdash e_2 \downarrow \tau_2 \Rightarrow e_2^*}{\phi;\Gamma \vdash \langle e_1, e_2\rangle \downarrow \tau_1 * \tau_2 \Rightarrow \langle e_1^*, e_2^*\rangle}$$

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

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

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

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

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

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

$$\frac{\phi;\Gamma \vdash e_1(e_2) \uparrow \sigma' \Rightarrow e^* \quad \phi \vdash \sigma' \equiv \sigma}{\phi;\Gamma \vdash e_1(e_2) \downarrow \sigma \Rightarrow e^*}$$

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

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

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

$$\frac{\phi;\Gamma, f:\tau \vdash v \downarrow \tau \Rightarrow v^* \quad \phi \vdash \tau \equiv \tau'}{\phi;\Gamma \vdash (\mathbf{fix}\ f:\tau.v) \downarrow \tau' \Rightarrow (\mathbf{fix}\ f:\tau.v^*)} \qquad \frac{\phi;\Gamma, f:\tau \vdash v \downarrow \tau \Rightarrow v^*}{\phi;\Gamma \vdash (\mathbf{fix}\ f.v) \downarrow \tau \Rightarrow (\mathbf{fix}\ f:\tau.v^*)}$$

$$\frac{\phi;\Gamma \vdash e \downarrow \tau \Rightarrow e^*}{\phi;\Gamma \vdash (e:\tau) \uparrow \tau \Rightarrow e^*} \qquad \frac{\phi;\Gamma \vdash (e:\tau) \uparrow \sigma_1 \Rightarrow e^* \quad \phi \vdash \sigma_1 \equiv \sigma_2}{\phi;\Gamma \vdash (e:\tau) \downarrow \sigma_2 \Rightarrow e^*}$$

Figure 6: The elaboration rules for DML(C)