

base types	$\delta ::= \mathbf{bool} \mid \mathbf{int} \mid \dots$
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.	$ms ::= (p_1 \Rightarrow e_1 \mid \dots \mid p_n \Rightarrow e_n)$
constants	$c ::= cc \mid cf$
expressions	$e ::= xf \mid c(e) \mid \langle \rangle \mid \langle e_1, e_2 \rangle \mid \mathbf{fst}(e) \mid \mathbf{snd}(e) \mid \mathbf{case} \ e \ \mathbf{of} \ ms \mid$ $\mathbf{lam} \ x. e \mid e_1(e_2) \mid \mathbf{fix} \ f. e \mid \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 \ \mathbf{end}$
values	$v ::= x \mid cc(v) \mid \langle \rangle \mid \langle v_1, v_2 \rangle \mid \mathbf{lam} \ x. e$
contexts	$\Gamma ::= \cdot \mid \Gamma, xf : \tau$
substitutions	$\theta ::= [] \mid \theta[x \mapsto v] \mid \theta[f \mapsto e]$

Fig. 3. The syntax for λ_{pat}

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

2 λ_{pat} : A starting point

We introduce a simply typed programming language λ_{pat} , which essentially extends the simply typed λ -calculus with pattern matching. We emphasize that there are no new contributions in this section. Instead, we primarily use λ_{pat} as an example to show how a type system is developed. In particular, we show how various properties of λ_{pat} are chained together in order to establish the type soundness of λ_{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 λ_{pat} . Except Lemma 2.14, all the results in this section are well-known and thus their proofs are omitted.

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

A \mathbf{lam} -bound variable is considered a value but a \mathbf{fix} -bound variable is not. We use the name *observable value* for a closed value that does not contain a lambda expression $\mathbf{lam} \ x. e$ 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 $\mathbf{1} \Rightarrow \mathbf{bool}$ and every integer i is assigned the type $\mathbf{1} \Rightarrow \mathbf{int}$.

Note that we do not treat the tuple constructor $\langle \cdot, \cdot \rangle$ as a special case of constructors. Instead, we introduce tuples into λ_{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.

$$\begin{array}{c}
\frac{}{x \downarrow \tau \Rightarrow x : \tau} \text{ (pat-var)} \\
\frac{}{\langle \rangle \downarrow \mathbf{1} \Rightarrow \emptyset} \text{ (pat-unit)} \\
\frac{p_1 \downarrow \tau_1 \Rightarrow \Gamma_1 \quad p_2 \downarrow \tau_2 \Rightarrow \Gamma_2}{\langle p_1, p_2 \rangle \downarrow \tau_1 * \tau_2 \Rightarrow \Gamma_1, \Gamma_2} \text{ (pat-prod)} \\
\frac{\vdash cc(\tau) : \delta \quad p \downarrow \tau \Rightarrow \Gamma}{cc(p) \downarrow \delta \Rightarrow \Gamma} \text{ (pat-const)}
\end{array}$$

Fig. 4. The typing rules for patterns in λ_{pat}

We use θ 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 \emptyset for the empty substitution and $\theta[xf \mapsto e]$ for the substitution that extends θ with a link from x to e , where it is assumed that x is not in the domain $\mathbf{dom}(\theta)$ of θ . Also, we may write $[x_1 \mapsto e_1, \dots, x_n \mapsto e_n]$ for a substitution that maps x_i to e_i for $1 \leq i \leq n$. 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 θ to \bullet .

We use \emptyset for the empty context and $\Gamma, xf : \tau$ for the context that extends Γ with one additional declaration $xf : \tau$, where we assume that xf is not already declared in Γ . A context $\Gamma = \emptyset, x_1 : \tau_1, \dots, x_n : \tau_n$ may also be treated as a finite mapping that maps x_i to τ_i for $1 \leq i \leq n$, and we use $\mathbf{dom}(\Gamma)$ for the domain of Γ . Also, we may use Γ, Γ' for the context $\emptyset, x_1 : \tau_1, \dots, x_n : \tau_n, x'_1 : \tau'_1, \dots, x'_n : \tau'_n$, where $\Gamma = \emptyset, x_1 : \tau_1, \dots, x_n : \tau_n$ and $\Gamma' = \emptyset, x'_1 : \tau'_1, \dots, x'_n : \tau'_n$ and all variables $x_1, \dots, x_n, x'_1, \dots, x'_n$ 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:

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

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 τ , 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 τ yields a context Γ . 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 $\mathbf{1} \Rightarrow \mathbf{intlist}$ and $\mathbf{int} * \mathbf{intlist} \Rightarrow \mathbf{intlist}$, respectively; then the following judgments

$$\begin{array}{c}
\frac{\Gamma(xf) = \tau}{\Gamma \vdash xf : \tau} \text{ (ty-var)} \\
\frac{\vdash c(\tau) : \delta \quad \Gamma \vdash e : \tau}{\Gamma \vdash c(e) : \delta} \text{ (ty-const)} \\
\frac{}{\Gamma \vdash \langle \rangle : \mathbf{1}} \text{ (ty-unit)} \\
\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 * \tau_2} \text{ (ty-prod)} \\
\frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash \mathbf{fst}(e) : \tau_1} \text{ (ty-fst)} \\
\frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash \mathbf{snd}(e) : \tau_2} \text{ (ty-snd)} \\
\frac{p \downarrow \tau_1 \Rightarrow \Gamma_1 \quad \Gamma, \Gamma_1 \vdash e : \tau_2}{\Gamma \vdash p \Rightarrow e : \tau_1 \rightarrow \tau_2} \text{ (ty-clause)} \\
\frac{\Gamma \vdash p_i \Rightarrow e_i : \tau_1 \rightarrow \tau_2 \text{ for } i = 1, \dots, n}{\Gamma \vdash (p_1 \Rightarrow e_1 \mid \dots \mid p_n \Rightarrow e_n) : \tau_1 \rightarrow \tau_2} \text{ (ty-clause-seq)} \\
\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash ms : \tau_1 \rightarrow \tau_2}{\Gamma \vdash \mathbf{case } e \text{ of } ms : \tau_2} \text{ (ty-case)} \\
\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathbf{lam } x. e : \tau_1 \rightarrow \tau_2} \text{ (ty-lam)} \\
\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1(e_2) : \tau_2} \text{ (ty-app)} \\
\frac{\Gamma, f : \tau \vdash e : \tau}{\Gamma \vdash \mathbf{fix } f. e : \tau} \text{ (ty-fix)} \\
\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{let } x = e_1 \text{ in } e_2 \text{ end} : \tau_2} \text{ (ty-let)}
\end{array}$$

Fig. 5. The typing rules for expressions in λ_{pat}

are derivable:

$$\begin{array}{l}
\mathit{cons}(\langle x, xs \rangle) \downarrow \mathbf{intlist} \Rightarrow x : \mathbf{int}, xs : \mathbf{intlist} \\
\mathit{cons}(\langle x, \mathit{nil}(\langle \rangle) \rangle) \downarrow \mathbf{intlist} \Rightarrow x : \mathbf{int}
\end{array}$$

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 τ_2 under the assumption that p is given the type τ_1 .

In the following presentation, given some form of judgment J , we use $\mathcal{D} :: J$ for a derivation of J . The structure of a derivation \mathcal{D} is a tree, and we use $\mathit{height}(\mathcal{D})$ 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 λ_{pat} .

Lemma 2.1 (Thinning)

Assume $\mathcal{D} :: \Gamma \vdash e : \tau$. Then there is a derivation $\mathcal{D}' :: \Gamma, xf : \tau' \vdash e : \tau$ such that $height(\mathcal{D}) = height(\mathcal{D}')$, where τ' 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 λ_{pat} .

Lemma 2.2 (Canonical Forms)

Assume that $\emptyset \vdash v : \tau$ is derivable.

1. If $\tau = \delta$ for some base type δ , 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 τ_1 and τ_2 , then v is of the form $\langle v_1, v_2 \rangle$.
4. If $\tau = \tau_1 \rightarrow \tau_2$ for some types τ_1 and τ_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 $\tau_1 \rightarrow \tau_2$ for some τ_1 and τ_2 . As a consequence, we can no longer claim that a value of the type $\tau_1 \rightarrow \tau_2$ for some τ_1 and τ_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 $\tau_1 \rightarrow \tau_2$.

Given Γ, Γ_0 and θ , 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 λ_{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 λ_{pat} through the use of evaluation contexts defined as follows.

Definition 2.4 (Evaluation Contexts)

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

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

$$\begin{array}{c}
\frac{}{\mathbf{match}(v, x) \Rightarrow [x \mapsto v]} \text{ (mat-var)} \\
\frac{}{\mathbf{match}(\langle \rangle, \langle \rangle) \Rightarrow []} \text{ (mat-unit)} \\
\frac{\mathbf{match}(v_1, p_1) \Rightarrow \theta_1 \quad \mathbf{match}(v_2, p_2) \Rightarrow \theta_2}{\mathbf{match}(\langle v_1, v_2 \rangle, \langle p_1, p_2 \rangle) \Rightarrow \theta_1 \cup \theta_2} \text{ (mat-prod)} \\
\frac{\mathbf{match}(v, p) \Rightarrow \theta}{\mathbf{match}(c(v), c(p)) \Rightarrow \theta} \text{ (mat-const)}
\end{array}$$

Fig. 6. The pattern matching rules for λ_{pat}

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.

Definition 2.5

We define evaluation redexes (or ev-redex, for short) and their reducts in λ_{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 $e[x \mapsto v]$.
- $\mathbf{case} \ v \ \mathbf{of} \ (p_1 \Rightarrow e_1 \mid \dots \mid p_n \Rightarrow e_n)$ is an ev-redex if $\mathbf{match}(v, p_k) \Rightarrow \theta$ is derivable for some $1 \leq k \leq n$, 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 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, \dots, p_n in a matching clause sequence $(p_1 \Rightarrow e_1 \mid \dots \mid p_n \Rightarrow e_n)$ be disjoint, that is, for $1 \leq i \neq j \leq n$, 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 **case** $cons(x, xs)$ **of** ($nil \Rightarrow true \mid x' \Rightarrow false$) and **case** x **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 **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 λ_{pat} . Given a closed expression e in λ_{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 **case** v **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 $e \hookrightarrow_{ev}^* \mathbf{Error}$, $e \hookrightarrow_{ev}^* \mathbf{Match}$ and $e \hookrightarrow_{ev}^* \mathbf{Undefined}$ 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 λ_{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 λ_{pat} . This is precisely the type soundness of λ_{pat} .

2.3 Type soundness

We are now ready to state the subject reduction theorem for λ_{pat} , which implies that the evaluation of a well-typed expression in λ_{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 Theorem 2.9.

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 $e \hookrightarrow_{ev}^* \mathbf{Error}$ can never happen for any well-typed expression e in λ_{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:

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

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

Definition 2.10

Given two expressions e_1 and e_2 in λ_{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 $\mathbf{lam} x : \mathbf{int} * \mathbf{int}. x$ in which the type $\mathbf{int} * \mathbf{int}$ is provided by the programmer; for some reason (to be made clear later), we may elaborate the program into the following one:

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

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

Definition 2.11

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

- either $G[e_2] \hookrightarrow_{ev}^* \mathbf{Error}$ 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 \leq_{dyn} .

Corollary 2.12

Assume that $e_1 \leq_{dyn} e_2$ holds. For any context G such that $G[e_2]$ is a closed well-typed expression in λ_{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 \leq_{dyn} and Theorem 2.9. \square

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

Definition 2.13

We define general redexes (or g-redexes, for short) and their reducts in λ_{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.
- $\mathbf{let} x = e \mathbf{in} E[x] \mathbf{end}$ is a g-redex if x has no free occurrences in E , and its reduct is $E[e]$.
- $\langle \mathbf{fst}(v), \mathbf{snd}(v) \rangle$ is a g-redex and its reduct is v .

index signatures	$\mathcal{S} ::= \emptyset \mid \mathcal{S}, C : (s_1, \dots, s_n) \Rightarrow s$
index base sorts	$b ::= \text{bool} \mid \dots$
index sorts	$s ::= b \mid s_1 * s_2 \mid s_1 \rightarrow s_2$
index terms	$I ::= a \mid C(I_1, \dots, I_n) \mid \langle I_1, I_2 \rangle \mid \pi_1(I) \mid \pi_2(I) \mid \lambda a : s. I \mid I_1(I_2)$
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

$\frac{\phi(a) = s}{\phi \vdash a : s} \text{ (st-var)}$
$\frac{\mathcal{S}(C) = (s_1, \dots, s_n) \Rightarrow s \quad \phi \vdash I_k : s_k \text{ for } 1 \leq k \leq n}{\phi \vdash C(I_1, \dots, I_n) : s} \text{ (st-const)}$
$\frac{\phi \vdash I_1 : s_1 \quad \phi \vdash I_2 : s_2}{\phi \vdash \langle I_1, I_2 \rangle : s_1 * s_2} \text{ (st-prod)}$
$\frac{\phi \vdash I : s_1 * s_2}{\phi \vdash \pi_1(I) : s_1} \text{ (st-fst)} \quad \frac{\phi \vdash I : s_1 * s_2}{\phi \vdash \pi_2(I) : s_2} \text{ (st-snd)}$
$\frac{\phi, a : s_1 \vdash I : s_2}{\phi \vdash \lambda a : s_1. I : s_1 \rightarrow s_2} \text{ (st-lam)}$
$\frac{\phi \vdash I_1 : s_1 \rightarrow s_2 \quad \phi \vdash I_2 : s_1}{\phi \vdash I_1(I_2) : s_2} \text{ (st-app)}$

Fig. 8. The sorting rules for type index terms

- **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 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 λ_{pat} that may contain free variables, $e \hookrightarrow_g^* e'$ implies $e' \leq_{dyn} e$.

Proof

A (lengthy) proof of the lemma is given in Appendix A. \square

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 λ_{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.

$$\begin{array}{c}
\frac{}{\phi; \vec{P} \models true} \text{ (reg-true)} \qquad \frac{}{\phi; \vec{P}, false \models P} \text{ (reg-false)} \\
\frac{\phi; \vec{P} \models P_0}{\phi, a : s; \vec{P} \models P_0} \text{ (reg-var-thin)} \qquad \frac{\phi \vdash P : bool \quad \phi; \vec{P} \models P_0}{\phi; \vec{P}, P \models P_0} \text{ (reg-prop-thin)} \\
\frac{\phi, a : s; \vec{P} \models P \quad \phi \vdash I : s}{\phi; \vec{P}[a \mapsto I] \models P[a \mapsto I]} \text{ (reg-subst)} \qquad \frac{\phi; \vec{P} \models P_0 \quad \phi; \vec{P}, P_0 \models P_1}{\phi; \vec{P} \models P_1} \text{ (reg-cut)} \\
\frac{\phi \vdash I : s}{\phi; \vec{P} \models I \doteq_s I} \text{ (reg-eq-refl)} \qquad \frac{\phi; \vec{P} \models I_1 \doteq_s I_2}{\phi; \vec{P} \models I_2 \doteq_s I_1} \text{ (reg-eq-symm)} \\
\frac{\phi; \vec{P} \models I_1 \doteq_s I_2 \quad \phi; \vec{P} \models I_2 \doteq_s I_3}{\phi; \vec{P} \models I_1 \doteq_s I_3} \text{ (reg-eq-tran)}
\end{array}$$

Fig. 9. The regularity rules

In this section, we show how a generic type index language \mathcal{L} can be formed and then present some concrete examples of type index languages. For generality, we will include both tuples and functions in \mathcal{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 \mathcal{L} itself is typed. In order to avoid potential confusion, we call the types in \mathcal{L} *type index sorts* (or *sorts*, for short). The syntax of \mathcal{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 $(s_1, \dots, s_n) \Rightarrow b$, which means that $C(I_1, \dots, I_n)$ is an index term of sort b if I_i are of sorts s_i for $i = 1, \dots, n$. 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 \mathcal{S} associated with \mathcal{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 ϕ), and \vec{P} for a sequence of propositions, where the ordering of the terms in this sequence is of no significance.

We may write $\phi \vdash \vec{P} : bool$ to mean that $\phi \vdash P : bool$ 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 ϕ, \vec{P}, P_0 such that both $\phi \vdash \vec{P} : bool$ and $\phi \vdash P_0 : bool$ 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 π_1 and π_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 \mathcal{L} . A frame is a collection $\{\mathbf{D}_s\}_{s \in \text{sort}}$ of nonempty domains \mathbf{D}_s , one for each sort s . We require that $\mathbf{D}_{bool} = \{\mathbf{tt}, \mathbf{ff}\}$, where \mathbf{tt} and \mathbf{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 $\mathbf{D}_{s_1 \rightarrow s_2}$ 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 $\langle \{\mathbf{D}_s\}_{s \in \text{sort}}, \mathbf{I} \rangle$ of \mathcal{L} consists of a frame $\{\mathbf{D}_s\}_{s \in \text{sort}}$ and a function \mathbf{I} that maps each constant C of c-sort $(s_1, \dots, s_n) \Rightarrow b$ to a function $\mathbf{I}(C)$ from $\mathbf{D}_{s_1} \times \dots \times \mathbf{D}_{s_n}$ 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

- $\mathbf{I}(true) = \mathbf{tt}$ and $\mathbf{I}(false) = \mathbf{ff}$, 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 $C(I_1, \dots, I_n) \doteq C(I'_1, \dots, I'_n)$

implies that $I_i \doteq I'_i$ for $1 \leq i \leq n$. Therefore, for each constructor C , we require that $\mathbf{I}(C)$ be an injective (a.k.a. 1-1) function.

An assignment η is a finite mapping from index variables to $\mathbf{D} = \cup_{s \in \text{sort}} \mathbf{D}_s$, and we use $\mathbf{dom}(\eta)$ for the domain of η . As usual, we use $[]$ for the empty mapping and $\eta[a \mapsto \mathbf{a}]$ for the mapping that extends η 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 $\mathcal{M} = \langle \{\mathbf{D}_s\}_{s \in \text{sort}}, \mathbf{I} \rangle$ of \mathcal{S} , which is the signature associated with \mathcal{L} , is a model for \mathcal{L} if there exists a (partial) binary function $\mathcal{V}_{\mathcal{M}}$ such that for each assignment η satisfying $\eta : \phi$ for some ϕ and each index term I , $\mathcal{V}_{\mathcal{M}}(\eta, I)$ is properly defined such that $\mathcal{V}_{\mathcal{M}}(\eta, I) \in \mathbf{D}_s$ holds whenever $\phi \vdash I : s$ is derivable for some sort s , and the following conditions are also met:

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

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

Proposition 3.2

The constraint relation $\models_{\mathcal{M}}$ is regular.

Proof

It is a simple routine to verify that each of the regularity rules listed in Figure 9 is valid. \square

Therefore, we have shown that for any given type index language \mathcal{L} , there always exists a regular constraint relation if a model can be constructed for \mathcal{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 \mathcal{L}_{alg}

We now describe a type index language \mathcal{L}_{alg} in which only algebraic terms can be formed. Suppose that there are some base sorts in \mathcal{L}_{alg} . For each base sort b , there

exists some constructors of c-sorts $(b_1, \dots, b_n) \Rightarrow b$ 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 \mathcal{L}_{alg} can be formed as follows,

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

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 \mathcal{L}_{alg} is of the following form:

$$a_1 : b_1, \dots, a_n : b_n; I_1 \doteq_{I'_1}, \dots, I_n \doteq_{I'_n} \mid= I \doteq_{I'}$$

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 \mathcal{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:

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

We can then use index terms of the sort stp to represent the types in a simply typed λ -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 \mathcal{L}_{alg} .

3.3.2 Another type index language \mathcal{L}_{int}

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

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

There are no tuples and functions (formed through λ -abstraction) in \mathcal{L}_{int} , and the constants C in \mathcal{L}_{int} together with their c-sorts are listed in Figure 10. Let \mathbf{D}_{int} be the domain (set) of integers and \mathcal{M}_{int} be $\langle \{\mathbf{D}_{bool}, \mathbf{D}_{int}\}, \mathbf{I}_{int} \rangle$, where \mathbf{I}_{int} maps each constant in \mathcal{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 \mathcal{M}_{int} is a model for \mathcal{L}_{int} . Therefore, the constraint relation $\mid=_{\mathcal{M}_{int}}$ is regular.

<i>true</i>	:	()	→	<i>bool</i>	
<i>false</i>	:	()	→	<i>bool</i>	
<i>i</i>	:	()	→	<i>int</i>	for every integer <i>i</i>
\neg	:	(<i>bool</i>)	→	<i>bool</i>	negation
\wedge	:	(<i>bool, bool</i>)	→	<i>bool</i>	conjunction
\vee	:	(<i>bool, bool</i>)	→	<i>bool</i>	disjunction
$+$:	(<i>int, int</i>)	→	<i>int</i>	
$-$:	(<i>int, int</i>)	→	<i>int</i>	
	:	(<i>int, int</i>)	→	<i>int</i>	
$/$:	(<i>int, int</i>)	→	<i>int</i>	
<i>max</i>	:	(<i>int, int</i>)	→	<i>int</i>	
<i>min</i>	:	(<i>int, int</i>)	→	<i>int</i>	
<i>mod</i>	:	(<i>int, int</i>)	→	<i>int</i>	modulo operation
\geq	:	(<i>int, int</i>)	→	<i>bool</i>	
$>$:	(<i>int, int</i>)	→	<i>bool</i>	
\leq	:	(<i>int, int</i>)	→	<i>bool</i>	
$<$:	(<i>int, int</i>)	→	<i>bool</i>	
$=$:	(<i>int, int</i>)	→	<i>bool</i>	
\neq	:	(<i>int, int</i>)	→	<i>bool</i>	
...	:	...			

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

```

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

Given a constraint $\phi; \vec{P} \models_{\mathcal{M}_{int}} P_0$, where $\phi = a_1 : int, \dots, a_n : int$, 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 \mathcal{L}_{int} in the design of a dependently type functional programming language where type equality between 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 \mathcal{L}_{alg} or \mathcal{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 \mathcal{L}_λ , which extends \mathcal{L}_{alg} with higher-order type indexes as follows:

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

Like in \mathcal{L}_{alg} , a constraint in \mathcal{L}_λ is of the following form:

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

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

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

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 \mathcal{L}_λ is undecidable as (a special case of) it can be reduced to the problem of higher-order unification. For instance, $\phi; I_1 \doteq I_2 \models \text{false}$ holds if and only if there exists no substitution $\Theta : \phi$ such that $I_1[\Theta]$ and $I_2[\Theta]$ are $\beta\eta$ -equivalent.

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

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

where for $1 \leq i \leq j \leq n$, 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 λ -terms, which is done by comparing whether the two λ -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 **EXP**(I). The value constructors associated with **EXP** are assigned the following c-types:

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

The intent is to use an index term I of sort *typ* to represent a type in the second-order polymorphic λ -calculus λ_2 (a.k.a. system F), and a value of type **EXP**(I) to represent a λ -term in λ_2 that can be assigned the type represented by I . For instance, the type $\forall \alpha. \alpha \rightarrow \alpha$ is represented as $\text{All}(\lambda a : \text{typ}. \text{Arrow}(a, a))$, and the following term:

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

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

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

which can be given the following type:

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

represents the λ -term $\Lambda \alpha_1. \Lambda \alpha_2. \lambda x : \alpha_1. \lambda y : \alpha_2. y(x)$. This is a form of higher-order abstract syntax (h.o.a.s.) representation for λ -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.

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

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

4.1 Syntax

Let us fix a type index language \mathcal{L} . We now present $\lambda_{pat}^{\Pi, \Sigma} = \lambda_{pat}^{\Pi, \Sigma}(\mathcal{L})$, which is an extension of λ_{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 \dots in the definition of types in $\lambda_{pat}^{\Pi, \Sigma}$ for the following definition of types in λ_{pat} :

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

We now use δ for base type families. We may write δ 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, \dots, I_n ; \vec{P} for a (possibly empty) sequence of index propositions P_1, \dots, P_n ; $\Pi\phi$ for a (possibly empty) sequence of quantifiers: $\Pi a_1 : s_1 \dots \Pi a_n : s_n$,

$$\begin{array}{c}
\frac{\vdash \delta(s_1, \dots, s_n) \quad \phi \vdash I_k : s_k \text{ for } 1 \leq k \leq n}{\phi \vdash \delta(I_1, \dots, I_n) [\mathbf{type}]} \text{ (tp-base)} \\
\frac{}{\phi \vdash \mathbf{1} [\mathbf{type}]} \text{ (tp-unit)} \\
\frac{\phi \vdash \tau_1 [\mathbf{type}] \quad \phi \vdash \tau_2 [\mathbf{type}]}{\phi \vdash \tau_1 * \tau_2 [\mathbf{type}]} \text{ (tp-prod)} \\
\frac{\phi \vdash \tau_1 [\mathbf{type}] \quad \phi \vdash \tau_2 [\mathbf{type}]}{\phi \vdash \tau_1 \rightarrow \tau_2 [\mathbf{type}]} \text{ (tp-fun)} \\
\frac{\phi \vdash P : \mathit{bool} \quad \phi \vdash \tau [\mathbf{type}]}{\phi \vdash P \supset \tau [\mathbf{type}]} \text{ (tp-}\supset\text{)} \\
\frac{\phi, a : s \vdash \tau [\mathbf{type}]}{\phi \vdash \Pi a : s. \tau [\mathbf{type}]} \text{ (tp-}\Pi\text{)} \\
\frac{\phi \vdash P : \mathit{bool} \quad \phi \vdash \tau [\mathbf{type}]}{\phi \vdash P \wedge \tau [\mathbf{type}]} \text{ (tp-}\wedge\text{)} \\
\frac{\phi, a : s \vdash \tau [\mathbf{type}]}{\phi \vdash \Sigma a : s. \tau [\mathbf{type}]} \text{ (tp-}\Sigma\text{)} \\
\frac{}{\phi \vdash \emptyset [\mathbf{ctx}]} \text{ (ctx-emp)} \\
\frac{\phi \vdash \Gamma [\mathbf{ctx}] \quad \phi \vdash \tau [\mathbf{type}] \quad xf \notin \mathit{dom}(\Gamma)}{\phi \vdash \Gamma, xf : \tau [\mathbf{ctx}]} \text{ (ctx-ext)}
\end{array}$$

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

where the index context ϕ is $a_1 : s_1, \dots, a_n : s_n$; $\vec{P} \supset \tau$ for $P_1 \supset (\dots (P_n \supset \tau) \dots)$ if $\vec{P} = P_1, \dots, P_n$.

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 τ , 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,

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

where $/$ stands for the integer division function in some type index language. The

$$\begin{array}{c}
\frac{\phi; \vec{P} \models I_1 \doteq I'_1 \quad \dots \quad \phi; \vec{P} \models I_n \doteq I'_n}{\phi; \vec{P} \models \delta(I_1, \dots, I_n) \leq_{tp}^s \delta(I'_1, \dots, I'_n)} \text{ (st-sub-base)} \\
\frac{}{\phi; \vec{P} \models \mathbf{1} \leq_{tp}^s \mathbf{1}} \text{ (st-sub-unit)} \\
\frac{\phi; \vec{P} \models \tau_1 \leq_{tp}^s \tau'_1 \quad \phi; \vec{P} \models \tau_2 \leq_{tp}^s \tau'_2}{\phi; \vec{P} \models \tau_1 * \tau_2 \leq_{tp}^s \tau'_1 * \tau'_2} \text{ (st-sub-prod)} \\
\frac{\phi; \vec{P} \models \tau'_1 \leq_{tp}^s \tau_1 \quad \phi; \vec{P} \models \tau_2 \leq_{tp}^s \tau'_2}{\phi; \vec{P} \models \tau_1 \rightarrow \tau_2 \leq_{tp}^s \tau'_1 \rightarrow \tau'_2} \text{ (st-sub-fun)} \\
\frac{\phi; \vec{P}, P' \models P \quad \phi; \vec{P}, P' \models \tau \leq_{tp}^s \tau'}{\phi; \vec{P} \models P \supset \tau \leq_{tp}^s P' \supset \tau'} \text{ (st-sub-}\supset\text{)} \\
\frac{\phi, a : s; \vec{P} \models \tau \leq_{tp}^s \tau'}{\phi; \vec{P} \models \Pi a : s. \tau \leq_{tp}^s \Pi a : s. \tau'} \text{ (st-sub-}\Pi\text{)} \\
\frac{\phi; \vec{P}, P \models P' \quad \phi; \vec{P}, P \models \tau \leq_{tp}^s \tau'}{\phi; \vec{P} \models P \wedge \tau \leq_{tp}^s P' \wedge \tau'} \text{ (st-sub-}\wedge\text{)} \\
\frac{\phi, a : s; \vec{P} \models \tau \leq_{tp}^s \tau'}{\phi; \vec{P} \models \Sigma a : s. \tau \leq_{tp}^s \Sigma a : s. \tau'} \text{ (st-sub-}\Sigma\text{)}
\end{array}$$

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

$$\begin{array}{c}
\frac{}{x \downarrow \tau \Rightarrow (\emptyset; \emptyset; x : \tau)} \text{ (pat-var)} \\
\frac{}{\langle \rangle \downarrow \mathbf{1} \Rightarrow (\emptyset; \emptyset; \emptyset)} \text{ (pat-unit)} \\
\frac{p_1 \downarrow \tau_1 \Rightarrow (\phi_1; \vec{P}_1; \Gamma_1) \quad p_2 \downarrow \tau_2 \Rightarrow (\phi_2; \vec{P}_2; \Gamma_2)}{\langle p_1, p_2 \rangle \downarrow \tau_1 * \tau_2 \Rightarrow (\phi_1, \phi_2; \vec{P}_1, \vec{P}_2; \Gamma_1, \Gamma_2)} \text{ (pat-prod)} \\
\frac{\phi_0; \vec{P}_0 \vdash cc(\tau) : \delta(I_1, \dots, I_n) \quad p \downarrow \tau \Rightarrow (\phi; \vec{P}; \Gamma)}{cc(p) \downarrow \delta(I'_1, \dots, I'_n) \Rightarrow (\phi_0, \phi; \vec{P}_0, \vec{P}, I_1 \doteq I'_1, \dots, I_n \doteq I'_n; \Gamma)} \text{ (pat-const)}
\end{array}$$

Fig. 15. The typing rules for patterns

following type is a rather interesting one:

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

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

$$\begin{array}{c}
\frac{\phi; \vec{P}; \Gamma \vdash e : \tau_1 \quad \phi; \vec{P} \models \tau_1 \leq_{tp}^s \tau_2}{\phi; \vec{P}; \Gamma \vdash e : \tau_2} \text{ (ty-sub)} \\
\frac{\phi \vdash \Gamma \text{ [ctx]} \quad \Gamma(xf) = \tau}{\phi; \vec{P}; \Gamma \vdash xf : \tau} \text{ (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])} \text{ (ty-const)} \\
\frac{\phi \vdash \Gamma \text{ [ctx]}}{\phi; \vec{P}; \Gamma \vdash \langle \rangle : \mathbf{1}} \text{ (ty-unit)} \quad \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} \text{ (ty-prod)} \\
\frac{\phi; \vec{P}; \Gamma \vdash e : \tau_1 * \tau_2}{\phi; \vec{P}; \Gamma \vdash \mathbf{fst}(e) : \tau_1} \text{ (ty-fst)} \quad \frac{\phi; \vec{P}; \Gamma \vdash e : \tau_1 * \tau_2}{\phi; \vec{P}; \Gamma \vdash \mathbf{snd}(e) : \tau_2} \text{ (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 \rightarrow \tau_2} \text{ (ty-clause)} \\
\frac{\phi; \vec{P}; \Gamma \vdash p_k \Rightarrow e_k : \tau_1 \rightarrow \tau_2 \text{ for } k = 1, \dots, n}{\phi; \vec{P}; \Gamma \vdash (p_1 \Rightarrow e_1 \mid \dots \mid p_n \Rightarrow e_n) : \tau_1 \rightarrow \tau_2} \text{ (ty-clause-seq)} \\
\frac{\phi; \vec{P}; \Gamma \vdash e : \tau_1 \quad \phi; \vec{P}; \Gamma \vdash ms : \tau_1 \rightarrow \tau_2}{\phi; \vec{P}; \Gamma \vdash \mathbf{case } e \text{ of } ms : \tau_2} \text{ (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 \rightarrow \tau_2} \text{ (ty-lam)} \\
\frac{\phi; \vec{P}; \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \phi; \vec{P}; \Gamma \vdash e_2 : \tau_1}{\phi; \vec{P}; \Gamma \vdash e_1(e_2) : \tau_2} \text{ (ty-app)} \\
\frac{\phi; \vec{P}; \Gamma, f : \tau \vdash e : \tau}{\phi; \vec{P}; \Gamma \vdash \mathbf{fix } f. e : \tau} \text{ (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 \text{ in } e_2 \text{ end} : \tau_2} \text{ (ty-let)}
\end{array}$$

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

sorts, which are formally defined as follows:

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

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 $\{a : int \mid a \geq 0\}$. In general, we may write $\{a : s \mid P_1, \dots, P_n\}$ for the subset sort \hat{s}_n defined as follows:

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

$$\begin{array}{c}
\frac{\phi; \vec{P}, P; \Gamma \vdash v : \tau}{\phi; \vec{P}; \Gamma \vdash \supset^+(v) : P \supset \tau} \text{ (ty-}\supset\text{-intro)} \\
\frac{\phi; \vec{P}; \Gamma \vdash e : P \supset \tau \quad \phi; \vec{P} \models P}{\phi; \vec{P}; \Gamma \vdash \supset^-(e) : \tau} \text{ (ty-}\supset\text{-elim)} \\
\frac{\phi, a : s; \vec{P}; \Gamma \vdash v : \tau}{\phi; \vec{P}; \Gamma \vdash \Pi^+(v) : \Pi a : s. \tau} \text{ (ty-}\Pi\text{-intro)} \\
\frac{\phi; \vec{P}; \Gamma \vdash e : \Pi a : s. \tau \quad \phi \vdash I : s}{\phi; \vec{P}; \Gamma \vdash \Pi^-(e) : \tau[a \mapsto I]} \text{ (ty-}\Pi\text{-elim)} \\
\frac{\phi; \vec{P}; \Gamma \vdash e : \tau \quad \phi; \vec{P} \models P}{\phi; \vec{P}; \Gamma \vdash \wedge(e) : P \wedge \tau} \text{ (ty-}\wedge\text{-intro)} \\
\frac{\phi; \vec{P}; \Gamma \vdash e_1 : P \wedge \tau_1 \quad \phi; \vec{P}, P; \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\phi; \vec{P}; \Gamma \vdash \mathbf{let} \wedge(x) = e_1 \mathbf{in} e_2 \mathbf{end} : \tau_2} \text{ (ty-}\wedge\text{-elim)} \\
\frac{\phi; \vec{P}; \Gamma \vdash e : \tau[a \mapsto I] \quad \phi \vdash I : s}{\phi; \vec{P}; \Gamma \vdash \Sigma(e) : \Sigma a : s. \tau} \text{ (ty-}\Sigma\text{-intro)} \\
\frac{\phi; \vec{P}; \Gamma \vdash e_1 : \Sigma a : s. \tau_1 \quad \phi, a : s; \vec{P}; \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\phi; \vec{P}; \Gamma \vdash \mathbf{let} \Sigma(x) = e_1 \mathbf{in} e_2 \mathbf{end} : \tau_2} \text{ (ty-}\Sigma\text{-elim)}
\end{array}$$

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

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

We use $\phi; \vec{P} \vdash I : \{a : s \mid P_1, \dots, P_n\}$ to mean that $\phi; \vec{P} \vdash I : s$ is derivable and $\phi; \vec{P} \vdash P_i[a \mapsto I]$ hold for $i = 1, \dots, n$. 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 $\Pi a_1 : \mathbf{nat}. \mathbf{int}(a_1) \rightarrow \Sigma a_2 : \mathbf{nat}. \mathbf{int}(a_2)$ for the following type:

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

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 $\vdash \delta(s_1, \dots, s_n)$ to indicate that we can construct a type $\delta(I_1, \dots, I_n)$ when given type index terms I_1, \dots, I_n of sorts s_1, \dots, s_n , respectively.

A judgment of the form $\phi \vdash \tau$ [**type**] means that τ is a well-formed type under the index context ϕ , and a judgment of the form $\phi \vdash \Gamma$ [**ctx**] means that Γ is a well-formed (expression) context under ϕ . The domain $\mathbf{dom}(\Gamma)$ of a context Γ is defined to be the set of variables declared in Γ . We write $\phi; \vec{P} \models P_0$ for a regular constraint relation in the fixed type index language \mathcal{L} .

In $\lambda_{pat}^{\Pi, \Sigma}$, type equality, that is, equality between types, is defined in terms of the static subtype relation \leq_{tp}^s : We say that τ and τ' are equal if both $\tau \leq_{tp}^s \tau'$ and $\tau' \leq_{tp}^s \tau$ hold. By overloading \models , we use $\phi; \vec{P} \models \tau \leq_{tp}^s \tau'$ 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 \leq_{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 $\tau_1 \leq_{tp}^s \tau_2$ does not hold (as \leq_{tp}^s is syntax-directed). We are to introduce in Section 4.6 another subtype relation \leq_{tp}^d , which is much stronger than \leq_{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 $\phi, \phi_0; \vec{P} \models \tau \leq_{tp}^s \tau'$ is derivable and $\phi \vdash \Theta : \phi_0$ holds, then $\phi; \vec{P}[\Theta] \models \tau[\Theta] \leq_{tp}^s \tau'[\Theta]$ is also derivable.
2. If $\phi; \vec{P}, \vec{P}_0 \models \tau \leq_{tp}^s \tau'$ is derivable and $\phi; \vec{P} \models \vec{P}_0$ holds, then $\phi; \vec{P} \models \tau \leq_{tp}^s \tau'$ is also derivable.

Proof

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

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

Proposition 4.2 (Reflexivity and Transitivity of \leq_{tp}^s)

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

Proof

Straightforward. \square

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 τ , if v matches p , that is, $\mathbf{match}(v, p) \Rightarrow \theta$ holds for some substitution θ , then there exists an index substitution Θ 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 ϕ_1 and ϕ_2 share no common index variables in their domains. In the rule (**pat-const**), we write $\phi_0; \vec{P}_0 \vdash cc(\tau) : \delta(I_1, \dots, I_n)$ to mean that cc is a constant constructor assigned (according to some signature for constants) the following c-type:

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

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 τ_2 contain only type index variables declared in ϕ . 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 Γ . Also, in the rule (**ty- Σ -elim**), the type index variable a is assumed to have no free occurrences in either \vec{P} , Γ or τ_2 . We now briefly go over some of the typing rules in Figure 17.

- If a value v can be assigned a type τ under an assumption P , then the typing rule (**ty- \supset -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 τ 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 τ_2 under the assumption that P holds and x is of type τ_1 . If e_1 is given the asserting type $P \wedge \tau_1$, then the rule (**ty- \wedge -elim**) assigns the type τ_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 $\mathcal{D} :: \phi; \vec{P}; \Gamma \vdash e : \tau$.

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

Proof

Straightforward. \square

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

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 δ , 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. \square

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)**:

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

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

Proof

(Sketch) By structural induction on the derivation \mathcal{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 $height(\mathcal{D}) = height(\mathcal{D}')$. \square

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

Proof

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

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. \square

4.3 Dynamic semantics

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

Definition 4.8 (Evaluation Contexts)

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

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 .
- $\mathbf{let} \ \wedge(x) = \wedge(v) \ \mathbf{in} \ e \ \mathbf{end}$ is a redex, and its reduct is $e[x \mapsto v]$.
- $\mathbf{let} \ \Sigma(x) = \Sigma(v) \ \mathbf{in} \ e \ \mathbf{end}$ 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 λ_{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 Θ 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 $p \downarrow \tau \Rightarrow (\phi; \vec{P}; \Gamma)$. \square

```

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

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 Θ 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. \square

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 λ_{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 $E[\mathbf{case} \ v \ \mathbf{of} \ ms]$ 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):

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

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 $\Sigma a : \text{nat}. (\tau) \mathbf{list}(a)$ for some type τ . 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:

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

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 `|·|` in Figure 19 that translates types, contexts and expressions in $\lambda_{pat}^{\Pi, \Sigma}$ into types, contexts and expressions in λ_{pat} , respectively. In particular, for each type family δ in $\lambda_{pat}^{\Pi, \Sigma}$, we assume that there is a corresponding type δ in λ_{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 λ_{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 λ_{pat} .

Proof

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

$ \delta(\vec{I}) $	$=$	δ
$ \mathbf{1} $	$=$	$\mathbf{1}$
$ \tau_1 * \tau_2 $	$=$	$ \tau_1 * \tau_2 $
$ \tau_1 \rightarrow \tau_2 $	$=$	$ \tau_1 \rightarrow \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 \dots \mid p_n \Rightarrow e_n) $	$=$	$\mathbf{case} e \mathbf{of} (p_1 \Rightarrow e_1 \mid \dots \mid p_n \Rightarrow e_n)$
$ \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} $	$=$	$\mathbf{let} x = e_1 \mathbf{in} e_2 \mathbf{end}$

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

Given a closed expression e_0 in λ_{pat} , we say that e_0 is typable in λ_{pat} if $\emptyset \vdash e_0 : \tau_0$ is derivable for some type τ_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 τ . Then by Theorem 4.13, we know that if an expression e in λ_{pat} is typable in $\lambda_{pat}^{\Pi, \Sigma}$ then it is already typable in λ_{pat} . In other words, $\lambda_{pat}^{\Pi, \Sigma}$ does not make more expressions in λ_{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 λ_{pat} .
2. If $|e| \hookrightarrow_{ev}^* v_0$ in λ_{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 λ_{pat} . Combining Theorem 4.13 and Theorem 4.14, we say that $\lambda_{pat}^{\Pi, \Sigma}$ is a conservative extension of λ_{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 \leq_{tp}^s and it plays a key role in Section 5, where an elaboration process is presented to facilitate program construction in $\lambda_{pat}^{\Pi, \Sigma}$.

Definition 4.15 (Dynamic Subtype Relation)

We write $\phi; \vec{P} \models E : \tau \leq_{tp}^d \tau'$ to mean that for any expression e and context Γ , if $\phi; \vec{P}; \Gamma \vdash e : \tau$ is derivable then both $\phi; \vec{P}; \Gamma \vdash E[e] : \tau'$ is derivable and $|e| \leq_{dyn} |E[e]|$ holds. We may write $\phi; \vec{P} \models \tau \leq_{tp}^d \tau'$ if, for some E , $\phi; \vec{P} \models E : \tau \leq_{tp}^d \tau'$ holds, where E can be thought of as a witness to $\tau \leq_{tp}^d \tau'$.

As is desired, the dynamic subtype relation \leq_{tp}^d is both reflexive and transitive.

Proposition 4.16 (Reflexivity and Transitivity of \leq_{tp}^d)

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

Proof

(Sketch) The proposition follows from the fact that the relation \leq_{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 $\lambda_{pat}^{\Pi, \Sigma}$:

$$\Pi a : int. a \geq 0 \supset (\mathbf{int}(a) \rightarrow \mathbf{int}(a + a))$$

corresponds to the following first-order formula:

$$\forall a : int. a \geq 0 \supset (\mathbf{int}(a) \rightarrow \mathbf{int}(a + a))$$

where \mathbf{int} is interpreted as some predicate on integers, and both \supset and \rightarrow stand for the implication connective in logic. However, it is not possible in $\lambda_{pat}^{\Pi, \Sigma}$ to form a dependent type of the form $\Pi a : \tau_1. \tau_2$, which on the other hand is allowed in a

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

Fig. 20. The syntax for DML_0

(full) dependent type system such as λP (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 \leq_{tp}^s in $\lambda_{pat}^{\Pi, \Sigma}$:

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

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, 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