*JFP* **17** (2): 215–286, 2007. © 2006 Cambridge University Press
doi:10.1017/S0956796806006216 First published online 11 December 2006 Printed in the United Kingdom

# *Dependent ML*
# *An approach to practical programming with dependent types*

HONGWEI XI*

*Computer Science Department, Boston University, Boston, MA, USA*
(*e-mail:* hwxi@cs.bu.edu)

**Abstract**

We present an approach to enriching the type system of ML with a restricted form of dependent types, where type index terms are required to be drawn from a given type index language $\mathscr{L}$ that is completely separate from run-time programs, leading to the DML($\mathscr{L}$) language schema. This enrichment allows for specification and inference of significantly more precise type information, facilitating program error detection and compiler optimization. The primary contribution of the paper lies in our language design, which can effectively support the use of dependent types in practical programming. In particular, this design makes it both natural and straightforward to accommodate dependent types in the presence of effects such as references and exceptions.

## 1 Introduction

In this paper, we report some research on supporting the use of dependent types in practical programming, drawing most of the results from (Xi, 1998). We do not attempt to incorporate into this paper some recent, closely related results (e.g., guarded recursive datatypes (Xi *et al.*, 2003), Applied Type System (Xi, 2004)), with which we only provide certain comparison.

Type systems for functional languages can be broadly classified into those for rich, realistic programming languages such as Standard ML (Milner *et al.*, 1997), Objective Caml (INRIA, n.d.), or Haskell (Peyton Jones *et al.*, 1999), and those for small, pure languages such as the ones underlying Coq (Dowek *et al.*, 1993), NuPrl (Constable *et al.*, 1986), or PX (Hayashi & Nakano, 1988). In practical programming, type-checking should be theoretically decidable as well as practically feasible for typical programs without requiring an overwhelmingly large number of type annotations. In order to achieve this, the type systems for realistic programming languages are often relatively simple, and only relatively elementary properties of programs can be expressed and thus checked by a type-checker. For instance, the error of taking the first element out of an empty list cannot be prevented by the type system of ML since it does not distinguish an empty list from a non-empty one.

*Supported in part by NSF grants no. CCR-0224244 and no. CCR-0229480.

```
datatype 'a list (int) =
    nil(0) | {n:nat} cons(n+1) of 'a * 'a list(n)

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

Fig. 1. An introductory example: appending lists.

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 type 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 *et al*., 1998). In this paper, we address the issue by designing a type system for practical programming that supports a restricted form of dependent types, allowing more program invariants to be captured by types. We conservatively extend the type system of ML by allowing some dependencies while maintaining practical and unintrusive type-checking. It will be shown that a program that is typable in the extended type system is already typable in ML. However, the program may be assigned a more precise type in the extended type system than in ML. It is in this sense we refer to the extended type system as a conservative extension of ML.

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 length $m$ and $n$, respectively. This property, however, cannot be captured by the type system of ML, and the 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. The declared type constructor **list** takes a type $\tau$ and a type index $n$ (of sort *int*) to form a type $(\tau)\mathbf{list}(n)$ for lists of length $n$ in which each element is of type $\tau$. The value constructors associated with **list** are then assigned certain dependent types:

- The syntax `nil(0)` states that the list constructor *nil* is assigned the type $\forall\alpha.(\alpha)\mathbf{list}(0)$, that is, *nil* is a list of length 0.
- The syntax `{n:nat} cons(n+1) of 'a * 'a list(n)` states that the list constructor *cons* is assigned the following type,

$$\forall\alpha.\Pi n:nat.\ \alpha * (\alpha)\mathbf{list}(n) \to (\alpha)\mathbf{list}(n+1)$$

that is, *cons* yields a list of length $n + 1$ when given a pair consisting of an element and a list of length $n$. We use *nat* for a subset sort defined as $\{a : int \mid a \geqslant 0\}$ and the syntax `{n:nat}` for a universal quantifier over type index variable $n$ of the subset sort *nat*.

```
fun ('a)
    filter p [] = []
  | filter p (x :: xs) = if p (x) then x :: filter p xs else filter p xs
withtype {m:nat} 'a list (m) -> [n:nat | n <= m] 'a list (n)
```

Fig. 2. Another introductory example: filtering lists.

The `withtype` clause in the definition of the function *append* is a type annotation, which precisely states that *append* returns a list of length $m+n$ when given a pair of lists of length $m$ and $n$, respectively. The annotated type can be formally written as follows:

$$\forall\alpha.\Pi m{:}nat.\Pi n{:}nat.\ (\alpha)\mathbf{list}(m) * (\alpha)\mathbf{list}(n) \to (\alpha)\mathbf{list}(m+n)$$

which we often call a universal dependent type. In general, the programmer is responsible for assigning dependent types to value constructors associated with a declared datatype constructor; he or she is also responsible for providing type annotations against which programs are automatically checked.

Adding dependent types to ML raises a number of theoretical and pragmatic questions. In particular, the kind of pure type inference in ML, which is certainly desirable in practice, becomes untenable, and a large portion of the paper is devoted to addressing various issues involved in supporting a form of partial type inference. We briefly summarize our results and design choices as follows.

The first question that arises is the meaning of expressions with effects when they occur as type index terms. In order to avoid the difficulty, we require that type index terms be pure. In fact, our type system is parameterized over a pure type index language from which type index terms are drawn. We can maintain this purity and still make the connection to run-time values by using *singleton types*, such as $\mathbf{int}(n)$, which is the type for integer expressions of value equal to $n$. This is critical for practical applications such as static elimination of array bound checks (Xi & 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 some type index language $\mathscr{L}$. The second step is to define an elaboration from DML($\mathscr{L}$), 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 clearly shows that existential dependent types, which are involved in nearly all the realistic examples in our experiments, are indispensable in practice. For instance, the function *filter* defined in Figure 2 is assigned the following types:

$$\forall\alpha.\Pi m{:}nat.\ (\alpha)\mathbf{list}(m) \to \Sigma n{:}\{a : nat \mid a \leqslant m\}.\ (\alpha)\mathbf{list}(n)$$

where $\{a : nat \mid a \leqslant m\}$ is a sort for natural numbers that are less than or equal to $m$. The type $\Sigma n : \{a : nat \mid a \leqslant m\}.\ (\alpha)\mathbf{list}(n)$, which is for lists of length less than or equal to $m$, is what we call an existential dependent type. The type assigned to *filter* simply means that the output list returned by *filter* cannot be longer than the input list taken by *filter*. Without existential dependent types, in order to assign a type to *filter*, we may have to compute in the type system the exact length of the output list returned by *filter* in terms of the input list and the predicate taken by *filter*. This would most likely make the type system too complicated for practical programming.

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, polymorphism, mutable references and exceptions. We have also implemented our design for a fragment of ML that encompasses all these features. In addition, we have experimented with different constraint domains and applications. Many non-trivial examples can be found in (Xi, 1999). At this point, we suggest that the reader first take a look at the examples in Section 7 so as to obtain a sense as to what can be effectively done in DML.

In our experience, DML($\mathscr{L}$) is acceptable from the pragmatic point of view: programs can often be annotated with little internal change and type annotations are usually concise and to the point. The resulting constraint simplification problems can be solved efficiently in practice once the type index language $\mathscr{L}$ is properly chosen. Also the type annotations are mechanically verified, and therefore can be fully trusted as program documentation.

The form of dependent types studied in this paper is substantially different from the usual form of dependent types in Martin-Löf's development of constructive type theory (Martin-Löf, 1984; Martin-Löf, 1985). In some earlier research work (Xi, 1998; Xi & Pfenning, 1999) on which this paper is largely based, the dependent types studied in this paper are called *a restricted form of dependent types*. From now on, we may also use the name *DML-style* dependent types to refer to such a restricted form of dependent types.

The remainder of the paper is organized as follows. In Section 2, we present as a starting point a simply typed language $\lambda_{pat}$, which essentially extends the simply typed $\lambda$-calculus with recursion and general pattern matching. We then formally describe in Section 3 how type index languages can be formed. In particular, we explain how constraint relations can be properly defined in type index languages. The core of the paper lies in Section 4, where a language $\lambda_{pat}^{\Pi,\Sigma}$ is introduced that extends $\lambda_{pat}$ with both universal and existential dependent types. We also formally prove the subject reduction theorem and the progress theorem for $\lambda_{pat}^{\Pi,\Sigma}$, thus establishing the type soundness of $\lambda_{pat}^{\Pi,\Sigma}$. In Section 5, we introduce an external language $DML_0$ designed for the programmer to construct programs that can be elaborated into $\lambda_{pat}^{\Pi,\Sigma}$. We present a set of elaboration rules and then justify these rules by proving that they preserve the dynamic semantics of programs. In support of the practicality of $\lambda_{pat}^{\Pi,\Sigma}$, we extend $\lambda_{pat}^{\Pi,\Sigma}$ in Section 6 with parametric polymorphism (as is supported in ML), exceptions and references. Also, we present some interesting examples in Section 7 to give the reader a feel as to how dependent types can be used in practice

| | | | |
|---|---|---|---|
| base types | $\delta$ | $::=$ | $\textbf{bool} \mid \textbf{int} \mid \ldots$ |
| types | $\tau$ | $::=$ | $\delta \mid \mathbf{1} \mid \tau_1 * \tau_2 \mid \tau_1 \rightarrow \tau_2$ |
| patterns | $p$ | $::=$ | $x \mid f \mid \langle\rangle \mid \langle p_1, p_2\rangle \mid cc(p)$ |
| matching clause seq. | $ms$ | $::=$ | $(p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n)$ |
| constants | $c$ | $::=$ | $cc \mid cf$ |
| expressions | $e$ | $::=$ | $xf \mid c(e) \mid \langle\rangle \mid \langle e_1, e_2\rangle \mid \textbf{fst}(e) \mid \textbf{snd}(e) \mid \textbf{case}\ e\ \textbf{of}\ ms \mid$ $\textbf{lam}\,x.\,e \mid e_1(e_2) \mid \textbf{fix}\,f.\,e \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2\ \textbf{end}$ |
| values | $v$ | $::=$ | $x \mid cc(v) \mid \langle\rangle \mid \langle v_1, v_2\rangle \mid \textbf{lam}\,x.\,e$ |
| contexts | $\Gamma$ | $::=$ | $\cdot \mid \Gamma, xf : \tau$ |
| substitutions | $\theta$ | $::=$ | $[] \mid \theta[x \mapsto v] \mid \theta[f \mapsto e]$ |

Fig. 3. The syntax for $\lambda_{pat}$.

to capture program invariants. We mention some closely related work in Section 8 and then conclude.

## 2 $\lambda_{pat}$: A starting point

We introduce a simply typed programming language $\lambda_{pat}$, which essentially extends the simply typed $\lambda$-calculus with pattern matching. We emphasize that there are no new contributions in this section. Instead, we primarily use $\lambda_{pat}$ as an example to show how a type system is developed. In particularly, we show how various properties of $\lambda_{pat}$ are chained together in order to establish the type soundness of $\lambda_{pat}$. The subsequent development of the dependent type system in Section 4 and all of its extensions will be done in parallel to the development of $\lambda_{pat}$. Except Lemma 2.14, all the results in this section are well-known and thus their proofs are omitted.

The syntax of $\lambda_{pat}$ is given in Figure 3. We use $\delta$ for base types such as **int** and **bool** and $\tau$ for types. We use $x$ for **lam**-bound variables and $f$ for **fix**-bound variables, and $xf$ for either $x$ or $f$. Given an expression $e$, we write FV($e$) for the set of free variables $xf$ in $e$, which is defined as usual.

A **lam**-bound variable is considered a value but a **fix**-bound variable is not. We use the name *observable value* for a closed value that does not contain a lambda expression $\textbf{lam}\,x.\,e$ as its substructure. We use $c$ for a constant, which is either a constant constructor $cc$ or a constant function $cf$. Each constant $c$ is assigned a constant type (or c-type, for short) of the form $\tau \Rightarrow \delta$. Note that a c-type is not regarded as a (regular) type. For each constant constructor $cc$ assigned the type $\mathbf{1} \Rightarrow \delta$, we may write $cc$ as a shorthand for $cc(\langle\rangle)$, where $\langle\rangle$ stands for the unit of the unit type $\mathbf{1}$. In the following presentation, we assume that the boolean values *true* and *false* are assigned the type $\mathbf{1} \Rightarrow \textbf{bool}$ and every integer $i$ is assigned the type $\mathbf{1} \Rightarrow \textbf{int}$.

Note that we do not treat the tuple constructor $\langle\cdot,\cdot\rangle$ as a special case of constructors. Instead, we introduce tuples into $\lambda_{pat}$ explicitly. The primary reason for this decision is that tuples are to be handled specially in Section 5, where an elaboration procedure is presented for supporting a form of partial type inference in the presence of dependent types.

$$\frac{}{x \downarrow \tau \Rightarrow x : \tau}\ \textbf{(pat-var)}$$

$$\frac{}{\langle\rangle \downarrow \mathbf{1} \Rightarrow \emptyset}\ \textbf{(pat-unit)}$$

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

$$\frac{\vdash cc(\tau) : \delta \quad p \downarrow \tau \Rightarrow \Gamma}{cc(p) \downarrow \delta \Rightarrow \Gamma}\ \textbf{(pat-const)}$$

Fig. 4. The typing rules for patterns in $\lambda_{pat}$.

We use $\theta$ for a substitution, which is a finite mapping that maps **lam**-bound variables $x$ to values and **fix**-bound variables to fixed-point expressions. We use [] for the empty substitution and $\theta[xf \mapsto e]$ for the substitution that extends $\theta$ with a link from $xf$ to $e$, where it is assumed that $xf$ is not in the domain $\mathbf{dom}(\theta)$ of $\theta$. Also, we may write $[xf_1 \mapsto e_1, \ldots, xf_n \mapsto e_n]$ for a substitution that maps $xf_i$ to $e_i$ for $1 \leqslant i \leqslant n$. We omit the further details on substitution, which are completely standard. Given a piece of syntax $\bullet$ (representing expressions, evaluation contexts, etc.), we use $\bullet[\theta]$ for the result of applying $\theta$ to $\bullet$.

We use $\emptyset$ for the empty context and $\Gamma, xf : \tau$ for the context that extends $\Gamma$ with one additional declaration $xf : \tau$, where we assume that $xf$ is not already declared in $\Gamma$. A context $\Gamma = \emptyset, xf_1 : \tau_1, \ldots, xf_n : \tau_n$ may also be treated as a finite mapping that maps $xf_i$ to $\tau_i$ for $1 \leqslant i \leqslant n$, and we use $\mathbf{dom}(\Gamma)$ for the domain of $\Gamma$. Also, we may use $\Gamma, \Gamma'$ for the context $\emptyset, xf_1 : \tau_1, \ldots, xf_n : \tau_n, xf'_1 : \tau'_1, \ldots, xf'_n : \tau'_{n'}$, where $\Gamma = \emptyset, xf_1 : \tau_1, \ldots, xf_n : \tau_n$ and $\Gamma' = \emptyset, xf'_1 : \tau'_1, \ldots, xf'_n : \tau'_{n'}$ and all variables $xf_1, \ldots, xf_n, xf'_1, \ldots, xf'_{n'}$ are distinct.

As a form of syntactic sugar, we may write **let** $\langle x_1, x_2\rangle = e_1$ **in** $e_2$ **end** for the following expression:

$$\textbf{let } x = e_1 \textbf{ in let } x_1 = \textbf{fst}(x) \textbf{ in let } x_2 = \textbf{snd}(x) \textbf{ in } e_2 \textbf{ end end end}$$

where $x$ is assumed to have no free occurrences in $e_1, e_2$.

## *2.1 Static semantics*

We use $p$ for patterns and require that a variable occur at most once in a pattern. Given a pattern $p$ and a type $\tau$, we can derive a judgment of the form $p \downarrow \tau \Rightarrow \Gamma$ with the rules in Figure 4, which reads that checking pattern $p$ against type $\tau$ yields a context $\Gamma$. Note that the rule **(pat-prod)** is unproblematic since $p_1$ and $p_2$ cannot share variables. Also note that we write $\vdash cc(\tau) : \delta$ in the rule **(pat-const)** to indicate that $cc$ is a constant constructor of c-type $\tau \Rightarrow \delta$. As an example, let us assume that **intlist** is a base type, and *nil* and *cons* are constructors of c-types $\mathbf{1} \Rightarrow$ **intlist** and **int** $*$ **intlist** $\Rightarrow$ **intlist**, respectively; then the following judgments are derivable:

$$\begin{array}{rcl} cons(\langle x, xs\rangle) & \downarrow & \textbf{intlist} \Rightarrow x : \textbf{int}, xs : \textbf{intlist} \\ cons(\langle x, nil(\langle\rangle)\rangle) & \downarrow & \textbf{intlist} \Rightarrow x : \textbf{int} \end{array}$$

$$\frac{\Gamma(xf) = \tau}{\Gamma \vdash xf : \tau}\ \textbf{(ty-var)}$$

$$\frac{\vdash c(\tau) : \delta \quad \Gamma \vdash e : \tau}{\Gamma \vdash c(e) : \delta}\ \textbf{(ty-const)}$$

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

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 * \tau_2}\ \textbf{(ty-prod)}$$

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

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

$$\frac{p \downarrow \tau_1 \Rightarrow \Gamma_1 \quad \Gamma, \Gamma_1 \vdash e : \tau_2}{\Gamma \vdash p \Rightarrow e : \tau_1 \rightarrow \tau_2}\ \textbf{(ty-clause)}$$

$$\frac{\Gamma \vdash p_i \Rightarrow e_i : \tau_1 \rightarrow \tau_2 \ \text{ for } i = 1, \ldots, n}{\Gamma \vdash (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) : \tau_1 \rightarrow \tau_2}\ \textbf{(ty-clause-seq)}$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash ms : \tau_1 \rightarrow \tau_2}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ ms : \tau_2}\ \textbf{(ty-case)}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathbf{lam}\, x.e : \tau_1 \rightarrow \tau_2}\ \textbf{(ty-lam)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1(e_2) : \tau_2}\ \textbf{(ty-app)}$$

$$\frac{\Gamma, f : \tau \vdash e : \tau}{\Gamma \vdash \mathbf{fix}\, f.e : \tau}\ \textbf{(ty-fix)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} : \tau_2}\ \textbf{(ty-let)}$$

Fig. 5. The typing rules for expressions in $\lambda_{pat}$.

We present the typing rules for expressions in Figure 5. The rule **(ty-clause)** is for assigning types to clauses. Generally speaking, a clause $p \Rightarrow e$ can be assigned the type $\tau_1 \rightarrow \tau_2$ if $e$ can be assigned the type $\tau_2$ under the assumption that $p$ is given the type $\tau_1$.

In the following presentation, given some form of judgment $J$, we use $\mathscr{D} :: J$ for a derivation of $J$. The structure of a derivation $\mathscr{D}$ is a tree, and we use $height(\mathscr{D})$ for its height, which is defined as usual.

The following standard lemma simply reflects that extra assumptions can be discarded in intuitionistic reasoning. It is needed, for instance, in the proof of Lemma 2.3, the Substitution Lemma for $\lambda_{pat}$.

*Lemma 2.1* (*Thinning*)
Assume $\mathscr{D} :: \Gamma \vdash e : \tau$. Then there is a derivation $\mathscr{D}' :: \Gamma, xf : \tau' \vdash e : \tau$ such that $height(\mathscr{D}) = height(\mathscr{D}')$, where $\tau'$ is any well-formed type.

The following lemma indicates a close relation between the type of a closed value and the form of the value. This lemma is needed to establish Theorem 2.9, the Progress Theorem for $\lambda_{pat}$.

*Lemma 2.2* (*Canonical Forms*)
Assume that $\emptyset \vdash v : \tau$ is derivable.

1. If $\tau = \delta$ for some base type $\delta$, then $v$ is of the form $cc(v_0)$, where $cc$ is a constant constructor assigned a c-type of the form $\tau_0 \Rightarrow \delta$.
2. If $\tau = \mathbf{1}$, then $v$ is $\langle\rangle$.
3. If $\tau = \tau_1 * \tau_2$ for some types $\tau_1$ and $\tau_2$, then $v$ is of the form $\langle v_1, v_2 \rangle$.
4. If $\tau = \tau_1 \to \tau_2$ for some types $\tau_1$ and $\tau_2$, then $v$ is of the form $\mathbf{lam}\, x.\, e$.

Note the need for c-types in the proof of Lemma 2.2 when the last case is handled. If c-types are not introduced, then a (primitive) constant function needs to be assigned a type of the form $\tau_1 \to \tau_2$ for some $\tau_1$ and $\tau_2$. As a consequence, we can no longer claim that a value of the type $\tau_1 \to \tau_2$ for some $\tau_1$ and $\tau_2$ must be of the form $\mathbf{lam}\, x.\, e$ as the value may also be a constant function. So the precise purpose of introducing c-types is to guarantee that only a value of the form $\mathbf{lam}\, x.\, e$ can be assigned a type of the form $\tau_1 \to \tau_2$.

Given $\Gamma, \Gamma_0$ and $\theta$, we write $\Gamma \vdash \theta : \Gamma_0$ to indicate that $\Gamma \vdash \theta(xf) : \Gamma_0(xf)$ is derivable for each $xf$ in $\mathbf{dom}(\theta) = \mathbf{dom}(\Gamma_0)$. The following lemma is often given the name *Substitution Lemma*, which is needed in the proof of Theorem 2.8, the Subject Reduction Theorem for $\lambda_{pat}$.

*Lemma 2.3* (*Substitution*)
Assume that $\Gamma \vdash \theta : \Gamma_0$ holds. If $\Gamma, \Gamma_0 \vdash e : \tau$ is derivable, then $\Gamma \vdash e[\theta] : \tau$ is also derivable.

## 2.2 *Dynamic semantics*

We assign dynamic semantics to expressions in $\lambda_{pat}$ through the use of evaluation contexts defined as follows.

*Definition 2.4* (*Evaluation Contexts*)

$$\begin{array}{lcl} \text{evaluation contexts} \quad E & ::= & [] \mid c(E) \mid \langle E, e \rangle \mid \langle v, E \rangle \mid \mathbf{fst}(E) \mid \mathbf{snd}(E) \mid \\ & & \mathbf{case}\ E\ \mathbf{of}\ ms \mid E(e) \mid v(E) \mid \mathbf{let}\ x = E\ \mathbf{in}\ e\ \mathbf{end} \end{array}$$

We use FV($E$) for the set of free variables $xf$ in $E$. Note that every evaluation context contains exactly one hole [] in it. Given an evaluation context $E$ and an expression $e$, we use $E[e]$ for the expression obtained from replacing the hole [] in $E$ with $e$. As the hole [] in no evaluation context can appear in the scope of a **lam**-binder or a **fix**-binder, there is no issue of capturing free variables in such a replacement.

Given a pattern $p$ and a value $v$, a judgment of the form $\mathbf{match}(v, p) \Rightarrow \theta$, which means that matching a value $v$ against a pattern $p$ yields a substitution for the variables in $p$, can be derived through the application of the rules in Figure 6. Note that the rule **(mat-prod)** is unproblematic because $p_1$ and $p_2$ can share no common variables as $\langle p_1, p_2 \rangle$ is a pattern.

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

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

$$\frac{\mathbf{match}(v_1, p_1) \Rightarrow \theta_1 \quad \mathbf{match}(v_2, p_2) \Rightarrow \theta_2}{\mathbf{match}(\langle v_1, v_2\rangle, \langle p_1, p_2\rangle) \Rightarrow \theta_1 \cup \theta_2}\ \textbf{(mat-prod)}$$

$$\frac{\mathbf{match}(v, p) \Rightarrow \theta}{\mathbf{match}(c(v), c(p)) \Rightarrow \theta}\ \textbf{(mat-const)}$$

Fig. 6. The pattern matching rules for $\lambda_{pat}$.

*Definition 2.5*
We define evaluation redexes (or ev-redex, for short) and their reducts in $\lambda_{pat}$ as follows:

- $\mathbf{fst}(\langle v_1, v_2\rangle)$ is an ev-redex, and its reduct is $v_1$.
- $\mathbf{snd}(\langle v_1, v_2\rangle)$ is an ev-redex, and its reduct is $v_2$.
- $(\mathbf{lam}\, x.\, e)(v)$ is an ev-redex, and its reduct is $e[x \mapsto v]$.
- $\mathbf{fix}\, f.\, e$ is an ev-redex, and its reduct is $e[f \mapsto \mathbf{fix}\, f.\, e]$.
- $\mathbf{let}\ x = v\ \mathbf{in}\ e\ \mathbf{end}$ is an ev-redex, and its reduct is $e[x \mapsto v]$.
- $\mathbf{case}\ v\ \mathbf{of}\ (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n)$ is an ev-redex if $\mathbf{match}(v, p_k) \Rightarrow \theta$ is derivable for some $1 \leqslant k \leqslant n$, and its reduct is $e_k[\theta]$.
- $cf(v)$ is an ev-redex if (1) $v$ is an observable value and (2) $cf(v)$ is defined to be some value $v'$. In this case, the reduct of $cf(v)$ is $v'$. Note that a value is observable if it does not contain any lambda expression $\mathbf{lam}\, x.\, e$ as its substructure.

The one-step evaluation relation $\hookrightarrow_{ev}$ is defined as follows: We write $e_1 \hookrightarrow_{ev} e_2$ if $e_1 = E[e]$ for some evaluation context $E$ and ev-redex $e$, and $e_2 = E[e']$, where $e'$ is a reduct of $e$. We use $\hookrightarrow_{ev}^*$ for the reflexive and transitive closure of $\hookrightarrow_{ev}$ and say that $e_1$ ev-reduces (or evaluates) to $e_2$ if $e_1 \hookrightarrow_{ev}^* e_2$ holds. There is a certain amount of nondeterminism in the evaluation of expressions: $\mathbf{case}\ v\ \mathbf{of}\ ms$ may reduce to $e[\theta]$ for any clause $p \Rightarrow e$ in $ms$ such that $\mathbf{match}(v, p) \Rightarrow \theta$ is derivable. This form of nondeterminism can cause various complications, which we want to avoid in the first place. In this paper, *we require that the patterns $p_1, \ldots, p_n$ in a matching clause sequence $(p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n)$ be disjoint, that is, for $1 \leqslant i \neq j \leqslant n$, there are no values $v$ that can match both $p_i$ and $p_j$.*

In the actual implementation, we do allow overlapping patterns in a matching clause sequence, and we avoid nondeterminism by performing pattern matching in a deterministic sequential manner. We could certainly do the same in the theoretical development, but this may complicate the evaluation of open programs, that is, programs containing free variables. For instance, let $e_1$ and $e_2$ be the following expressions $\mathbf{case}\ cons(x, xs)\ \mathbf{of}\ (nil \Rightarrow true \mid x' \Rightarrow false)$ and $\mathbf{case}\ x\ \mathbf{of}\ (nil \Rightarrow true \mid x' \Rightarrow false)$, respectively. Clearly, we should evaluate $e_1$ to $false$, but we should not evaluate $e_2$ to $false$ as we do not know whether $x$ matches $nil$ or not. This complication is simply avoided when patterns in a matching clause sequence are required to be disjoint.

The meaning of a judgment of the form $p \downarrow \tau \Rightarrow \Gamma$ is captured precisely by following lemma.

*Lemma 2.6*
Assume that the typing judgment $\emptyset \vdash v : \tau$ is derivable. If $p \downarrow \tau \Rightarrow \Gamma$ and $\mathbf{match}(v, p) \Rightarrow \theta$ are derivable, then $\emptyset \vdash \theta : \Gamma$ holds.

*Definition 2.7*
We introduce some forms to classify closed expressions in $\lambda_{pat}$. Given a closed expression $e$ in $\lambda_{pat}$, which may or may not be well-typed,

- $e$ is in V-form if $e$ is a value.
- $e$ is in R-form if $e = E[e_0]$ for some evaluation context $E$ and ev-redex $e_0$. So if $e$ is in R-form, then it can be evaluated further.
- $e$ is in M-form if $e = E[\mathbf{case}\ v\ \mathbf{of}\ ms]$ such that $\mathbf{case}\ v\ \mathbf{of}\ ms$ is not an ev-redex. This is a case where pattern matching fails because none of the involved patterns match $v$.
- $e$ is in U-form if $e = E[cf(v)]$ and $cf(v)$ is undefined. For instance, division by zero is such a case.
- $e$ is in E-form otherwise. We will prove that this is a case that can never occur during the evaluation of a well-typed program.

We introduce three symbols **Error**, **Match** and **Undefined**, and use **EMU** for the set $\{\mathbf{Error}, \mathbf{Match}, \mathbf{Undefined}\}$ and **EMUV** for the union of **EMU** and the set of observable values. We write $e \hookrightarrow_{ev}^{*}$ **Error**, $e \hookrightarrow_{ev}^{*}$ **Match** and $e \hookrightarrow_{ev}^{*}$ **Undefined** if $e \hookrightarrow_{ev}^{*} e'$ for some $e'$ in E-form, M-form and U-form, respectively.

It can be readily checked that the evaluation of a (not necessarily well-typed) program in $\lambda_{pat}$ may either continue forever or reach an expression in V-form, M-form, U-form, or E-form. We will show that an expression in E-form can never be encountered if the evaluation starts with a well-typed program in $\lambda_{pat}$. This is precisely the type soundness of $\lambda_{pat}$.

## 2.3 *Type soundness*

We are now ready to state the subject reduction theorem for $\lambda_{pat}$, which implies that the evaluation of a well-typed expression in $\lambda_{pat}$ does not alter the type of the expression.

For each constant function $cf$ of c-type $\tau \Rightarrow \delta$, if $\emptyset \vdash v : \tau$ is derivable and $c(v)$ is defined to be $v'$, then we require that $\emptyset \vdash v' : \delta$ be also derivable. In other words, we require that each constant function meet its specification, that is, the c-type assigned to it.

*Theorem 2.8* (*Subject Reduction*)
Assume that $\emptyset \vdash e_1 : \tau$ is derivable and $e_1 \hookrightarrow_{ev} e_2$ holds. Then $\emptyset \vdash e_2 : \tau$ is also derivable.

Lemma 2.3 is used in the proof of Theorem 2.8.

*Theorem 2.9* (*Progress*)
Assume that $\emptyset \vdash e_1 : \tau$ is derivable. Then there are only four possibilities:

- $e_1$ is a value, or
- $e_1$ is in M-form, or
- $e_1$ is in U-form, or
- $e_1 \hookrightarrow_{ev} e_2$ holds for some expression $e_2$.

Note that it is implied here that $e_1$ cannot be in E-form.

Lemma 2.2 is needed in the proof of Theorem2.9.

By Theorem 2.8 and Theorem 2.9, we can readily claim that for a well-typed closed expression $e$, either $e$ evaluates to a value, or $e$ evaluates to an expression in M-form, or $e$ evaluates to an expression in U-form, or $e$ evaluates forever. In particular, it is guaranteed that $e \hookrightarrow_{ev}^{*}$ **Error** can never happen for any well-typed expression $e$ in $\lambda_{pat}$.

## 2.4 Operational equivalence

We will present an elaboration procedure in Section 5, which maps a program written in an external language into one in an internal language. We will need to show that the elaboration of a program preserves the operational semantics of the program. For this purpose, we first introduce the notion of general contexts as follows:

$$\begin{array}{l}
\text{general contexts} \quad G \quad ::= \\
\quad [\,] \mid c(G) \mid \langle G, e\rangle \mid \langle e, G\rangle \mid \mathbf{fst}(G) \mid \mathbf{snd}(G) \mid \mathbf{lam}\, x.\, G \mid G(e) \mid e(G) \mid \\
\quad \mathbf{case}\ G\ \mathbf{of}\ (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) \mid \\
\quad \mathbf{case}\ e\ \mathbf{of}\ (p_1 \Rightarrow e_1 \mid \cdots \mid p_{i-1} \Rightarrow e_{i-1} \mid p_i \Rightarrow G \mid p_{i+1} \Rightarrow e_{i+1} \mid \cdots \mid p_n \Rightarrow e_n) \mid \\
\quad \mathbf{fix}\, f.\, G \mid \mathbf{let}\ x = G\ \mathbf{in}\ e\ \mathbf{end} \mid \mathbf{let}\ x = e\ \mathbf{in}\ G\ \mathbf{end}
\end{array}$$

Given a general context $G$ and an expression $e$, $G[e]$ stands for the expression obtained from replacing with $e$ the hole $[\,]$ in $G$. We emphasize that this replacement may capture free variables in $e$. For instance, $G[x] = \mathbf{lam}\, x.\, x$ if $G = \mathbf{lam}\, x.\, [\,]$. The notion of operational equivalence can then be defined as follows.

*Definition 2.10*
Given two expressions $e_1$ and $e_2$ in $\lambda_{pat}$, which may contain free variables, we say that $e_1$ is operationally equivalent to $e_2$ if the following holds.

- Given any context $G$, $G[e_1] \hookrightarrow_{ev}^{*} v^*$ holds if and only if $G[e_2] \hookrightarrow_{ev}^{*} v^*$, where $v^*$ ranges over **EMUV**, that is, the union of **EMU** and the set of observable values.

We write $e_1 \cong e_2$ if $e_1$ is operationally equivalent to $e_2$, which is clearly an equivalence relation.

Unfortunately, this operational equivalence relation is too strong to suit our purpose. The reason can be explained with a simple example. Suppose we have a program $\textbf{lam}\, x : \textbf{int} * \textbf{int}.\, x$ in which the type $\textbf{int} * \textbf{int}$ is provided by the programmer; for some reason (to be made clear later), we may elaborate the program into the following one:

$$e = \textbf{lam}\, x.\, \textbf{let}\ \langle x_1, x_2 \rangle = x\ \textbf{in}\ \langle x_1, x_2 \rangle\ \textbf{end}$$

Note that if we erase the type $\textbf{int} * \textbf{int}$ in the original program, we obtain the expression $\textbf{lam}\, x.\, x$, which is *not* operationally equivalent to $e$; for instance they are distinguished by the simple context $G = [](\langle\rangle)$. To address this rather troublesome issue, we introduce a reflexive and transitive relation $\leqslant_{dyn}$ on expressions in $\lambda_{pat}$.

*Definition 2.11*
Given two expressions $e_1$ and $e_2$ in $\lambda_{pat}$, which may contain free variables, we say that $e_1 \leqslant_{dyn} e_2$ holds if for any context $G$,

- either $G[e_2] \hookrightarrow^*_{ev} \textbf{Error}$ holds, or
- $G[e_1] \hookrightarrow^*_{ev} v^*$ if and only if $G[e_2] \hookrightarrow^*_{ev} v^*$, where $v^*$ ranges over **EMUV**, that is, the union of **EMU** and the set of observable values.

It is straightforward to verify the reflexivity and transitivity of $\leqslant_{dyn}$.

*Corollary 2.12*
Assume that $e_1 \leqslant_{dyn} e_2$ holds. For any context $G$ such that $G[e_2]$ is a closed well-typed expression in $\lambda_{pat}$, $G[e_1]$ evaluates to $v^*$ if and only if $G[e_2]$ evaluates to $v^*$, where $v^*$ ranges over **EMUV**.

*Proof*
This simply follows the definition of $\leqslant_{dyn}$ and Theorem 2.9. □

In other words, $e_1 \leqslant_{dyn} e_2$ implies that $e_1$ and $e_2$ are operationally indistinguishable in a typed setting. We now present an approach to establishing the relation $\leqslant_{dyn}$ in certain special cases.

*Definition 2.13*
We define general redexes (or g-redexes, for short) and their reducts in $\lambda_{pat}$ as follows:

- An ev-redex is a g-redex, and the reduct of the ev-redex is also the reduct of the g-redex.
- $\textbf{let}\ x = e\ \textbf{in}\ E[x]\ \textbf{end}$ is a g-redex if $x$ has no free occurrences in $E$, and its reduct is $E[e]$.
- $\langle \textbf{fst}(v), \textbf{snd}(v) \rangle$ is a g-redex and its reduct is $v$.
- $\textbf{lam}\, x.\, v(x)$ is a g-redex and its reduct is $v$.

We write $e_1 \hookrightarrow_g e_2$ if $e_1 = G[e]$ for some general context $G$ and g-redex $e$, and $e_2 = G[e']$, where $e'$ is a reduct of $e$. We use $\hookrightarrow^*_g$ for the reflexive and transitive

| | | | |
|---|---|---|---|
| index signatures | $\mathscr{S}$ | $::=$ | $\emptyset \mid \mathscr{S}, C : (s_1, \ldots, s_n) \Rightarrow s$ |
| index base sorts | $b$ | $::=$ | $bool \mid \ldots$ |
| index sorts | $s$ | $::=$ | $b \mid s_1 * s_2 \mid s_1 \rightarrow s_2$ |
| index terms | $I$ | $::=$ | $a \mid C(I_1, \ldots, I_n) \mid \langle I_1, I_2 \rangle \mid \pi_1(I) \mid \pi_2(I) \mid$ $\lambda a : s.I \mid I_1(I_2)$ |
| index contexts | $\phi$ | $::=$ | $\emptyset \mid \phi, a : s$ |
| index substitutions | $\Theta$ | $::=$ | $[] \mid \Theta[a \mapsto I]$ |

Fig. 7. The syntax for a generic type index language.

closure of $\hookrightarrow_g$ and say that $e_1$ g-reduces to $e_2$ if $e_1 \hookrightarrow_g^* e_2$ holds. We now mention a lemma as follows:

*Lemma 2.14*
Given two expressions $e$ and $e'$ in $\lambda_{pat}$ that may contain free variables, $e \hookrightarrow_g^* e'$ implies $e' \leqslant_{dyn} e$.

*Proof*
A (lengthy) proof of the lemma is given in Appendix A. □

This lemma is to be of important use in Section 5, where we need to establish that the dynamic semantics of a program cannot be altered by elaboration.

## 3 Type index language

We are to enrich $\lambda_{pat}$ with a restricted form of dependent types. The enrichment is to parameterize over a type index language from which type index terms are drawn. In this section, we show how a generic type index language $\mathscr{L}$ can be formed and then present some concrete examples of type index languages. For generality, we will include both tuples and functions in $\mathscr{L}$. However, we emphasize that a type index language can but does not necessarily have to support tuples or functions.

The generic type index language $\mathscr{L}$ itself is typed. In order to avoid potential confusion, we call the types in $\mathscr{L}$ *type index sorts* (or *sorts*, for short). The syntax of $\mathscr{L}$ is given in Figure 7. We use $b$ for base sorts. In particular, there is a base sort *bool* for boolean values. We use $a$ for index variables and $C$ for constants, which are either constant functions or constant constructors. Each constant is assigned a constant sort (or c-sort, for short) of the form $(s_1, \ldots, s_n) \Rightarrow b$, which means that $C(I_1, \ldots, I_n)$ is an index term of sort $b$ if $I_i$ are of sorts $s_i$ for $i = 1, \ldots, n$. For instance, *true* and *false* are assigned the c-sort $() \Rightarrow bool$. We may write $C$ for $C()$ if $C$ is a constant of c-sort $() \Rightarrow b$ for some base sort $b$. We assume that the c-sorts of constants are declared in some signature $\mathscr{S}$ associated with $\mathscr{L}$, and for each sort $s$, there is a constant function $\doteq_s$ of the c-sort $(s, s) \Rightarrow bool$. We may use $\doteq$ to mean $\doteq_s$ for some sort $s$ if there is no risk of confusion.

We present the sorting rules for type index terms in Figure 8, which are mostly standard. We use $P$ for index propositions, which are index terms that can be assigned the sort *bool* (under some index context $\phi$), and $\vec{P}$ for a sequence of propositions, where the ordering of the terms in this sequence is of no significance.

$$\frac{\phi(a) = s}{\phi \vdash a : s}\ \textbf{(st-var)}$$

$$\frac{\mathscr{S}(C) = (s_1, \ldots, s_n) \Rightarrow s \quad \phi \vdash I_k : s_k \ \text{ for } 1 \leqslant k \leqslant n}{\phi \vdash C(I_1, \ldots, I_n) : s}\ \textbf{(st-const)}$$

$$\frac{\phi \vdash I_1 : s_1 \quad \phi \vdash I_2 : s_2}{\phi \vdash \langle I_1, I_2 \rangle : s_1 * s_2}\ \textbf{(st-prod)}$$

$$\frac{\phi \vdash I : s_1 * s_2}{\phi \vdash \pi_1(I) : s_1}\ \textbf{(st-fst)} \qquad \frac{\phi \vdash I : s_1 * s_2}{\phi \vdash \pi_2(I) : s_2}\ \textbf{(st-snd)}$$

$$\frac{\phi, a : s_1 \vdash I : s_2}{\phi \vdash \lambda a : s_1.I : s_1 \rightarrow s_2}\ \textbf{(st-lam)}$$

$$\frac{\phi \vdash I_1 : s_1 \rightarrow s_2 \quad \phi \vdash I_2 : s_1}{\phi \vdash I_1(I_2) : s_2}\ \textbf{(st-app)}$$

Fig. 8. The sorting rules for type index terms.

$$\frac{}{\phi; \vec{P} \models \mathit{true}}\ \textbf{(reg-true)} \qquad \frac{}{\phi; \vec{P}, \mathit{false} \models P}\ \textbf{(reg-false)}$$

$$\frac{\phi; \vec{P} \models P_0}{\phi, a : s; \vec{P} \models P_0}\ \textbf{(reg-var-thin)} \qquad \frac{\phi \vdash P : \mathit{bool} \quad \phi; \vec{P} \models P_0}{\phi; \vec{P}, P \models P_0}\ \textbf{(reg-prop-thin)}$$

$$\frac{\phi, a : s; \vec{P} \models P \quad \phi \vdash I : s}{\phi; \vec{P}[a \mapsto I] \models P[a \mapsto I]}\ \textbf{(reg-subst)} \qquad \frac{\phi; \vec{P} \models P_0 \quad \phi; \vec{P}, P_0 \models P_1}{\phi; \vec{P} \models P_1}\ \textbf{(reg-cut)}$$

$$\frac{\phi \vdash I : s}{\phi; \vec{P} \models I \doteq_s I}\ \textbf{(reg-eq-refl)} \qquad \frac{\phi; \vec{P} \models I_1 \doteq_s I_2}{\phi; \vec{P} \models I_2 \doteq_s I_1}\ \textbf{(reg-eq-symm)}$$

$$\frac{\phi; \vec{P} \models I_1 \doteq_s I_2 \quad \phi; \vec{P} \models I_2 \doteq_s I_3}{\phi; \vec{P} \models I_1 \doteq_s I_3}\ \textbf{(reg-eq-tran)}$$

Fig. 9. The regularity rules.

We may write $\phi \vdash \vec{P} : \mathit{bool}$ to mean that $\phi \vdash P : \mathit{bool}$ is derivable for every $P$ in $\vec{P}$. In addition, we may use $\phi \vdash \Theta : \phi_0$ to indicate that $\phi \vdash \Theta(a) : \phi_0(a)$ holds for each $a$ in $\mathbf{dom}(\Theta) = \mathbf{dom}(\phi_0)$.

### *3.1 Regular constraint relation*

A constraint relation $\phi; \vec{P} \models P_0$ is defined on triples $\phi, \vec{P}, P_0$ such that both $\phi \vdash \vec{P} : \mathit{bool}$ and $\phi \vdash P_0 : \mathit{bool}$ are derivable. We may also write $\phi; \vec{P} \models \vec{P}_0$ to mean that $\phi; \vec{P} \models P_0$ holds for each $P_0$ in $\vec{P}_0$. We say that a constraint relation $\phi; \vec{P} \models P_0$ is regular if all the regularity rules in Figure 9 are valid, that is, the conclusion of a regularity rule holds whenever all the premises of the regularity rule do. Note that the rules **(reg-eq-refl)**, **(reg-eq-symm)** and **(reg-eq-tran)** indicate that for each sort $s$, $\doteq_s$ needs to be interpreted as an equivalence relation on expressions of the sort $s$.

Essentially, we want to treat a constraint relation as an abstract notion. However, in order to use it, we need to specify certain properties it possesses, and this is precisely the motivation for introducing regularity rules. For instance, we need the regularity rules to prove the following lemma.

*Lemma 3.1* (*Substitution*)

- Assume $\phi, \phi_0; \vec{P} \models P_0$ and $\phi \vdash \Theta : \phi_0$. Then $\phi; \vec{P}[\Theta] \models P_0[\Theta]$ holds.
- Assume $\phi; \vec{P}, \vec{P}_0 \models P_0$ and $\phi; \vec{P} \models \vec{P}_0$. Then $\phi; \vec{P} \models P_0$ holds.

Note that these two properties are just simple iterations of the rules **(reg-subst)** and **(reg-cut)**.

In the rest of this section, we first present a model-theoretic approach to establishing the consistency of a regular constraint relation, and then show some concrete examples of type index languages. At this point, an alternative is for the reader to proceed directly to the next section and then return at a later time.

## *3.2 Models for type index languages*

We now present an approach to constructing regular constraint relations for type index languages. The approach, due to Henkin (Henkin, 1950), is commonly used in the construction of models for simple type theories. The presentation of this approach given below is entirely adopted from Chapter 5 (Andrews, 1986). Also, some details on constructing Henkin models can be found in (Andrews, 1972; Mitchell & Scott, 1989).

We use $\mathbf{D}$ for domains (sets). Given two domains $\mathbf{D}_1$ and $\mathbf{D}_2$, we use $\mathbf{D}_1 \times \mathbf{D}_2$ for the usual product set $\{\langle \mathbf{a}_1, \mathbf{a}_2 \rangle \mid \mathbf{a}_1 \in \mathbf{D}_1 \text{ and } \mathbf{a}_2 \in \mathbf{D}_2\}$, and $\pi_1$ and $\pi_2$ for the standard projection functions from $\mathbf{D}_1 \times \mathbf{D}_2$ to $\mathbf{D}_1$ and $\mathbf{D}_2$, respectively.

Let **sort** be the (possibly infinite) set of all sorts in $\mathscr{L}$. A frame is a collection $\{\mathbf{D}_s\}_{s \in \mathbf{sort}}$ of nonempty domains $\mathbf{D}_s$, one for each sort $s$. We require that $\mathbf{D}_{bool} = \{\mathbf{tt}, \mathbf{ff}\}$, where **tt** and **ff** refer to two distinct elements representing truth and falsehood, respectively, and $\mathbf{D}_{s_1 * s_2} = \mathbf{D}_{s_1} \times \mathbf{D}_{s_2}$ and $\mathbf{D}_{s_1 \to s_2}$ be some collection of functions from $\mathbf{D}_{s_1}$ to $\mathbf{D}_{s_2}$ (but not necessarily all the functions from $\mathbf{D}_{s_1}$ to $\mathbf{D}_{s_2}$). An interpretation $\langle \{\mathbf{D}_s\}_{s \in \mathbf{sort}}, \mathbf{I} \rangle$ of $\mathscr{L}$ consists of a frame $\{\mathbf{D}_s\}_{s \in \mathbf{sort}}$ and a function $\mathbf{I}$ that maps each constant $C$ of c-sort $(s_1, \ldots, s_n) \Rightarrow b$ to a function $\mathbf{I}(C)$ from $\mathbf{D}_{s_1} \times \ldots \times \mathbf{D}_{s_n}$ into $\mathbf{D}_b$ (or to an element in $\mathbf{D}_b$ if $n = 0$), where $b$ stands for a base sort. In particular, we require that

- $\mathbf{I}(\mathit{true}) = \mathbf{tt}$ and $\mathbf{I}(\mathit{false}) = \mathbf{ff}$, and
- $\mathbf{I}(\doteq_s)$ be the equality function of the domain $\mathbf{D}_s$ for each sort $s$.

Assume that the arity of a constructor $C$ is $n$. Then $C(I_1, \ldots, I_n) \doteq C(I'_1, \ldots, I'_n)$ implies that $I_i \doteq I'_i$ for $1 \leqslant i \leqslant n$. Therefore, for each constructor $C$, we require that $\mathbf{I}(C)$ be an injective (a.k.a. 1-1) function.

An assignment $\eta$ is a finite mapping from index variables to $\mathbf{D} = \cup_{s \in \mathbf{sort}} \mathbf{D}_s$, and we use $\mathbf{dom}(\eta)$ for the domain of $\eta$. As usual, we use [] for the empty mapping and $\eta[a \mapsto \mathbf{a}]$ for the mapping that extends $\eta$ with one additional link from $a$ to $\mathbf{a}$, where $a \notin \mathbf{dom}(\eta)$ is assumed. We write $\eta : \phi$ if $\eta(a) \in \mathbf{D}_{\phi(a)}$ holds for each $a \in \mathbf{dom}(\eta) = \mathbf{dom}(\phi)$.

An interpretation $\mathscr{M} = \langle \{\mathbf{D}_s\}_{s \in \mathbf{sort}}, \mathbf{I} \rangle$ of $\mathscr{S}$, which is the signature associated with $\mathscr{L}$, is a model for $\mathscr{L}$ if there exists a (partial) binary function $\mathscr{V}_{\mathscr{M}}$ such that for each assignment $\eta$ satisfying $\eta : \phi$ for some $\phi$ and each index term $I$, $\mathscr{V}_{\mathscr{M}}(\eta, I)$ is properly defined such that $\mathscr{V}_{\mathscr{M}}(\eta, I) \in \mathbf{D}_s$ holds whenever $\phi \vdash I : s$ is derivable for some sort $s$, and the following conditions are also met:

1. $\mathscr{V}_{\mathscr{M}}(\eta, a) = \eta(a)$ for each $a \in \mathbf{dom}(\eta)$, and
2. $\mathscr{V}_{\mathscr{M}}(\eta, C(I_1, \ldots, I_n)) = I(C)(\mathscr{V}_{\mathscr{M}}(\eta, I_1), \ldots, \mathscr{V}_{\mathscr{M}}(\eta, I_n))$, and
3. $\mathscr{V}_{\mathscr{M}}(\eta, \langle I_1, I_2 \rangle) = \langle \mathscr{V}_{\mathscr{M}}(\eta, I_1), \mathscr{V}_{\mathscr{M}}(\eta, I_2) \rangle$, and
4. $\mathscr{V}_{\mathscr{M}}(\eta, \pi_1(I)) = \pi_1(\mathscr{V}_{\mathscr{M}}(\eta, I))$ whenever $\phi \vdash I : s_1 * s_2$ is derivable for some sorts $s_1$ and $s_2$, and
5. $\mathscr{V}_{\mathscr{M}}(\eta, \pi_2(I)) = \pi_2(\mathscr{V}_{\mathscr{M}}(\eta, I))$, whenever $\phi \vdash I : s_1 * s_2$ is derivable for some sorts $s_1$ and $s_2$, and
6. $\mathscr{V}_{\mathscr{M}}(\eta, I_1(I_2)) = \mathscr{V}_{\mathscr{M}}(\eta, I_1)(\mathscr{V}_{\mathscr{M}}(\eta, I_2))$ whenever $\phi \vdash I_1(I_2) : s$ is derivable for some sort $s$, and
7. $\mathscr{V}_{\mathscr{M}}(\eta, \lambda a : s_1.I)$ is the function that maps each element $\mathbf{a}$ in the domain $\mathbf{D}_{s_1}$ to $\mathscr{V}_{\mathscr{M}}(\eta[a \mapsto \mathbf{a}], I)$ whenever $\phi \vdash \lambda a : s_1.I : s_1 \to s_2$ is derivable for some sort $s_2$.

Note that not all interpretations are models (Andrews, 1972). Given a model $\mathscr{M}$ for $\mathscr{L}$, we can define a constraint relation $\models_{\mathscr{M}}$ as follows: $\phi; \vec{P} \models_{\mathscr{M}} P_0$ holds if and only if for each assignment $\eta$ such that $\eta : \phi$ holds, $\mathscr{V}_{\mathscr{M}}(\eta, P_0) = \mathbf{tt}$ or $\mathscr{V}_{\mathscr{M}}(\eta, P) = \mathbf{ff}$ for some $P \in \vec{P}$.

*Proposition 3.2*
The constraint relation $\models_{\mathscr{M}}$ is regular.

*Proof*
It is a simple routine to verify that each of the regularity rules listed in Figure 9 is valid. □

Therefore, we have shown that for any given type index language $\mathscr{L}$, there always exists a regular constraint relation if a model can be constructed for $\mathscr{L}$. Of course, in practice, we need to focus on regular constraint relations that can be decided in an algorithmically effective manner.

## *3.3 Some examples of type index languages*

### *3.3.1 A type index language $\mathscr{L}_{alg}$*

We now describe a type index language $\mathscr{L}_{alg}$ in which only algebraic terms can be formed. Suppose that there are some base sorts in $\mathscr{L}_{alg}$. For each base sort $b$, there exists some constructors of c-sorts $(b_1, \ldots, b_n) \Rightarrow b$ for constructing terms of the base sort $b$, and we say that these constructors are associated with the sort $b$. In general, the terms in $\mathscr{L}_{alg}$ can be formed as follows,

$$\text{index terms} \quad I \quad ::= \quad a \mid C(I_1, \ldots, I_n)$$

where $C$ is a constructor or an equality constant function $\doteq_s$ for some sort $s$. For instance, we may have a sort *Nat* and two constructors $Z$ and $S$ of c-sorts $() \Rightarrow Nat$ and $(Nat) \Rightarrow Nat$, respectively, for constructing terms of sort *Nat*. A constraint in $\mathscr{L}_{alg}$ is of the following form:

$$a_1 : b_1, \ldots, a_n : b_n; I_1 \doteq I'_1, \ldots, I_n \doteq I'_n \models I \doteq I'$$

where each $\doteq$ is $\doteq_s$ for some sort $s$. A simple rule-based algorithm for solving this kind of constraints can be found in (Xi *et al.*, 2003), where algebraic terms are used to represent types.

In practice, we can provide a mechanism for adding into $\mathscr{L}_{alg}$ a new base sort $b$ as well as the constructors associated with $b$. As an example, we may use the following concrete syntax:

```
datasort stp =
  Bool | Integer | Arrow of (stp, stp) | Pair of (stp, stp)
```

to introduce a sort *stp* and then associate with it some constructors of the following c-sorts:

$$\begin{array}{rcl} Bool & : & () \Rightarrow stp \\ Integer & : & () \Rightarrow stp \\ Arrow & : & (stp, stp) \Rightarrow stp \\ Pair & : & (stp, stp) \Rightarrow stp \end{array}$$

We can then use index terms of the sort *stp* to represent the types in a simply typed $\lambda$-calculus where tuples are supported and there are also base types for booleans and integers. In Section 7.3, we will present a concrete programming example involving the type index language $\mathscr{L}_{alg}$.

### 3.3.2 *Another type index language* $\mathscr{L}_{int}$

We now formally describe another type index language $\mathscr{L}_{int}$ in which we can form integer expressions. The syntax for $\mathscr{L}_{int}$ is given as follows:

$$\begin{array}{llcl} \text{index sorts} & s & ::= & bool \mid int \\ \text{index terms} & I & ::= & a \mid C(I_1, \ldots, I_n) \end{array}$$

There are no tuples and functions (formed through $\lambda$-abstraction) in $\mathscr{L}_{int}$, and the constants $C$ in $\mathscr{L}_{int}$ together with their c-sorts are listed in Figure 10. Let $\mathbf{D}_{int}$ be the domain (set) of integers and $\mathscr{M}_{int}$ be $\langle\{\mathbf{D}_{bool}, \mathbf{D}_{int}\}, \mathbf{I}_{int}\rangle$, where $\mathbf{I}_{int}$ maps each constant in $\mathscr{L}_{int}$ to its standard interpretation. For instance, $\mathbf{I}(+)$ and $\mathbf{I}(-)$ are the standard addition and subtraction functions on integers, respectively. It can be readily verified that $\mathscr{M}_{int}$ is a model for $\mathscr{L}_{int}$. Therefore, the constraint relation $\models_{\mathscr{M}_{int}}$ is regular.

Given a constraint $\phi; \vec{P} \models_{\mathscr{M}_{int}} P_0$, where $\phi = a_1 : int, \ldots, a_n : int$, and each $P$ in $\vec{P}$ is a linear inequality on integers, and $P_0$ is also a linear inequality on integers, we can use linear integer programming to solve such a constraint. We will mention later that we can make use of the type index language $\mathscr{L}_{int}$ in the design of a dependently type functional programming language where type equality between

| | | | |
|---|---|---|---|
| *true* | : | $() \to bool$ | |
| *false* | : | $() \to bool$ | |
| $i$ | : | $() \to int$ | for every integer $i$ |
| $\neg$ | : | $(bool) \to bool$ | negation |
| $\wedge$ | : | $(bool, bool) \to bool$ | conjunction |
| $\vee$ | : | $(bool, bool) \to bool$ | disjunction |
| $+$ | : | $(int, int) \to int$ | |
| $-$ | : | $(int, int) \to int$ | |
| $*$ | : | $(int, int) \to int$ | |
| $/$ | : | $(int, int) \to int$ | |
| *max* | : | $(int, int) \to int$ | |
| *min* | : | $(int, int) \to int$ | |
| *mod* | : | $(int, int) \to int$ | modulo operation |
| $\geqslant$ | : | $(int, int) \to bool$ | |
| $>$ | : | $(int, int) \to bool$ | |
| $\leqslant$ | : | $(int, int) \to bool$ | |
| $<$ | : | $(int, int) \to bool$ | |
| $=$ | : | $(int, int) \to bool$ | |
| $\neq$ | : | $(int, int) \to bool$ | |
| $\ldots$ | : | $\ldots$ | |

Fig. 10. The constants and their c-sorts in $\mathscr{L}_{int}$.

two types can be decided through linear integer programming. Though the problem of linear integer programming itself is NP-complete, we have observed that the overwhelming majority of constraints encountered in practice can be solved in a manner that is efficient enough to support realistic programming.

### *3.3.3 Higher-order type index terms*

There are no higher-order type indexes, that is, type index terms of function sorts, in either $\mathscr{L}_{alg}$ or $\mathscr{L}_{int}$. In general, the constraint relation involving higher-order type indexes are often difficult or simply intractable to solve. We now present a type index language $\mathscr{L}_{\lambda}$, which extends $\mathscr{L}_{alg}$ with higher-order type indexes as follows:

$$\text{index terms} \quad I \quad ::= \quad \ldots \mid \lambda a : s.I \mid I_1(I_2)$$

Like in $\mathscr{L}_{alg}$, a constraint in $\mathscr{L}_{\lambda}$ is of the following form:

$$a_1 : b_1, \ldots, a_n : b_n; I_1 \doteq I'_1, \ldots, I_n \doteq I'_n \models I \doteq I'$$

For instance, we may ask whether the following constraint holds:

$$a_1 : b \to b, a_2 : b; a_1(a_1(a_2)) \doteq a_1(a_2) \models a_1(a_2) = a_2$$

If there are two distinct constants $C_1$ and $C_2$ of sort $b$, then the answer is negative since a counterexample can be constructed by letting $a_1$ and $a_2$ be $\lambda a : b.C_1$ and $C_2$, respectively. Clearly, the problem of solving constraints in $\mathscr{L}_{\lambda}$ is undecidable as (a special case of) it can be reduced to the problem of higher-order unification. For instance, $\phi; I_1 \doteq I_2 \models false$ holds if and only if there exists no substitution $\Theta : \phi$ such that $I_1[\Theta]$ and $I_2[\Theta]$ are $\beta\eta$-equivalent.

```
datasort typ = Arrow of (typ, typ) | All of (typ -> typ)

datatype EXP (typ) =
  | {a1:typ, a2:typ} EXPlam (Arrow (a1, a2)) of (EXP (a1) -> EXP (a2))
  | {a1:typ, a2:typ} EXPapp (a2) of (EXP (Arrow (a1, a2)), EXP (a1))
  | {f:typ -> typ} EXPalli (All (f)) of ({a:typ} EXP (f a))
  | {f:typ -> typ,a:typ} EXPalle (f a) of (EXP (All f))
```

Fig. 11. An example involving higher-order type index terms.

In practice, we can decide to only handle constraints of the following simplified form:

$$\phi; a_1 \doteq I_1, \ldots, a_n \doteq I_n \models I \doteq I'$$

where for $1 \leqslant i \leqslant j \leqslant n$, there are no free occurrences of $a_j$ in $I_i$. Solving such a constraint can essentially be reduced to deciding the $\beta\eta$-equality on two simply typed $\lambda$-terms, which is done by comparing whether the two $\lambda$-terms have the same long $\beta\eta$-normal form.

We now present an example that makes use of higher-order type indexes. The constraints on type indexes involved in this example have the above simplified form and thus can be easily solved using $\beta\eta$-normalization. The concrete syntax in Figure 11 declares a sort *typ* and a type constructor **EXP** that takes an index term $I$ of sort *typ* to form a type $\mathbf{EXP}(I)$. The value constructors associated with **EXP** are assigned the following c-types:

$$\begin{array}{rcl}
EXPlam & : & \Pi a_1 : typ. \Pi a_2 : typ. \\
 & & \quad (\mathbf{EXP}(a_1) \to \mathbf{EXP}(a_2)) \Rightarrow \mathbf{EXP}(Arrow(a_1, a_2)) \\
EXPapp & : & \Pi a_1 : typ. \Pi a_2 : typ. \\
 & & \quad (\mathbf{EXP}(Arrow(a_1, a_2)), \mathbf{EXP}(a_1)) \Rightarrow \mathbf{EXP}(a_2) \\
EXPalli & : & \Pi f : typ \to typ. \\
 & & \quad (\Pi a : typ.\ \mathbf{EXP}(f(a))) \Rightarrow \mathbf{EXP}(All(f)) \\
EXPalle & : & \Pi f : typ \to typ. \Pi a : typ. \\
 & & \quad (\mathbf{EXP}(All(f))) \Rightarrow \mathbf{EXP}(f(a))
\end{array}$$

The intent is to use an index term $I$ of sort *typ* to represent a type in the second-order polymorphic $\lambda$-calculus $\lambda_2$ (a.k.a. system F), and a value of type $\mathbf{EXP}(I)$ to represent a $\lambda$-term in $\lambda_2$ that can be assigned the type represented by $I$. For instance, the type $\forall \alpha.\ \alpha \to \alpha$ is represented as $All(\lambda a : typ.\ Arrow(a, a))$, and the following term:

$$EXPalli(\Pi^+(EXPalli(\Pi^+(EXPlam(\mathbf{lam}\, x.\ EXPlam(\mathbf{lam}\, y.\ EXPapp(y, x)))))))$$

which can be given the following type:

$$\mathbf{EXP}(All(\lambda a_1 : typ.\ All(\lambda a_2 : typ.\ Arrow(a_1, Arrow(Arrow(a_1, a_2), a_2)))))$$

represents the $\lambda$-term $\Lambda\alpha_1.\Lambda\alpha_2.\lambda x : \alpha_1.\lambda y : \alpha_1 \to \alpha_2.y(x)$. This is a form of higher-order abstract syntax (h.o.a.s.) representation for $\lambda$-terms (Church, 1940; Pfenning & Elliott, 1988; Pfenning, n.d.). As there is some unfamiliar syntax involved in this example, we suggest that the reader revisit it after studying Section 4.

| | | | |
|---|---|---|---|
| types | $\tau$ | $::=$ | $\ldots \mid \delta(\vec{I}) \mid P \supset \tau \mid P \wedge \tau \mid \Pi a{:}s.\ \tau \mid \Sigma a{:}s.\ \tau$ |
| expressions | $e$ | $::=$ | $\ldots \mid \supset^+(v) \mid \supset^-(e) \mid \Pi^+(v) \mid \Pi^-(e) \mid$ |
| | | | $\wedge(e) \mid$ **let** $\wedge(x) = e_1$ **in** $e_2$ **end** $\mid$ |
| | | | $\Sigma(e) \mid$ **let** $\Sigma(x) = e_1$ **in** $e_2$ **end** |
| values | $v$ | $::=$ | $\ldots \mid \supset^+(v) \mid \Pi^+(v) \mid \wedge(v) \mid \Sigma(v)$ |

Fig. 12. The syntax for $\lambda_{pat}^{\Pi,\Sigma}$.

## 4 $\lambda_{pat}^{\Pi,\Sigma}$: Extending $\lambda_{pat}$ with dependent types

In this section, we introduce both universal and existential dependent types into the type system of $\lambda_{pat}$, leading to the design of a programming language schema $\lambda_{pat}^{\Pi,\Sigma}(\mathscr{L})$ that parameterizes over a given type index language $\mathscr{L}$.

### *4.1 Syntax*

Let us fix a type index language $\mathscr{L}$. We now present $\lambda_{pat}^{\Pi,\Sigma} = \lambda_{pat}^{\Pi,\Sigma}(\mathscr{L})$, which is an extension of $\lambda_{pat}$ with universal and existential dependent types. The syntax of $\lambda_{pat}^{\Pi,\Sigma}$ is given in Figure 12, which extends the syntax in Figure 3. For instance, we use $\ldots$ in the definition of types in $\lambda_{pat}^{\Pi,\Sigma}$ for the following definition of types in $\lambda_{pat}$:

$$\mathbf{1} \mid \tau_1 * \tau_2 \mid \tau_1 \to \tau_2$$

We now use $\delta$ for base type families. We may write $\delta$ for $\delta()$, which is just an unindexed type. We do not specify here as to how new type families can actually be declared. In our implementation, we do provide a means for the programmer to declare type families. For instance, in Section 1, there is such a declaration in the example presented in Figure 1.

We use the names *universal (dependent) types*, *existential (dependent) types*, *guarded types* and *asserting types* for types of the forms $\Pi a : s.\ \tau$, $\Sigma a : s.\ \tau$, $P \supset \tau$ and $P \wedge \tau$, respectively. Note that the type constructor $\wedge$ is asymmetric. In addition, we use the names *universal expressions*, *existential expressions*, *guarded expressions* and *asserting expressions* for expressions of the forms $\Pi^+(v)$, $\Sigma(e)$, $\supset^+(v)$ and $\wedge(e)$, respectively.

In the following presentation, we may write $\vec{I}$ for a (possibly empty) sequence of index terms $I_1, \ldots, I_n$; $\vec{P}$ for a (possibly empty) sequence of index propositions $P_1, \ldots, P_n$; $\Pi\phi$ for a (possibly empty) sequence of quantifiers: $\Pi a_1 : s_1 \ldots \Pi a_n : s_n$, where the index context $\phi$ is $a_1 : s_1, \ldots, a_n : s_n$; $\vec{P} \supset \tau$ for $P_1 \supset (\ldots(P_n \supset \tau)\ldots)$ if $\vec{P} = P_1, \ldots, P_n$.

Notice that a form of value restriction is imposed in $\lambda_{pat}^{\Pi,\Sigma}$: It is required that e be a value in order to form expressions $\Pi^+(e)$ and $\supset^+(e)$. This form of value restriction can in general greatly simplify the treatment of effectful features such as references (Wright, 1995), which are to be added into $\lambda_{pat}^{\Pi,\Sigma}$ in Section 6. We actually need to slightly relax this form of value restriction in Section 6.3 by only requiring that $e$ be a value-equivalent expression (instead of a value) when $\Pi^+(e)$ or $\supset^+(e)$ is formed. Generally speaking, a value-equivalent expression, which is to be formally defined later, refers to an expression that is operationally equivalent to a value.

Intuitively, in order to turn a value of a guarded type $P \supset \tau$ into a value of type $\tau$, we must establish the proposition $P$; if a value of an asserting type $P \wedge \tau$ is generated, then we can assume that the proposition $P$ holds. For instance, the following type can be assigned to the usual division function on integers,

$$\Pi a_1 : int. \Pi a_2 : int.\ (a_2 \neq 0) \supset (\mathbf{int}(a_1) * \mathbf{int}(a_2) \to \mathbf{int}(a_1/a_2))$$

where / stands for the integer division function in some type index language. The following type is a rather interesting one:

$$\Pi a : bool.\ \mathbf{bool}(a) \to (a \doteq true) \wedge \mathbf{1}$$

This type can be assigned to a function that checks at run-time whether a boolean expression holds. In the case where the boolean expression fails to hold, some form of exception is to be raised. Therefore, this function acts as a verifier for run-time assertions made in programs.

In practice, we also have a notion of subset sort. We use $\hat{s}$ to range over subset sorts, which are formally defined as follows:

$$\text{subset sort} \quad \hat{s} \quad ::= \quad s \mid \{a : \hat{s} \mid P\}$$

where the index variable $a$ in $\{a : \hat{s} \mid P\}$ binds the free occurrences of $a$ in $P$. Note that subset sorts, which extend sorts, are just a form of syntactic sugar. Intuitively, the subset sort $\{a : \hat{s} \mid P\}$ is for index terms $I$ of subset sort $\hat{s}$ that satisfy the proposition $P[a \mapsto I]$. For instance, the subset sort *nat* is defined to be $\{a : int \mid a \geqslant 0\}$. In general, we may write $\{a : s \mid P_1, \ldots, P_n\}$ for the subset sort $\hat{s}_n$ defined as follows:

$$\hat{s}_0 = s \qquad\qquad \hat{s}_k = \{a : \hat{s}_{k-1} \mid P_k\}$$

where $k = 1, \ldots, n$.

We use $\phi; \vec{P} \vdash I : \{a : s \mid P_1, \ldots, P_n\}$ to mean that $\phi; \vec{P} \vdash I : s$ is derivable and $\phi; \vec{P} \vdash P_i[a \mapsto I]$ hold for $i = 1, \ldots, n$. Given a subset sort $\hat{s}$, we write $\Pi a : \hat{s}.\ \tau$ for $\Pi a : s.\ \tau$ if $\hat{s}$ is $s$, or for $\Pi a : \hat{s}_1.\ P \supset \tau$ if $\hat{s}$ is $\{a : \hat{s}_1 \mid P\}$. Similarly, we write $\Sigma a : \hat{s}.\ \tau$ for $\Sigma a : s.\ \tau$ if $\hat{s}$ is $s$, or for $\Sigma a : \hat{s}_1.\ P \wedge \tau$ if $\hat{s}$ is $\{a : \hat{s}_1 \mid P\}$. For instance, we write $\Pi a_1 : nat.\ \mathbf{int}(a_1) \to \Sigma a_2 : nat.\ \mathbf{int}(a_2)$ for the following type:

$$\Pi a_1 : int\ .(a_1 \geqslant 0) \supset (\mathbf{int}(a_1) \to \Sigma a_2 : int\ .(a_2 \geqslant 0) \wedge \mathbf{int}(a_2)),$$

which is for functions that map natural numbers to natural numbers.

## *4.2 Static semantics*

We start with the rules for forming types and contexts, which are listed in Figure 13. We use the syntax $\vdash \delta(s_1, \ldots, s_n)$ to indicate that we can construct a type $\delta(I_1, \ldots, I_n)$ when given type index terms $I_1, \ldots, I_n$ of sorts $s_1, \ldots, s_n$, respectively.

A judgment of the form $\phi \vdash \tau$ [**type**] means that $\tau$ is a well-formed type under the index context $\phi$, and a judgment of the form $\phi \vdash \Gamma$ [**ctx**] means that $\Gamma$ is a well-formed (expression) context under $\phi$. The domain $\mathbf{dom}(\Gamma)$ of a context $\Gamma$ is

$$\frac{\vdash \delta(s_1,\ldots,s_n) \quad \phi \vdash I_k : s_k \text{ for } 1 \leqslant k \leqslant n}{\phi \vdash \delta(I_1,\ldots,I_n)\ \textbf{[type]}}\ \textbf{(tp-base)}$$

$$\frac{}{\phi \vdash \mathbf{1}\ \textbf{[type]}}\ \textbf{(tp-unit)}$$

$$\frac{\phi \vdash \tau_1\ \textbf{[type]} \quad \phi \vdash \tau_2\ \textbf{[type]}}{\phi \vdash \tau_1 * \tau_2\ \textbf{[type]}}\ \textbf{(tp-prod)}$$

$$\frac{\phi \vdash \tau_1\ \textbf{[type]} \quad \phi \vdash \tau_2\ \textbf{[type]}}{\phi \vdash \tau_1 \rightarrow \tau_2\ \textbf{[type]}}\ \textbf{(tp-fun)}$$

$$\frac{\phi \vdash P : bool \quad \phi \vdash \tau\ \textbf{[type]}}{\phi \vdash P \supset \tau\ \textbf{[type]}}\ \textbf{(tp-}\supset\textbf{)}$$

$$\frac{\phi, a : s \vdash \tau\ \textbf{[type]}}{\phi \vdash \Pi a{:}s.\ \tau\ \textbf{[type]}}\ \textbf{(tp-}\Pi\textbf{)}$$

$$\frac{\phi \vdash P : bool \quad \phi \vdash \tau\ \textbf{[type]}}{\phi \vdash P \wedge \tau\ \textbf{[type]}}\ \textbf{(tp-}\wedge\textbf{)}$$

$$\frac{\phi, a : s \vdash \tau\ \textbf{[type]}}{\phi \vdash \Sigma a{:}s.\ \tau\ \textbf{[type]}}\ \textbf{(tp-}\Sigma\textbf{)}$$

$$\frac{}{\phi \vdash \emptyset\ \textbf{[ctx]}}\ \textbf{(ctx-emp)}$$

$$\frac{\phi \vdash \Gamma\ \textbf{[ctx]} \quad \phi \vdash \tau\ \textbf{[type]} \quad xf \notin \mathbf{dom}(\Gamma)}{\phi \vdash \Gamma, xf : \tau\ \textbf{[ctx]}}\ \textbf{(ctx-ext)}$$

Fig. 13. The type and context formation rules in $\lambda_{pat}^{\Pi,\Sigma}$.

defined to be the set of variables declared in $\Gamma$. We write $\phi;\vec{P} \models P_0$ for a regular constraint relation in the fixed type index language $\mathscr{L}$.

In $\lambda_{pat}^{\Pi,\Sigma}$, type equality, that is, equality between types, is defined in terms of the static subtype relation $\leqslant_{tp}^{s}$: We say that $\tau$ and $\tau'$ are equal if both $\tau \leqslant_{tp}^{s} \tau'$ and $\tau' \leqslant_{tp}^{s} \tau$ hold. By overloading $\models$, we use $\phi;\vec{P} \models \tau \leqslant_{tp}^{s} \tau'$ for a static subtype judgment and present the rules for deriving such a judgment in Figure 14. Note that all of these rules are syntax-directed.

The static subtype relation $\leqslant_{tp}^{s}$ is often too weak in practice. For instance, we may need to use a function of the type $\tau_1 = \Pi a{:}int.\ \mathbf{int}(a) \rightarrow \mathbf{int}(a)$ as a function of the type $\tau_2 = (\Sigma a{:}int.\ \mathbf{int}(a)) \rightarrow (\Sigma a{:}int.\ \mathbf{int}(a))$, but it is clear that $\tau_1 \leqslant_{tp}^{s} \tau_2$ does not hold (as $\leqslant_{tp}^{s}$ is syntax-directed). We are to introduce in Section 4.6 another subtype relation $\leqslant_{tp}^{d}$, which is much stronger than $\leqslant_{tp}^{s}$ and is given the name *dynamic subtype relation*.

The following lemma, which is parallel to Lemma 3.1, essentially states that the rules in Figure 14 are closed under substitution.

*Lemma 4.1*

1. If $\phi, \phi_0; \vec{P} \models \tau \leqslant_{tp}^{s} \tau'$ is derivable and $\phi \vdash \Theta : \phi_0$ holds, then $\phi; \vec{P}[\Theta] \models \tau[\Theta] \leqslant_{tp}^{s} \tau'[\Theta]$ is also derivable.
2. If $\phi; \vec{P}, \vec{P}_0 \models \tau \leqslant_{tp}^{s} \tau'$ is derivable and $\phi; \vec{P} \models \vec{P}_0$ holds, then $\phi; \vec{P} \models \tau \leqslant_{tp}^{s} \tau'$ is also derivable.

$$\frac{\phi;\vec{P} \models I_1 \doteq I'_1 \quad \cdots \quad \phi;\vec{P} \models I_n \doteq I'_n}{\phi;\vec{P} \models \delta(I_1,\ldots,I_n) \leqslant^s_{tp} \delta(I'_1,\ldots,I'_n)}\ \textbf{(st-sub-base)}$$

$$\frac{}{\phi;\vec{P} \models \mathbf{1} \leqslant^s_{tp} \mathbf{1}}\ \textbf{(st-sub-unit)}$$

$$\frac{\phi;\vec{P} \models \tau_1 \leqslant^s_{tp} \tau'_1 \quad \phi;\vec{P} \models \tau_2 \leqslant^s_{tp} \tau'_2}{\phi;\vec{P} \models \tau_1 * \tau_2 \leqslant^s_{tp} \tau'_1 * \tau'_2}\ \textbf{(st-sub-prod)}$$

$$\frac{\phi;\vec{P} \models \tau'_1 \leqslant^s_{tp} \tau_1 \quad \phi;\vec{P} \models \tau_2 \leqslant^s_{tp} \tau'_2}{\phi;\vec{P} \models \tau_1 \to \tau_2 \leqslant^s_{tp} \tau'_1 \to \tau'_2}\ \textbf{(st-sub-fun)}$$

$$\frac{\phi;\vec{P},P' \models P \quad \phi;\vec{P},P' \models \tau \leqslant^s_{tp} \tau'}{\phi;\vec{P} \models P \supset \tau \leqslant^s_{tp} P' \supset \tau'}\ \textbf{(st-sub-}\supset\textbf{)}$$

$$\frac{\phi, a : s;\vec{P} \models \tau \leqslant^s_{tp} \tau'}{\phi;\vec{P} \models \Pi a{:}s.\ \tau \leqslant^s_{tp} \Pi a{:}s.\ \tau'}\ \textbf{(st-sub-}\Pi\textbf{)}$$

$$\frac{\phi;\vec{P},P \models P' \quad \phi;\vec{P},P \models \tau \leqslant^s_{tp} \tau'}{\phi;\vec{P} \models P \wedge \tau \leqslant^s_{tp} P' \wedge \tau'}\ \textbf{(st-sub-}\wedge\textbf{)}$$

$$\frac{\phi, a : s;\vec{P} \models \tau \leqslant^s_{tp} \tau'}{\phi;\vec{P} \models \Sigma a{:}s.\ \tau \leqslant^s_{tp} \Sigma a{:}s.\ \tau'}\ \textbf{(st-sub-}\Sigma\textbf{)}$$

Fig. 14. The static subtype rules in $\lambda^{\Pi,\Sigma}_{pat}$.

*Proof*
(Sketch) (1) and (2) are proven by structural induction on the derivations of $\phi, \phi_0;\vec{P} \models \tau \leqslant^s_{tp} \tau'$ and $\phi;\vec{P},\vec{P}_0 \models \tau \leqslant^s_{tp} \tau'$, respectively. Lemma 3.1 is needed in the proof. □

As can be expected, the static subtype relation is both reflexive and transitive.

*Proposition 4.2* (*Reflexitivity and Transitivity of* $\leqslant^s_{tp}$)

1. $\phi;\vec{P} \models \tau \leqslant^s_{tp} \tau$ holds for each $\tau$ such that $\phi \vdash \tau$ **[type]** is derivable.
2. $\phi;\vec{P} \models \tau_1 \leqslant^s_{tp} \tau_3$ holds if $\phi;\vec{P} \models \tau_1 \leqslant^s_{tp} \tau_2$ and $\phi;\vec{P} \models \tau_2 \leqslant^s_{tp} \tau_3$ do.

*Proof*
Straightforward. □

We now present the typing rules for patterns in Figure 15 and then the typing rules for expressions in Figure 16 and Figure 17.

The typing judgments for patterns are of the form $p \downarrow \tau \Rightarrow (\phi;\vec{P};\Gamma)$, and the rules for deriving such judgments are given in Figure 15. A judgment of the form $p \downarrow \tau \Rightarrow (\phi;\vec{P};\Gamma)$ means that for any value $v$ of the type $\tau$, if $v$ matches $p$, that is, $\mathbf{match}(v, p) \Rightarrow \theta$ holds for some substitution $\theta$, then there exists an index substitution $\Theta$ such that $\emptyset \vdash \Theta : \phi$, $\emptyset;\emptyset \models \vec{P}[\Theta]$ and $(\emptyset;\emptyset;\emptyset) \vdash \theta : \Gamma[\Theta]$. This is captured precisely by Lemma 4.10. In the rule **(pat-prod)**, it is required that $\phi_1$ and $\phi_2$ share no common index variables in their domains. In the rule **(pat-const)**, we

$$\frac{}{x \downarrow \tau \Rightarrow (\emptyset; \emptyset; x : \tau)}\ \textbf{(pat-var)}$$

$$\frac{}{\langle\rangle \downarrow \mathbf{1} \Rightarrow (\emptyset; \emptyset; \emptyset)}\ \textbf{(pat-unit)}$$

$$\frac{p_1 \downarrow \tau_1 \Rightarrow (\phi_1; \vec{P}_1; \Gamma_1) \quad p_2 \downarrow \tau_2 \Rightarrow (\phi_2; \vec{P}_2; \Gamma_2)}{\langle p_1, p_2\rangle \downarrow \tau_1 * \tau_2 \Rightarrow (\phi_1, \phi_2; \vec{P}_1, \vec{P}_2; \Gamma_1, \Gamma_2)}\ \textbf{(pat-prod)}$$

$$\frac{\phi_0; \vec{P}_0 \vdash cc(\tau) : \delta(I_1, \ldots, I_n) \quad p \downarrow \tau \Rightarrow (\phi; \vec{P}; \Gamma)}{cc(p) \downarrow \delta(I'_1, \ldots, I'_n) \Rightarrow (\phi_0, \phi; \vec{P}_0, \vec{P}, I_1 \doteq I'_1, \ldots, I_n \doteq I'_n; \Gamma)}\ \textbf{(pat-const)}$$

Fig. 15. The typing rules for patterns.

$$\frac{\phi; \vec{P}; \Gamma \vdash e : \tau_1 \quad \phi; \vec{P} \models \tau_1 \leqslant^s_{tp} \tau_2}{\phi; \vec{P}; \Gamma \vdash e : \tau_2}\ \textbf{(ty-sub)}$$

$$\frac{\phi \vdash \Gamma\ [\textbf{ctx}] \quad \Gamma(xf) = \tau}{\phi; \vec{P}; \Gamma \vdash xf : \tau}\ \textbf{(ty-var)}$$

$$\frac{\phi_0; \vec{P}_0 \vdash c(\tau) : \delta(\vec{I}_0) \quad \phi \vdash \Theta : \phi_0 \quad \phi; \vec{P} \models \vec{P}_0[\Theta] \quad \phi; \vec{P}; \Gamma \vdash e : \tau[\Theta]}{\phi; \vec{P}; \Gamma \vdash c(e) : \delta(\vec{I}_0[\Theta])}\ \textbf{(ty-const)}$$

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

$$\frac{\phi; \vec{P}; \Gamma \vdash e : \tau_1 * \tau_2}{\phi; \vec{P}; \Gamma \vdash \mathbf{fst}(e) : \tau_1}\ \textbf{(ty-fst)} \qquad \frac{\phi; \vec{P}; \Gamma \vdash e : \tau_1 * \tau_2}{\phi; \vec{P}; \Gamma \vdash \mathbf{snd}(e) : \tau_2}\ \textbf{(ty-snd)}$$

$$\frac{p \downarrow \tau_1 \Rightarrow (\phi_0; \vec{P}_0, \Gamma_0) \quad \phi, \phi_0; \vec{P}; \vec{P}_0; \Gamma, \Gamma_0 \vdash e : \tau_2}{\phi; \vec{P}; \Gamma \vdash p \Rightarrow e : \tau_1 \to \tau_2}\ \textbf{(ty-clause)}$$

$$\frac{\phi; \vec{P}; \Gamma \vdash p_k \Rightarrow e_k : \tau_1 \to \tau_2 \ \text{ for } k = 1, \ldots, n}{\phi; \vec{P}; \Gamma \vdash (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) : \tau_1 \to \tau_2}\ \textbf{(ty-clause-seq)}$$

$$\frac{\phi; \vec{P}; \Gamma \vdash e : \tau_1 \quad \phi; \vec{P}; \Gamma \vdash ms : \tau_1 \to \tau_2}{\phi; \vec{P}; \Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ ms : \tau_2}\ \textbf{(ty-case)}$$

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

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

$$\frac{\phi; \vec{P}; \Gamma, f : \tau \vdash e : \tau}{\phi; \vec{P}; \Gamma \vdash \mathbf{fix}\, f. e : \tau}\ \textbf{(ty-fix)}$$

$$\frac{\phi; \vec{P}; \Gamma \vdash e_1 : \tau_1 \quad \phi; \vec{P}; \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\phi; \vec{P}; \Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} : \tau_2}\ \textbf{(ty-let)}$$

Fig. 16. The typing rules for $\lambda^{\Pi,\Sigma}_{pat}$ (1).

$$\frac{\phi;\vec{P},P;\Gamma\vdash v:\tau}{\phi;\vec{P};\Gamma\vdash\supset^+(v):P\supset\tau}\ \textbf{(ty-}\supset\textbf{-intro)}$$

$$\frac{\phi;\vec{P};\Gamma\vdash e:P\supset\tau\quad \phi;\vec{P}\models P}{\phi;\vec{P};\Gamma\vdash\supset^-(e):\tau}\ \textbf{(ty-}\supset\textbf{-elim)}$$

$$\frac{\phi,a:s;\vec{P};\Gamma\vdash v:\tau}{\phi;\vec{P};\Gamma\vdash\Pi^+(v):\Pi a{:}s.\ \tau}\ \textbf{(ty-}\Pi\textbf{-intro)}$$

$$\frac{\phi;\vec{P};\Gamma\vdash e:\Pi a{:}s.\ \tau\quad \phi\vdash I:s}{\phi;\vec{P};\Gamma\vdash\Pi^-(e):\tau[a\mapsto I]}\ \textbf{(ty-}\Pi\textbf{-elim)}$$

$$\frac{\phi;\vec{P};\Gamma\vdash e:\tau\quad \phi;\vec{P}\models P}{\phi;\vec{P};\Gamma\vdash\wedge(e):P\wedge\tau}\ \textbf{(ty-}\wedge\textbf{-intro)}$$

$$\frac{\phi;\vec{P};\Gamma\vdash e_1:P\wedge\tau_1\quad \phi;\vec{P},P;\Gamma,x:\tau_1\vdash e_2:\tau_2}{\phi;\vec{P};\Gamma\vdash\textbf{let}\ \wedge(x)=e_1\ \textbf{in}\ e_2\ \textbf{end}:\tau_2}\ \textbf{(ty-}\wedge\textbf{-elim)}$$

$$\frac{\phi;\vec{P};\Gamma\vdash e:\tau[a\mapsto I]\quad \phi\vdash I:s}{\phi;\vec{P};\Gamma\vdash\Sigma(e):\Sigma a{:}s.\ \tau}\ \textbf{(ty-}\Sigma\textbf{-intro)}$$

$$\frac{\phi;\vec{P};\Gamma\vdash e_1:\Sigma a{:}s.\ \tau_1\quad \phi,a:s;\vec{P};\Gamma,x:\tau_1\vdash e_2:\tau_2}{\phi;\vec{P};\Gamma\vdash\textbf{let}\ \Sigma(x)=e_1\ \textbf{in}\ e_2\ \textbf{end}:\tau_2}\ \textbf{(ty-}\Sigma\textbf{-elim)}$$

Fig. 17. The typing rules for $\lambda_{pat}^{\Pi,\Sigma}$ (2).

write $\phi_0;\vec{P}_0\vdash cc(\tau):\delta(I_1,\ldots,I_n)$ to mean that $cc$ is a constant constructor assigned (according to some signature for constants) the following c-type:

$$\Pi\phi_0.\vec{P}_0\supset(\tau\Rightarrow\delta(I_1,\ldots,I_n))$$

In other words, given a constant constructor $cc$, we can form a rule **(pat-const)** for this particular $cc$ based on the c-type assigned to $cc$.

The typing rules given in Figure 16 are mostly expected. The rule **(ty-clause)** requires that $\tau_2$ contain only type index variables declared in $\phi$. For universal dependent types, existential dependent types, guarded types, and assertion types, the typing rules are given in Figure 17. Note that we have omitted certain obvious side conditions that need to be attached to some of these rules. For instance, in the rule **(ty-Π-intro)**, the type index variable $a$ is assumed to have no free occurrences in either $\vec{P}$ or $\Gamma$. Also, in the rule **(ty-Σ-elim)**, the type index variable $a$ is assumed to have no free occurrences in either $\vec{P}$, $\Gamma$ or $\tau_2$. We now briefly go over some of the typing rules in Figure 17.

- If a value $v$ can be assigned a type $\tau$ under an assumption $P$, then the typing rule **(ty-⊃-intro)** assigns $\supset^+(v)$ the guarded type $P\supset\tau$. Notice the presence of value restriction here.

- Given an expression $e$ of type $P \supset \tau$, the typing rule **(ty-$\supset$-elim)** states that the expression $\supset^-(e)$ can be formed if the proposition $P$ holds. Intuitively, a guarded expression is useful only if the guard can be discharged.
- If $e$ can be assigned a type $\tau$ and $P$ holds, then the typing rule **(ty-$\wedge$-intro)** assigns $\wedge(e)$ the asserting type $P \wedge \tau$.
- The elimination rule for the type constructor $\wedge$ is **(ty-$\wedge$-elim)**. Assume that $e_2$ can be assigned a type $\tau_2$ under the assumption that $P$ holds and $x$ is of type $\tau_1$. If $e_1$ is given the asserting type $P \wedge \tau_1$, then the rule **(ty-$\wedge$-elim)** assigns the type $\tau_2$ to the expression **let** $\wedge(x) = e_1$ **in** $e_2$ **end**. Clearly, this rule resembles the treatment of existentially quantified packages (Mitchell & Plotkin, 1988).

The following lemma is parallel to Lemma 2.1. We need to make use of the assumption that the constraint relation involved here is regular when proving the first two statements in this lemma.

*Lemma 4.3* (*Thinning*)
Assume $\mathscr{D} :: \phi; \vec{P}; \Gamma \vdash e : \tau$.

1. For every index variable $a$ that is not declared in $\phi$, we have a derivation $\mathscr{D}' :: \phi, a : s; \vec{P}; \Gamma \vdash e : \tau$ such that $\mathit{height}(\mathscr{D}) = \mathit{height}(\mathscr{D}')$.
2. For every $P$ such that $\phi \vdash P : \mathit{bool}$ is derivable, we have a derivation $\mathscr{D}' :: \phi; \vec{P}, P; \Gamma \vdash e : \tau$ such that $\mathit{height}(\mathscr{D}) = \mathit{height}(\mathscr{D}')$.
3. For every variable $xf$ that is not declared in $\Gamma$ and $\tau'$ such that $\phi \vdash \tau'$ **[type]** is derivable, we have a derivation $\mathscr{D}' :: \phi; \vec{P}; \Gamma, xf : \tau' \vdash e : \tau$ such that $\mathit{height}(\mathscr{D}) = \mathit{height}(\mathscr{D}')$.

*Proof*
Straightforward. □

The following lemma indicates a close relation between the type of a closed value in $\lambda^{\Pi,\Sigma}_{pat}$ and the form of the value, which is needed in the proof of Theorem 4.12, the Progress Theorem for $\lambda^{\Pi,\Sigma}_{pat}$.

*Lemma 4.4* (*Canonical Forms*)
Assume that $\emptyset; \emptyset; \emptyset \vdash v : \tau$ is derivable.

1. If $\tau = \delta(\vec{I})$ for some type family $\delta$, then $v$ is of the form $cc(v_0)$, where $cc$ is a constant constructor assigned a c-type of the form $\Pi\phi.\vec{P} \supset (\tau_0 \Rightarrow \delta(\vec{I}_0))$.
2. If $\tau = \mathbf{1}$, then $v$ is $\langle\rangle$.
3. If $\tau = \tau_1 * \tau_2$, then $v$ is of the form $\langle v_1, v_2 \rangle$.
4. If $\tau = \tau_1 \rightarrow \tau_2$, then $v$ is of the form $\mathbf{lam}\, x.\, e$.
5. If $\tau = P \supset \tau_0$, then $v$ is of the form $\supset^+(v_0)$.
6. If $\tau = \Pi a{:}s.\ \tau_0$, then $v$ is of the form $\Pi^+(v_0)$.
7. If $\tau = P \wedge \tau_0$, then $v$ is of the form $\wedge(v_0)$.
8. If $\tau = \Sigma a{:}s.\ \tau_0$, then $v$ is of the form $\Sigma(v_0)$.

*Proof*
By a thorough inspection of the typing rules in Figure 16 and Figure 17. □

Clearly, the following rule is admissible in $\lambda_{pat}^{\Pi,\Sigma}$ as it is equivalent to the rule **(ty-var)** followed by the rule **(ty-sub)**:

$$\frac{\phi \vdash \Gamma\ [\mathbf{ctx}] \quad \Gamma(xf) = \tau \quad \phi; \vec{P} \models \tau \leqslant_{tp}^{s} \tau'}{\phi; \vec{P}; \Gamma \vdash xf\ : \tau'}\ \textbf{(ty-var')}$$

In the following presentation, we retire the rule **(ty-var)** and simply replace it with the rule **(ty-var')**.

The following technical lemma is needed for establishing Lemma 4.6.

*Lemma 4.5*
Assume $\mathscr{D} :: \phi; \vec{P}; \Gamma, xf\ :\ \tau_1 \vdash e\ :\ \tau_2$. If $\phi; \vec{P} \models \tau_1' \leqslant_{tp}^{s} \tau_1$, then there exists $\mathscr{D}' :: \phi; \vec{P}; \Gamma, xf\ : \tau_1' \vdash e : \tau_2$ such that $height(\mathscr{D}) = height(\mathscr{D}')$.

*Proof*
(Sketch) By structural induction on the derivation $\mathscr{D}$. We need to make use of the fact that the rule **(ty-var)** is replaced with the rule **(ty-var')** in order to show $height(\mathscr{D}) = height(\mathscr{D}')$. □

The following lemma is needed in the proof of Theorem 4.11, the Subject Reduction Theorem for $\lambda_{pat}^{\Pi,\Sigma}$.

*Lemma 4.6*
Assume $\mathscr{D} :: \phi; \vec{P}; \Gamma \vdash v : \tau$. Then there exists a derivation $\mathscr{D}' :: \phi; \vec{P}; \Gamma \vdash v : \tau$ such that $height(\mathscr{D}') \leqslant height(\mathscr{D})$ and the last typing rule applied in $\mathscr{D}'$ is not **(ty-sub)**.

*Proof*
(Sketch) The proof proceeds by structural induction on $\mathscr{D}$. When handling the case where the last applied rule in $\mathscr{D}$ is **(ty-lam)**, we make use of Lemma 4.5 and thus see the need for replacing **(ty-var)** with **(ty-var')**. □

Note that the value $v$ in Lemma 4.6 cannot be replaced with an arbitrary expression. For instance, if we replace $v$ with an expression of the form $\Pi^{-}(e)$, then the lemma cannot be proven.

The following lemma plays a key role in the proof of Theorem 4.11, the Subject Reduction Theorem for $\lambda_{pat}^{\Pi,\Sigma}$.

*Lemma 4.7* (*Substitution*)
1. Assume that $\phi, \phi_0; \vec{P}; \Gamma \vdash e\ :\ \tau$ is derivable. If $\phi \vdash \Theta\ :\ \phi_0$ holds, then $\phi; \vec{P}[\Theta]; \Gamma[\Theta] \vdash e : \tau[\Theta]$ is also derivable.
2. Assume that $\phi; \vec{P}, \vec{P}_0; \Gamma \vdash e : \tau$ is derivable. If $\phi; \vec{P} \models \vec{P}_0$ holds, then $\phi; \vec{P}; \Gamma \vdash e : \tau$ is also derivable.
3. Assume that $\phi; \vec{P}; \Gamma, \Gamma_0 \vdash e\ :\ \tau$ is derivable. If $\phi; \vec{P}; \Gamma \vdash \theta\ :\ \Gamma_0$ holds, then $\phi; \vec{P}; \Gamma \vdash e[\theta]\ : \tau$ is also derivable.

*Proof*
(Sketch) All (1), (2) and (3) are proven straightforwardly by structural induction on the derivations of the typing judgments $\phi, \phi_0; \vec{P}; \Gamma \vdash e\ :\ \tau$, and $\phi; \vec{P}, \vec{P}_0; \Gamma \vdash e\ :\ \tau$, and $\phi; \vec{P}; \Gamma, \Gamma_0 \vdash e\ :\ \tau$, respectively. □

### 4.3 Dynamic semantics

We now need to extend the definition of evaluation contexts (Definition 2.4) as follows.

*Definition 4.8* (*Evaluation Contexts*)

$$\begin{array}{lll} \text{evaluation contexts} & E & ::= & \ldots \mid \supset^{+}(E) \mid \supset^{-}(E) \mid \Pi^{+}(E) \mid \Pi^{-}(E) \mid \\ & & & \wedge(E) \mid \mathbf{let}\ \wedge(x) = E\ \mathbf{in}\ e\ \mathbf{end} \mid \\ & & & \Sigma(E) \mid \mathbf{let}\ \Sigma(x) = E\ \mathbf{in}\ e\ \mathbf{end} \end{array}$$

We are also in need of extending the definition of redexes and their reducts (Definition 2.5).

*Definition 4.9*
In addition to the forms of redexes in Definition 2.5, we have the following new forms of redexes:

- $\supset^{-}(\supset^{+}(v))$ is a redex, and its reduct is $v$.
- $\Pi^{-}(\Pi^{+}(v))$ is a redex, and its reduct is $v$.
- **let** $\wedge(x) = \wedge(v)$ **in** $e$ **end** is a redex, and its reduct is $e[x \mapsto v]$.
- **let** $\Sigma(x) = \Sigma(v)$ **in** $e$ **end** is a redex, and its reduct is $e[x \mapsto v]$.

Note that Definition 2.7, where V-form, R-form, M-form, U-form and E-form are defined, can be readily carried over from $\lambda_{pat}$ into $\lambda_{pat}^{\Pi,\Sigma}$.

The following lemma captures the meaning of the typing judgments for patterns; such judgments can be derived according to the rules in Figure 15.

*Lemma 4.10*
Assume that $\emptyset;\emptyset;\emptyset \vdash v : \tau$ is derivable. If $p \downarrow \tau \Rightarrow (\phi; \vec{P}; \Gamma)$ and $\mathbf{match}(v, p) \Rightarrow \theta$ are also derivable, then there exists $\Theta$ satisfying $\emptyset \vdash \Theta : \phi$ such that both $\emptyset;\emptyset \models \vec{P}[\Theta]$ and $(\emptyset;\emptyset;\emptyset) \vdash \theta : \Gamma[\Theta]$ hold.

*Proof*
(Sketch) By structural induction on the derivation of $p \downarrow \tau \Rightarrow (\phi; \vec{P}; \Gamma)$. □

### 4.4 Type soundness

In order to establish the type soundness for $\lambda_{pat}^{\Pi,\Sigma}$, we make the following assumption: For each constant function $cf$ assigned c-type $\Pi\phi.\vec{P} \supset (\tau \Rightarrow \delta(\vec{I}))$, if $\emptyset;\emptyset \models \vec{P}[\Theta]$ holds for some substitution $\Theta$ satisfying $\emptyset \vdash \Theta : \phi$ and $\emptyset;\emptyset;\emptyset \vdash v : \tau[\Theta]$ is derivable and $cf(v)$ is defined to be $v'$, then $\emptyset;\emptyset;\emptyset \vdash v' : \delta(\vec{I}[\Theta])$ is also derivable. In other words, we assume that each constant function meets its specification. That is, each constant function respects its c-type assignment.

*Theorem 4.11* (*Subject Reduction*)
Assume $\emptyset;\emptyset;\emptyset \vdash e_1 : \tau$ and $e_1 \hookrightarrow_{ev} e_2$. Then $\emptyset;\emptyset;\emptyset \vdash e_2 : \tau$ is also derivable.

*Proof*
A completed proof of this theorem is given in Appendix B. □

```
fun zip (nil, nil) = nil
  | zip (cons (x, xs), cons (y, ys)) = (x, y) :: zip (xs, ys)
```

Fig. 18. An example of exhaustive pattern matching.

*Theorem 4.12* (*Progress*)
Assume that $\emptyset;\emptyset;\emptyset \vdash e_1 : \tau$ is derivable. Then there are only four possibilities:

- $e_1$ is a value, or
- $e_1$ is in M-form, or
- $e_1$ is in U-form, or
- $e_1 \hookrightarrow_{ev} e_2$ holds for some expression $e_2$.

In particular, this implies that $e_1$ cannot be in E-form.

*Proof*
(Sketch) The proof immediately follows from structural induction on the derivation of $\emptyset;\emptyset;\emptyset \vdash e_1 : \tau$. Lemma 4.4 plays a key role in this proof. $\square$

By Theorem 4.11 and Theorem 4.12, we can readily claim that for a well-typed closed expression $e$ in $\lambda_{pat}^{\Pi,\Sigma}$, either $e$ evaluates to a value, or $e$ evaluates to an expression in M-form, or $e$ evaluates to an expression in U-form, or $e$ evaluates forever.

When compared to $\lambda_{pat}$, it is interesting to see what progress we have made in $\lambda_{pat}^{\Pi,\Sigma}$. We may now assign a more accurate type to a constant functions $cf$ to eliminate the occurrences of undefined $cf(v)$ for certain values $v$. For instance, if the division function on integers is assigned the following c-type:

$$\Pi a_1 : int.\Pi a_2 : int.\ (a_2 \neq 0) \supset (\mathbf{int}(a_1) * \mathbf{int}(a_2) \Rightarrow \mathbf{int}(a_1/a_2))$$

then division by zero causes to a type error and thus can never occur at run-time. Similarly, we may now assign a more accurate type to a function to eliminate some occurrences of expressions of the form **case** $v$ **of** $ms$ that are not ev-redexes. For instance, when applied to two lists of unequal length, the function *zip* in Figure 18 evaluates to some expression of the form $E[$**case** $v$ **of** $ms]$ where **case** $v$ **of** $ms$ is not an ev-redex. If we annotate the definition of *zip* with the following type annotation,

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

that is, we assign *zip* the following type (which requires the feature of parametric polymorphism that we are to introduce in Section 6):

$$\forall\alpha_1.\forall\alpha_2.\Pi a : nat.\ (\alpha_1)\mathbf{list}(a) * (\alpha_2)\mathbf{list}(a) \rightarrow (\alpha_1 * \alpha_2)\mathbf{list}(a)$$

then *zip* can no longer be applied to two lists of unequal length. In short, we can now use dependent types to eliminate various (but certainly not all) occurrences of expressions in M-form or U-form, which would not have been possible previously.

Now suppose that we have two lists *xs* and *ys* of unknown length, that is, they are of the type $\Sigma a\,{:}\,nat.\ (\tau)\mathbf{list}(a)$ for some type $\tau$. In order to apply *zip* to *xs* and *ys*, we can insert a run-time check as follows:

```
let
   val m = length (xs) and n = length (ys)
in
   if m = n then zip (xs, ys) else raise UnequalLength
end
```

where the integer equality function = and the list length function *length* are assumed to be of the following types:

$$\begin{array}{rcl} = & : & \Pi a_1\,{:}\,int.\Pi a_2\,{:}\,int.\ \mathbf{int}(a_1) * \mathbf{int}(a_2) \rightarrow \mathbf{bool}(a_1 \doteq a_2) \\ length & : & \forall\alpha.\Pi a\,{:}\,nat.\ (\alpha)\mathbf{list}(a) \rightarrow \mathbf{int}(a) \end{array}$$

Of course, we also have the option to implement another zip function that can directly handle lists of unequal length, but this implementation is less efficient than the one given in Figure 18.

### *4.5 Type index erasure*

In general, there are two directions for extending a type system such as the one in ML: One is to extend it so that more programs can be admitted as type-correct, and the other is to extend it so that programs can be assigned more accurate types. In this paper, we are primarily interested in the latter as is shown below.

We can define a function $|\cdot|$ in Figure 19 that translates types, contexts and expressions in $\lambda_{pat}^{\Pi,\Sigma}$ into types, contexts and expressions in $\lambda_{pat}$, respectively. In particular, for each type family $\delta$ in $\lambda_{pat}^{\Pi,\Sigma}$, we assume that there is a corresponding type $\delta$ in $\lambda_{pat}$, and for each constant $c$ of c-type $\Pi\phi.\vec{P} \supset (\tau \Rightarrow \delta(\vec{I}))$ in $\lambda_{pat}^{\Pi,\Sigma}$, we assume that $c$ is assigned the c-type $|\tau| \Rightarrow \delta$ in $\lambda_{pat}$.

*Theorem 4.13*
Assume that $\phi;\vec{P};\Gamma \vdash e : \tau$ is derivable in $\lambda_{pat}^{\Pi,\Sigma}$. Then $|\Gamma| \vdash |e| : |\tau|$ is derivable in $\lambda_{pat}$.

*Proof*
(Sketch) By structural induction on the derivation of $\phi;\vec{P};\Gamma \vdash e : \tau$. □

Given a closed expression $e_0$ in $\lambda_{pat}$, we say that $e_0$ is typable in $\lambda_{pat}$ if $\emptyset \vdash e_0 : \tau_0$ is derivable for some type $\tau_0$; and we say that $e_0$ is typable in $\lambda_{pat}^{\Pi,\Sigma}$ if there exists an expression $e$ in $\lambda_{pat}^{\Pi,\Sigma}$ such that $|e| = e_0$ and $\emptyset;\emptyset;\emptyset \vdash e : \tau$ is derivable for some type $\tau$. Then by Theorem 4.13, we know that if an expression $e$ in $\lambda_{pat}$ is typable in $\lambda_{pat}^{\Pi,\Sigma}$ then it is already typable in $\lambda_{pat}$. In other words, $\lambda_{pat}^{\Pi,\Sigma}$ does not make more expressions in $\lambda_{pat}$ typable.

*Theorem 4.14*
Assume that $\emptyset;\emptyset;\emptyset \vdash e : \tau$ is derivable.

1. If $e \hookrightarrow_{ev}^{*} v$ in $\lambda_{pat}^{\Pi,\Sigma}$, then $|e| \hookrightarrow_{ev}^{*} |v|$ in $\lambda_{pat}$.

$$
\begin{array}{rcl}
|\delta(\vec{I})| & = & \delta \\
|\mathbf{1}| & = & \mathbf{1} \\
|\tau_1 * \tau_2| & = & |\tau_1| * |\tau_2| \\
|\tau_1 \to \tau_2| & = & |\tau_1| \to |\tau_2| \\
|P \supset \tau| & = & |\tau| \\
|\Pi a{:}s.\ \tau| & = & |\tau| \\
|P \wedge \tau| & = & |\tau| \\
|\Sigma a{:}s.\ \tau| & = & |\tau| \\
|\emptyset| & = & \emptyset \\
|\Gamma, xf : \tau| & = & |\Gamma|, xf : |\tau| \\
|xf| & = & xf \\
|c(e)| & = & c(|e|) \\
|\mathbf{case}\ e\ \mathbf{of}\ (p_1 \Rightarrow e_1 \mid \ldots \mid p_n \Rightarrow e_n)| & = & \mathbf{case}\ |e|\ \mathbf{of}\ (p_1 \Rightarrow |e_1| \mid \ldots \mid p_n \Rightarrow |e_n|) \\
|\langle\rangle| & = & \langle\rangle \\
|\langle e_1, e_2\rangle| & = & \langle |e_1|, |e_2| \rangle \\
|\mathbf{fst}(e)| & = & \mathbf{fst}(|e|) \\
|\mathbf{snd}(e)| & = & \mathbf{snd}(|e|) \\
|\mathbf{lam}\, x.\, e| & = & \mathbf{lam}\, x.\, |e| \\
|e_1(e_2)| & = & |e_1|(|e_2|) \\
|\mathbf{fix}\, f.\, e| & = & \mathbf{fix}\, f.\, |e| \\
|\supset^+(e)| & = & |e| \\
|\supset^-(e)| & = & |e| \\
|\Pi^+(e)| & = & |e| \\
|\Pi^-(e)| & = & |e| \\
|\wedge(e)| & = & |e| \\
|\mathbf{let}\ \wedge(x) = e_1\ \mathbf{in}\ e_2\ \mathbf{end}| & = & \mathbf{let}\ x = |e_1|\ \mathbf{in}\ |e_2|\ \mathbf{end} \\
|\Sigma(e)| & = & |e| \\
|\mathbf{let}\ \Sigma(x) = e_1\ \mathbf{in}\ e_2\ \mathbf{end}| & = & \mathbf{let}\ x = |e_1|\ \mathbf{in}\ |e_2|\ \mathbf{end}
\end{array}
$$

Fig. 19. The erasure function $|\cdot|$ on types, contexts and expressions in $\lambda_{pat}^{\Pi,\Sigma}$.

2. If $|e| \hookrightarrow_{ev}^{*} v_0$ in $\lambda_{pat}$, then there is a value $v$ such that $e \hookrightarrow_{ev}^{*} v$ in $\lambda_{pat}^{\Pi,\Sigma}$ and $|v| = v_0$.

*Proof*

(Sketch) It is straightforward to prove (1). As for (2), it follows from structural induction on the derivation of $\emptyset;\emptyset;\emptyset \vdash e : \tau$. $\square$

Theorem 4.14 indicates that we can evaluate a well-typed program in $\lambda_{pat}^{\Pi,\Sigma}$ by first erasing all the markers $\Pi^+(\cdot)$, $\Pi^-(\cdot)$, $\supset^+(\cdot)$, $\supset^-(\cdot)$, $\Sigma(\cdot)$ and $\wedge(\cdot)$ in the program and then evaluating the erasure in $\lambda_{pat}$. Combining Theorem 4.13 and Theorem 4.14, we say that $\lambda_{pat}^{\Pi,\Sigma}$ is a conservative extension of $\lambda_{pat}$ in terms of both static and dynamic semantics.

### 4.6 Dynamic subtype relation

The dynamic subtype relation defined below is much stronger than the static subtype relation $\leqslant^s_{tp}$ and it plays a key role in Section 5, where an elaboration process is presented to facilitate program construction in $\lambda^{\Pi,\Sigma}_{pat}$.

*Definition 4.15* (*Dynamic Subtype Relation*)
We write $\phi;\vec{P} \models E : \tau \leqslant^d_{tp} \tau'$ to mean that for any expression $e$ and context $\Gamma$, if $\phi;\vec{P};\Gamma \vdash e : \tau$ is derivable then both $\phi;\vec{P};\Gamma \vdash E[e] : \tau'$ is derivable and $|e| \leqslant_{dyn} |E[e]|$ holds. We may write $\phi;\vec{P} \models \tau \leqslant^d_{tp} \tau'$ if, for some $E$, $\phi;\vec{P} \models E : \tau \leqslant^d_{tp} \tau'$ holds, where $E$ can be thought of as a witness to $\tau \leqslant^d_{tp} \tau'$.

As is desired, the dynamic subtype relation $\leqslant^d_{tp}$ is both reflexive and transitive.

*Proposition 4.16* (*Reflexitivity and Transitivity of* $\leqslant^d_{tp}$)

1. $\phi;\vec{P} \models [] : \tau \leqslant^d_{tp} \tau$ holds for each $\tau$ such that $\phi \vdash \tau$ **[type]** is derivable.
2. $\phi;\vec{P} \models E_2[E_1] : \tau_1 \leqslant^d_{tp} \tau_3$ holds if $\phi;\vec{P} \models E_1 : \tau_1 \leqslant^d_{tp} \tau_2$ and $\phi;\vec{P} \models E_2 : \tau_2 \leqslant^d_{tp} \tau_3$ do, where $E_2[E_1]$ is the evaluation context formed by replacing the hole $[]$ in $E_2$ with $E_1$.

*Proof*
(Sketch) The proposition follows from the fact that the relation $\leqslant_{dyn}$ is both reflexive and transitive. $\square$

### 4.7 A restricted form of dependent types

Generally speaking, we use the name *dependent types* to refer to a form of types that correspond to formulas in some first-order many-sorted logic. For instance, the following type in $\lambda^{\Pi,\Sigma}_{pat}$:

$$\Pi a{:}int.\ a \geqslant 0 \supset (\mathbf{int}(a) \to \mathbf{int}(a + a))$$

corresponds to the following first-order formula:

$$\forall a : int.a \geqslant 0 \supset (\mathbf{int}(a) \to \mathbf{int}(a + a))$$

where **int** is interpreted as some predicate on integers, and both $\supset$ and $\to$ stand for the implication connective in logic. However, it is not possible in $\lambda^{\Pi,\Sigma}_{pat}$ to form a dependent type of the form $\Pi a{:}\tau_1.\ \tau_2$, which on the other hand is allowed in a (full) dependent type system such as $\lambda$P (Barendregt, 1992). To see the difficulty in supporting practical programming with such types that may depend on programs, let us recall the following rule that is needed for determining the static subtype relation $\leqslant^s_{tp}$ in $\lambda^{\Pi,\Sigma}_{pat}$:

$$\frac{\phi;\vec{P} \models I \doteq I'}{\phi;\vec{P} \models \delta(I) \leqslant^s_{tp} \delta(I')}$$

If $I$ and $I'$ are programs, then $I \doteq I'$ is an equality on programs. In general, if recursion is allowed in program construction, then it is not just undecidable to determine whether two programs are equal; it is simply intractable. In addition,

$$
\begin{array}{llll}
\text{expressions} & \underline{e} & ::= & x \mid c(\underline{e}) \mid \textbf{case}\ \underline{e}\ \textbf{of}\ (p_1 \Rightarrow \underline{e}_1 \mid \dots p_n \Rightarrow \underline{e}_n) \mid \\
& & & \langle\rangle \mid \langle \underline{e}_1, \underline{e}_2 \rangle \mid \textbf{fst}(\underline{e}) \mid \textbf{snd}(\underline{e}) \mid \\
& & & \textbf{lam}\, x.\, \underline{e} \mid \textbf{lam}\, x : \tau.\, \underline{e} \mid \underline{e}_1(\underline{e}_2) \mid \\
& & & \textbf{fix}\, f : \tau.\, \underline{e} \mid \textbf{let}\ x = \underline{e}_1\ \textbf{in}\ \underline{e}_2\ \textbf{end} \mid \\
& & & \lambda a : \hat{s}.\ \underline{e} \mid \underline{e}[I] \mid (\underline{e} : \tau)
\end{array}
$$

Fig. 20. The syntax for $DML_0$.

such a design means that the type system of a programming language can be rather unstable as adding a new programming feature into the programming language may significantly affect the type system. For instance, if some form of effect (e.g., exceptions, references) is added, then equality on programs can at best become rather intricate to define and is in general impractical to reason about. Currently, there are various studies aiming at addressing these difficulties in order to support full dependent types in practical programming. For instance, a plausible design is to separate pure expressions from potentially effectful ones by employing monads and then require that only pure expressions be used to form types. As for deciding equalities on (pure) expressions, the programmer may be asked to provide proofs of these equalities. Please see (McBride, n.d.; Westbrook *et al.*, 2005) for further details.

We emphasize that the issue of supporting the use of dependent types in practical programming is largely not shared by Martin-Löf's development of constructive type theory (Martin-Löf, 1984; Martin-Löf, 1985), where the principal objective is to give a constructive foundation of mathematics. In such a pure setting, it is perfectly reasonable to define type equality in terms of equality on programs (or more accurately, proofs).

## 5 Elaboration

We have so far presented an *explicitly typed* language $\lambda_{pat}^{\Pi,\Sigma}$. This presentation has a serious drawback from the point of view of a programmer: *One may quickly be overwhelmed with the need for writing types when programming in such a setting*. It then becomes apparent that it is necessary to provide an *external language* $DML_0$ together with a mapping from $DML_0$ to the *internal language* $\lambda_{pat}^{\Pi,\Sigma}$, and we call such a mapping *elaboration*. We may also use the phrase *type-checking* loosely to mean elaboration, sometimes.

We are to introduce a set of rules to perform elaboration. The elaboration process itself is nondeterministic. Nonetheless, we can guarantee based on Theorem 5.3 that if $\underline{e}$ in $DML_0$ can be elaborated into $e$ in $\lambda_{pat}^{\Pi,\Sigma}$, then $\underline{e}$ and $e$ are operationally equivalent. In other words, elaboration cannot alter the dynamic semantics of a program. This is what we call *the soundness of elaboration*, which is considered a major contribution of the paper. We are to perform elaboration with bi-directional strategy that casually resembles the one adopted by Pierce and Turner in their study on local type inference (Pierce & Turner, 1998), where the primary focus is on the interaction between polymorphism and subtyping.

We present the syntax for $DML_0$ in Figure 20, which is rather similar to that of $\lambda_{pat}^{\Pi,\Sigma}$. In general, it should not be difficult to relate the concrete syntax used in our program examples to the formal syntax of $DML_0$. We now briefly explain as to how some concrete syntax can be used to provide type annotations for functions. We essentially support two forms of type annotations for functions, both of which are given below:

```
fun succ1 (x) = x + 1
withtype {a:int | a >= 0} int (a) -> int (a+1)


fun succ2 {a:int | a >= 0} (x: int(a)): int(a+1) = x + 1
```

The first form of annotation allows the programmer to easily read out the type of the annotated function while the second form makes it more convenient to handle a case where the body of a function needs to access some bound type index variables in the type annotation. The concrete syntax for the definition of $succ_1$ translates into the following formal syntax,

$$\mathbf{fix}\, f : \tau.\, \lambda a : \hat{s}.\mathbf{lam}\, x : \mathbf{int}(a).\, (x + 1 : \mathbf{int}(a + 1))$$

where $\hat{s} = \{a : int \mid a \geqslant 0\}$, and so does the concrete syntax for the definition of $succ_2$. As an example, both forms of annotation are involved in the following program, which computes the length of a given list:

```
fun length {n:nat} (xs: 'a list n): int n =
  let // this is a tail-recursive implementation
     fun aux xs j = case xs of
       | nil => j
       | cons (_, xs) => aux xs (j+1)
     withtype {i:nat, j:nat | i+j=n} 'a list i -> int j -> int n
  in
     aux xs 0
  end
```

Note that the type index variable $n$ is used in the type annotation for the inner auxiliary function $aux$.

In the following presentation, we may use $\supset^+_n(\cdot)$ for $\supset^+(\ldots(\supset^+(\cdot))\ldots)$, where there are $n$ occurrences of $\supset^+$, and $\wedge_n(\cdot)$ for $\wedge(\ldots(\wedge(\cdot))\ldots)$, where there are $n$ occurrences of $\wedge$, and **let** $\Sigma(\wedge_0(x)) = e_1$ **in** $e_2$ **end** for **let** $\Sigma(x) = e_1$ **in** $e_2$ **end**, and **let** $\Sigma(\wedge_{n+1}(x)) = e_1$ **in** $e_2$ **end** for the following expression:

$$\mathbf{let}\ \Sigma(\wedge_n(x)) = (\mathbf{let}\ \wedge(x) = e_1\ \mathbf{in}\ x\ \mathbf{end})\ \mathbf{in}\ e_2\ \mathbf{end},$$

where $n$ ranges over natural numbers.

*Proposition 5.1*
We have $|\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end}| \leqslant_{dyn} |\mathbf{let}\ \Sigma(\wedge_n(x)) = e_1\ \mathbf{in}\ e_2\ \mathbf{end}|$.

*Proof*
This immediately follows from Lemma 2.14 and the observation that

$$|\mathbf{let}\ \Sigma(\wedge_n(x)) = e_1\ \mathbf{in}\ e_2\ \mathbf{end}| \hookrightarrow^*_g |\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end}|$$

holds. □

$$\frac{\phi;\vec{P} \models I_1 \doteq I'_1 \quad \cdots \quad \phi;\vec{P} \models I_n \doteq I'_n}{\phi;\vec{P} \vdash [] : \delta(I_1,\ldots,I_n) \leqslant \delta(I'_1,\ldots,I'_n)} \ \textbf{(dy-sub-base)}$$

$$\frac{}{\phi;\vec{P} \vdash [] : \mathbf{1} \leqslant \mathbf{1}} \ \textbf{(dy-sub-unit)}$$

$$\frac{\phi;\vec{P};x_1 : \tau_1, x_2 : \tau_2 \vdash \langle x_1, x_2\rangle \downarrow \tau \Rightarrow e}{\phi;\vec{P} \vdash \textbf{let}\ \langle x_1, x_2\rangle = []\ \textbf{in}\ e\ \textbf{end} : \tau_1 * \tau_2 \leqslant \tau} \ \textbf{(dy-sub-prod)}$$

$$\frac{\phi;\vec{P};x : \tau, x_1 : \tau_1 \vdash x(x_1) \downarrow \tau_2 \Rightarrow e}{\phi;\vec{P} \vdash \textbf{let}\ x = []\ \textbf{in}\ \textbf{lam}\, x_1.\, e\ \textbf{end} : \tau \leqslant \tau_1 \to \tau_2} \ \textbf{(dy-sub-fun)}$$

$$\frac{\hat{s} = \{a : s \mid P_1,\ldots,P_n\} \quad \phi, a : s;\vec{P}, P_1,\ldots,P_n \vdash E : \tau \leqslant \tau'}{\phi;\vec{P} \vdash \Pi^+(\supset_n^+(E)) : \tau \leqslant \Pi a{:}\hat{s}.\ \tau'} \ \textbf{(dy-sub-}\Pi\textbf{-r)}$$

$$\frac{\hat{s} = \{a : s \mid P_1,\ldots,P_n\} \quad \phi, a : s;\vec{P}, P_1,\ldots,P_n \vdash E : \tau \leqslant \tau'}{\phi;\vec{P} \vdash \textbf{let}\ \Sigma(\wedge_n(x)) = []\ \textbf{in}\ E[x]\ \textbf{end} : \Sigma a{:}\hat{s}.\ \tau \leqslant \tau'} \ \textbf{(dy-sub-}\Sigma\textbf{-l)}$$

$$\frac{\hat{s} = \{a : s \mid P_1,\ldots,P_n\} \quad \phi \vdash I : \hat{s} \quad \phi;\vec{P} \vdash E : \tau[a \mapsto I] \leqslant \tau'}{\phi;\vec{P} \vdash E[\supset_n^-(\Pi^-([]))] : \Pi a{:}\hat{s}.\ \tau \leqslant \tau'} \ \textbf{(dy-sub-}\Pi\textbf{-l)}$$

$$\frac{\hat{s} = \{a : s \mid P_1,\ldots,P_n\} \quad \phi \vdash I : \hat{s} \quad \phi;\vec{P} \vdash E : \tau \leqslant \tau'[a \mapsto I]}{\phi;\vec{P} \vdash \Sigma(\wedge_n(E)) : \tau \leqslant \Sigma a{:}\hat{s}.\ \tau'} \ \textbf{(dy-sub-}\Sigma\textbf{-r)}$$

Fig. 21. The dynamic subtype rules in $\lambda_{pat}^{\Pi,\Sigma}$.

## *5.1 The judgments and rules for elaboration*

We introduce a new form of judgment $\phi;\vec{P} \vdash E : \tau_1 \leqslant \tau_2$, which we call *dynamic subtype judgment*. We may write $\phi;\vec{P} \vdash \tau_1 \leqslant \tau_2$ to mean $\phi;\vec{P} \vdash E : \tau_1 \leqslant \tau_2$ for some evaluation context $E$. The rules for deriving such a new form of judgment are given in Figure 21. We are to establish that if $\phi;\vec{P} \vdash E : \tau \leqslant \tau'$ is derivable, then $\phi;\vec{P} \models E : \tau \leqslant_{tp}^{d} \tau'$ holds, that is, for any expression $e$ of type $\tau$, $E[e]$ can be assigned the type $\tau'$ and $|e| \leqslant_{dyn} |E[e]|$ holds.

There is another new form of judgment $\phi;\vec{P};\Gamma \vdash \underline{e} \downarrow \tau \Rightarrow e$ involved in the rule **(dy-sub-prod)** and the rule **(dy-sub-fun)**, and the rules for deriving such a judgment, which we call *analysis elaboration judgment*, are to be presented next.

We actually have two forms of elaboration judgments involved in the process of elaborating expressions from $DML_0$ to $\lambda_{pat}^{\Pi,\Sigma}$.

- A synthesis elaboration judgment is of the form $\phi;\vec{P};\Gamma \vdash \underline{e} \uparrow \tau \Rightarrow e$, which means that given $\phi, \vec{P}, \Gamma$ and $\underline{e}$, we can find a type $\tau$ and an expression $e$ such that $\phi;\vec{P};\Gamma \vdash e : \tau$ is derivable and $|\underline{e}| \leqslant_{dyn} |e|$ holds. Intuitively, $\tau$ can be thought of as being synthesized through an inspection on the structure of $\underline{e}$.
- An analysis elaboration judgment is of the form $\phi;\vec{P};\Gamma \vdash \underline{e} \downarrow \tau \Rightarrow e$, which means that given $\phi;\vec{P};\Gamma, \underline{e}$ and $\tau$, we can find an expression $e$ such that $\phi;\vec{P};\Gamma \vdash e : \tau$ is derivable and $|\underline{e}| \leqslant_{dyn} |e|$ holds.

$$\frac{\hat{s} = \{a : s \mid P_1, \ldots, P_n\} \quad \phi; \vec{P} \vdash I : \hat{s} \quad \phi; \vec{P}; \Gamma \vdash \underline{e} \uparrow \Pi a{:}\hat{s}.\ \tau \Rightarrow e}{\phi; \vec{P}; \Gamma \vdash \underline{e} \uparrow \tau[a \mapsto I] \Rightarrow \supset_n^-(\Pi^-(e))} \quad \textbf{(elab-up-}\Pi\textbf{-elim-1)}$$

$$\frac{\hat{s} = \{a : s \mid P_1, \ldots, P_n\} \quad \phi; \vec{P} \vdash I : \hat{s} \quad \phi; \vec{P}; \Gamma \vdash \underline{e} \uparrow \Pi a{:}\hat{s}.\ \tau \Rightarrow e}{\phi; \vec{P}; \Gamma \vdash \underline{e}[I] \uparrow \tau[a \mapsto I] \Rightarrow \supset_n^-(\Pi^-(e))} \quad \textbf{(elab-up-}\Pi\textbf{-elim-2)}$$

$$\frac{\hat{s} = \{a : s \mid P_1, \ldots, P_n\} \quad \phi, a : s; \vec{P}, P_1, \ldots, P_n; \Gamma \vdash \underline{e} \uparrow \tau \Rightarrow e}{\phi; \vec{P}; \Gamma \vdash \lambda a : \hat{s}.\ \underline{e} \uparrow \Pi a{:}\hat{s}.\ \tau \Rightarrow \Pi^+(\supset_n^+(e))} \quad \textbf{(elab-up-}\Pi\textbf{-intro)}$$

$$\frac{\phi; \vec{P}; \Gamma \vdash \underline{e} \downarrow \tau \Rightarrow e}{\phi; \vec{P}; \Gamma \vdash (\underline{e} : \tau) \uparrow \tau \Rightarrow e} \quad \textbf{(elab-up-anno)}$$

$$\frac{\phi \vdash \Gamma\ [\textbf{ctx}] \quad \Gamma(xf) = \tau}{\phi; \vec{P}; \Gamma \vdash xf \uparrow \tau \Rightarrow xf} \quad \textbf{(elab-up-var)}$$

$$\frac{\phi_0; \vec{P}_0 \vdash c(\tau_0) : \delta(\vec{I}_0) \quad \phi \vdash \Theta : \phi_0 \quad \phi \models \vec{P}_0[\Theta] \quad \phi; \vec{P}; \Gamma \vdash \underline{e} \downarrow \tau_0[\Theta] \Rightarrow e}{\phi; \vec{P}; \Gamma \vdash c(\underline{e}) \uparrow \delta(\vec{I}_0[\Theta]) \Rightarrow c(e)} \quad \textbf{(elab-up-const)}$$

$$\frac{\phi; \vec{P}; \Gamma \vdash \underline{e}_1 \uparrow \tau_1 \Rightarrow e_1 \quad \phi; \vec{P}; \Gamma \vdash \underline{e}_2 \uparrow \tau_2 \Rightarrow e_2}{\phi; \vec{P}; \Gamma \vdash \langle \underline{e}_1, \underline{e}_2 \rangle \uparrow \tau_1 * \tau_2 \Rightarrow \langle e_1, e_2 \rangle} \quad \textbf{(elab-up-prod)}$$

$$\frac{\phi; \vec{P}; \Gamma \vdash \underline{e} \uparrow \tau_1 * \tau_2 \Rightarrow e}{\phi; \vec{P}; \Gamma \vdash \textbf{fst}(\underline{e}) \uparrow \tau_1 \Rightarrow \textbf{fst}(e)} \quad \textbf{(elab-up-fst)}$$

$$\frac{\phi; \vec{P}; \Gamma \vdash \underline{e} \uparrow \tau_1 * \tau_2 \Rightarrow e}{\phi; \vec{P}; \Gamma \vdash \textbf{snd}(\underline{e}) \uparrow \tau_2 \Rightarrow \textbf{snd}(e)} \quad \textbf{(elab-up-snd)}$$

$$\frac{\phi; \vec{P}; \Gamma, x : \tau_1 \vdash \underline{e} \uparrow \tau_2 \Rightarrow \textbf{lam}\, x.\, e}{\phi; \vec{P}; \Gamma \vdash \textbf{lam}\, x : \tau_1.\, \underline{e} \uparrow \tau_1 \to \tau_2 \Rightarrow \textbf{lam}\, x.\, e} \quad \textbf{(elab-up-lam)}$$

$$\frac{\phi; \vec{P}; \Gamma \vdash \underline{e}_1 \uparrow \tau_1 \to \tau_2 \Rightarrow e_1 \quad \phi; \vec{P}; \Gamma \vdash \underline{e}_2 \downarrow \tau_1 \Rightarrow e_2}{\phi; \vec{P}; \Gamma \vdash \underline{e}_1(\underline{e}_2) \uparrow \tau_2 \Rightarrow e_1(e_2)} \quad \textbf{(elab-up-app-1)}$$

$$\frac{\begin{array}{c}\phi; \vec{P}; \Gamma \vdash \underline{e}_1 \uparrow \tau \Rightarrow e_1 \qquad \phi; \vec{P}; \Gamma \vdash \underline{e}_2 \uparrow \tau_1 \Rightarrow e_2 \\ \phi; \vec{P}; x_1 : \tau, x_2 : \tau_1 \vdash x_1(x_2) \uparrow \tau_2 \Rightarrow e\end{array}}{\phi; \vec{P}; \Gamma \vdash \underline{e}_1(\underline{e}_2) \uparrow \tau_2 \Rightarrow \textbf{let}\ x_1 = e_1\ \textbf{in let}\ x_2 = e_2\ \textbf{in}\ e\ \textbf{end end}} \quad \textbf{(elab-up-app-2)}$$

$$\frac{\phi; \vec{P}; \Gamma, f : \tau \vdash e \downarrow \tau \Rightarrow e}{\phi; \vec{P}; \Gamma \vdash \textbf{fix}\, f : \tau.\, e \uparrow \tau \Rightarrow \textbf{fix}\, f.\, e} \quad \textbf{(elab-up-fix)}$$

$$\frac{\phi; \vec{P}; \Gamma \vdash \underline{e}_1 \uparrow \tau_1 \Rightarrow e_1 \quad \phi; \vec{P}; \Gamma, x : \tau_1 \vdash \underline{e}_2 \uparrow \tau_2 \Rightarrow e_2}{\phi; \vec{P}; \Gamma \vdash \textbf{let}\ x = \underline{e}_1\ \textbf{in}\ \underline{e}_2\ \textbf{end} \uparrow \tau_2 \Rightarrow \textbf{let}\ x = e_1\ \textbf{in}\ e_2\ \textbf{end}} \quad \textbf{(elab-up-let)}$$

$$\frac{\phi; \vec{P}; \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash \underline{e}[x \mapsto \langle x_1, x_2 \rangle] \uparrow \tau \Rightarrow e}{\phi; \vec{P}; \Gamma, x : \tau_1 * \tau_2 \vdash \underline{e} \uparrow \tau \Rightarrow \textbf{let}\ \langle x_1, x_2 \rangle = x\ \textbf{in}\ e\ \textbf{end}} \quad \textbf{(elab-up-prod-left)}$$

$$\frac{\hat{s} = \{a : s \mid P_1, \ldots, P_n\} \quad \phi, a : s; \vec{P}, P_1, \ldots, P_n; \Gamma, x : \tau_1 \vdash \underline{e} \uparrow \tau_2 \Rightarrow e}{\phi; \vec{P}; \Gamma, x : \Sigma a{:}\hat{s}.\ \tau_1 \vdash \underline{e} \uparrow \Sigma a{:}\hat{s}.\ \tau_2 \Rightarrow \textbf{let}\ \Sigma(\wedge_n(x)) = x\ \textbf{in}\ \Sigma(\wedge_n(e))\ \textbf{end}} \quad \textbf{(elab-up-}\Sigma\textbf{-left)}$$

Fig. 22. The rules for synthesis elaboration from $DML_0$ to $\lambda_{pat}^{\Pi,\Sigma}$.

We use $|\underline{e}|$ for the erasure of an expression $\underline{e}$ in $DML_0$, which is obtained from erasing in $\underline{e}$ all occurrences of the markers $\Pi^+(\cdot)$, $\Pi^-(\cdot)$, $\supset^+(\cdot)$, $\supset^-(\cdot)$, $\Sigma(\cdot)$ and $\wedge(\cdot)$. The erasure function is formally defined in Figure 24.

The rules for deriving synthesis and analysis elaboration judgments are given in Figure 22 and Figure 23, respectively. Note that there are various occasions where the two forms of elaboration judgments meet. For instance, when using the rule

$$\frac{\hat{s} = \{a : s \mid P_1,\ldots,P_n\} \quad \phi, a : s;\vec{P},P_1,\ldots,P_n;\Gamma \vdash \underline{e} \downarrow \tau \Rightarrow e}{\phi;\vec{P};\Gamma \vdash \underline{e} \downarrow \Pi a{:}\hat{s}.\,\tau \Rightarrow \Pi^+(\supset_n^+(e))}\ \textbf{(elab-dn-}\Pi\textbf{-intro)}$$

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

$$\frac{\phi;\vec{P};\Gamma, x : \tau_1 \vdash \underline{e} \downarrow \tau_2 \Rightarrow \mathbf{lam}\, x.\, e}{\phi;\vec{P};\Gamma \vdash \mathbf{lam}\, x.\, \underline{e} \downarrow \tau_1 \rightarrow \tau_2 \Rightarrow \mathbf{lam}\, x.\, e}\ \textbf{(elab-dn-lam)}$$

$$\frac{p \downarrow \tau_1 \Rightarrow (\phi_0;\vec{P}_0;\Gamma_0) \quad \phi,\phi_0;\vec{P},\vec{P}_0;\Gamma,\Gamma_0 \vdash \underline{e} \downarrow \tau_2 \Rightarrow e}{\phi;\vec{P};\Gamma \vdash (p \Rightarrow \underline{e}) \downarrow (\tau_1 \rightarrow \tau_2) \Rightarrow (p \Rightarrow e)}\ \textbf{(elab-dn-clause)}$$

$$\frac{\begin{array}{c}\phi;\vec{P};\Gamma \vdash (p_i \Rightarrow \underline{e_i}) \downarrow (\tau_1 \rightarrow \tau_2) \Rightarrow (p_i \Rightarrow e_i) \ \text{ for } 1 \leqslant i \leqslant n \\ \underline{ms} = (p_1 \Rightarrow \underline{e_1} \mid \ldots \mid p_n \Rightarrow \underline{e_n}) \qquad ms = (p_1 \Rightarrow e_1 \mid \ldots \mid p_n \Rightarrow e_n)\end{array}}{\phi;\vec{P};\Gamma \vdash \underline{ms} \downarrow \tau_1 \rightarrow \tau_2 \Rightarrow ms}\ \textbf{(elab-dn-clause-seq)}$$

$$\frac{\phi;\vec{P};\Gamma \vdash \underline{e} \uparrow \tau_1 \Rightarrow e \quad \phi;\vec{P};\Gamma \vdash ms \downarrow \tau_1 \rightarrow \tau_2 \Rightarrow \underline{ms}}{\phi;\vec{P};\Gamma \vdash \mathbf{case}\ \underline{e}\ \mathbf{of}\ \underline{ms} \downarrow \tau_2 \Rightarrow \mathbf{case}\ e\ \mathbf{of}\ ms}\ \textbf{(elab-dn-case)}$$

$$\frac{\phi;\vec{P};\Gamma \vdash \underline{e} \uparrow \tau_1 \Rightarrow e \quad \phi;\vec{P} \vdash E : \tau_1 \leqslant \tau_2}{\phi;\vec{P};\Gamma \vdash \underline{e} \downarrow \tau_2 \Rightarrow E[e]}\ \textbf{(elab-dn-up)}$$

$$\frac{\phi;\vec{P};\Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash \underline{e}[x \mapsto \langle x_1,x_2\rangle] \downarrow \tau \Rightarrow e}{\phi;\vec{P};\Gamma, x : \tau_1 * \tau_2 \vdash \underline{e} \downarrow \tau \Rightarrow \mathbf{let}\ \langle x_1,x_2\rangle = x\ \mathbf{in}\ e\ \mathbf{end}}\ \textbf{(elab-dn-prod-left)}$$

$$\frac{\hat{s} = \{a : s \mid P_1,\ldots,P_n\} \quad \phi, a : s;\vec{P},P_1,\ldots,P_n;\Gamma, x : \tau_1 \vdash \underline{e} \downarrow \tau_2 \Rightarrow e}{\phi;\vec{P};\Gamma, x : \Sigma a{:}\hat{s}.\,\tau_1 \vdash \underline{e} \downarrow \tau_2 \Rightarrow \mathbf{let}\ \Sigma(\wedge_n(x)) = x\ \mathbf{in}\ e\ \mathbf{end}}\ \textbf{(elab-dn-}\Sigma\textbf{-left)}$$

Fig. 23. The rules for analysis elaboration from $DML_0$ to $\lambda_{pat}^{\Pi,\Sigma}$.

$$\begin{array}{rcl}
|xf| & = & xf \\
|c(\underline{e})| & = & c(|\underline{e}|) \\
|\mathbf{case}\ e\ \mathbf{of}\ (p_1 \Rightarrow \underline{e_1} \mid \ldots \mid p_n \Rightarrow \underline{e_n})| & = & \mathbf{case}\ |\underline{e}|\ \mathbf{of}\ (p_1 \Rightarrow |\underline{e_1}| \mid \ldots \mid p_n \Rightarrow |\underline{e_n}|) \\
|\langle\rangle| & = & \langle\rangle \\
|\langle \underline{e_1},\underline{e_2}\rangle| & = & \langle |\underline{e_1}|, |\underline{e_2}| \rangle \\
|\mathbf{fst}(\underline{e})| & = & \mathbf{fst}(|\underline{e}|) \\
|\mathbf{snd}(\underline{e})| & = & \mathbf{snd}(|\underline{e}|) \\
|\mathbf{lam}\, x.\, \underline{e}| & = & \mathbf{lam}\, x.\, |\underline{e}| \\
|\mathbf{lam}\, x : \tau.\, \underline{e}| & = & \mathbf{lam}\, x.\, |\underline{e}| \\
|\underline{e_1}(\underline{e_2})| & = & |\underline{e_1}|(|\underline{e_2}|) \\
|\mathbf{fix}\, f : \tau.\, \underline{e}| & = & \mathbf{fix}\, f.\, |\underline{e}| \\
|\mathbf{let}\ x = \underline{e_1}\ \mathbf{in}\ \underline{e_2}\ \mathbf{end}| & = & \mathbf{let}\ x = |\underline{e_1}|\ \mathbf{in}\ |\underline{e_2}|\ \mathbf{end} \\
|(\underline{e} : \tau)| & = & |\underline{e}| \\
|\lambda a : \hat{s}.\ \underline{e}| & = & |\underline{e}| \\
|\underline{e}[I]| & = & |\underline{e}|
\end{array}$$

Fig. 24. The erasure function on expressions in $DML_0$.

**(elab-up-app-1)** to elaborate $e_1(e_2)$, we may first synthesize a type $\tau_1 \rightarrow \tau_2$ for $e_1$ and then check $e_2$ against $\tau_1$.

We next present some explanation on the elaboration rules. First and foremost, we emphasize that many elaboration rules are not syntax-directed. If in a case there are two or more elaboration rules applicable, the actual elaboration procedure should determine (based on some implementation strategies) which elaboration rule is to be chosen. We are currently not positioned to argue which implementation strategies are better than others, though we shall mention some key points about the strategies we have implemented. Given that the elaboration is not a form of pure type inference,[1] it is difficult to even formalize the question as to whether an implementation of the elaboration is complete or not.

### *5.2 Some explanation on synthesis elaboration rules*

The rules for synthesis elaboration judgments are presented in Figure 22. The purpose of the rules **(elab-up-Π-elim-1)** and **(elab-up-Π-elim-2)** is for eliminating Π quantifiers. For instance, let us assume that we are elaborating an expression $\underline{e}_1(\underline{e}_2)$, and a type of the form $\Pi a : \hat{s}.\ \tau$ is already synthesized for $\underline{e}_1$; then we need to apply the rule **(elab-up-Π-elim-1)** so as to eliminate the Π quantifier in $\Pi a : \hat{s}.\ \tau$; we continue to do so until the synthesized type for $\underline{e}_1$ does not begin with a Π quantifier. In some (rare) occasions, the programmer may write $e[I]$ to indicate an explicit elimination of a Π quantifier, and the rule **(elab-up-Π-elim-2)** is designed for this purpose.

The rule **(elab-up-anno)** turns a need for a synthesis elaboration judgment into a need for an analysis elaboration judgment. For instance, we may encounter a situation where we need to synthesize a type for some expression $\mathbf{lam}\, x.\underline{e}$; however, there is no rule for such a synthesis as the involved expression is a **lam**-expression; to address the issue, the programmer may provide a type annotation by writing $(\mathbf{lam}\, x.\underline{e} : \tau)$ instead; synthesizing a type for $(\mathbf{lam}\, x.\underline{e} : \tau)$ is then reduced to analyzing whether $\mathbf{lam}\, x.\underline{e}$ can be assigned the type $\tau$.

The rule **(elab-up-app-1)** is fairly straightforward. When synthesizing a type for $\underline{e}_1(\underline{e}_2)$, we can first synthesize a type for $\underline{e}_1$; if the type is of the form $\tau_1 \rightarrow \tau_2$, we can then analyze whether $\underline{e}_2$ can be assigned the type $\tau_1$; if the analysis succeeds, then we claim that the type $\tau_2$ is synthesized for $\underline{e}_2$.

The rule **(elab-up-app-2)** is rather intricate but of great importance in practice, and we provide some explanation for it. When synthesizing a type for $\underline{e}_1(\underline{e}_2)$, we may first synthesize a type $\tau$ for $\underline{e}_1$ that is not of the form $\tau_1 \rightarrow \tau_2$; for instance, $\tau$ may be a universally quantified type; if this is the case, we can next synthesize a type for $\underline{e}_2$ and then apply the rule **(elab-up-app-2)**. Let us now see a concrete example involving **(elab-up-app-2)**. Suppose that $f$ is given the following type:

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

where $nat = \{a : int \mid a \geqslant 0\}$, and we need to elaborate the expression $f(1)$. By applying the rule **(elab-Π-elim-1)** we can synthesize the type $\mathbf{int}(1) \rightarrow \Sigma a_2 : nat.\ \mathbf{int}(a_2)$ for $f$; then we can analyze that 1 has the type $\mathbf{int}(1)$ and thus synthesize the type $\Sigma a_2 : nat.\ \mathbf{int}(a_2)$ for $f(1)$; note that $f(1)$ elaborates into $\supset^-(\Pi^-(f))(1)$, which can be assigned the type $\Sigma a_2 : nat.\ \mathbf{int}(a_2)$. Now suppose that we need to elaborate the

[1] By pure type inference, we refer to the question that asks whether a given expression in $\lambda_{pat}$ is typable in $\lambda_{pat}^{\Pi,\Sigma}$, that is, whether a given expression in $\lambda_{pat}$ can be the erasure of some typable expression in $\lambda_{pat}^{\Pi,\Sigma}$.

$$
\begin{array}{rcl}
\tau_1 & = & \Pi a_1{:}nat.\ \mathbf{int}(a_1) \to \Sigma a_2{:}nat.\ \mathbf{int}(a_2) \\
\tau_2(a) & = & \mathbf{int}(a) \to \Sigma a_2{:}nat.\ \mathbf{int}(a_2) \\
\tau_3 & = & \Sigma a_2{:}nat.\ \mathbf{int}(a_2) \\
e_1 & = & \supset^-(\Pi^-(f))(1) \\
e_2 & = & \supset^-(\Pi^-(x_1)) \\
e_3 & = & \mathbf{let}\ \Sigma(\wedge(x_2)) = x_2\ \mathbf{in}\ \supset^-(\Pi^-(x_1))(x_2)\ \mathbf{end} \\
e_4 & = & \mathbf{let}\ x_1 = f\ \mathbf{in}\ \mathbf{let}\ x_2 = e_1\ \mathbf{in}\ e_3\ \mathbf{end}\ \mathbf{end}
\end{array}
$$

$$
\frac{}{\mathscr{D}_0 :: \emptyset;\emptyset;\emptyset, f : \tau_1 \vdash f \uparrow \tau_1 \Rightarrow f}\ \textbf{(elab-up-var)}
$$

$$
\frac{\mathscr{D}_0 \qquad \emptyset;\emptyset \vdash 1 : nat}{\mathscr{D}_1 :: \emptyset;\emptyset;\emptyset, f : \tau_1 \vdash f \uparrow \tau_2(1) \Rightarrow \supset^-(\Pi^-(f))}\ \textbf{(elab-up-}\boldsymbol{\Pi}\textbf{-elim-1)}
$$

$$
\frac{}{\mathscr{D}_2 :: \emptyset;\emptyset;\emptyset, f : \tau_1 \vdash 1 \uparrow \mathbf{int}(1) \Rightarrow 1}\ \textbf{(elab-up-const)}
$$

$$
\cfrac{\mathscr{D}_1 \qquad \cfrac{\mathscr{D}_2 \qquad \cfrac{\emptyset;\emptyset \models 1 \doteq 1}{\emptyset;\emptyset \vdash [] : \mathbf{int}(1) \leqslant \mathbf{int}(1)}\ \textbf{(dy-sub-base)}}{\emptyset;\emptyset;\emptyset, f : \tau_1 \vdash 1 \downarrow \mathbf{int}(1) \Rightarrow 1}\ \textbf{(elab-dn-up)}}{\mathscr{D}_3 :: \emptyset;\emptyset;\emptyset, f : \tau_1 \vdash f(1) \uparrow \tau_3 \Rightarrow e_1}\ \textbf{(elab-up-app-1)}
$$

$$
\frac{}{\mathscr{D}_4 :: \emptyset, a_2 : int;\emptyset, a_2 \geqslant 0;\emptyset, x_1 : \tau_1, x_2 : \mathbf{int}(a_2) \vdash x_1 \uparrow \tau_1 \Rightarrow x_1}\ \textbf{(elab-up-var)}
$$

$$
\frac{\mathscr{D}_4 \qquad \emptyset, a_2 : int;\emptyset, a_2 \geqslant 0 \vdash a_2 : nat}{\mathscr{D}_5 :: \emptyset, a_2 : int;\emptyset, a_2 \geqslant 0;\emptyset, x_1 : \tau_1, x_2 : \mathbf{int}(a_2) \vdash x_1 \uparrow \tau_2(a_2) \Rightarrow e_2}\ \textbf{(elab-up-}\boldsymbol{\Pi}\textbf{-elim-1)}
$$

$$
\frac{}{\mathscr{D}_6 :: \emptyset, a_2 : int;\emptyset, a_2 \geqslant 0;\emptyset, x_1 : \tau_1, x_2 : \mathbf{int}(a_2) \vdash x_2 \uparrow \mathbf{int}(a_2) \Rightarrow x_2}\ \textbf{(elab-up-var)}
$$

$$
\cfrac{\cfrac{\mathscr{D}_5 \qquad \cfrac{\mathscr{D}_6 \qquad \cfrac{\emptyset, a_2 : int;\emptyset, a_2 \geqslant 0 \models a_2 \doteq a_2}{\emptyset, a_2 : int;\emptyset, a_2 \geqslant 0 \vdash [] : \mathbf{int}(a_2) \leqslant \mathbf{int}(a_2)}\ \textbf{(dy-sub-base)}}{\emptyset, a_2 : int;\emptyset, a_2 \geqslant 0;\emptyset, x_1 : \tau_1, x_2 : \mathbf{int}(a_2) \vdash x_2 \downarrow \mathbf{int}(a_2) \Rightarrow x_2}\ \textbf{(elab-dn-up)}}{\emptyset, a_2 : int;\emptyset, a_2 \geqslant 0;\emptyset, x_1 : \tau_1, x_2 : \mathbf{int}(a_2) \vdash x_1(x_2) \uparrow \tau_3 \Rightarrow e_2(x_2)}\ \textbf{(elab-up-app-1)}}{\mathscr{D}_7 :: \emptyset;\emptyset;\emptyset, x_1 : \tau_1, x_2 : \tau_3 \vdash x_1(x_2) \uparrow \Sigma a{:}nat.\ \tau_3 \Rightarrow e_3}\ \textbf{(elab-up-}\boldsymbol{\Sigma}\textbf{-left)}
$$

$$
\frac{\mathscr{D}_0 \qquad \mathscr{D}_3 \qquad \mathscr{D}_7}{\mathscr{D}_8 :: \emptyset;\emptyset;\emptyset, x_1 : \tau_1, x_2 : \tau_3 \vdash f(f(1)) \uparrow \Sigma a{:}nat.\ \tau_3 \Rightarrow e_4}\ \textbf{(elab-up-app-2)}
$$

Fig. 25. An example of elaboration.

expression $f(f(1))$. If we simply synthesize a type of the form $\mathbf{int}(I) \to \Sigma a_2{:}nat.\ \mathbf{int}(a_2)$ for the first occurrence of $f$ in $f(f(1))$, then the elaboration for $f(f(1))$ cannot succeed as it is impossible to elaborate $f(1)$ into an expression in $\lambda_{pat}^{\Pi,\Sigma}$ of the type $\mathbf{int}(I)$ for any type index $I$. With the rule **(elab-up-app-2)**, we are actually able to elaborate $f(f(1))$ into the following expression $e$ in $\lambda_{pat}^{\Pi,\Sigma}$:

$$\mathbf{let}\ x_1 = f\ \mathbf{in}\ \mathbf{let}\ x_2 = \supset^-(\Pi^-(f))(1)\ \mathbf{in}\ e'\ \mathbf{end}\ \mathbf{end}$$

where $e' = \mathbf{let}\ \Sigma(\wedge(x_2)) = x_2\ \mathbf{in}\ \supset^-(\Pi^-(x_1))(x_2)\ \mathbf{end}$. Please find that the entire elaboration is formally carried out in Figure 25. Clearly, the erasure of $e$ is operationally equivalent to $f(f(1))$.

The rules **(elab-up-prod-left)** and **(elab-up-Σ-left)** are for simplifying the types assigned to variables in a dynamic context. In practice, we apply these rules during elaboration whenever possible.

## *5.3 Some explanation on analysis elaboration rules*

The rules for analysis elaboration judgments are presented in Figure 23. For instance, if $\underline{e} = \langle \underline{e}_1, \underline{e}_2 \rangle$ and $\tau = \langle \tau_1, \tau_2 \rangle$, then the rule **(elab-dn-prod)** reduces the question whether $\underline{e}$ can be assigned the type $\tau$ to the questions whether $\underline{e}_i$ can be assigned the types $\tau_i$ for $i = 1, 2$. Most of analysis elaboration rules are straightforward. In the rule **(elab-dn-up)**, we see that the three forms of judgments (dynamic subtype judgment, synthesis elaboration judgment and analysis elaboration judgment) meet. This rule simply means that when analyzing whether an expression $\underline{e}$ can be given a type $\tau_2$, we may first synthesize a type $\tau_1$ for $\underline{e}$ and then show that $\tau_1$ is a dynamic subtype of $\tau_2$ (by showing that $E : \tau_1 \leqslant \tau_2$ is derivable for some evaluation context $E$). In practice, we apply the rule **(elab-dn-up)** only if all other analysis elaboration rules are inapplicable.

We now show some actual use of analysis elaboration rules by presenting in Figure 26 a derivation of the following judgment for some $E$:

$$\emptyset; \emptyset \vdash E : \mathbf{Nat} * \mathbf{Nat} \leqslant \Sigma a_1 : nat. \Sigma a_2 : nat.\ \mathbf{int}(a_1) * \mathbf{int}(a_2)$$

where $\mathbf{Nat} = \Sigma a : nat.\ \mathbf{int}(a)$. In this derivation, we assume the existence of a derivation $\mathscr{D}_0 :: \phi_2; \vec{P}_2 \vdash E_0 : \tau_0 \leqslant \tau_2$ for the following evaluation context $E_0$:

$$\Sigma(\wedge(\Sigma(\wedge(\mathbf{let}\ \langle x_1, x_2 \rangle = []\ \mathbf{in}\ \langle x_1, x_2 \rangle\ \mathbf{end}))))$$

which can be readily constructed.

As another example, the interested reader can readily derive the following judgment for some $E$:

$$\emptyset; \emptyset \vdash E : \Pi a : int.\ \mathbf{int}(a) \to \mathbf{int}(a) \leqslant \mathbf{Int} \to \mathbf{Int}$$

where $\mathbf{Int} = \Sigma a : int.\ \mathbf{int}(a)$. Therefore, we can always use a function of the type $\Pi a : int.\ \mathbf{int}(a) \to \mathbf{int}(a)$ as a function of the type $\mathbf{Int} \to \mathbf{Int}$.

## *5.4 The soundness of elaboration*

We now prove the soundness of elaboration, that is, elaboration cannot alter the dynamic semantics of a program. To make the statement more precise, we define in Figure 24 an erasure function $|\cdot|$ from $DML_0$ to $\lambda_{pat}$. The following lemma is the key to establishing the soundness of elaboration.

*Lemma 5.2*
Given $\phi, \vec{P}, \Gamma, \underline{e}, \tau, \tau'$ and $e$, we have the following:

1. If $\phi; \vec{P} \vdash E : \tau \leqslant \tau'$ is derivable, then $\phi; \vec{P} \models E : \tau \leqslant^d_{tp} \tau'$ holds.
2. If $\phi; \vec{P}; \Gamma \vdash \underline{e} \uparrow \tau \Rightarrow e$ is derivable, then $\phi; \vec{P}; \Gamma \vdash e : \tau$ is derivable in $\lambda^{\Pi,\Sigma}_{pat}$, and $|e| \hookrightarrow^*_g |\underline{e}|$ holds.
3. If $\phi; \vec{P}; \Gamma \vdash \underline{e} \downarrow \tau \Rightarrow e$ is derivable, then $\phi; \vec{P}; \Gamma \vdash e : \tau$ is derivable in $\lambda^{\Pi,\Sigma}_{pat}$, and $|e| \hookrightarrow^*_g |\underline{e}|$ holds.

$$\begin{array}{lcl}
\tau_0 & = & \mathbf{int}(a_1) * \mathbf{int}(a_2) \\
\tau_1 & = & \Sigma a_2 {:} nat.\ \mathbf{int}(a_1) * \mathbf{int}(a_2) \\
\tau_2 & = & \Sigma a_1 {:} nat.\Sigma a_2 {:} nat.\ \mathbf{int}(a_1) * \mathbf{int}(a_2) \\
\phi_1 & = & \emptyset, a_1 : int \\
\phi_2 & = & \emptyset, a_1 : int, a_2 : int \\
\vec{P}_1 & = & \emptyset, a_1 \geqslant 0 \\
\vec{P}_2 & = & \emptyset, a_1 \geqslant 0, a_2 \geqslant 0 \\
E_0 & = & \Sigma(\wedge(\Sigma(\wedge(\mathbf{let}\ \langle x_1, x_2 \rangle = []\ \mathbf{in}\ \langle x_1, x_2 \rangle\ \mathbf{end})))) \\
e_1 & = & E_0[\langle x_1, x_2 \rangle] \\
e_2 & = & \mathbf{let}\ \Sigma(\wedge(x_2)) = x_2\ \mathbf{in}\ e_1\ \mathbf{end} \\
e_3 & = & \mathbf{let}\ \Sigma(\wedge(x_1)) = x_1\ \mathbf{in}\ e_2\ \mathbf{end} \\
E & = & \mathbf{let}\ \langle x_1, x_2 \rangle = []\ \mathbf{in}\ e_3\ \mathbf{end}
\end{array}$$

$$\frac{}{\mathscr{D}_1 :: \phi_2; \vec{P}_2; x_1 : \mathbf{int}(a_1), x_2 : \mathbf{int}(a_2) \vdash x_1 \uparrow \mathbf{int}(a_1) \Rightarrow x_1}\ \textbf{(elab-var-up)}$$

$$\frac{}{\mathscr{D}_2 :: \phi_2; \vec{P}_2; x_1 : \mathbf{int}(a_1), x_2 : \mathbf{int}(a_2) \vdash x_2 \uparrow \mathbf{int}(a_2) \Rightarrow x_2}\ \textbf{(elab-var-up)}$$

$$\frac{\mathscr{D}_1 \quad \dfrac{\phi_2; \vec{P}_2 \models a_1 \doteq a_1}{\phi_2; \vec{P}_2 \vdash [] : \mathbf{int}(a_1) \leqslant \mathbf{int}(a_1)}}{\mathscr{D}_3 :: \phi_2; \vec{P}_2; x_1 : \mathbf{int}(a_1), x_2 : \mathbf{int}(a_2) \vdash x_1 \downarrow \mathbf{int}(a_1) \Rightarrow x_1}\ \textbf{(elab-dn-up)}$$

$$\frac{\mathscr{D}_2 \quad \dfrac{\phi_2; \vec{P}_2 \models a_2 \doteq a_2}{\phi_2; \vec{P}_2 \vdash [] : \mathbf{int}(a_2) \leqslant \mathbf{int}(a_2)}}{\mathscr{D}_4 :: \phi_2; \vec{P}_2; x_1 : \mathbf{int}(a_1), x_2 : \mathbf{int}(a_2) \vdash x_2 \downarrow \mathbf{int}(a_2) \Rightarrow x_2}\ \textbf{(elab-dn-up)}$$

$$\frac{\mathscr{D}_3 \quad \mathscr{D}_4}{\mathscr{D}_5 :: \phi_2; \vec{P}_2; x_1 : \mathbf{int}(a_1), x_2 : \mathbf{int}(a_2) \vdash \langle x_1, x_2 \rangle \downarrow \tau_0 \Rightarrow \langle x_1, x_2 \rangle}\ \textbf{(elab-dn-prod)}$$

$$\cfrac{\cfrac{\cfrac{\cfrac{\mathscr{D}_5 \quad \mathscr{D}_0 :: \phi_2; \vec{P}_2 \vdash E_0 : \tau_0 \leqslant \tau_2}{\mathscr{D}_6 :: \phi_2; \vec{P}_2; x_1 : \mathbf{int}(a_1), x_2 : \mathbf{int}(a_2) \vdash \langle x_1, x_2 \rangle \downarrow \tau_2 \Rightarrow e_1}\ \textbf{(elab-dn-up)}}{\phi_1; \vec{P}_1; x_1 : \mathbf{int}(a_1), x_2 : \mathbf{Nat} \vdash \langle x_1, x_2 \rangle \downarrow \tau_2 \Rightarrow e_2}\ \textbf{(elab-dn-}\Sigma\textbf{-left)}}{\emptyset; \emptyset; x_1 : \mathbf{Nat}, x_2 : \mathbf{Nat} \vdash \langle x_1, x_2 \rangle \downarrow \tau_2 \Rightarrow e_3}\ \textbf{(elab-dn-}\Sigma\textbf{-left)}}{\mathscr{D}_7 :: \emptyset; \emptyset \vdash E : \mathbf{Nat} * \mathbf{Nat} \leqslant \tau_2}\ \textbf{(dy-sub-prod)}$$

Fig. 26. Another example of elaboration.

*Proof*

(Sketch) (1), (2) and (3) are proven simultaneously by structural induction on the derivations of $\phi; \vec{P} \vdash E : \tau \leqslant \tau'$, $\phi; \vec{P}; \Gamma \vdash \underline{e} \uparrow \tau \Rightarrow e$ and $\phi; \vec{P}; \Gamma \vdash \underline{e} \downarrow \tau \Rightarrow e$. □

The soundness of elaboration is justified by the following theorem:

*Theorem 5.3*

Assume that $\emptyset; \emptyset; \emptyset \vdash \underline{e} \uparrow \tau \Rightarrow e$ is derivable. Then $\emptyset; \emptyset; \emptyset \vdash e : \tau$ is derivable and $|\underline{e}| \leqslant_{dyn} |e|$.

*Proof*

This follows from Lemma 5.2 and Lemma 2.14 immediately. □

### 5.5 *Implementing elaboration*

A typed programming language ATS is currently under development (Xi, 2005), and its type system supports the form of dependent types in $\lambda_{pat}^{\Pi,\Sigma}$. The elaboration process in ATS is implemented in a manner that follows the presented elaboration rules closely, providing concrete evidence in support of the practicality of these rules. We now mention some strategies adopted in this implementation to address nondeterminism in elaboration.

- The dynamic subtype rules in Figure 21 are applied according to the order in which they are listed. In other words, if two or more dynamic subtype rules are applicable, then the one listed first is chosen. It is important to always choose **(dy-sub-Π-r)** and **(dy-sub-Σ-l)** over **(dy-sub-Π-l)** and **(dy-sub-Σ-r)**, respectively. For instance, this is necessary when we prove $\emptyset; \emptyset \vdash \tau \leqslant \tau$ for $\tau = \Pi a : int.\ \mathbf{int}(a) \rightarrow \mathbf{int}(a)$ and also for $\tau = \Sigma a : int.\ \mathbf{int}(a)$.
- The following "left" rules:
  — **(elab-up-Σ-left)** and **(elab-dn-Σ-left)**
  — **(elab-up-prod-left)** and **(elab-dn-prod-left)**

  are chosen whenever they are applicable.
- The rule **(elab-up-app-2)** is in general chosen over the rule **(elab-up-app-1)**. However, we also provide some special syntax to allow the programmer to indicate that the rule **(elab-up-app-1)** is preferred in a particular case. For instance, the special syntax for doing this in ATS is $\{\ldots\}$: we write $\underline{e}_1\{...\}(\underline{e}_2)$ to indicate that a type of the form $\tau_1 \rightarrow \tau_2$ needs to be synthesized out of $\underline{e}_1$ and then $\underline{e}_2$ is to be checked against $\tau_1$. This kind of elaboration is mostly used in a case where the expression $\underline{e}_1$ is a higher-order function, saving the need for explicitly annotating the expression $\underline{e}_2$.
- We choose the rule **(elab-dn-up)**, which turns analysis into synthesis, only when no other analysis elaboration rules are applicable. The general principle we follow is to prefer analysis over synthesis as the former often makes better use of type annotations and yields more accurate error message report.

While the description of elaboration in terms of the rules in Figure 21, Figure 22 and Figure 23 is intuitively appealing, there is still a substantial gap between the description and its implementation. For instance, the elaboration rules are further refined in (Xi, 1998) to generate constraints when applied, and there are also various issues of reporting error messages as informative as possible. As these issues are mostly concerned with an actual implementation of elaboration, they are of relatively little theoretical significance and thus we plan to address them elsewhere in different contexts.

## 6 Extensions

We extend $\lambda_{pat}^{\Pi,\Sigma}$ with parametric polymorphism, exceptions and references in this section, attesting to the adaptability and practicality of our proposed approach to supporting the use of dependent types in the presence of realistic programming features.

## *6.1 Parametric polymorphism*

We first extend the syntax of $\lambda_{pat}^{\Pi,\Sigma}$ as follows:

$$\begin{array}{llll} \text{types} & \tau & ::= & \ldots \mid \alpha \\ \text{type schemes} & \sigma & ::= & \forall\vec{\alpha}.\ \tau \\ \text{contexts} & \Gamma & ::= & \cdot \mid \Gamma, xf : \sigma \end{array}$$

where we use $\alpha$ to range over type variables. A c-type is now of the following form:

$$\forall\vec{\alpha}.\Pi\phi.\vec{P} \supset (\tau \Rightarrow (\vec{\tau})\delta(\vec{I}))$$

and the typing rules **(ty-var)** and **(ty-const)** are modified as follows:

$$\frac{\Gamma(xf) = \forall\vec{\alpha}.\ \tau \quad \phi \vdash \vec{\tau}\ [\mathbf{type}]}{\phi;\vec{P};\Gamma \vdash xf : \tau[\vec{\alpha} \mapsto \vec{\tau}]}\ \textbf{(ty-var)}$$

$$\frac{\begin{array}{c}\vec{\alpha};\phi_0;\vec{P}_0 \vdash c(\tau) : (\vec{\tau}_0)\delta(\vec{I}_0) \quad \phi \vdash \vec{\tau}\ [\mathbf{type}] \\ \phi \vdash \Theta : \phi_0 \quad \phi;\vec{P} \models \vec{P}_0[\Theta] \quad \phi;\vec{P};\Gamma \vdash e : \tau[\vec{\alpha} \mapsto \vec{\tau}][\Theta]\end{array}}{\phi;\vec{P};\Gamma \vdash c(e) : (\vec{\tau}_0[\vec{\alpha} \mapsto \vec{\tau}])\delta(\vec{I}_0[\Theta])}\ \textbf{(ty-const)}$$

We write $\vec{\alpha};\phi_0;\vec{P}_0 \vdash c(\tau) : (\vec{\tau}_0)\delta(\vec{I}_0)$ to indicate that the constant $c$ is assigned the c-type $\forall\vec{\alpha}.\Pi\phi_0.\vec{P}_0 \supset (\tau \Rightarrow \delta(\vec{I}_0))$, and $\phi \vdash \vec{\tau}$ **[type]** to mean that $\phi \vdash \tau$ **[type]** is derivable for each $\tau$ in $\vec{\tau}$, and $[\vec{\alpha} \mapsto \vec{\tau}]$ for a substitution that maps $\vec{\alpha} = \alpha_1,\ldots,\alpha_n$ to $\vec{\tau} = \tau_1,\ldots,\tau_n$. We now need the following static subtype rule to deal with type variables:

$$\frac{}{\phi;\vec{P} \models \alpha \leqslant_{tp}^{s} \alpha}\ \textbf{(st-sub-var)}$$

In addition, the rule **(st-sub-base)** needs to be modified as follows:

$$\frac{\begin{array}{c}\phi;\vec{P} \models \tau_1 =_{tp}^{s} \tau_1' \quad \cdots \quad \phi;\vec{P} \models \tau_m =_{tp}^{s} \tau_m' \\ \phi;\vec{P} \models I_1 \doteq I_1' \quad \cdots \quad \phi;\vec{P} \models I_n \doteq I_n'\end{array}}{\phi;\vec{P} \models (\tau_1,\ldots,\tau_m)\delta(I_1,\ldots,I_n) \leqslant_{tp}^{s} (\tau_1',\ldots,\tau_m')\delta(I_1',\ldots,I_n')}\ \textbf{(st-sub-base)}$$

where for types $\tau$ and $\tau'$, $\tau =_{tp}^{s} \tau'$ means both $\tau \leqslant_{tp}^{s} \tau'$ and $\tau' \leqslant_{tp}^{s} \tau$ hold. It is possible to replace $\tau_i =_{tp}^{s} \tau_i'$ with $\tau_i \leqslant_{tp}^{s} \tau_i'$ ($\tau_i' \leqslant_{tp}^{s} \tau_i$) if $\delta$ is covariant (contravariant) with respect to its $i^{\text{th}}$ type argument. However, we do not entertain this possibility in this paper (but do so in implementation).

The following typing rules **(ty-poly)** and **(ty-let)** are introduced for handling let-polymorphism as is supported in ML:

$$\frac{\phi;\vec{P};\Gamma \vdash e : \tau \quad \vec{\alpha} \# \Gamma}{\phi;\vec{P};\Gamma \vdash e : \forall\vec{\alpha}.\ \tau}\ \textbf{(ty-poly)}$$

$$\frac{\phi;\vec{P};\Gamma \vdash e_1 : \sigma_1 \quad \phi;\vec{P};\Gamma, x : \sigma_1 \vdash e_2 : \sigma_2}{\phi;\vec{P};\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} : \sigma_2}\ \textbf{(ty-let)}$$

Obviously, we need to associate with the rule **(ty-poly)** a side condition that requires no free occurrences of $\vec{\alpha}$ in $\Gamma$. This condition is written as $\vec{\alpha} \# \Gamma$.

$$
\begin{array}{rcl}
c(\mathbf{raise}(v)) & \hookrightarrow_{ev} & \mathbf{raise}(v) \\
\langle \mathbf{raise}(v), e \rangle & \hookrightarrow_{ev} & \mathbf{raise}(v) \\
\langle v_0, \mathbf{raise}(v) \rangle & \hookrightarrow_{ev} & \mathbf{raise}(v) \\
(\mathbf{raise}(v))(e) & \hookrightarrow_{ev} & \mathbf{raise}(v) \\
v_0(\mathbf{raise}(v)) & \hookrightarrow_{ev} & \mathbf{raise}(v) \\
\mathbf{case}\ \mathbf{raise}(v)\ \mathbf{of}\ ms & \hookrightarrow_{ev} & \mathbf{raise}(v) \\
\mathbf{let}\ x = \mathbf{raise}(v)\ \mathbf{in}\ e\ \mathbf{end} & \hookrightarrow_{ev} & \mathbf{raise}(v) \\
\supset^{-}(\mathbf{raise}(v)) & \hookrightarrow_{ev} & \mathbf{raise}(v) \\
\Pi^{-}(\mathbf{raise}(v)) & \hookrightarrow_{ev} & \mathbf{raise}(v) \\
\mathbf{let}\ \wedge(x) = \mathbf{raise}(v)\ \mathbf{in}\ e\ \mathbf{end} & \hookrightarrow_{ev} & \mathbf{raise}(v) \\
\mathbf{let}\ \Sigma(x) = \mathbf{raise}(v)\ \mathbf{in}\ e\ \mathbf{end} & \hookrightarrow_{ev} & \mathbf{raise}(v) \\
\mathbf{try}\ \mathbf{raise}(v)\ \mathbf{with}\ ms & \hookrightarrow_{ev} & \begin{cases} e[\theta] & \text{if } \mathbf{match}(v,p) \Rightarrow \theta \text{ holds for} \\ & \text{some } p \Rightarrow e \text{ in } ms; \\ \mathbf{raise}(v) & \text{otherwise.} \end{cases}
\end{array}
$$

Fig. 27. Additional forms of redexes and their reducts.

As usual, the type soundness of this extension is established by the subject reduction theorem and the progress theorem stated as follows:

*Theorem 6.1* (*Subject Reduction*)
Assume that $\mathcal{D} :: \emptyset;\emptyset;\emptyset \vdash e_1 : \sigma$ is derivable and $e_1 \hookrightarrow_{ev} e_2$ holds. Then $\emptyset;\emptyset;\emptyset \vdash e_2 : \sigma$ is also derivable.

*Theorem 6.2* (*Progress*)
Assume that $\emptyset;\emptyset;\emptyset \vdash e_1 : \sigma$ is derivable. Then there are the following possibilities:

- $e_1$ is a value, or
- $e_1$ is in M-form, or
- $e_1$ is in U-form, or
- $e_1 \hookrightarrow_{ev} e_2$ holds for some expression $e_2$.

We omit the proofs for these two theorems, which are essentially the same as the ones for Theorem 4.11 and Theorem 4.12.

## *6.2 Exceptions*

We further extend $\lambda_{pat}^{\Pi,\Sigma}$ with exceptions. First, we introduce the following additional syntax, where **exn** is the type for values representing exceptions.

$$
\begin{array}{lccl}
\text{types} & \tau & ::= & \ldots \mid \mathbf{exn} \\
\text{expressions} & e & ::= & \ldots \mid \mathbf{raise}(e) \mid \mathbf{try}\ e\ \mathbf{with}\ ms \\
\text{answers} & ans & ::= & v \mid \mathbf{raise}(v)
\end{array}
$$

An answer of the form **raise**($v$) is called a *raised exception*, where $v$ is the exception being raised. We also introduce in Figure 27 some new forms of ev-redexes and their reducts, which are needed for propagating a raised exception to the top level. In addition, we introduce a new form of evaluation context to allow a raised exception to be captured (potentially):

$$
\text{evaluation contexts} \quad E \quad ::= \quad \ldots \mid \mathbf{try}\ E\ \mathbf{with}\ ms
$$

The following typing rules are needed for handling the newly added language constructs:

$$\frac{\phi;\vec{P};\Gamma \vdash e : \mathbf{exn}}{\phi;\vec{P};\Gamma \vdash \mathbf{raise}(e) : \sigma}\ \textbf{(ty-raise)}$$

$$\frac{\phi;\vec{P};\Gamma \vdash e : \tau \quad \phi;\vec{P};\Gamma \vdash ms : \mathbf{exn} \to \tau}{\phi;\vec{P};\Gamma \vdash \mathbf{try}\ e\ \mathbf{with}\ ms : \tau}\ \textbf{(ty-try)}$$

Again, the type soundness of this extension rests upon the following two theorems:

*Theorem 6.3* (*Subject Reduction*)
Assume $\mathscr{D} :: \emptyset;\emptyset;\emptyset \vdash e_1 : \sigma\ e_1 \hookrightarrow_{ev} e_2$. Then $\emptyset;\emptyset;\emptyset \vdash e_2 : \sigma$ is also derivable.

*Theorem 6.4* (*Progress*)
Assume that $\emptyset;\emptyset;\emptyset \vdash e_1 : \sigma$ is derivable. Then there are the following possibilities:

- $e_1$ is a value, or
- $e_1$ is a raised exception, or
- $e_1$ is in M-form, or
- $e_1$ is in U-form, or
- $e_1 \hookrightarrow_{ev} e_2$ holds for some expression $e_2$.

We omit the proofs for these two theorems, which are essentially the same as the ones for Theorem 4.11 and Theorem 4.12.

Assume the existence of two exception constants *Match* and *Undefined* that are assigned the c-type $() \Rightarrow \mathbf{exn}$. We can then introduce the following evaluation rules for handling expressions in M-form or U-form:

$$\begin{array}{rcl} \mathbf{case}\ v\ \mathbf{of}\ ms & \hookrightarrow_{ev} & \mathbf{raise}(\mathit{Match})\ \text{ if } v \text{ matches none of the patterns in } ms. \\ cf(v) & \hookrightarrow_{ev} & \mathbf{raise}(\mathit{Undefined})\ \text{ if } cf(v) \text{ is undefined.} \end{array}$$

Then the progress theorem can be stated as follows:

*Theorem 6.5* (*Progress*)
Assume that $\emptyset;\emptyset;\emptyset \vdash e_1 : \sigma$ is derivable. Then there are the following possibilities:

- $e_1$ is a value, or
- $e_1$ is a raised exception, or
- $e_1 \hookrightarrow_{ev} e_2$ holds for some expression $e_2$.

So we can now claim that the evaluation of a well-typed program either terminates with an answer, that is, a value or a raised exception, or goes on forever.

## 6.3 *References*

In this section, we add into $\lambda^{\Pi,\Sigma}_{pat}$ another effectful programming feature: references. We first introduce a unary type constructor **ref** that takes a type $\tau$ to form a reference type $(\tau)\mathbf{ref}$. We need the following static subtype rule for dealing with the type constructor **ref**:

$$\frac{\phi;\vec{P} \models \tau_1 \leqslant^s_{tp} \tau_2 \quad \phi;\vec{P} \models \tau_2 \leqslant^s_{tp} \tau_1}{\phi;\vec{P} \models (\tau_1)\mathbf{ref} \leqslant^s_{tp} (\tau_2)\mathbf{ref}}\ \textbf{(st-sub-ref)}$$

which takes into account that **ref** is nonvariant on its type argument. We also assume the existence of the following predefined functions *ref*, ! (prefix) and := (infix) with the assigned c-types:

$$\begin{array}{rcl} \mathit{ref} & : & \forall\alpha.(\alpha) \Rightarrow (\alpha)\mathbf{ref} \\ ! & : & \forall\alpha.((\alpha)\mathbf{ref}) \Rightarrow \alpha \\ := & : & \forall\alpha.((\alpha)\mathbf{ref}, \alpha) \Rightarrow \mathbf{1} \end{array}$$

We use $l$ to range over an infinite set of reference constants $\mathbf{l}_1, \mathbf{l}_2, \ldots$, which one may simply assume are represented as natural numbers. We use $M$ and $\mu$ for stores and store types, respectively, which are defined below as finite mappings:

$$\begin{array}{lccl} \text{stores} & M & ::= & [] \mid M[l \mapsto v] \\ \text{store types} & \mu & ::= & [] \mid \mu[l \mapsto \tau] \end{array}$$

Note that we do allow type variables to occur in a store type. In other words, for each $l \in \mathbf{dom}(\mu)$, $\mu(l)$ may contain free type variables.

We say that a store $M'$ extends another store $M$ if $M(l) = M'(l)$ for every $l \in \mathbf{dom}(M) \subseteq \mathbf{dom}(M')$. Similarly, we say that a store type $\mu'$ extends another store type $\mu$ if $\mu(l) = \mu'(l)$ for every $l \in \mathbf{dom}(\mu) \subseteq \mathbf{dom}(\mu')$.

*Definition 6.6 (Stateful Reduction)*
The stateful reduction relation $(M_1, e_1) \hookrightarrow_{ev/st} (M_2, e_2)$ is defined as follows:

- If $e_1 \hookrightarrow_{ev} e_2$ holds, then we have $(M, e_1) \hookrightarrow_{ev/st} (M, e_2)$.
- If $e_1 = E[\mathit{ref}(v)]$, then we have $(M, e_1) \hookrightarrow_{ev/st} (M[l \mapsto v], E[\langle\rangle])$ for any reference constant $l \notin \mathbf{dom}(M)$. So nondeterminism appears to be involved in this case. This form of nondeterminism can be eliminated if we equate $(M, e)$ and $(M', e')$ whenever one can be obtained from the other by properly renaming the reference constants. The precise definition of such a renaming algorithm is omitted as it is not needed in this paper.
- If $e_1 = E[!l]$ and $M(l) = v$, then we have $(M, e_1) \hookrightarrow_{ev/st} (M, E[v])$.
- If $e_1 = E[l := v]$ and $l \in \mathbf{dom}(M)$, then we have $(M, e_1) \hookrightarrow_{ev/st} (M', E[\langle\rangle])$, where $M'$ is a store such that $\mathbf{dom}(M') = \mathbf{dom}(M)$ and $M'(l) = v$ and $M'(l') = M(l')$ for every $l'$ in $\mathbf{dom}(M)$ that is not $l$.

As usual, we use $\hookrightarrow^*_{ev/st}$ for the reflexive and transitive closure of $\hookrightarrow_{ev/st}$.

Given an answer *ans*, we say that *ans* is observable if $\mathit{ans} = v$ or $\mathit{ans} = \mathbf{raise}(v)$ for some observable value $v$.

*Definition 6.7*
Given two expressions $e_1$ and $e_2$ in $\lambda_{pat}$ extended with polymorphism, exceptions and references, we say that $e_1 \leqslant_{dyn} e_2$ holds if for any store $M_1$ and any context $G$, either $(M_1, G[e_2]) \hookrightarrow^*_{ev/st} (M_2, \mathbf{Error})$ holds for some store $M_2$, or $(M_1, G[e_1]) \hookrightarrow^*_{ev/st} (M_2, \mathit{ans}^*)$ if and only if $(M_1, G[e_2]) \hookrightarrow^*_{ev/st} (M_2, \mathit{ans}^*)$, where $M_2$ ranges over stores and $\mathit{ans}^*$ ranges over the set of observable answers.

The definition of the dynamic subtype relation $\leqslant^d_{tp}$ (Definition 4.15) can be modified according to the above definition of $\leqslant_{dyn}$. In particular, we can readily verify that

Lemma 2.14 still holds (as the generate reduction relation $\hookrightarrow_g$ is still defined in the same manner).

We now outline as follows an approach to typing references, which is largely based upon the one presented in (Harper, 1994). A typing judgment is now of the form $\phi; \vec{P}; \Gamma \vdash_\mu e : \sigma$, and all the previous typing rules need to be modified accordingly. Also, we introduce the following typing rule for assigning types to reference constants:

$$\frac{\mu(l) = \tau}{\phi; \vec{P}; \Gamma \vdash_\mu l : (\tau)\mathbf{ref}} \; \textbf{(ty-ref)}$$

We say that an expression $e$ is value-equivalent if $|e| \leqslant_{dyn} v$ holds for some value $v$. A form of value restriction is imposed by the following rules:

$$\frac{\phi; \vec{P}, P; \Gamma \vdash_\mu e : \tau \quad e \text{ is value-equivalent}}{\phi; \vec{P}; \Gamma \vdash_\mu \supset^+(e) : P \supset \tau} \; \textbf{(ty-}\supset\textbf{-intro)}$$

$$\frac{\phi, a : s; \vec{P}; \Gamma \vdash_\mu e : \tau \quad e \text{ is value-equivalent}}{\phi; \vec{P}; \Gamma \vdash_\mu \Pi^+(e) : \Pi a{:}s.\ \tau} \; \textbf{(ty-}\Pi\textbf{-intro)}$$

$$\frac{\phi; \vec{P}; \Gamma \vdash_\mu e : \tau \quad \vec{\alpha} \# \Gamma \quad \vec{\alpha} \# \mu \quad e \text{ is value-equivalent}}{\phi; \vec{P}; \Gamma \vdash_\mu e : \forall \vec{\alpha}.\ \tau} \; \textbf{(ty-poly)}$$

In the rule **(ty-poly)**, $\vec{\alpha} \# \mu$ means that there is no free occurrence of $\alpha$ in $\mu(l)$ for any $\alpha \in \vec{\alpha}$, where $l$ ranges over $\mathbf{dom}(\mu)$. Also, we need to extend the definition of evaluation contexts as follows:

$$E \quad ::= \quad \ldots \mid \supset^+(E) \mid \Pi^+(E)$$

As an example, when applying the rule **(ty-Π-intro)** to an expression, we need to verify that the expression must be value-equivalent. This is slightly different from the usual form of value restriction (Wright, 1995) imposed, for instance, in ML. The minor change is needed since the elaboration of a value may not necessarily be a value. For instance, this may happen if the rule **(elab-dn-up)** is applied. By Lemma 5.2 and Lemma 2.14, we know that the elaboration of a value is always value-equivalent.

Given a store $M$ and a store type $\mu$, we write $M : \mu$ to mean that the store $M$ can be assigned the store type $\mu$, which is formally defined as follows:

$$\frac{\emptyset; \emptyset; \emptyset \vdash_\mu M(l) : \mu(l) \ \text{ for every } l \in \mathbf{dom}(M) = \mathbf{dom}(\mu)}{M : \mu} \; \textbf{(ty-store)}$$

Again, the type soundness of this extension rests upon the following two theorems:

*Theorem 6.8* (*Subject Reduction*)
Assume $M_1 : \mu_1$ holds and $\emptyset; \emptyset; \emptyset \vdash_{\mu_1} e_1 : \sigma$ is derivable. If $(M_1, e_1) \hookrightarrow_{ev/st} (M_2, e_2)$ holds, then there exists a store typing $\mu_2$ that extends $\mu_1$ such that $M_2 : \mu_2$ holds and $\emptyset; \emptyset; \emptyset \vdash_{\mu_2} e_2 : \sigma$ is derivable.

*Theorem 6.9* (*Progress*)
Assume that $M : \mu$ holds and $\emptyset;\emptyset;\emptyset \vdash_{\mu} e : \sigma$ is derivable. Then there are the following possibilities:

- $e$ is a value $v$, or
- $e$ is a raised exception **raise**$(v)$, or
- $(M, e) \hookrightarrow_{ev/st} (M', e')$ holds for some store $M'$ and expression $e'$ such that $M'$ extends $M$.

The proofs for these two theorems are essentially the same as the ones for Theorem 4.11 and Theorem 4.12, and some related details can also be found in (Harper, 1994). In Appendix C, we provide a proof sketch for Theorem 6.8 that clearly demonstrates some involvement of value restriction.

## 7 Some programming examples

We have finished prototyping a language Dependent ML (DML), which essentially extends ML with a form of dependent types in which type index terms are drawn from the type index languages $\mathscr{L}_{int}$ and $\mathscr{L}_{alg}$ presented in Section 3.3.2 and Section 3.3.1, respectively. At this moment, DML has already become a part of ATS, a programming language with a type system rooted in the framework *Applied Type System* (Xi, 2004). The current implementation of ATS is available on-line (Xi, 2005), which includes a type-checker and a compiler (from ATS to C) and a substantial library (containing more than 25K lines of code written in ATS itself).

When handling integer constraints, we reject nonlinear ones outrightly rather than postpone them as *hard constraints* (Michaylov, 1992), which is planned for future work. This decision of rejecting nonlinear integer constraints may seem *ad hoc*, and it can be too restrictive, sometimes, in a situation where nonlinear constraints (e.g., $\forall n : int.\ n * n \geqslant 0$) need to be dealt with. To address this issue, an approach to combining programming with theorem proving has been proposed recently (Chen & Xi, 2005a).

If the constraints are linear, we negate them and test for unsatisfiability. For instance, the following is a sample constraint generated when an implementation of binary search on arrays is type-checked:

$$\phi; \vec{P} \quad \models \quad l + (h - l)/2 + 1 \leqslant sz$$

where

$$\begin{aligned} \phi &= h : int, l : int, sz : int \\ \vec{P} &= l \geqslant 0, sz \geqslant 0, 0 \leqslant h + 1, h + 1 \leqslant sz, 0 \leqslant l, l \leqslant sz, h \geqslant l \end{aligned}$$

The employed technique for solving linear constraints is mainly based on the Fourier-Motzkin variable elimination approach (Dantzig & Eaves, 1973), but there are many other practical methods available for this purpose such as the SUP-INF method (Shostak, 1977) and the well-known simplex method. We have chosen Fourier-Motzkin's method mainly for its simplicity.[2]

[2] Recently, we have also implemented a constraint solver based the simplex method. Our experience indicates that Fourier-Motzkin's method is almost always superior to the simplex method due to the nature of the constraints encountered in practice.

We now briefly explain this method. We use $x$ for integer variables, $a$ for integers, and $l$ for linear expressions. Given a set of inequalities $S$, we would like to show that $S$ is unsatisfiable. We fix a variable $x$ and transform all the linear inequalities into one of the two forms: $l \leqslant ax$ and $ax \leqslant l$, where $a \geqslant 0$ is assumed. For every pair $l_1 \leqslant a_1x$ and $a_2x \leqslant l_2$, where $a_1, a_2 > 0$, we introduce a new inequality $a_2l_1 \leqslant a_1l_2$ into $S$, and then remove from $S$ all the inequalities involving $x$. Clearly, this is a sound but incomplete procedure. If $x$ were a real variable, then the procedure would also be complete.

In order to handle modulo arithmetic, we also perform another operation to rule out non-integer solutions: we transform an inequality of form

$$a_1x_1 + \cdots + a_nx_n \leqslant a$$

into

$$a_1x_1 + \cdots + a_nx_n \leqslant a',$$

where $a'$ is the largest integer such that $a' \leqslant a$ and the greatest common divisor of $a_1, \ldots, a_n$ divides $a'$. The method can be extended to become both sound and complete while remaining practical (see, for example, (Pugh & Wonnacott, 1992; Pugh & Wonnacott, 1994)).

In DML, we do allow patterns in a matching clause sequence to be overlapping, and sequential pattern matching is performed at run-time. This design can lead to some complications in type-checking, which will be mentioned in Section 7.2. Please refer to (Xi, 2003) for more details on this issue.

We now present some programing examples taken from a prototype implementation of DML, giving the reader some concrete feel as to how dependent types can actually be used to capture programming invariants in practice.

## *7.1 Arrays*

Arrays are a widely used data structure in practice. We use **array** as a built-in type constructor that takes a type $\tau$ and a natural number $n$ to form the type $(\tau)\mathbf{array}(n)$ for arrays of size $n$ in which each element has the type $\tau$.[3] We also have the built-in functions *sub*, *update* and *make*, which are given the following c-types:

$$\begin{array}{rcl} sub & : & \forall\alpha.\Pi n{:}nat.\Pi i{:}\{a : nat \mid a < n\}.\ (\alpha)\mathbf{array}(n) * \mathbf{int}(i) \Rightarrow \alpha \\ update & : & \forall\alpha.\Pi n{:}nat.\Pi i{:}\{a : nat \mid a < n\}.\ (\alpha)\mathbf{array}(n) * \mathbf{int}(i) * \alpha \Rightarrow \mathbf{1} \\ make & : & \forall\alpha.\Pi n{:}nat.\ \mathbf{int}(n) * \alpha \Rightarrow (\alpha)\mathbf{array}(n) \end{array}$$

There is no built-in function for computing the size of an array. Notice that the c-type of *sub* indicates that the function can be applied to an array and an index only if the value of the index is a natural number less than the size of the array. In other words, the quantification $\Pi n{:}nat.\Pi i{:}\{a : nat \mid a < n\}$ acts like a pre-condition for the function *sub*. The c-type of *update* imposes a similar requirement. We may, however, encounter a situation where the programmer knows or believes for some reason that the value of the index is within the bounds of the array, but this

[3] Each valid index of an array is a natural number less than the size of the array

```
datatype 'a Array with nat = {n:nat} Array(n) of int(n) * 'a array(n)

exception Subscript

fun('a) arraySub (Array(n, a), i) =
  if (i < 0) then raise Subscript
  else if (i >= n) then raise Subscript
       else sub (a, i)
withtype {n:nat,i:int} 'a Array(n) * int(i) -> 'a

fun('a) arrayUpdate (Array(n, a), i, x) =
  if (i < 0) then raise Subscript
  else if (i >= n) then raise Subscript
       else update (a, i, x)
withtype {n:nat,i:int} 'a Array(n) * int(i) * 'a -> unit

fun('a) makeArray (n, x) = Array (n, make (n, x))
withtype {n:nat} int(n) * 'a -> 'a Array(n)

fun('a) arrayLength (Array(n, _)) = n
withtype {n:nat} 'a Array(n) -> int(n)
```

Fig. 28. A datatype for arrays with size information and some related functions.

property is difficult or even impossible to be captured in the type system of DML. In such a situation, the programmer may need to use run-time array bound checks to overcome the difficulty. We now present some type-theoretical justification for run-time array bound checking in DML.

In Figure 28, we declare a type constructor **Array** for forming types for arrays with size information. The only value constructor *Array* associated with the type constructor **Array** is assigned the following c-type:

$$\forall\alpha.\Pi n{:}nat.\ \mathbf{int}(n) * (\alpha)\mathbf{array}(n) \Rightarrow (\alpha)\mathbf{Array}(n)$$

The defined functions *arraySub*, *arrayUpdate* and *makeArray* correspond to the functions *sub*, *update* and *make*, respectively. Note that run-time array bound checks are inserted in the implementation of *arraySub* and *arrayUpdate*. For an array carrying size information, the function *arrayLength* simply extracts out the information. Additional examples can be found in (Xi & Pfenning, 1998) that makes use of dependent types in eliminating run-time array bound checks.

Clearly, the programmer now has the option to decide which subscripting (updating) function should be used: *sub* or *arraySub* (*update* or *arrayUpdate*)? When compared to the former, the latter is certainly less efficient and may incur a run-time exception. However, in order to use the former, the programmer often needs to capture more program invariants by supplying type annotations. This point is shown clearly when we compare the two (essentially identical) implementations of the standard binary search on integer arrays in Figure 29. In the first implementation, we use the array subscripting function *arraySub*, which incurs run-time array bound checks. In the second implementation, we instead use *sub*, which incurs no run-time

```
datatype ORDER = LESS | EQUAL | GREATER

fun binarySearch cmp (key, Vec) = let (* require run-time bound checks *)
  fun loop (l, u) =
    if u < l then NONE
    else let
      val m = l + (u-l) / 2
      val x = arraySub (Vec, m) (* require bound checks *)
    in case cmp (x, key) of
       LESS => loop (m+1, u)
     | GREATER => loop (l, m-1)
     | EQUAL => SOME (m)
    end
  withtype int * int -> int option
in loop (0, length Vec - 1) end
withtype {n:nat} ('a * 'a -> Bool) -> ('a * 'a Array(n)) -> int option

fun binarySearch cmp (key, Vec) = let (* require NO run-time bound checks *)
  val Array (n, vec) = Vec
  fun loop (l, u) =
    if u < l then NONE
    else let
      val m = l + (u-l) / 2
      val x = sub (vec, m) (* require no bound checks *)
    in case cmp (x, key) of
       LESS => loop (m+1, u)
     | GREATER => loop (l, m-1)
     | EQUAL => SOME (m)
    end
    withtype {i:nat,j:int | i <= j+1 <= n} int(i) * int(j) -> int option
in loop (0, n-1) end
withtype {n:nat} ('a * 'a -> Bool) -> ('a * 'a Array(n)) -> int option
```

Fig. 29. Two implementations of binary search on integer arrays in DML.

array bound checks. Clearly, the second implementation is superior to the first one when either safety or efficiency is of the concern. However, the programmer needs to provide a more informative type for the inner function *loop* in order to eliminate the array bound checks. In this case, the provided type captures the invariant that $i \leqslant j+1 \leqslant n$ holds whenever *loop* is called, where $i$ and $j$ are integer values of $l$ and $u$, respectively, and $n$ is the size of the array being searched.

## *7.2 Red-black trees*

We now show a typical use of dependent types in capturing certain inherent invariants in data structures.

A red-black tree (RBT) is a balanced binary tree that satisfies the following conditions:

1. All leaves are marked black and all other nodes are marked either red or black;

```
sort color = {a:int | 0 <= a <= 1} (* sort declaration *)

datatype 'a rbtree (color, nat, nat) = (* color, black height, violation *)
    E(0, 0, 0)
  | {cl:color, cr:color, bh:nat}
    B(0, bh+1, 0) of 'a rbtree(cl, bh, 0) * 'a * 'a rbtree(cr, bh, 0)
  | {cl:color, cr:color, bh:nat}
    R(1, bh, cl+cr) of 'a rbtree(cl, bh, 0) * 'a * 'a rbtree(cr, bh, 0)

fun restore (R(R(a, x, b), y, c), z, d) = R(B(a, x, b), y, B(c, z, d))
  | restore (R(a, x, R(b, y, c)), z, d) = R(B(a, x, b), y, B(c, z, d))
  | restore (a, x, R(R(b, y, c), z, d)) = R(B(a, x, b), y, B(c, z, d))
  | restore (a, x, R(b, y, R(c, z, d))) = R(B(a, x, b), y, B(c, z, d))
  | restore (a, x, b) = B(a, x, b)
withtype {cl:color, cr:color, bh:nat, vl:nat, vr:nat | vl+vr <= 1}
  'a rbtree(cl, bh, vl) * 'a * 'a rbtree(cr, bh, vr) ->
  [c:color] 'a rbtree(c, bh+1, 0)

exception ItemAlreadyExists

fun insert cmp (x, t) = let
  fun ins (E) = R(E, x, E)
    | ins (B (a, y, b)) = (
        case cmp (x, y) of
            LESS => restore (ins a, y, b)
          | GREATER => restore(a, y, ins b)
          | EQUAL => raise ItemAlreadyExists
      )
    | ins (R (a, y, b)) = (
        case cmp (x, y) of
            LESS => R (ins a, y, b)
          | GREATER => R(a, y, ins b)
          | EQUAL => raise ItemAlreadyExists
      )
  withtype {c:color, bh:nat}
    'a rbtree(c, bh, 0) -> [c':color, v:nat | v <= c] 'a rbtree(c', bh, v)
in case ins t of R(a, y, b) => B(a, y, b) | t => t end
withtype {c:color, bh:nat} ('a * 'a -> ORDER) ->
  key * 'a rbtree(c, bh, 0) -> [bh':nat] 'a rbtree(0, bh', 0)
```

Fig. 30. A red-black tree implementation.

2. 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;
3. the two children of every red node are black.

It is a common practice to use the RBT data structure to implement a dictionary. We declare a datatype in Figure 30, which precisely captures the above three properties of being a RBT.

A sort *color* is declared for the type index terms representing the colors of nodes. We use 0 for black and 1 for red. The type constructor **rbtree** is indexed with a triple

$(c, bh, v)$, where $c, bh, v$ stand for the color of the node, the black height of the tree rooted at the node, and the number of color violations in the tree, respectively. We record one color violation if a red node is followed by another red one, and thus a valid RBT must have no color violations. Clearly, the types of value constructors associated with the type constructor **rbtree** indicate that color violations can only occur at the top node. Also, notice that a leaf, that is, $E$, is considered black. Given the datatype declaration and the explanation, it should be clear that the type of a RBT in which all keys are of type $\tau$ is simply:

$$\Sigma c : color. \Sigma bh : nat.\ (\tau)\mathbf{rbtree}(c, bh, 0),$$

that is, a RBT is a tree that has some top node color $c$ and some black height $bh$ but no color violations.

It is an involved task to implement RBT. The implementation we present in Figure 30 is largely adopted from one in (Okasaki, 1998), though there are some minor modifications. We explain how the insertion operation on a RBT is implemented. Clearly, the invariant we intend to capture is that inserting an entry into a RBT yields another RBT. In other words, we intend to declare that the insertion operation has the following type:

$$\forall \alpha.(\alpha * \alpha \to \mathbf{Bool}) \to \alpha * (\alpha)\mathbf{RBT} \to (\alpha)\mathbf{RBT}$$

where **Bool** is the type for booleans and $(\alpha)\mathbf{RBT}$ is defined to be:

$$\Sigma c : color. \Sigma bh : nat.\ (\alpha)\mathbf{rbtree}(c, bh, 0)$$

If we insert an entry into a RBT, some properties on RBT may be invalidated, and the invalidated properties can then be restored through some rotation operations. The function *restore* in Figure 30 is defined for this purpose.

The type of *restore*, though long, is easy to understand. It states that this function takes a tree with at most one color violation, an entry and a RBT, and returns a RBT. The two trees in the argument must have the same black height $bh$ for some natural number $bh$ and the black height of the returned RBT is $bh + 1$. This information can be of great help for understanding the code. It is not trivial at all to verify the information manually, and we could imagine that almost everyone who did this would appreciate the availability of a type-checker to perform it automatically.

There is a substantial difference between type-checking a matching clause sequence in DML and in ML. The operational semantics of ML requires that pattern matching be performed sequentially, that is, the chosen pattern matching clause is always the first one that matches a given value. For instance, in the definition of the function *restore*, if the last clause is chosen at run-time, then we know the argument of *restore* does not match any one of the clauses ahead of the last one. This must be taken into account when we type-check pattern matching in DML. One approach is to expand patterns into disjoint ones. For instance, the pattern $(a, x, b)$ expands into 36 patterns $(pattern_1, x, pattern_2)$, where $pattern_1$ and $pattern_2$ range over the following six patterns:

$$R(B\_, \_, B\_), R(B\_, \_, E), R(E, \_, B\_), R(E, \_, E), B\_, E$$

Unfortunately, such an expansion may lead to combinatorial explosion. An alternative is to require the programmer to indicate whether such an expansion is needed. Neither of these was available in the original implementation of DML, and the author had to take the inconvenience to expand patterns into disjoint ones when necessary. Recently, we have implemented the alternative mentioned above. For instance, the last clause in the definition of *restore* can be written as follows:

```
  | restore (a, x, b) == B(a, x, b)
```

where the special symbol == indicates to the type-checker that the pattern involved here needs to be (automatically) expanded into ones that are disjoint from the patterns in the previous clauses. For a thorough study on the issue of type-checking pattern matching clauses in DML, please refer to (Xi, 2003).

The complete implementation of the insertion operation follows immediately. Notice that the type of the function *ins* indicates that *ins* may return a tree with one color violation if it is applied to a tree with a red top node. This violation can be eliminated by replacing the top node with a black one for every returned tree with a red top node.

Moreover, we can use an extra index to capture the size information of a RBT. If we do so, we can then show that the *insert* function always returns a RBT of size $n + 1$ when given a RBT of size $n$ (note that an exception is raised if the entry to be inserted already exists in the tree). A complete implementation of red-black trees is available on-line (Xi, 2005), which includes deletion and join operations as well. Also, several examples that make use of dependent types in capturing invariants in other data structures (e.g., Braun trees, random-access lists, binomial heaps) can be found in (Xi, 1999).

We point out that it is also possible to capture the invariants of being a RBT by using nested datatypes (Kahrs, 2001). This is a rather different approach as it, to a large extent, employs run-time checking (in the form of pattern matching) to ensure that a binary tree meets the criteria of being a red-black tree. The use of nested datatypes essentially guarantees the adequacy of such run-time checking. A more systematic study on making use of nested types in capturing program invariants can be found in (Hinze, 2001).

## *7.3 A type-preserving evaluator*

We now implement an evaluator for an object language based on the simply typed $\lambda$-calculus, capturing in the type system of DML that the evaluator is type-preserving at the object level. Apart form using integer expressions as type indexes in the previous examples, we employ algebraic terms as type indexes in this example.

We use the following syntax to define a sort *ty* for representing simple types in the object language:

```
datasort ty = Bool | Int | Arrow of (ty, ty)
```

where we assume that *Bool* and *Int* represent two simple base types $\hat{bool}$ and $\hat{int}$, respectively, and *Arrow* represents (the overloaded) constructor $\rightarrow$ for forming simple

```
datatype EXP (ty) =
    EXPint (Int) of int
  | EXPbool (Bool) of bool
  | EXPadd (Int) of EXP (Int) * EXP (Int)
  | EXPsub (Int) of EXP (Int) * EXP(Int)
  | EXPmul(Int) of EXP (Int) * EXP (Int)
  | EXPdiv(Int) of EXP (Int) * EXP (Int)
  | EXPzero (Bool) of EXP (Int)
  | {a: ty} EXPif (a) of EXP (Bool) * EXP (a) * EXP (a)
  | {a1: ty, a2: ty} EXPlam (Arrow (a1, a2)) of (EXP (a1) -> EXP (a2))
  | {a1: ty, a2: ty} EXPapp (a2) of (EXP (Arrow (a1, a2)), EXP (a1))
  | {a1: ty, a2: ty} EXPlet (a2) of (EXP (a1), (EXP(a1) -> EXP(a2)))
  | {a: ty} EXPfix (a) of (EXP (a) -> EXP (a))
```

Fig. 31. A datatype for higher-order abstract syntax.

function types. For instance, we use the term $Arrow(Int, Arrow(Int, Bool))$ to represent the simple type $\hat{int} \rightarrow (\hat{int} \rightarrow \hat{bool})$ in the object language, where $\hat{bool}$ and $\hat{int}$ are two simple base types and (the overloaded) $\rightarrow$ is a simple type constructor. We use a form of higher-order abstract syntax (h.o.a.s) (Church, 1940; Pfenning & Elliott, 1988; Pfenning, n.d.) to represent expressions in the object language. In Figure 31, we declare a type constructor **EXP**, which takes a type index term $I$ of sort $ty$ to form a type $\mathbf{EXP}(I)$ for the values that represent closed expressions in the object language that can be assigned the type represented by $I$. For example, the function $\lambda x : \hat{int}.x + x$ in the object language is represented as $EXPlam(\mathbf{lam}\, x. EXPadd(x, x))$, which can be given the type $\mathbf{EXP}(Arrow(Int, Int))$. The usual factorial function can be represented as follows (in the concrete syntax of DML),

```
EXPfix (lam f =>
  EXPlam (lam x =>
    EXPif (EXPzero (x),
           EXPint(1),
           EXPmul (x, EXPapp (f, EXPsub (x, EXPint(1)))))))
```

which can also be given the type $\mathbf{EXP}(Arrow(Int, Int))$. We often refer to such a representation as a form of typeful representation since the type of an expression in the object language is now reflected in the type of the representation of the expression.

We now implement a function *evaluate* in Figure 32. The function is an evaluator for the object language, taking (the representation of) an object expression and returning (the representation of) the value of the object expression. Notice that the function is assigned the type $\Pi a : ty.\ \mathbf{EXP}(a) \rightarrow \mathbf{EXP}(a)$, indicating that the function is type-preserving at the object level. Also, we point out that (extended) type-checking in DML guarantees that no pattern matching failure can occur in this example.

Clearly, a natural question is whether we can also implement a type-preserving evaluator for an object language based on the second-order polymorphic $\lambda$-calculus or system F (Girard, 1972). In order to do so, we need to go beyond algebraic terms, employing $\lambda$-terms to encode polymorphic types in the object language. First

```
fun evaluate (v as EXPint _) = v

  | evaluate (v as EXPbool _) = v

  | evaluate (EXPadd (e1, e2)) = let // no pattern matching failure
      val EXPint (i1) = evaluate e1 and EXPint (i2) = evaluate e2
    in EXPint (i1+i2) end

  (* the cases for EXPsub, EXPmul, EXPdiv are omitted *)

  | evaluate (EXPzero e) = let // no pattern matching failure
      val EXPint (n) = evaluate e
    in EXPbool (n=0) end

  | evaluate (EXPif (e0, e1, e2)) = let // no pattern matching failure
      val EXPbool (b) = evaluate e0
    in if b then evaluate e1 else evaluate e2 end

  | evaluate (EXPapp (e1, e2)) = let // no pattern matching failure
      val EXPlam (f) = evaluate e1
    in evaluate (f (evaluate e2)) end

  (* the case for EXPlet is omitted *)

  | evaluate (v as EXPlam _) = v

  | evaluate (e as EXPfix f) = evaluate (f e)

withtype {a: ty} EXP (a) -> EXP (a)
```

Fig. 32. An implementation of a type-preserving evaluation function in DML.

we extend the definition of the sort *ty* as follows so that universally quantified types can also be represented:

```
datasort ty = ... | All of (ty -> ty)
```

Given a term $f$ of sort $ty \to ty$, $All(f)$ represents the type $\forall\alpha.\ \tau$ if for each type $\tau_0$, $f(t)$ represents the type $\tau[\alpha \mapsto \tau_0]$ as long as $t$ represents the type $\tau_0$. For instance, $All(\lambda a.Arrow(a, Arrow(a, Int)))$ represents the type $\forall\alpha.\alpha \to \alpha \to \mathbf{int}$; the term $All(\lambda a.(All(\lambda b.Arrow(a, Arrow(b, a)))))$ represents the type $\forall\alpha.\forall\beta.\alpha \to \beta \to \alpha$. With this strategy, we have no difficulty in implementing a type-preserving evaluator for an object language based on the second-order polymorphic language calculus. We have actually already done this in the programming language ATS (Xi, 2005). Note that the type indexes involved in this example are drawn from $\mathscr{L}_\lambda$.

It is also possible to implement a type-preserving evaluator through the use of first-order abstract syntax (f.o.a.s), and further details on this subject can be found in (Chen & Xi, 2003; Chen *et al.*, 2005), where some interesting typeful program transformations (e.g., a call-by-value continuation-passing style (CPS) transformation (Meyer & Wand, 1985; Griffin, 1990)) are studied.

In (Xi *et al.*, 2003), a typeful implementation of simply typed $\lambda$-calculus based on guarded recursive (g.r.) datatypes is presented. There, a g.r. datatype constructor **HOAS** (of the kind *type* $\rightarrow$ *type*) is declared such that for each simply typed $\lambda$-expression of some simple type $T$, its representation can be assigned the type $(\underline{T})$**HOAS**, where $\underline{T}$ is the representation of $T$. More precisely, $\underline{T}$ can be defined as follows:

$$\underline{b} \;=\; b \qquad\qquad \underline{T_1 \rightarrow T_2} \;=\; \underline{T_1} \rightarrow \underline{T_2}$$

where each simple base type $b$ is represented by a type $\underline{b}$ (in the implementation language). For instance, the type for the representation of the simply typed expression $\lambda x : \hat{int}.x$ is $(\underline{\hat{int}} \rightarrow \underline{\hat{int}})$**HOAS**, where $\hat{int}$ is a simple base type. With this representation for simply typed $\lambda$-calculus, an evaluation function of the type $\forall\alpha.(\alpha)$**HOAS** $\rightarrow \alpha$ can be implemented. A particular advantage of this implementation is that we can use native tagless values in the implementation language to directly represent values of object expressions. This can be of great use in a setting (e.g., meta-programming) where the object language needs to interact with the implementation language (Chen & Xi, 2005b). Given that DML is a conservative extension of ML, this is clearly something that cannot be achieved in DML. The very reason for this is that DML does not allow type equalities like $\tau_1 \doteq \tau_2$ (meaning both $\tau_1 \leqslant \tau_2$ and $\tau_2 \leqslant \tau_1$) to appear in index contexts $\phi$. In ATS, this restriction is lifted, resulting in a much more expressive type system but also a (semantically) much more complicated constraint relation (on types and type indexes) (Xi, 2004).

## 8 Related work

Our work falls in between full program verification, either in type theory or systems such as PVS (Owre *et al.*, 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 Ph.D. Thesis (Zenger, 1998) (an earlier version of which is described in Zenger (1997)). He worked in the context of lazy functional programming. His language is simple and clean 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 existentially quantified 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 (much) more explicit. For instance, one may introduce the following datatype to represent the existentially quantified type $\Sigma a : int.$ **int**$(a)$:

```
datatype IntType = {a: int} Int of int (a)
```

where the value constructor *Int* is assigned the c-type $\Pi a : int.\ \mathbf{int}(a) \Rightarrow IntType$. If one also wants a type for natural numbers, then another datatype needs to be introduced as follows:

```
datatype NatType = {a: int | a >= 0} Nat of int (a)
```

where *Nat* is assigned the c-type $\forall a : int.a \geqslant 0 \supset (\mathbf{int}(a) \Rightarrow NatType)$. If types for positive integers, negative integers, etc. are wanted, then corresponding datatypes have to be introduced accordingly. Also, one may have to define functions between these datatypes. For example, a function from *NatType* to *IntType* is needed to turn natural numbers into integers. At this point, we have strong doubts about the viability of such an approach to handling existentially quantified types, especially, in cases where the involved type index terms are drawn from a (rich) type index language such as $\mathscr{L}_{int}$. Also, since the language in Zenger (1998) is pure, the issue of supporting indexed types in the presence of effects is not studied there.

When compared to traditional type systems for programming languages, perhaps the most closely related work is refinement types (Freeman & 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 studies 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). In Dunfield (2002), dependent types as formulated earlier (Xi, 1998; Xi & Pfenning, 1999) are combined with refinement types via regular tree grammar (Freeman & Pfenning, 1991), and this combination shows that these two forms of types can coexist naturally. Subsequently, a pure type assignment system that includes intersection and dependent types, as well as union and existential types, is constructed in Dunfield & Pfenning (2003). This is a rather different approach when compared with the one presented in the paper as it does not employ elaboration as a central part of the development. In particular, type-checking is undecidable, and the issue of undecidable type-checking is addressed in Dunfield & Pfenning (2004), where a new reconstruction of the rules for *indefinite* types (existential, union, empty types) using evaluation contexts is given. This new reconstruction avoids elaboration and is decidable in theory. However, its effectiveness in practice is yet to be substantiated. In particular, the effectiveness of handling existential types through the use contextual type annotations in this reconstruction requires further investigation.

Parent (1995) proposed to reverse the process of extracting programs from constructive proofs in Coq (Dowek *et al.*, 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 direct interaction with a theorem prover.

*Extended ML* (Sannella & 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 design requires theorem proving during extended type-checking. In contrast, we specify and check less information about functions, thus avoiding general theorem proving.

*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 price for this is undecidable type-checking in Cayenne. For instance, the `printf` in *C*, which is not directly typable in ML,[4] can be made typable in Cayenne, and modules can be replaced with records, but the notion of datatype refinement does not exist. As a pure language, Cayenne also does not address issue of supporting dependent types in the presence of effects. This clearly separates our language design from that of Cayenne.

The notion of sized types is introduced in Hughes *et al.* (1996) for proving the correctness of reactive systems. Though there exist some similarities between sized types and datatype refinement in DML($\mathscr{L}$) for some type index language $\mathscr{L}$ over the domain of natural numbers, the differences are also substantial. We feel that the language presented in *et al.* (1996) seems too restrictive for general programming as its type system 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($\mathscr{L}$) at this moment.

Jay & Sekanina (1996) have introduced a technique for array bounds checking elimination 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($\mathscr{L}$), where constraints are linear inequalities on integers. We feel that their design is more restrictive and seems more promising for languages based on iteration schema rather than general recursion.

A crucial feature in DML($\mathscr{L}$) that does not exist in either of the above two systems is *existential dependent types*, or more precisely, *existentially quantified dependent types*, which is indispensable in our experiment.

The work on local type inference by Pierce and Turner (Pierce & Turner, 1998), which includes some empirical studies, is also based on a similar bi-directional strategy for elaboration, although they are mostly concerned with the interaction between polymorphism and subtyping, while we are concerned with dependent types. This work is further extended by Odersky, Zenger and Zenger in their study on colored local type inference (Odersky *et al.*, 2001). However, we emphasize that the use of constraints for index domains is quite different from the use of constraints to model subtyping constraints (Sulzmann *et al.*, 1997), for example.

[4] In ML, it is possible to implement a function similar to `printf`, which, instead of applying to a format string, applies to a function argument corresponding to a parsed format string. Please see (Danvy, 1998) for further details.

Along a different but closely related line of research, a new notion of types called *guarded recursive (g.r.) datatypes* are introduced (Xi *et al*., 2003). Also, *phantom types* are studied in Cheney & Hinze (2003), which are largely identical to g.r. datatypes. Recently, this notion of types are given the name *generalized algebraic datatypes* (GADTs). On the syntactic level, GADTs are of great similarity to universal dependent datatypes in $\lambda_{pat}^{\Pi,\Sigma}$, essentially using types as type indexes. However, unlike DML-style dependent types, ML extended with GADTs is no longer a conservative extension over ML as strictly more programs can be typed in the presence of GADTs. On the semantic level, g.r. datatypes are a great deal more complex than dependent types. At this moment, we are not aware of any model-theoretical explanation of GADTs.

Many examples in DML($\mathscr{L}$) can also be handled in terms of GADTS. As an example, suppose we want to use types to represent natural numbers; we can introduce a type $Z$ and a type constructor $S$ of the kind *type* → *type*; for each natural number $n$, we use $S^n(Z)$ to represent $n$, where $S^n$ means $n$ applications of $S$. There are some serious drawbacks with this approach. For instance, it cannot rule out forming a type like $S(Z * Z)$, which does not represent any natural number. More importantly, the programmer may need to supply proofs in a program in order for the program to pass type-checking (Sheard, 2004). There are also various studies on type inference addressing GADTs (Pottier & Régis-Gianas, 2006; Jones *et al*., 2005), which are of rather different focus and style from the elaboration in Section 5.

Noting the close resemblance between DML-style dependent types and the guarded recursive datatypes, we immediately initiated an effort to unify these two forms of types in a single framework, leading to the design and formalization of *Applied Type System* ($\mathscr{ATS}$) (Xi, 2004). Compared to $\lambda_{pat}^{\Pi,\Sigma}$, $\mathscr{ATS}$ is certainly much more general and expressive, but it is also much more complicated, especially, semantically. For instance, unlike in $\lambda_{pat}^{\Pi,\Sigma}$, the definition of type equality in $\mathscr{ATS}$ involves impredicativity. In DML, we impose certain restrictions on the syntactic form of constraints so that some effective means can be found for solving constraints automatically. Evidently, this is a rather *ad hoc* design in its nature. In ATS (Xi, 2005), a language with a type system rooted in $\mathscr{ATS}$, we adopt a different design. Instead of imposing syntactical restrictions on constraints, we provide a means for the programmer to construct proofs to attest to the validity of constraints. In particular, we accommodate a programming paradigm in ATS that enables the programmer to combine programming with theorem proving (Chen & Xi, 2005a).

## 9 Conclusion

We have presented an approach that can effectively support the use of dependent types in practical programming, allowing for specification and inference of significantly more precise type information and thus facilitating program error detection and compiler optimization. By separating type index terms from programs, we make it both natural and straightforward to accommodate dependent types in the presence of realistic programming features such as (general) recursion and effects (e.g., exceptions and references). In addition, we have formally established the type soundness of $\lambda_{pat}^{\Pi,\Sigma}$, the core dependent type system in our development, and have also justified the correctness of a set of elaboration rules, which play a crucial role in reducing (not eliminating) the amount of explicit type annotation needed in practice.

On another front, we have finished a prototype implementation of Dependent ML (DML), which essentially extends ML with a restricted form of dependent types such that the type index terms are required to be integer expressions drawn from the type index language $\mathscr{L}_{int}$ presented in Section 3. A variety of programming examples have been constructed in support of the practicality of DML, some of which are shown in Section 7.

Lastly, we point out that $\lambda_{pat}^{\Pi,\Sigma}$ can be classified as an applied type system in the framework $\mathscr{ATS}$ (Xi, 2004). At this moment, DML has already been fully incorporated into ATS (Xi, 2005).

**Acknowledgments**

The current paper is partly based on the author's doctoral dissertation (Xi, 1998) supervised by Frank Pfenning, and an extended abstract of the dissertation is already in publication (Xi & Pfenning, 1999). The author sincerely thanks Frank Pfenning for his suggestion of the research topic and his guidance in the research conducted subsequently. In addition, the author acknowledges many discussions with Chiyan Chen regarding the subject of elaboration presented in Section 5 and thanks him for his efforts on proofreading a draft of paper. Also, the author thanks the anonymous referees for their voluminous constructive comments, which have undoubtedly raised the quality of the paper significantly.

## A Proof of Lemma 2.14

The key step in the proof of Lemma 2.14 is to show that if $e \hookrightarrow_g^* e'$ and $e \hookrightarrow_{ev} e_1$ hold then there exists $e_1'$ such that both $e_1 \hookrightarrow_g^* e_1'$ and $e' \hookrightarrow_{ev}^* e_1'$ hold. We are to employ a notion of parallel reduction (Takahashi, 1995) to complete this key step.

*Definition A.1* (*Parallel general reduction*)
Given two expressions $e$ and $e'$ in $\lambda_{pat}$, we say that $e$ g-reduces to $e'$ *in parallel* if $e \hookrightarrow\!\!\!\rightarrow_g e'$ can be derived according to the rules in Figure A 1.

Note that the symbol $\hookrightarrow\!\!\!\rightarrow_g$ is overloaded to also mean parallel reduction on evaluation contexts, matching clause sequences and substitutions.

Intuitively, $e \hookrightarrow\!\!\!\rightarrow_g e'$ means that $e'$ can be obtained from reducing (many) g-redexes in $e$ simultaneously. Clearly, if $e$ g-reduces to $e'$, that is, $e \hookrightarrow_g e'$ holds, then $e$ g-reduces to $e'$ in parallel, that is $e \hookrightarrow\!\!\!\rightarrow_g e'$ holds.

*Proposition A.2*
Assume that $e$ and $e'$ are two expressions in $\lambda_{pat}$ such that $e \hookrightarrow\!\!\!\rightarrow_g e'$ holds.

1. If $e$ is an observable value, then $e = e'$.
2. If $e$ is in V-form, then so is $e'$.
3. If $e$ is in M-form, then so is $e'$.
4. If $e$ is in U-form, then so is $e'$.

$$\frac{}{xf \hookrightarrow\!\!\!\!\twoheadrightarrow_g xf} \qquad \frac{e \hookrightarrow\!\!\!\!\twoheadrightarrow_g e'}{c(e) \hookrightarrow\!\!\!\!\twoheadrightarrow_g c(e')} \qquad \frac{}{\langle\rangle \hookrightarrow\!\!\!\!\twoheadrightarrow_g \langle\rangle} \qquad \frac{e_1 \hookrightarrow\!\!\!\!\twoheadrightarrow_g e_1' \quad e_2 \hookrightarrow\!\!\!\!\twoheadrightarrow_g e_2'}{\langle e_1, e_2\rangle \hookrightarrow\!\!\!\!\twoheadrightarrow_g \langle e_1', e_2'\rangle}$$

$$\frac{e \hookrightarrow\!\!\!\!\twoheadrightarrow_g e'}{\mathbf{fst}(e) \hookrightarrow\!\!\!\!\twoheadrightarrow_g \mathbf{fst}(e')} \qquad \frac{e \hookrightarrow\!\!\!\!\twoheadrightarrow_g e'}{\mathbf{snd}(e) \hookrightarrow\!\!\!\!\twoheadrightarrow_g \mathbf{snd}(e')} \qquad \frac{e \hookrightarrow\!\!\!\!\twoheadrightarrow_g e' \quad ms \hookrightarrow\!\!\!\!\twoheadrightarrow_g ms'}{\mathbf{case}\ e\ \mathbf{of}\ ms \hookrightarrow\!\!\!\!\twoheadrightarrow_g \mathbf{case}\ e'\ \mathbf{of}\ ms'}$$

$$\frac{e \hookrightarrow\!\!\!\!\twoheadrightarrow_g e'}{\mathbf{lam}\, x.\, e \hookrightarrow\!\!\!\!\twoheadrightarrow_g \mathbf{lam}\, x.\, e'} \qquad \frac{e_1 \hookrightarrow\!\!\!\!\twoheadrightarrow_g e_1' \quad e_2 \hookrightarrow\!\!\!\!\twoheadrightarrow_g e_2'}{e_1(e_2) \hookrightarrow\!\!\!\!\twoheadrightarrow_g e_1'(e_2')}$$

$$\frac{e \hookrightarrow\!\!\!\!\twoheadrightarrow_g e'}{\mathbf{fix}\, f.\, e \hookrightarrow\!\!\!\!\twoheadrightarrow_g \mathbf{fix}\, f.\, e'} \qquad \frac{e_1 \hookrightarrow\!\!\!\!\twoheadrightarrow_g e_1' \quad e_2 \hookrightarrow\!\!\!\!\twoheadrightarrow_g e_2'}{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} \hookrightarrow\!\!\!\!\twoheadrightarrow_g \mathbf{let}\ x = e_1'\ \mathbf{in}\ e_2'\ \mathbf{end}}$$

$$\frac{v_1 \hookrightarrow\!\!\!\!\twoheadrightarrow_g v_1'}{\mathbf{fst}(\langle v_1, v_2\rangle) \hookrightarrow\!\!\!\!\twoheadrightarrow_g v_1'} \qquad \frac{v_2 \hookrightarrow\!\!\!\!\twoheadrightarrow_g v_2'}{\mathbf{snd}(\langle v_1, v_2\rangle) \hookrightarrow\!\!\!\!\twoheadrightarrow_g v_2'}$$

$$\frac{e \hookrightarrow\!\!\!\!\twoheadrightarrow_g e' \quad v \hookrightarrow\!\!\!\!\twoheadrightarrow_g v'}{(\mathbf{lam}\, x.\, e)(v) \hookrightarrow\!\!\!\!\twoheadrightarrow_g e'[x \mapsto v']} \qquad \frac{e \hookrightarrow\!\!\!\!\twoheadrightarrow_g e'}{\mathbf{fix}\, f.\, e \hookrightarrow\!\!\!\!\twoheadrightarrow_g e'[f \mapsto \mathbf{fix}\, f.\, e']}$$

$$\frac{e \hookrightarrow\!\!\!\!\twoheadrightarrow_g e' \quad v \hookrightarrow\!\!\!\!\twoheadrightarrow_g v'}{\mathbf{let}\ x = v\ \mathbf{in}\ e\ \mathbf{end} \hookrightarrow\!\!\!\!\twoheadrightarrow_g e'[x \mapsto v']}$$

$$\frac{\mathbf{match}(v, p_k) \Rightarrow \theta \quad e_k \hookrightarrow\!\!\!\!\twoheadrightarrow_g e_k' \quad \theta \hookrightarrow\!\!\!\!\twoheadrightarrow_g \theta'}{\mathbf{case}\ v\ \mathbf{of}\ (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) \hookrightarrow\!\!\!\!\twoheadrightarrow_g e_k'[\theta']}$$

$$\frac{x \notin \mathrm{FV}(E) \quad e \hookrightarrow\!\!\!\!\twoheadrightarrow_g e' \quad E \hookrightarrow\!\!\!\!\twoheadrightarrow_g E'}{\mathbf{let}\ x = e\ \mathbf{in}\ E[x]\ \mathbf{end} \hookrightarrow\!\!\!\!\twoheadrightarrow_g E'[e']}$$

$$\frac{v \hookrightarrow\!\!\!\!\twoheadrightarrow_g v'}{\langle \mathbf{fst}(v), \mathbf{snd}(v)\rangle \hookrightarrow\!\!\!\!\twoheadrightarrow_g v'} \qquad \frac{v \hookrightarrow\!\!\!\!\twoheadrightarrow_g v'}{\mathbf{lam}\, x.\, v(x) \hookrightarrow\!\!\!\!\twoheadrightarrow_g v'}$$

$$\frac{}{[] \hookrightarrow\!\!\!\!\twoheadrightarrow_g []} \qquad \frac{E \hookrightarrow\!\!\!\!\twoheadrightarrow_g E'}{c(E) \hookrightarrow\!\!\!\!\twoheadrightarrow_g c(E')} \qquad \frac{E \hookrightarrow\!\!\!\!\twoheadrightarrow_g E' \quad e \hookrightarrow\!\!\!\!\twoheadrightarrow_g e'}{\langle E, e\rangle \hookrightarrow\!\!\!\!\twoheadrightarrow_g \langle E', e'\rangle} \qquad \frac{v \hookrightarrow\!\!\!\!\twoheadrightarrow_g v' \quad E \hookrightarrow\!\!\!\!\twoheadrightarrow_g E'}{\langle v, E\rangle \hookrightarrow\!\!\!\!\twoheadrightarrow_g \langle v', E'\rangle}$$

$$\frac{E \hookrightarrow\!\!\!\!\twoheadrightarrow_g E'}{\mathbf{fst}(E) \hookrightarrow\!\!\!\!\twoheadrightarrow_g \mathbf{fst}(E')} \qquad \frac{E \hookrightarrow\!\!\!\!\twoheadrightarrow_g E'}{\mathbf{snd}(E) \hookrightarrow\!\!\!\!\twoheadrightarrow_g \mathbf{snd}(E')} \qquad \frac{E \hookrightarrow\!\!\!\!\twoheadrightarrow_g E' \quad ms \hookrightarrow\!\!\!\!\twoheadrightarrow_g ms'}{\mathbf{case}\ E\ \mathbf{of}\ ms \hookrightarrow\!\!\!\!\twoheadrightarrow_g \mathbf{case}\ E'\ \mathbf{of}\ ms'}$$

$$\frac{E \hookrightarrow\!\!\!\!\twoheadrightarrow_g E' \quad e \hookrightarrow\!\!\!\!\twoheadrightarrow_g e'}{E(e) \hookrightarrow\!\!\!\!\twoheadrightarrow_g E'(e')} \qquad \frac{v \hookrightarrow\!\!\!\!\twoheadrightarrow_g v' \quad E \hookrightarrow\!\!\!\!\twoheadrightarrow_g E'}{v(E) \hookrightarrow\!\!\!\!\twoheadrightarrow_g v'(E')}$$

$$\frac{E \hookrightarrow\!\!\!\!\twoheadrightarrow_g E' \quad e \hookrightarrow\!\!\!\!\twoheadrightarrow_g e'}{\mathbf{let}\ x = E\ \mathbf{in}\ e\ \mathbf{end} \hookrightarrow\!\!\!\!\twoheadrightarrow_g \mathbf{let}\ x = E'\ \mathbf{in}\ e'\ \mathbf{end}}$$

$$\frac{e_1 \hookrightarrow\!\!\!\!\twoheadrightarrow_g e_1' \quad \cdots \quad e_n \hookrightarrow\!\!\!\!\twoheadrightarrow_g e_n'}{(p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) \hookrightarrow\!\!\!\!\twoheadrightarrow_g (p_1 \Rightarrow e_1' \mid \cdots \mid p_n \Rightarrow e_n')}$$

$$\frac{\theta(xf) \hookrightarrow\!\!\!\!\twoheadrightarrow_g \theta'(xf) \ \text{ for each } xf \text{ in } \mathbf{dom}(\theta) = \mathbf{dom}(\theta')}{\theta \hookrightarrow\!\!\!\!\twoheadrightarrow_g \theta'}$$

Fig. A 1. The rules for the parallel general reduction $\hookrightarrow\!\!\!\!\twoheadrightarrow_g$.

*Proof*
Straightforward. □

Note that if $e$ is in E-form and $e \hookrightarrow\!\!\!\!\twoheadrightarrow_g e'$ holds, then $e'$ is not necessarily in E-form. For instance, assume $e = \langle \mathbf{fst}(\mathbf{lam}\, x.\, x), \mathbf{snd}(\mathbf{lam}\, x.\, x)\rangle(\langle\rangle)$. Then $e$ is in E-form. Note that $e \hookrightarrow\!\!\!\!\twoheadrightarrow_g e'$ holds for $e' = (\mathbf{lam}\, x.\, x)(\langle\rangle)$, which is not in E-form ($e'$ is actually in R-form).

The essence of parallel general reduction is captured in the following proposition.

*Proposition A.3*

1. Assume $E \hookrightarrow\!\!\!\!\rightarrow_g E'$ and $e \hookrightarrow\!\!\!\!\rightarrow_g e'$ for some evaluation contexts $E, E'$ and expressions $e, e'$ in $\lambda_{pat}$. Then we have $E[e] \hookrightarrow\!\!\!\!\rightarrow_g E'[e']$.
2. Assume $e \hookrightarrow\!\!\!\!\rightarrow_g e'$ and $\theta \hookrightarrow\!\!\!\!\rightarrow_g \theta'$ for some expressions $e, e'$ and substitutions $\theta, \theta'$ in $\lambda_{pat}$. Then we have $e[\theta] \hookrightarrow\!\!\!\!\rightarrow_g e'[\theta']$.

*Proof*
(Sketch) By structural induction. $\square$

In the proof of Proposition A.3, it needs to be verified that for each evaluation context $E$ and $\theta$, $E[\theta]$, the evaluation context obtained from applying $\theta$ to $E$, is also an evaluation context. This follows from the fact that $\theta$ maps each **lam**-variable $x$ (treated as a value) in its domain to a value.

*Lemma A.4*
Assume that $\mathbf{match}(v, p) \Rightarrow \theta$ is derivable in $\lambda_{pat}$, where $v, p, \theta$ are a value, a pattern and a substitution, respectively. If $v \hookrightarrow\!\!\!\!\rightarrow_g v'$ holds for some value $v'$, then we can derive $\mathbf{match}(v', p) \Rightarrow \theta'$ for some $\theta'$ such that $\theta \hookrightarrow\!\!\!\!\rightarrow_g \theta'$ holds.

*Proof*
(Sketch) By structural induction on the derivation of $\mathbf{match}(v, p) \Rightarrow \theta$. $\square$

The following lemma is the key step to proving Lemma 2.14. Given two expressions $e, e'$, we write $e \hookrightarrow^{0/1}_{ev} e'$ to mean that $e = e'$ or $e \hookrightarrow_{ev} e'$, that is, $e$ ev-reduces to $e'$ in either 0 or 1 step.

*Lemma A.5*
Assume that $e_1$ and $e'_1$ are two expressions in $\lambda_{pat}$ such that $e_1 \hookrightarrow\!\!\!\!\rightarrow_g e'_1$ holds. If we have $e_1 \hookrightarrow_{ev} e_2$ for some $e_2$, then there exists $e'_2$ such that both $e'_1 \hookrightarrow^{0/1}_{ev} e'_2$ and $e_2 \hookrightarrow\!\!\!\!\rightarrow_g e'_2$ hold.

*Proof*
(Sketch) The proof proceeds by structural induction on the derivation $\mathcal{D}$ of $e_1 \hookrightarrow\!\!\!\!\rightarrow_g e'_1$, and we present a few interesting cases as follows.

- $\mathcal{D}$ is of the following form:

$$\frac{e_{10} \hookrightarrow\!\!\!\!\rightarrow_g e'_{10} \qquad ms \hookrightarrow\!\!\!\!\rightarrow_g ms'}{\mathbf{case}\ e_{10}\ \mathbf{of}\ ms \hookrightarrow\!\!\!\!\rightarrow_g \mathbf{case}\ e'_{10}\ \mathbf{of}\ ms'}$$

  where $e_1 = \mathbf{case}\ e_{10}\ \mathbf{of}\ ms$ and $e'_1 = \mathbf{case}\ e'_{10}\ \mathbf{of}\ ms'$. We have two subcases.

  — We have $e_{10} \hookrightarrow_{ev} e_{20}$ for some expression $e_{20}$ and $e_1 \hookrightarrow_{ev} e_2$, where $e_2 = \mathbf{case}\ e_{20}\ \mathbf{of}\ ms$. By induction hypothesis on the derivation of $e_{10} \hookrightarrow\!\!\!\!\rightarrow_g e'_{10}$, we can find an expression $e'_{20}$ such that both $e'_{10} \hookrightarrow^{0/1}_{ev} e'_{20}$ and $e_{20} \hookrightarrow\!\!\!\!\rightarrow_g e'_{20}$ hold. Let $e'_2$ be $\mathbf{case}\ e'_{20}\ \mathbf{of}\ ms'$, and we are done.

  — We have $e_1 \hookrightarrow_{ev} e_2 = e_{1k}[\theta]$, where $e_{10} = v$ for some value $v$, and $ms = (p_1 \Rightarrow e_{11} \mid \cdots \mid p_n \Rightarrow e_{1n})$ for some patterns $p_1, \ldots, p_n$ and expressions $e_{11}, \ldots, e_{1n}$, and $\mathbf{match}(v, p_k) \Rightarrow \theta$ is derivable. By Proposition A.2 (2), we know that $e'_{10}$ is in V-form (as $e_{10}$ is V-form). Let $v'$ be $e'_{10}$. By Lemma A.4, we have $\mathbf{match}(v', p_k) \Rightarrow \theta'$ for some substitution $\theta'$ such that $\theta \hookrightarrow\!\!\!\!\!\rightarrow_g \theta'$ holds. Note that $ms'$ is of the form $(p_1 \Rightarrow e'_{11} \mid \cdots \mid p_n \Rightarrow e'_{1n})$, where we have $e_{11} \hookrightarrow\!\!\!\!\!\rightarrow_g e'_{11}, \ldots, e_{1n} \hookrightarrow\!\!\!\!\!\rightarrow_g e'_{1n}$. Let $e'_2$ be $e'_{1k}[\theta']$. Clearly, we have $e'_1 \hookrightarrow_{ev} e'_2$. By Proposition A.3 (2), we also have $e_2 \hookrightarrow\!\!\!\!\!\rightarrow_g e'_2$.

- $\mathcal{D}$ is of the following form

$$\frac{x \notin \mathrm{FV}(E) \quad e_{10} \hookrightarrow\!\!\!\!\!\rightarrow_g e'_{10} \quad E \hookrightarrow\!\!\!\!\!\rightarrow_g E'}{\mathbf{let}\ x = e_{10}\ \mathbf{in}\ E[x]\ \mathbf{end} \hookrightarrow\!\!\!\!\!\rightarrow_g E'[e'_{10}]}$$

  where $e_1 = \mathbf{let}\ x = e_{10}\ \mathbf{in}\ E[x]\ \mathbf{end}$ and $e'_1 = E'[e'_{10}]$. We have two subcases.

  — We have $e_{10} \hookrightarrow_{ev} e_{20}$ and $e_1 \hookrightarrow_{ev} e_2 = \mathbf{let}\ x = e_{20}\ \mathbf{in}\ E[x]\ \mathbf{end}$. By induction hypothesis on the derivation of $e_{10} \hookrightarrow\!\!\!\!\!\rightarrow_g e'_{10}$, we can find an expression $e'_{20}$ such that both $e'_{10} \hookrightarrow^{0/1}_{ev} e'_{20}$ and $e_{20} \hookrightarrow\!\!\!\!\!\rightarrow_g e'_{20}$ hold. Let $e'_2$ be $E'[e'_{20}]$, and we are done.

  — We have $e_1 \hookrightarrow_{ev} e_2 = E[v]$, where $e_{10} = v$ for some value $v$. By Proposition A.2 (2), $e'_{10}$ is a value. Let $v'$ be $e'_{10}$ and $e'_2 = E'[v']$. Then we $e'_1 \hookrightarrow_{ev} e'_2$. By Proposition A.3 (1), we also have $e_2 \hookrightarrow\!\!\!\!\!\rightarrow_g e'_2$.

All other cases can be treated similarly. $\square$

*Lemma A.6*
Assume that $e \hookrightarrow\!\!\!\!\!\rightarrow_g e'$ holds for expressions $e, e'$ in $\lambda_{pat}$. If $e \hookrightarrow^*_{ev} v^*$ holds for some $v^*$ in **EMUV**, the union of **EMU** and the set of observable values, then $e' \hookrightarrow^*_{ev} v^*$ also holds.

*Proof*
The proof proceeds by induction on $n$, the number of steps in $e \hookrightarrow^*_{ev} v^*$.

- $n = 0$. This case immediately follows from Proposition A.2.
- $n > 0$. Then we have $e \hookrightarrow_{ev} e_1 \hookrightarrow^*_{ev} v^*$ for some expression $e_1$. By Lemma A.5, we have an expression $e'_1$ such that both $e' \hookrightarrow^{0/1}_{ev} e'_1$ and $e_1 \hookrightarrow\!\!\!\!\!\rightarrow_g e'_1$ hold. By induction hypothesis, $e'_1 \hookrightarrow^*_{ev} v^*$ holds, which implies $e' \hookrightarrow^*_{ev} v^*$.

$\square$

We are now ready to present the proof of Lemma 2.14.

*Proof*
(of Lemma 2.14) In order to prove $e' \leqslant_{dyn} e$, we need to show that for any context $G$, either $G[e'] \hookrightarrow^*_{ev}$ **Error**, or $G[e'] \hookrightarrow^*_{ev} v^*$ if and only if $G[e] \hookrightarrow^*_{ev} v^*$, where $v^*$ ranges over **EMUV**, that is, the union of **EMU** and the set of observable values.

Let $G$ be a context, and we have three possibilities.

- $G[e] \hookrightarrow^*_{ev}$ **Error** holds.
- $G[e] \hookrightarrow^*_{ev} v^*$ for some $v^*$ in **EMUV**. By Lemma A.6, we have $G[e'] \hookrightarrow^*_{ev} v^*$ since $G[e] \hookrightarrow\!\!\!\!\!\rightarrow_g G[e']$ holds.
- There exists an infinite evaluation reduction sequence from $G[e]$ :

$$G[e] = e_0 \hookrightarrow_{ev} e_1 \hookrightarrow_{ev} e_2 \hookrightarrow_{ev} \ldots$$

By Lemma A.5, we have the following evaluation reduction sequence:

$$G[e'] = e'_0 \hookrightarrow^{0/1}_{ev} e'_1 \hookrightarrow^{0/1}_{ev} e'_2 \hookrightarrow^{0/1}_{ev} \cdots$$

where $G[e'] = e'_0$ and $e_i \hookrightarrow\!\!\!\rightarrow_g e'_i$ for $i = 0, 1, 2, \ldots$. We now need to show that there exist infinitely many nonempty steps in the above evaluation sequence. This can be done by introducing a notion of residuals of g-redexes under ev-reduction, analogous to the notion of residuals of $\beta$-redex under $\beta$-reduction developed in the study of pure $\lambda$-calculus (Barendregt, 1984). The situation here is nearly identical to the one encountered in the proof of Conservation Theorem (Theorem 11.3.4 (Barendregt, 1984)), and we thus omit further routine but rather lengthy details.

After inspecting these three possibilities, we clearly see that this lemma holds. □

## B Proof of Theorem 4.11

*Proof*

Let $\mathscr{D}$ be the typing derivation of $\emptyset;\emptyset;\emptyset \vdash e_1 : \tau$. The proof proceeds by induction on the height of $\mathscr{D}$. Assume that the last applied rule in $\mathscr{D}$ is **(ty-sub)**. Then $\mathscr{D}$ is of the following form:

$$\frac{\mathscr{D}_1 :: \emptyset;\emptyset;\emptyset \vdash e_1 : \tau_1 \quad \emptyset;\emptyset \vdash \tau_1 \leqslant^s_{tp} \tau}{\emptyset;\emptyset;\emptyset \vdash e_1 : \tau} \ \textbf{(ty-sub)}$$

By induction hypothesis on $\mathscr{D}_1$, $\emptyset;\emptyset;\emptyset \vdash e_2 : \tau_1$ is derivable. Hence, $\emptyset;\emptyset;\emptyset \vdash e_2 : \tau$ is also derivable.

In the rest of the proof, we assume that the last applied rule in $\mathscr{D}$ is *not* **(ty-sub)**. Let $e_1 = E[e_0]$ and $e_2 = E[e'_0]$ for some evaluation context $E$, where $e_0$ is a redex and $e'_0$ is the reduct of $e'_0$. We proceed by analyzing the structure of $E$.

As an example, let us assume that $E$ is **let** $x = E_0$ **in** $e$ **end** for some evaluation context $E_0$ and expression $e$. Then $e_1$ is **let** $x = E_0[e_0]$ **in** $e$ **end** and the typing derivation $\mathscr{D}$ is of the following form:

$$\frac{\mathscr{D}_1 :: \emptyset;\emptyset;\emptyset \vdash E_0[e_0] : \tau_1 \quad \emptyset;\emptyset;\emptyset, x : \tau_1 \vdash e : \tau_2}{\emptyset;\emptyset;\emptyset \vdash \textbf{let } x = E_0[e_0] \textbf{ in } e \textbf{ end} : \tau_2} \ \textbf{(ty-let)}$$

where $\tau_2 = \tau$. By induction hypothesis on $\mathscr{D}_1$, we can derive $\emptyset;\emptyset;\emptyset \vdash E_0[e'_0] : \tau_1$. Hence, we can also derive $\emptyset;\emptyset;\emptyset \vdash$ **let** $x = E_0[e'_0]$ **in** $e$ **end** $: \tau_2$. Note that $e_2$ is **let** $x = E_0[e'_0]$ **in** $e$ **end**, and we are done.

We skip all other cases except the most interesting one where $E = []$, that is, $e_1$ is a redex and $e_2$ is the reduct of $e_1$. In this case, we proceed by inspecting the structure of $\mathscr{D}$.

- $e_1 = \textbf{fst}(\langle v_1, v_2\rangle)$ and $e_2 = v_1$. Then $\mathscr{D}$ is of the following form:

$$\frac{\mathscr{D}_1 :: \emptyset;\emptyset;\emptyset \vdash \langle v_1, v_2\rangle : \tau_1 * \tau_2}{\emptyset;\emptyset;\emptyset \vdash \textbf{fst}(\langle v_1, v_2\rangle) : \tau_1} \ \textbf{(ty-fst)}$$

where $\tau = \tau_1$. By Lemma 4.6, we may assume that the last rule applied in $\mathscr{D}_1$ is not **(ty-sub)**. Hence, $\mathscr{D}_1$ is of the following form:

$$\frac{\emptyset;\emptyset;\emptyset \vdash v_1 : \tau_1 \quad \emptyset;\emptyset;\emptyset \vdash v_2 : \tau_2}{\emptyset;\emptyset;\emptyset \vdash \langle v_1, v_2 \rangle : \tau_1 * \tau_2} \textbf{(ty-prod)}$$

and therefore $\emptyset;\emptyset;\emptyset \vdash e_2 : \tau_1$ is derivable as $e_2 = v_1$.

- $e_1 = \textbf{snd}(\langle v_1, v_2 \rangle)$ and $e_2 = v_2$. This case is symmetric to the previous one.
- $e_1 = (\textbf{lam}\, x.\, e)(v)$ and $e_2 = e[x \mapsto v]$. Then $\mathscr{D}$ is of the following form:

$$\frac{\mathscr{D}_1 :: \emptyset;\emptyset;\emptyset \vdash \textbf{lam}\, x.\, e : \tau_1 \rightarrow \tau_2 \quad \emptyset;\emptyset;\emptyset \vdash v : \tau_1}{\emptyset;\emptyset;\emptyset \vdash (\textbf{lam}\, x.\, e)(v) : \tau_2} \textbf{(ty-app)}$$

where $\tau = \tau_2$. By Lemma 4.6, we may assume that the last rule applied in $\mathscr{D}_1$ is not **(ty-sub)**. Hence, $\mathscr{D}_1$ is of the following form

$$\frac{\emptyset;\emptyset;\emptyset, x : \tau_1 \vdash e : \tau_2}{\emptyset;\emptyset;\emptyset \vdash \textbf{lam}\, x.\, e : \tau_1 \rightarrow \tau_2} \textbf{(ty-lam)}$$

By Lemma 4.7 (3), we know that the typing judgment $\emptyset;\emptyset;\emptyset \vdash e[x \mapsto v] : \tau_2$ is derivable.

- $e_1 = \textbf{case}\ v\ \textbf{of}\ ms$ and $e_2 = e[\theta]$ for some clause $p \Rightarrow e$ in $ms$ such that $\textbf{match}(v, p) \Rightarrow \theta$ is derivable. Let $\mathscr{D}_1, \mathscr{D}_2, \mathscr{D}_3$ be derivations of $\emptyset;\emptyset;\emptyset \vdash v : \tau_1$, $p \downarrow \tau_1 \Rightarrow (\phi_0; \vec{P}_0; \Gamma_0)$, and $\phi_0; \vec{P}_0; \Gamma_0 \vdash e : \tau_2$, respectively, where $\tau = \tau_2$. By Lemma 4.10, we have a substitution $\Theta$ satisfying $\emptyset \vdash \Theta : \phi_0$ such that both $\emptyset \models \vec{P}_0[\Theta]$ and $\emptyset \vdash \theta : \Gamma_0[\Theta]$ hold. By Lemma 4.7 (1), we know that $\emptyset; \vec{P}_0[\Theta]; \Gamma_0[\Theta] \vdash e : \tau_2$ is derivable as $\tau_2$ contains no free occurrences of the index variables declared in $\phi_0$. By Lemma 4.7 (2), we know that $\emptyset;\emptyset;\Gamma_0[\Theta] \vdash e : \tau_2$ is derivable. By Lemma 4.7 (3), we know that $\emptyset;\emptyset;\emptyset \vdash e[\theta] : \tau_2$ is derivable.
- $e_1 = \supset^-(\supset^+(v))$ for some value $v$. Then $\mathscr{D}$ is of the following form:

$$\frac{\mathscr{D}_1 :: \emptyset;\emptyset;\emptyset \vdash \supset^+(v) : P \supset \tau \quad \emptyset \models P}{\emptyset;\emptyset;\emptyset \vdash \supset^-(\supset^+(v)) : \tau} \textbf{(ty-}\supset\textbf{-elim)}$$

By Lemma 4.6, we may assume that the last rule applied in $\mathscr{D}_1$ is not **(ty-sub)**. Hence, $\mathscr{D}_1$ is of the following form:

$$\frac{\mathscr{D}_2 :: \emptyset; P; \emptyset \vdash v : \tau}{\emptyset;\emptyset;\emptyset \vdash \supset^+(v) : P \supset \tau} \textbf{(ty-}\supset\textbf{-intro)}$$

By Lemma 4.7 (2), the typing judgment $\emptyset;\emptyset;\emptyset \vdash v : \tau$ is derivable. Note that $e_2 = v$, and we are done.

- $e_1 = \Pi^-(\Pi^+(v))$ for some value $v$. Then $\mathscr{D}$ is of the following form:

$$\frac{\mathscr{D}_1 :: \emptyset;\emptyset;\emptyset \vdash \Pi^+(v) : \Pi a{:}s.\, \tau_0 \quad \emptyset \vdash I : s}{\emptyset;\emptyset;\emptyset \vdash \Pi^-(\Pi^+(v)) : \tau_0[a \mapsto I]} \textbf{(ty-}\Pi\textbf{-elim)}$$

where $\tau = \tau_0[a \mapsto I]$. By Lemma 4.6, we may assume that the last rule applied in $\mathscr{D}_1$ is not **(ty-sub)**. Hence, $\mathscr{D}_1$ is of the following form:

$$\frac{\mathscr{D}_2 :: \emptyset, a : s; \emptyset; \emptyset \vdash v : \tau_0}{\emptyset; \emptyset; \emptyset \vdash \Pi^+(v) : \Pi a{:}s.\ \tau_0}\ \textbf{(ty-}\Pi\textbf{-intro)}$$

By Lemma 4.7 (1), the typing judgment $\emptyset; \emptyset; \emptyset \vdash v : \tau_0[a \mapsto I]$ is derivable. Note that $e_2 = v$, and we are done.

- $e_1 = \textbf{let}\ \wedge(x) = \wedge(v)\ \textbf{in}\ e\ \textbf{end}$ for some value $v$ and expression $e$. Then $\mathscr{D}$ is of the following form:

$$\frac{\mathscr{D}_1 :: \emptyset; \emptyset; \emptyset \vdash \wedge(v) : P \wedge \tau_1 \quad \mathscr{D}_2 :: \emptyset; P; \emptyset, x : \tau_1 \vdash e : \tau_2}{\emptyset; \emptyset; \emptyset \vdash \textbf{let}\ \wedge(x) = v\ \textbf{in}\ e\ \textbf{end} : \tau_2}\ \textbf{(ty-}\wedge\textbf{-elim)}$$

  where $\tau = \tau_2$. By Lemma 4.6, we may assume that the last rule applied in $\mathscr{D}_1$ is not **(ty-sub)**. Hence, $\mathscr{D}_1$ is of the following form:

$$\frac{\mathscr{D}_3 :: \emptyset; \emptyset; \emptyset \vdash v : \tau_1 \quad \emptyset \models P}{\emptyset; \emptyset; \emptyset \vdash \wedge(v) : P \wedge \tau_1}\ \textbf{(ty-}\wedge\textbf{-intro)}$$

  By Lemma 4.7 (2), $\emptyset; \emptyset; \emptyset, x : \tau_1 \vdash e : \tau_2$ is derivable, and by Lemma 4.7 (3), $\emptyset; \emptyset; \emptyset \vdash e[x \mapsto v] : \tau_2$ is also derivable. Note that $e_2 = e[x \mapsto v]$, and we are done.

- $e_1 = \textbf{let}\ \Sigma(x) = \Sigma(v)\ \textbf{in}\ e\ \textbf{end}$ for some value $v$ and expression $e$. Then $\mathscr{D}$ is of the following form:

$$\frac{\mathscr{D}_1 :: \emptyset; \emptyset; \emptyset \vdash \Sigma(v) : \Sigma a{:}s.\ \tau_1 \quad \mathscr{D}_2 :: \emptyset, a : s; \emptyset; \emptyset, x : \tau_1 \vdash e : \tau_2}{\emptyset; \emptyset; \emptyset \vdash \textbf{let}\ \Sigma(x) = v\ \textbf{in}\ e\ \textbf{end} : \tau_2}\ \textbf{(ty-}\Sigma\textbf{-elim)}$$

  where $\tau = \tau_2$. By Lemma 4.6, we may assume that the last rule applied in $\mathscr{D}_1$ is not **(ty-sub)**. Hence, $\mathscr{D}_1$ is of the following form:

$$\frac{\mathscr{D}_3 :: \emptyset; \emptyset; \emptyset \vdash v :: \tau_1[a \mapsto I] \quad \emptyset \vdash I : s}{\emptyset; \emptyset; \emptyset \vdash \Sigma(v) : \Sigma a{:}s.\ \tau_1}\ \textbf{(ty-}\Sigma\textbf{-intro)}$$

  By Lemma 4.7 (1), $\emptyset; \emptyset; \emptyset, x : \tau_1[a \mapsto I] \vdash e : \tau_2$ is derivable as $\tau_2$ contains no free occurrence of $a$. Then by Lemma 4.7 (3), $\emptyset; \emptyset; \emptyset \vdash e[x \mapsto v] : \tau_2$ is also derivable. Note that $e_2 = e[x \mapsto v]$, and we are done.

We thus conclude the proof of Theorem 4.11. $\square$

## C Proof Sketch of Theorem 6.8

We outline in this section a proof of Theorem 6.8. Though we see no fundamental difficulty in handling exceptions, we will not attempt to do it here as this would significantly complicate the presentation of the proof.

We first state some basic properties about typing derivations in $\lambda_{pat}^{\Pi,\Sigma}$ extended with references.

*Proposition C.1*
Assume that $\mathscr{D} :: \phi; \vec{P}; \Gamma \vdash_\mu e : \sigma$ is derivable and there is no free occurrence of $\alpha$ in either $\Gamma$ or $\mu$. Then there is derivation of $\mathscr{D}'$ of $\phi; \vec{P}; \Gamma \vdash_\mu e : \sigma[\alpha \mapsto \tau]$ such that $height(\mathscr{D}) = height(\mathscr{D}')$ holds.

*Proof*
(Sketch) By induction on the height of $\mathscr{D}$. □

*Proposition C.2*
Assume that $\mathscr{D}_1 : \phi; \vec{P}; \Gamma \vdash_{\mu_1} e : \sigma$ is derivable and $\mu_2$ extends $\mu_1$. Then there is a derivation $\mathscr{D}_2$ of $\phi; \vec{P}; \Gamma \vdash_{\mu_2} e : \sigma$ such that $height(\mathscr{D}_1) = height(\mathscr{D}_2)$ holds.

*Proof*
(Sketch) The proof proceeds by induction on the height of $\mathscr{D}_1$. We present the only interesting case in this proof, where $\sigma = \forall \vec{\alpha}.\ \tau$ for some type $\tau$ and $\mathscr{D}_1$ is of the following form:

$$\frac{\mathscr{D}_{10} :: \phi; \vec{P}; \Gamma \vdash_{\mu_1} e : \tau \quad \vec{\alpha} \# \Gamma \quad \vec{\alpha} \# \mu_1 \quad e \text{ is value-equivalent}}{\phi; \vec{P}; \Gamma \vdash_{\mu_1} e : \forall \vec{\alpha}.\ \tau} \textbf{(ty-poly)}$$

Let us choose $\vec{\alpha}'$ such that there is no $\alpha$ in $\vec{\alpha}'$ that has any free occurrences in $\Gamma$, $\tau$ or $\mu_2$. Applying Proposition C.1 (repeatedly if needed), we can obtain a derivation $\mathscr{D}'_{10}$ of $\phi; \vec{P}; \Gamma \vdash_{\mu_1} e : \tau[\vec{\alpha} \mapsto \vec{\alpha}']$ such that $height(\mathscr{D}_{10}) = height(\mathscr{D}'_{10})$. By induction hypothesis, we have a derivation $\mathscr{D}'_{20}$ of $\phi; \vec{P}; \Gamma \vdash_{\mu_2} e : \tau[\vec{\alpha} \mapsto \vec{\alpha}']$ such that $height(\mathscr{D}'_1) = height(\mathscr{D}'_2)$. Let $\mathscr{D}_2$ be the following derivation:

$$\frac{\mathscr{D}'_{20} :: \phi; \vec{P}; \Gamma \vdash_{\mu_2} e : \tau[\vec{\alpha} \mapsto \vec{\alpha}'] \quad \vec{\alpha}' \# \Gamma \quad \vec{\alpha}' \# \mu_2 \quad e \text{ is value-equivalent}}{\phi; \vec{P}; \Gamma \vdash_{\mu_2} e : \forall \vec{\alpha}'.\ \tau[\vec{\alpha} \mapsto \vec{\alpha}']} \textbf{(ty-poly)}$$

Note that $\sigma = \forall \vec{\alpha}'.\ \tau[\vec{\alpha} \mapsto \vec{\alpha}']$, and we are done. □

The following lemma states that evaluation not involving references is type-preserving.

*Lemma C.3*
Assume that $\phi; \vec{P}; \emptyset \vdash_\mu e_1 : \sigma$ is derivable. If $e_1 \hookrightarrow_{ev} e_2$ holds, then $\phi; \vec{P}; \emptyset \vdash_\mu e_2 : \sigma$ is also derivable.

*Proof*
(Sketch) This proof can be handled in precisely the same manner as the proof of Theorem 4.11 in Appendix B. □

Lemma C.3 can actually be strengthened to state that evaluation not involving reference creation is type-preserving.

We are now ready to prove Theorem 6.8.

*Proof*
(of Theorem 6.8) (Sketch) We have the following four possibilities according to the definition of $\hookrightarrow_{ev/st}$.

- $e_1 \hookrightarrow_{ev} e_2$. This case follows from Lemma C.3 immediately.

- $e_1 = E[\mathit{ref}(v)]$ for some evaluation context $E$ and value $v$. This case is handled by analyzing the structure of $E$. Obviously, $e_1$ is not value-equivalent since $e_1 \hookrightarrow^*_{ev} v$ does not hold for any value. This means that $E$ cannot be of either the form $\supset^+(E_1)$ or the form $\Pi^+(E_1)$. We encourage the reader to figure out what would happen if these two forms of evaluation contexts were not ruled out. Among the rest of the cases, the only interesting one is where $E$ is $[]$, that is, $e_1 = \mathit{ref}(v)$. In this case, we know that $\sigma$ cannot be a type scheme (since $e_1$ is not value-equivalent). Hence, $\sigma$ is of the form $(\tau)\mathbf{ref}$ for some type $\tau$ and $\emptyset;\emptyset;\emptyset \vdash_{\mu_1} v : \tau$ is derivable. Also, we have $M_2 = M_1[l \mapsto v]$ for some reference constant $l$ not in the domain of $M_1$ and $e_2 = l$. Let $\mu_2$ be $\mu_1[l \mapsto \tau]$, and we have $M_2 : \mu_2$. Clearly, $\emptyset;\emptyset;\emptyset \vdash_{\mu_2} e_2 : (\tau)\mathbf{ref}$ is derivable.
- $e_1 = E[!l]$ for some evaluation context $E$ and reference constant $l$. This case can be handled like the previous one.
- $e_1 = E[l := v]$ for some evaluation context $E$, reference constant $l$ and value $v$. This case can handled like the previous one.

□

In order to fully appreciate the notion of value restriction, it is probably helpful to see what can happen if there is no value restriction. Assume that the constructor *nil* is given the c-type $\forall\alpha.\ \mathbf{1} \Rightarrow (\alpha)\mathbf{list}$. Clearly, we have a derivation $\mathcal{D}$ of the following judgment:

$$\emptyset;\emptyset;\emptyset \vdash_{[]} \mathit{ref}(\mathit{nil}) : ((\alpha)\mathbf{list})\mathbf{ref}$$

where $\alpha$ is some type variable. With no value restriction, the following derivation can be constructed

$$\frac{\mathcal{D} :: \emptyset;\emptyset;\emptyset \vdash_{[]} \mathit{ref}(\mathit{nil}) : ((\alpha)\mathbf{list})\mathbf{ref}}{\emptyset;\emptyset;\emptyset \vdash_{[]} \mathit{ref}(\mathit{nil}) : \forall\alpha.((\alpha)\mathbf{list})\mathbf{ref}}$$

Certainly, we have $([], \mathit{ref}(\mathit{nil})) \hookrightarrow_{ev/st} ([l \mapsto \mathit{nil}], l)$ for any reference constant $l$. However, there is simply no store type $\mu$ such that $[l \mapsto \mathit{nil}] : \mu$ holds and $\emptyset;\emptyset;\emptyset \vdash_\mu l : \forall\alpha.\ ((\alpha)\mathbf{list})\mathbf{ref}$ is also derivable. For instance, let us choose $\mu$ to be $[l \mapsto (\alpha)\mathbf{list}]$. Then we can derive $\emptyset;\emptyset;\emptyset \vdash_\mu l : ((\alpha)\mathbf{list})\mathbf{ref}$, but this does not lead to a derivation of $\emptyset;\emptyset;\emptyset \vdash_\mu l : \forall\alpha.\ ((\alpha)\mathbf{list})\mathbf{ref}$ as $\alpha \# \mu$ does not hold, that is, $\alpha$ does have a free occurrence in $\mu$. Hence, without value restriction, the theorem of subject reduction can no longer be established.

## References

Andrews, P. B. (1972) General Models, Descriptions and Choice in Type Theory. *Journal of Symbolic Logic*, **37**, 385–394.

Andrews, P. B. (1986) *An Introduction to Mathematical Logic: To Truth through Proof.* Orlando, Florida: Academic Press.

Augustsson, L. (1998) Cayenne – a language with dependent types. *Pages 239–250 of: Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming.*

Barendregt, H. P. (1984) *The lambda calculus, its syntax and semantics.* Revised edition edn. Studies in Logic and the Foundations of Mathematics, vol. 103. Amsterdam: North-Holland.

Barendregt, H. P. (1992) Lambda Calculi with Types. *Pages 117–441 of:* Abramsky, S., Gabbay, Dov M., & Maibaum, T.S.E. (eds), *Handbook of Logic in Computer Science*, vol. II. Oxford: Clarendon Press.

Chen, C. & Xi, H. (2003) Implementing Typeful Program Transformations. *Pages 20–28 of: Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Semantics based Program Manipulation.*

Chen, C. & Xi, H. (2005a) Combining Programming with Theorem Proving. *Pages 66–77 of: Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming.*

Chen, C. & Xi, H. (2005b) Meta-Programming through Typeful Code Representation. *Journal of Functional Programming*, **15**(6), 1–39.

Chen, C., Shi, R. & Xi, H. (2005) Implementing Typeful Program Transformations. *Fundamenta informaticae*, **69**(1–2), 103–121.

Cheney, J. & Hinze, R. (2003) *Phantom Types.* Technical Report CUCIS-TR2003-1901. Cornell University. Available at `http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cis/TR2003-1901`.

Church, A. (1940) A formulation of the simple type theory of types. *Journal of Symbolic Logic*, **5**, 56–68.

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

Dantzig, G. B. & Eaves, B. C. (1973) Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory (A)*, **14**, 288–297.

Danvy, O. (1998) Functional unparsing. *Journal of Functional Programming*, **8**(6), 621–625.

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

Dunfield, J. (2002) *Combining two forms of type refinement.* Tech. rept. CMU-CS-02-182. Carnegie Mellon University.

Dunfield, J. & Pfenning, F. (2003) Type assignment for intersections and unions in call-by-value languages. *Pages 250–266 of:* Gordon, A. D. (ed), *Proceedings of the 6th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'03).* Warsaw, Poland: Springer-Verlag LNCS 2620.

Dunfield, J. & Pfenning, F. (2004) Tridirectional typechecking. *Pages 281–292 of:* Leroy, X. (ed), *Conference Record of the 31st Annual Symposium on Principles of Programming Languages (POPL'04).* Venice, Italy: ACM Press. Extended version available as Technical Report CMU-CS-04-117, March 2004.

Freeman, T. & Pfenning, F. (1991) Refinement types for ML. *Pages 268–277 of: ACM SIGPLAN Conference on Programming Language Design and Implementation.*

Girard, J.-Y. (1972) *Interprétation fonctionnelle et Élimination des coupures dans l'arithmétique d'ordre supérieur.* Thèse de doctorat d'état, Université de Paris VII, Paris, France.

Griffin, T. (1990) A Formulae-as-Types Notion of Control. *Pages 47–58 of: Conference Record of POPL '90: 17th ACM SIGPLAN Symposium on Principles of Programming Languages.*

Harper, R. (1994) A simplified account of polymorphic references. *Information Processing Letters*, **51**, 201–206.

Hayashi, S. & Nakano, H. (1988) *PX: A computational logic.* The MIT Press.

Henkin, L. (1950) Completeness in the theory of types. *Journal of Symbolic Logic*, **15**, 81–91.

Hinze, R. (2001) Manufacturing Datatypes. *Journal of Functional Programming*, **11**(5), 493–524.

Hughes, J., Pareto, L. & Sabry, A. (1996) Proving the Correctness of Reactive Systems Using Sized Types. *Pages 410–423 of: Proceeding of 23rd Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '96).*

INRIA. *Objective Caml.* `http://caml.inria.fr`.

Jay, C. B. & Sekanina, M. (1996) *Shape checking of array programs.* Tech. rept. 96.09. University of Technology, Sydney, Australia.

Jones, S. P., Vytiniotis, D., Weirich, S. & Washburn, G. (2005) *Simple Unification-based Type Inference for GADTS.*

Kahrs, S. (2001) Red-black trees with types. *Journal of Functional Programming*, **11**(4), 425–432.

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

Martin-Löf, P. (1984) *Intuitionistic type theory.* Naples, Italy: Bibliopolis.

Martin-Löf, P. (1985) Constructive mathematics and computer programming. Hoare, C. R. A. (ed), *Mathematical logic and programming languages.* Prentice-Hall.

McBride, C. *Epigram.* Available at: `http://www.dur.ac.uk/CARG/epigram`.

Meyer, A. & Wand, M. (1985) Continuation Semantics in Typed Lambda Calculi (summary). *Pages 219–224 of:* Parikh, R. (ed), *Logics of Programs.* Springer-Verlag LNCS 224.

Michaylov, S. (1992) *Design and implementation of practical constraint logic programming systems.* Ph.D. thesis, Carnegie Mellon University. Available as Technical Report CMU-CS-92-168.

Milner, R., Tofte, M., Harper, R. W. & MacQueen, D. (1997) *The Definition of Standard ML (revised).* Cambridge, Massachusetts: MIT Press.

Mitchell, J. C. & Plotkin, G. D. (1988) Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, **10**(3), 470–502.

Mitchell, J. C. & Scott, P. J. (1989) Typed lambda models and cartesian closed categories (preliminary version). *Pages 301–316 of:* Gray, J. W. & Scedrov, A. (eds), *Categories in Computer Science and Logic.* Contemporary Mathematics, vol. 92. Boulder, Colorado: American Mathematical Society.

Odersky, M., Zenger, C. & Zenger, M. (2001) Colored Local Type Inference. *Pages 41–53 of: Proceedings of the 28th Annual ACM SIGPPLAN-SIGACT Symposium on Principles of Programming Languages.*

Okasaki, C. (1998) *Purely Functional Data Structures.* Cambridge University Press.

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

Parent, C. (1995) Synthesizing proofs from programs in the calculus of inductive constructions. *Pages 351–379 of: Proceedings of the International Conference on Mathematics for Programs Constructions.* Springer-Verlag LNCS 947.

Peyton Jones, S. *et al.*. (1999) *Haskell 98 – A non-strict, purely functional language.* Available at `http://www.haskell.org/onlinereport/`.

Pfenning, F. *Computation and Deduction.* Cambridge University Press. (To appear).

Pfenning, F. & Elliott, C. (1988) Higher-order abstract syntax. *Pages 199–208 of: Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation.*

Pierce, B. & Turner, D. (1998) Local type inference. *Pages 252–265 of: Proceedings of 25th Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '98).*

Pottier, F. & Régis-Gianas, Y. (2006) Stratified type inference for generalized algebraic data types. *Pages 232–244 of: Proceedings of the 33rd ACM Symposium on Principles of Programming Languages (POPL'06).*

Pugh, W. & Wonnacott, D. (1992) Eliminating false data dependences using the Omega test. *Pages 140–151 of: ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.* ACM Press.

Pugh, W. & Wonnacott, D. (1994) *Experience with constraint-based array dependence analysis.* Tech. rept. CS-TR-3371. University of Maryland.

Sannella, D. & Tarlecki, A. (1989) *Toward formal development of ML programs: Foundations and methodology.* Tech. rept. ECS-LFCS-89-71. Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh.

Sheard, T. (2004) Languages of the future. *Proceedings of the Onward! Track of Objected-oriented Programming Systems, Languages, Applications (OOPSLA).* Vancouver, BC: ACM Press.

Shostak, R. E. (1977) On the SUP-INF method for proving Presburger formulas. *Journal of the ACM*, **24**(4), 529–543.

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

Takahashi, M. (1995) Parallel Reduction. *Information & Computation*, **118**, 120–127.

Westbrook, E., Stump, A. & Wehrman, I. (2005) A Language-Based Approach to Functionally Correct Imperative Programming. *Pages 268–279 of: Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming.*

Wright, A. (1995) Simple imperative polymorphism. *Journal of Lisp and Symbolic Computation*, **8**(4), 343–355.

Xi, H. (1998) *Dependent types in practical programming.* Ph.D. thesis, Carnegie Mellon University. pp. viii+189. Available at `http://www.cs.cmu.edu/~hwxi/DML/thesis.ps`.

Xi, H. (1999) Dependently Typed Data Structures. *Pages 17–33 of: Proceedings of Workshop on Algorithmic Aspects of Advanced Programming Languages.*

Xi, Hongwei. (2003). Dependently Typed Pattern Matching. *Journal of Universal Computer Science*, **9**(8), 851–872.

Xi, H. (2004) Applied Type System (extended abstract). *Pages 394–408 of: Post-Workshop Proceedings of Types 2003.* Springer-Verlag LNCS 3085.

Xi, H. (2005) *Applied Type System.* Available at: `http://www.cs.bu.edu/~hwxi/ATS`.

Xi, H. & Pfenning, F. (1998) Eliminating array bound checking through dependent types. *Pages 249–257 of: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation.*

Xi, H. & Pfenning, F. (1999) Dependent Types in Practical Programming. *Pages 214–227 of: Proceedings of 26th ACM SIGPLAN Symposium on Principles of Programming Languages.* San Antonio, Texas: ACM press.

Xi, H., Chen, C. & Chen, G. (2003) Guarded Recursive Datatype Constructors. *Pages 224–235 of: Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages.* New Orleans, LA: ACM press.

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

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