Dependently Typed Pattern Matching

Hongwei Xi[*]

Computer Science Department
Boston University
Boston, MA 02215, USA

hwxi@cs.bu.edu

Abstract. *The mechanism for declaring datatypes to model data structures in functional programming languages such as Standard ML and Haskell can offer both convenience in programming and clarity in code. With the introduction of dependent datatypes in DML, the programmer can model data structures with more accuracy, thus capturing more program invariants. In this paper, we study some practical aspects of dependent datatypes that affect both type-checking and compiling pattern matching. The results, which have already been tested, demonstrate that dependent datatype can not only offer various programming benefits but also lead to performance gains, yielding a concrete case where safer programs run faster.*

1 Introduction

In functional programming languages such as Standard ML (SML) (Milner, Tofte, Harper, and MacQueen 1997) and Haskell (Peyton Jones et al. 1999), the programmer can declare datatypes to model the data structures needed in programming and then use pattern matching to decompose the values of the declared datatypes. This is a mechanism that can offer both convenience in programming and clarity in code. For instance, we can declare the following datatype in SML to represent random-access (RA) lists and then implement a function that takes $O(\log n)$ time to access the nth element in a given RA list (Xi 1999b).

```
datatype 'a ralist =
  Nil | One of 'a
| Even of 'a ralist * 'a ralist
| Odd of 'a ralist * 'a ralist
```

Note that we intend to use *Nil* for the empty list, *One*(x) for the singleton list consisting of x, *Even*(l_1, l_2) for the list $x_1, y_1, \ldots, x_n, y_n$ ($n > 0$) where l_1 and l_2 represent lists $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$, respectively, and *Odd*(l_1, l_2) for $x_1, y_1, \ldots, x_n, y_n, x_{n+1}$ ($n > 0$) where l_1 and l_2 represent lists $x_1, \ldots, x_n, x_{n+1}$ and $y_1, \ldots, y_n$, respectively. However, this datatype declaration is unsatisfactory. For instance, it cannot enforce the invariant that the two arguments of *Even* need to represent two nonempty lists containing the same number of elements.

We have extended ML to Dependent ML (DML) with a restricted form of dependent types (Xi and Pfenning 1999; Xi 1998), introducing a notion of dependent datatypes that

[*]Partially supported by NSF grants no. CCR-0224244 and no. CCR-0229480

```
fun('a)
    uncons (One x) = (x,Nil)
  | uncons (Even (l1,l2)) =
    (case uncons l1 of (x,Nil) => (x,l2) | (x,l1) => (x,Odd (l2,l1)))
  | uncons (Odd (l1,l2)) =
    let val (x,l1) = uncons l1 in (x,Even (l2,l1)) end
withtype {n:pos} 'a ralist(n) -> 'a * 'a ralist(n-1)
```

Figure 1: The function *uncons* in DML

allows the programmer to model data structures with more accuracy. In DML, we can declare a dependent datatype as follows.

```
datatype 'a ralist (nat) =
  Nil (0) | One (1) of 'a
| {n:pos} Even (n+n) of 'a ralist(n) * 'a ralist(n)
| {n:pos} Odd (n+n+1) of 'a ralist(n+1) * 'a ralist(n)
```

In this declaration, the datatype `'a ralist` is indexed with a natural number, which stands for the length of a RA list in this case. For instance, the above syntax indicates that

- *One* is assigned the type scheme $\forall\alpha.\alpha \to (\alpha)\mathit{ralist}(1)$, which means that *One* forms a RA list of length 1 when given an element, and
- *Even* is assigned the following type scheme:

  $$\forall\alpha.\Pi n : pos.(\alpha)\mathit{ralist}(n) * (\alpha)\mathit{ralist}(n) \to (\alpha)\mathit{ralist}(n+n),$$

  which states that *Even* yields a RA list of length $n+n$ when given a pair of RA lists of length n. We use `{n:pos}` to indicate that n is universally quantified over positive integers, which is usually written as $\Pi n : pos$ in a dependent type theory.

Now we can implement in DML a function *uncons* that takes a nonempty RA list and returns a pair consisting of the head and the tail of the RA list. We present the implementation in Figure 1. Unfortunately, the implementation does *not* type-check in DML for a simple reason. Notice that we need to prove the list l_1 is a nonempty RA list in order to type-check the following clause in the implementation.

$$(x, l_1) \Rightarrow (x, \mathit{Odd}(l_2, l_1))$$

In DML, pattern matching is performed sequentially at run-time. Therefore, we know that l_1 cannot be *Nil* when the above clause is chosen at run-time. On the other hand, type-checking in DML, like in ML, assumes nondeterministic pattern matching, ignoring the fact that the above clause is chosen only if the result of $\mathit{uncons}(l_1)$ does not match the pattern (x, Nil). This example clearly illustrates a gap between static and dynamic semantics of pattern matching in DML.

Obviously, if all patterns in a sequence of pattern matching clauses are mutually disjoint, nondeterministic pattern matching is equivalent to sequential pattern matching. A straightforward approach, which was adopted in DML, is to require that the programmer replace the above clause with three clauses as follows, where p ranges over the three patters $\mathit{One}(\_)$, $\mathit{Even}(\_)$ and $\mathit{Odd}(\_)$.

$$(x, l_1 \textbf{ as } p) \Rightarrow (x, \mathit{Odd}(l_2, l_1))$$

While this is a simple approach, it can cause a great deal of inconvenience in programming as well as performance loss at run-time. For instance, in an implementation of red/black trees, one pattern needs to be expanded into 36 disjoint patterns in order to make the implementation type-check and the expansion causes the Caml-light compiler to produce significantly inferior code.

In this paper, we present an approach that bridges the gap between static and dynamic semantics of pattern matching in DML. Given patterns $p_1, \ldots, p_m$ and p, we intend to find patterns $p'_1, \ldots, p'_n$ such that a value matches p but none of p_i for $i = 1, \ldots, m$ if and only if it matches p'_j for some $1 \leq j \leq n$. Note that $p'_i, \ldots, p'_n$ do not have to be mutually disjoint. We emphasize that resolving sequentiality in pattern matching is only needed for type-checking in DML; it is not needed for compiling programs in DML. Similar problems have already been extensively studied in the context of lazy pattern matching compilation (Augustsson 1985; Puel and Suárez 1993; Laville 1990; Maranget 1994). In this paper, we essentially follow Laville's approach to resolving sequentiality in pattern matching.[1] However, there remains a significant issue in our case that has never been studied before. We need an approach that can produce the least n so as to minimize the number of constraints generated during type-checking. We prove that we have found such an approach, which is the main technical contribution of the paper.

There is yet another issue. When type-checking the implementation in Figure 1 (after the above expansion), the DML type-checker generates a warning message stating that the pattern matching is nonexhaustive as it assumes that *uncons* may be applied to *Nil*. We can eliminate this bogus warning message by verifying that *Nil* can never have a type (τ)*ralist*(n) for any positive integer n. This immediately implies that there is no need to insert a tag check for checking whether the argument of *uncons* is *Nil* at run-time when we compile the implementation in Figure 1. We will explain how such tag checks can be eliminated in Section 4. Note that this is an issue similar to array bound check elimination. Along this line, we can find cases such as an interpreter for a simply typed functional programming language where no run-time tag checks are needed for decomposing the values of certain datatypes.

The rest of the paper is organized as follows. In Section 2, we present some basics on types and pattern matching in DML. We then state in Section 3 a problem on pattern matching in DML and present a solution to this problem. We also prove that the solution is optimal according to a reasonable criterion. In Section 4, we study pattern matching compilation in the presence of dependent types and present an example. We also present some experimental results in Section 5 and provide some brief explanation. Lastly, we mention some related work and conclude.

2 Preliminaries

We present in this section some features in DML that are necessary for our study. Please find more details in (Xi and Pfenning 1999; Xi 1998).

[1]Laville's approach is simple but can be expensive. In theory, the approach leads to an algorithm that is exponential on the size of input patterns. Also, the approach cannot handle integer patterns because of the existence of infinitely many integer constants, but this is not a problem in our setting, which we will explain shortly.

index expressions	i, j	$::=$	$a \mid c \mid i + j \mid i - j \mid i * j \mid i \div j$
index propositions	P	$::=$	$i < j \mid i \leq j \mid i \geq j \mid i > j \mid i = j \mid i \neq j \mid P_1 \wedge P_2 \mid P_1 \vee P_2$
index sorts	γ	$::=$	*int* $\mid \{a : \gamma \mid P\} \mid \gamma_1 * \gamma_2$
index variable contexts	ϕ	$::=$	$\cdot \mid \phi, a : \gamma \mid \phi, P$
satisfaction relation			$\phi \models P$
index constraints	Φ	$::=$	$P \mid P \supset \Phi \mid \forall a : \gamma.\Phi$

Figure 2: The syntax for type index expressions

2.1 Types in DML

Intuitively speaking, dependent types are types which depend on the values of language expressions. For instance, we may form a type $int(i)$ for each integer i to mean that every integer expression of this type must have value i, that is, $int(i)$ is a singleton type. Note that i is the expression on which this type depends. We use the name *type index expression* for such an expression. There are various compelling reasons, such as practical type-checking, for imposing restrictions on expressions that can be chosen as type index expressions. A novelty in DML is to require that type index expressions be drawn only from a given constraint domain. For instance, the syntax for type index expressions in some integer domain is given in Figure 2, where we use a for type index variables and c for fixed integers. Note that the language for type index expressions is typed. We use the name *sort* for a type in this language so as to avoid potential confusion. We use $\cdot$ for the empty index variable context and omit the standard sorting rules for this language. We write $\{a : \gamma \mid P\}$ to denote the subset sort for those elements of sort γ that satisfy the proposition P. For example, we use *nat* as an abbreviation for the subset sort $\{a : int \mid a \geq 0\}$. We write $\phi \models \Phi$ to mean that the constraint Φ is satisfied under the index context ϕ, that is, the formula $(\phi)\Phi$ is satisfiable in the domain of integers, where $(\phi)\Phi$ is defined below.

$$\begin{array}{cc} (\cdot)\Phi = \Phi & (\phi, a : int)\Phi = (\phi)\forall a : int.\Phi \\ (\phi, P)\Phi = (\phi)(P \supset \Phi) & (\phi, \{a : \gamma \mid P\})\Phi = (\phi, a : \gamma)(P \supset \Phi) \end{array}$$

For instance, the satisfiability relation $a : nat, b : int, a + 1 = b \models b > 0$ holds since the following formula is true in the integer domain: $\forall a : int.a \geq 0 \supset \forall b : int.a + 1 = b \supset b > 0$ Note that the decidability of the satisfaction relation depends on the constraint domain. For the integer constraint domain we use here, the satisfaction relation is decidable as we do not accept nonlinear integer constraints.

The types and type schemes in DML are formed as follows. We use α for type variables and δ for type constructors. Also, we use $\vec{\tau}$ and $\vec{\imath}$ for (possibly empty) sequences of types and type indexes.

$$\begin{array}{rrcl} \text{types} & \tau & ::= & \alpha \mid (\vec{\tau})\delta(\vec{\imath}) \mid \mathbf{1} \mid \tau_1 * \tau_2 \mid \tau_1 \rightarrow \tau_2 \mid \Pi a : \gamma.\tau \mid \Sigma a : \gamma.\tau \\ \text{type schemes} & \sigma & ::= & \tau \mid \forall \alpha.\sigma \end{array}$$

For instance, *list* is a type constructor and (int)*list*(n) is the type for integer lists of length n. We use $\Pi a : \gamma.\tau$ ($\Sigma a : \gamma.\tau$) for a universal (an existential) dependent type. As an example, the universal dependent type $\Pi a : nat.($*int*$)$*list*$(a) \rightarrow ($*int*$)$*list*(a) captures the invariant of a function which, for every natural number a, returns an integer list of length

$$\frac{}{\phi \vdash \alpha \equiv \alpha} \qquad \frac{}{\phi \vdash \mathbf{1} \equiv \mathbf{1}}$$

$$\frac{\phi \models \tau_1 \equiv \tau_1' \quad \cdots \quad \tau_n \equiv \tau_n' \quad \phi \models i_1 \doteq i_1' \cdots \phi \models i_n \doteq i_n'}{\phi \vdash (\tau_1, \ldots, \tau_m)\delta(i_1, \ldots, i_n) \equiv (\tau_1', \ldots, \tau_m')\delta(i_1', \ldots, i_n')}$$

$$\frac{\phi \vdash \tau_1 \equiv \tau_1' \quad \phi \vdash \tau_2 \equiv \tau_2'}{\phi \vdash \tau_1 * \tau_2 \equiv \tau_1' * \tau_2'} \qquad \frac{\phi \vdash \tau_1' \equiv \tau_1 \quad \phi \vdash \tau_2 \equiv \tau_2'}{\phi \vdash \tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'}$$

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

Figure 3: Type conversion rules

a when given an integer list of length a. Also we use the existential dependent type $\Sigma a : nat.(\textit{int})\textit{list}(a)$ for integer lists of unknown length.

The typing rules for this language should be familiar from a dependently typed λ-calculus (such as the one underlying Coq or NuPrl). The critical notion of *type conversion* uses the judgment $\phi \vdash \tau_1 \equiv \tau_2$, which is a congruent extension of equality on index expressions to arbitrary types. The type conversion rules are listed in Figure 3. Notice that constraints may be generated when these rules are applied. For instance, the constraint $\phi \models (a + n) + 1 \doteq m + n$ is generated in order to derive $\phi \vdash (\textit{int})\textit{list}((a + n) + 1) \equiv (\textit{int})\textit{list}(m + n)$.

2.2 Pattern Matching in DML

We briefly present some formalism and a concrete example in this section to explain how a pattern matching clause is type-checked in DML.

We omit the definition for expressions e in DML, which are essentially expressions in a λ-calculus enriched with pattern matching. A pattern in DML is defined as follows, where we use x for variables, $\bullet$ for wildcard and c for constructors.[2]

$$\text{patterns } p \ ::= \ x \mid \bullet \mid c(p) \mid \langle\rangle \mid \langle p_1, p_2 \rangle \mid p_1 \ \textbf{as} \ p_2$$

We may write $c(p_1, \ldots, p_n)$ for $c(\langle p_1, \ldots, p_n \rangle)$ and c for $c(\langle\rangle)$. Note that a variable is allowed to occur at most once in a pattern. If p contains no variables (but may contain wildcards), then we call p a *constant* pattern. We use p_1 **as** p_2 for a composite pattern, where we require p_2 to be more specific than p_1 as is defined in Definition 3.1, and a value v matches the pattern p_1 **as** p_2 if v matches both p_1 and p_2.

In DML, a typing judgment is of the form $\phi; \Gamma \vdash e : \tau$, which states that the expression e can be assigned the type τ under the index variable context ϕ and the expression variable context Γ. The rule **(type-match)** for typing a pattern matching clause $p \Rightarrow e$ is given as follows:

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

[2]We assume that each constructor is unary, that is, it takes exactly one argument. For a constructor taking no argument, we can treat it as a constructor taking the unit $\langle\rangle$ as its argument.

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

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

$$\frac{p_1 \downarrow \tau \rhd (\phi_1, \Gamma_2) \quad p_2 \downarrow \tau \rhd (\phi_2, \Gamma_2)}{p_1 \textbf{ as } p_2 \downarrow \tau \rhd (\phi_1, \phi_2; \Gamma_1, \Gamma_2)} \textbf{ (pat-as)}$$

$$\frac{\mathcal{S}(c) = \Pi \vec{a} : \vec{\gamma}.(\tau \to (\vec{\alpha})\delta(\vec{\imath})) \quad p \downarrow \tau[\vec{\alpha} := \vec{\tau}] \rhd (\phi; \Gamma)}{c(p) \downarrow (\vec{\tau})\delta(\vec{j}) \rhd (\vec{a} : \vec{\gamma}, \vec{\imath} \doteq \vec{\jmath}, \phi; \Gamma)} \textbf{ (pat-cons)}$$

Figure 4: Typing rules for patterns

A judgment of the form $p \downarrow \tau \rhd (\phi; \Gamma)$, which reads *checking* p *against* τ *yields* $\phi; \Gamma$, means that if p is required to have type τ then we need to form index and expression variable contexts ϕ and Γ so that $\phi; \Gamma \vdash p : \tau$ is derivable. The rules for deriving such a judgment are listed in Figure 4, where $\mathcal{S}(c)$ denotes the type assigned to the constructor c in the signature $\mathcal{S}$. For instance, given a pattern $p = cons(\langle x, xs \rangle)$ and a type $\tau = (\alpha)\textit{list}(n)$, we can derive $p \downarrow \tau \rhd \phi; \Gamma$ for the following ϕ and Γ:

$$\phi = (a : nat, a + 1 = n) \text{ and } \Gamma = (x : \alpha, xs : (\alpha)\textit{list}(a)),$$

where we assume that $cons$ is assigned the following type scheme:

$$\Pi a : nat.\alpha * (\alpha)\textit{list}(a) \to (\alpha)\textit{list}(a + 1).$$

A pattern matching 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 required to have the type τ_1.

3 Resolving Sequentiality

In this section, we bridge the gap between dynamic and static semantics of pattern matching in DML. Given patterns $p_1, \ldots, p_m$ and p of type τ, what we need is essentially to form an index variable context ϕ to record the constraints that must be satisfied if a value is to match p but none of $p_1, \ldots, p_m$. For integer constants $i_1, \ldots, i_m$ and an integer pattern variable x, we can simply form the index variable context $\phi = (a : int, a \neq i_1, \ldots, a \neq i_m)$ if we know an integer of type *int*(a) matches x but none of $i_1, \ldots, i_m$. However, there seems no such a strategy for general patterns. Instead, we are to find patterns $p'_1, \ldots, p'_n$ such that a value matches p'_i for some $1 \leq i \leq n$ if and only if it matches p but none of $p_1, \ldots, p_m$. This task itself requires no use of dependent types. For simplicity, we assume that we are working in a simply type language ML_0 (Xi 1998), which is essentially mini-ML (Clément, Despeyroux, Despeyroux, and Kahn 1986) extended with general pattern matching. In particular, a case-expression in ML_0 is written as follows.

$$\textbf{case } e_0 \textbf{ of } p_1 \Rightarrow e_1 \mid \cdots \mid p_i \Rightarrow e_i \mid \cdots \mid p_n \Rightarrow e_n$$

A typing judgment in ML_0 is of the form $\Gamma \vdash e : \tau$, which states that expression e is given type τ under context Γ in which all free expression variables in e are declared, and values are defined as follows.

$$\text{values} \quad v \quad ::= \quad x \mid c(v) \mid \langle\rangle \mid \langle v_1, v_2 \rangle \mid \textbf{lam } x : \tau.e$$

$$\frac{e_0 \hookrightarrow v_0 \quad v_0 \downarrow p_i \rhd \theta \quad e_i[\theta] \hookrightarrow v}{\textbf{case } e_0 \textbf{ of } p_1 \Rightarrow e_1 \mid \cdots \mid p_i \Rightarrow e_i \mid \cdots \mid p_n \Rightarrow e_n \hookrightarrow v} \textbf{(case-nd)}$$

$$\frac{e_0 \hookrightarrow v_0 \quad v_0 \downarrow p_i \rhd \theta \quad v_0 \not\downarrow p_j \text{ for } 1 \leq j < i \quad e_i[\theta] \hookrightarrow v}{\textbf{case } e_0 \textbf{ of } p_1 \Rightarrow e_1 \mid \cdots \mid p_i \Rightarrow e_i \mid \cdots \mid p_n \Rightarrow e_n \hookrightarrow v} \textbf{(case-seq)}$$

Figure 5: The nondeterministic and sequential evaluation rules for case-expressions

A value is closed if it contains no free expression variables. Given a value v and a pattern p, we write $v \downarrow p \rhd \theta$ to mean that matching value v against pattern p yields a substitution θ. The rules for deriving such a judgment are given as follows.

$$\frac{}{v \downarrow \bullet \rhd []} \qquad \frac{}{v \downarrow x \rhd [x \mapsto v]}$$

$$\frac{}{\langle\rangle \downarrow \langle\rangle \rhd []} \qquad \frac{v_1 \downarrow p_1 \rhd \theta_1 \quad v_2 \downarrow p_2 \rhd \theta_2}{\langle v_1, v_2\rangle \downarrow \langle p_1, p_2\rangle \rhd \theta_1 \cup \theta_2}$$

$$\frac{v \downarrow p \rhd \theta}{c(v) \downarrow c(p) \rhd \theta} \qquad \frac{v \downarrow p_1 \rhd \theta_1 \quad v \downarrow p_2 \rhd \theta_2}{v \downarrow p_1 \textbf{ as } p_2 \rhd \theta_1 \cup \theta_2}$$

We use $[]$ for the empty substitution and $\theta[x \mapsto v]$ for the substitution that extends θ with a mapping from x to v. Also we use $\theta_1 \cup \theta_2$ for the union of two substitutions with distinct domains. We write $v \downarrow p$ if v matches p, i.e., $v \downarrow p \rhd \theta$ holds for some θ, and $v \not\downarrow p$ otherwise. We say two patterns p_1 and p_2 are disjoint if there exists no value v such that both $v \downarrow p_1$ and $v \downarrow p_2$ hold.

Definition 3.1 *Given two patterns p_1 and p_2, $p_1 \leq p_2$ holds if we can derive $p_1 \downarrow p_2$ by treating p_1 as a value. We write $p_1 < p_2$ if $p_1 \leq p_2$ but not $p_2 \leq p_1$. Intuitively, $p_1 < p_2$ means that p_1 is more specific than p_2.*

We write $e \hookrightarrow v$ to mean that expression e evaluates to v, which can be defined in the style of natural semantics (Kahn 1987). We present both nondeterministic and sequential rules for evaluating a case-expression in Figure 5. Note that DML uses the rule **(case-seq)** for evaluating a case-expression.

Given the following case-expression,

$$\textbf{case } e \textbf{ of } p_1 \Rightarrow e_1 \mid \cdots \mid p_i \Rightarrow e_i \mid \cdots \mid p_n \Rightarrow e_n$$

we are interested in finding constant patterns $p'_{i,1}, \ldots, p'_{i,n_i}$ for each p_i such that a value matches $p'_{i,j}$ for some $1 \leq j \leq n_i$ if and only if it matches p_i but none of $p_1, \ldots, p_{i-1}$. Note that a constant pattern is one that does not contain pattern variables (but may contain wildcards). This allows us to replace $p_i \Rightarrow e_i$ with a sequence of pattern matching clauses

$$p_{i,1} \textbf{ as } p_i \Rightarrow e_i \mid \cdots \mid p_{i,n_i} \textbf{ as } p_i \Rightarrow e_i.$$

while preserving the dynamic semantics of the case-expression.[3] Notice that $p_{i,j}$ and $p_{i',j'}$ must be disjoint patterns for $i \neq i'$. Therefore, this replacement closes the gap

[3]We present no proof for this fact, which, though straightforward, requires a formal definition of operational equivalence.

in DML between the dynamic semantics, where pattern matching is done sequentially, and the static semantics, where pattern matching is done nondeterministically. Clearly, for the sake of efficient type-checking, it is desirable to keep n_i as small as possible for $1 \leq i \leq n$ as this minimizes the number of constraints generated for type-checking the clauses $p_i \Rightarrow e_i$.

3.1 The Approach

We present a simple example before formally describing the proposed approach to resolving sequentiality in pattern matching. Suppose $p = \bullet$ and $p_1 = (nil, nil)$. Also suppose nil and $cons$ are the only constructors associated with the datatype $(\alpha)list$. The problem is to find patterns $p'_1, \ldots, p'_n$ such that a value v matches p but not p_1 if and only if v matches p'_j for some $1 \leq j \leq n$. In this case, the minimum value of n is 2 and p'_1, p'_2 are $(\mathit{cons}(\bullet), \bullet)$, $(\bullet, \mathit{cons}(\bullet))$. Note that p'_1 and p'_2 are *not* (required to be) disjoint.

Definition 3.2 *Given a type τ and a pattern p, p is a τ-pattern if $p \Downarrow \tau$ is derivable with the following rules.*

$$\frac{}{\vdash \bullet \Downarrow \tau} \qquad \frac{}{\vdash x \Downarrow \tau} \qquad \frac{}{\vdash \langle\rangle \Downarrow \mathbf{1}}$$

$$\frac{\vdash p_1 \Downarrow \tau_1 \quad \vdash p_2 \Downarrow \tau_2}{\vdash \langle p_1, p_2 \rangle \Downarrow \tau_1 * \tau_2} \qquad \frac{\vdash p_1 \Downarrow \tau \quad \vdash p_2 \Downarrow \tau}{\vdash p_1 \ \mathbf{as} \ p_2 \Downarrow \tau}$$

$$\frac{\mathcal{S}(c) = \tau \to (\vec{\alpha})\delta \quad \vdash p \Downarrow \tau[\vec{\alpha} := \vec{\tau}]}{\vdash c(p) \Downarrow (\vec{\tau})\delta}$$

Clearly, if $p \Downarrow \tau$ is derivable, then there exists Γ such that $\Gamma \vdash p : \tau$ is derivable.

For the rest of this section, we assume that for every τ-pattern p there exists a closed value v matching p if τ is closed, that is, τ contains no free type variables. This allows us to rule out some pathological cases.[4]

Definition 3.3 *Given n patterns $p_1, \ldots, p_n$, where $n \geq 1$, we define $p_1 \vee \cdots \vee p_n$ as a disjunctive pattern such that a value v matches this pattern if and only if v matches p_i for some $1 \leq i \leq n$. We use $[p]_\tau$ for the set of closed values of type τ that match p, and $[p_1 \vee \cdots \vee p_n]_\tau = [p_1]_\tau \cup \cdots \cup [p_n]_\tau$.*

Formally speaking, we intend to find an approach that, when given a list of τ-patterns $p, p_1, \ldots, p_n$, can yield patterns $p'_1, \ldots, p'_n$ such that

$$[p]_\tau \setminus [p_1 \vee \cdots \vee p_n]_\tau = [p'_1 \vee \cdots \vee p'_{n'}]_\tau.$$

We regard a solution to be *optimal* if it finds the minimal n'. Without loss of generality, we can assume that all the patterns contain no variables (but they may contain wildcards) in the rest of the section. Note that this assumption makes it no longer necessary to consider composite patterns.

Definition 3.4 *Given two patterns p_1 and p_2, a judgment of the form $p_1 \ \& \ p_2 \rhd p$ can be derived with the following rules.*

$$\frac{}{\vdash p \ \& \ \bullet \ \rhd p} \qquad \frac{}{\vdash \bullet \ \& \ p \ \rhd \ p} \qquad \frac{}{\vdash \langle\rangle \ \& \ \langle\rangle \rhd \langle\rangle}$$

$$\frac{\vdash p_{11} \ \& \ p_{21} \rhd p_1 \quad \vdash p_{12} \ \& \ p_{22} \rhd p_2}{\vdash \langle p_{11}, p_{12} \rangle \ \& \ \langle p_{21}, p_{22} \rangle \rhd \langle p_1, p_2 \rangle} \qquad \frac{\vdash p_1 \ \& \ p_2 \rhd p}{\vdash c(p_1) \ \& \ c(p_2) \rhd c(p)}$$

[4] In ML, it is possible to declare a datatype that contains no values and such a datatype seems useless in practice.

$$\overline{\bullet} = \emptyset$$
$$\overline{c(p)} = \{c'(\bullet) \mid \delta(c') = \delta(c) \text{ and } c' \neq c\} \cup \{c(p') \mid p' \in \overline{p}\}$$
$$\overline{\langle p_1, p_2 \rangle} = \{\langle p, \bullet \rangle \mid p \in \overline{p_1}\} \cup \{\langle \bullet, p \rangle \mid p \in \overline{p_2}\}$$
$$\overline{p_1 \vee \cdots \vee p_n} = \{p'_1 \wedge \cdots \wedge p'_n \mid p'_1 \in \overline{p_1}, \cdots, p'_n \in \overline{p_n}\}$$

Figure 6: The complement of patterns

If $p_1 \ \& \ p_2 \rhd p$ is derivable, we use $p_1 \wedge p_2$ for the pattern p; otherwise, $p_1 \wedge p_2$ is undefined. The intuition is that a value v matches both p_1 and p_2 if and only if v matches $p_1 \wedge p_2$.

Proposition 3.5 *Given patterns p_1 and p_2, we have the following.*

1. *If $v \downarrow p_1$ and $v \downarrow p_2$ for some value v, then $v \downarrow p_1 \wedge p_2$.*
2. *If $p \leq p_1$ and $p \leq p_2$ for some pattern p, then $p \leq p_1 \wedge p_2$.*

Proof This is straightforward. ■

Definition 3.6 *Let V be a set of closed values of type τ. A τ-pattern p is a (V, τ)-pattern if $[p]_\tau \subset V$. Given a (V, τ)-pattern p, if $[p]_\tau \subseteq [p']_\tau$ implies $[p]_\tau = [p']_\tau$ for every (V, τ)-pattern p', then p is (V, τ)-maximal.*

Proposition 3.7 *We have the following.*

1. *Given two τ-patterns p_1 and p_2, $p_1 \leq p_2$ implies $[p_1]_\tau \subseteq [p_2]_\tau$ for every type τ.*
2. *Let V be a set of closed values of type τ and p be a (V, τ)-pattern. Then there is a maximal (V, τ)-pattern p' such that $p \leq p'$*
3. *Let $p_1, \ldots p_n$ be τ-patterns and $V = [p_1 \vee \cdots \vee p_n]_\tau$. If p is a (V, τ)-maximal pattern, then $p_i \leq p$ holds for some $1 \leq i \leq n$.*

Proof (1) is straightforward. (2) follows the fact that there cannot exist an infinite chain like $p < p_1 < p_2 < \ldots$. (3) follows from a structural induction on p. ■

Definition 3.8 *For every constructor c in ML_0, we use $\delta(c)$ for the type constructor δ such that c is assigned a type of the form $\tau \rightarrow (\vec{\alpha})\delta$. We define the complement $\overline{p}$ of pattern p in Figure 6, which is a set of patterns. We define $p \setminus (p_1 \vee \cdots \vee p_n)$ as*

$$\{p \wedge p' \mid p' \in \overline{p_1 \vee \cdots \vee p_n}\}.$$

We assume that for each δ there are only finitely many constructors c such that $\delta(c) = \delta$. Integer patterns are handled differently as is explained at the beginning of the section.

Proposition 3.9 *We have the following.*

1. *Let $p, p_1, \ldots, p_n$ be τ-patterns for a type τ. Then every pattern in $p \setminus (p_1 \vee \cdots \vee p_n)$ is also a τ-pattern.*
2. *Given a type τ, every value in $[p]_\tau \setminus [p_1 \vee \cdots \vee p_n]_\tau$ matches some pattern in $p \setminus (p_1 \vee \cdots \vee p_n)$.*
3. *Given a value v of type τ, if v matches some pattern in $p \setminus (p_1 \vee \cdots \vee p_n)$ then v is in $[p]_\tau \setminus [p_1 \vee \cdots \vee p_n]_\tau$.*

Proof This is straightforward. ■

Lemma 3.10 *(Main Lemma) Let $p_1, \dots, p_n$ be τ-patterns and $V = [p]_\tau \setminus [p_1 \vee \cdots \vee p_n]_\tau$. Then for each given (V, τ)-pattern p' we can find a pattern $p'' \in p \setminus (p_1 \vee \cdots \vee p_n)$ such that $p' \leq p''$ holds.*

Proof By Proposition 3.5 (2), it is enough to prove this for $p = \bullet$ and $n = 1$. We proceed by a structural induction on p_1.

- $p_1 = \bullet$. Then it is trivial.
- $p_1 = c(p_{11})$. This case immediately follows from induction hypothesis on p_{11}.
- $p_1 = \langle p_{11}, p_{12} \rangle$. Then $\tau = \tau_1 * \tau_2$ for some types τ_1, τ_2. We can assume that $p' = \langle p'_1, p'_2 \rangle$ where p_i are some τ_i-patterns for $i = 1, 2$. Let $V_i = [\bullet]_{\tau_i} \setminus [p_{1i}]_{\tau_i}$ for $i = 1, 2$. We have several cases.
  - p'_1 is a (V_1, τ_1)-pattern. By induction hypothesis, there exists $p''_1 \in \overline{p_1}$ such that $p'_1 \leq p''_1$. Hence, $\langle p'_1, p'_2 \rangle \leq \langle p''_1, \bullet \rangle \in \overline{p}$.
  - p'_2 is a (V_2, τ_2)-pattern. This is similar to the previous case.
  - Neither of p'_i is a (V_i, τ_i)-pattern for $i = 1, 2$. This implies that we can find values v_i of type τ_i such that $v_i \notin V_i$ hold for $i = 1, 2$. Thus, $v_i \downarrow p'_i$ hold for $i = 1, 2$, and this yields $v = \langle v_1, v_2 \rangle \downarrow \langle p'_1, p'_2 \rangle = p'$. Also note that $v_i \in [p_{1i}]_{\tau_i}$ for $i = 1, 2$, which implies $v = \langle v_1, v_2 \rangle \in [p_1]_\tau$. This contradicts p' being a (V, τ)-pattern. Therefore, this case can never occur.

■

Theorem 3.11 *Let $p, p_1, \dots, p_n$ be τ-patterns and $V = [p]_\tau \setminus [p_1 \vee \cdots \vee p_n]_\tau$ for a given type τ. Let $p'_1, \dots, p'_m$ be a list of patterns in $p \setminus (p_1 \vee \cdots \vee p_n)$ such that (a) $p'_i \not\leq p'_j$ for $1 \leq i \neq j \leq m$ and (b) for every $p' \in p \setminus (p_1 \vee \cdots \vee p_n)$ we have $p' \leq p'_k$ for some $1 \leq k \leq m$. Then $V = [p'_1 \vee \cdots \vee p'_m]_\tau$. If $V = [p''_1 \vee \cdots \vee p''_{m'}]_\tau$ for some patterns $p''_1, \cdots, p''_{m'}$, then $m \leq m'$.*

Proof By Lemma 3.10, it is clear that every p'_i is (V, τ)-maximal for $i = 1, \dots, m$.

Assume $v \in V$. By Proposition 3.9 (2), v matches some p' in $p \setminus (p_1 \vee \cdots \vee p_n)$, that is, $v \in [p']_\tau$. Since $p' \leq p'_k$ for some k, we have $v \in [p'_k]_\tau$ by Proposition 3.7 (1). Hence, $V \subseteq [p'_1]_\tau \cup \ldots \cup [p'_m]_\tau = [p'_1 \vee \cdots \vee p'_m]_\tau$. Obviously, $[p'_1 \vee \cdots \vee p'_m]_\tau \subseteq V$ by Proposition 3.9 (3). Therefore, $V = [p'_1 \vee \cdots \vee p'_m]$.

Assume $V = [p''_1 \vee \cdots \vee p''_{m'}]$. For every $1 \leq i \leq m'$, $p''_i \leq p'_{k_i}$ holds for some $1 \leq k_i \leq m'$ by the definition of $p'_1, \dots, p'_m$ and Lemma 3.10. Therefore, $V = [p'_{k_1} \vee \cdots \vee p'_{k_{m'}}]$. Assume $m' < m$. Then there exists $1 \leq j \leq m$ such that $j \neq k_i$ for every $1 \leq i \leq m'$. Since p'_j is (V, τ)-maximal, we have $p'_{k_i} \leq p'_j$ for some $1 \leq i \leq m'$ by Proposition 3.7 (3). This contradicts to the definition of $p'_1, \dots, p'_m$. Thus, $m \leq m'$. ■

Therefore, given patterns $p, p_1, \dots, p_n$, Theorem 3.11 gives us a method to compute patterns $p'_1, \dots, p'_{n'}$ such that a value v matches some p'_i for $1 \leq i \leq n$ if and only if v matches p but none of $p_1, \dots, p_n$ and this method always yields the minimal n'.

Note that the presented approach does not apply to integer patterns since there are infinitely many integer constants. However, there is also no need for applying the approach to integer patterns since we can use the simple strategy at the beginning of this section to deal with integer patterns.

```
fun restore (R(R t, y, c), z, d) = R(B t, 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 t)) = R(B(a, x, b), y, B t)
  | restore t == B t (* == : indication for resolving sequentiality *)
```

Figure 7: An example in DML

3.2 Some Applications

The code in Figure 7 is extracted from a red-black tree implementation in DML. The function *restore* essentially restores through tree rotations some invariants of a red-black tree that are destroyed after an element is inserted.

We find it useful to allow the programmer to decide whether sequentiality in pattern matching needs to be resolved. If the programmer knows pattern matching in some implementation can be done nondeterministically (and may want to *test* it), then there is simply no need for resolving sequentiality. For instance, it is clearly such a case where no dependent datatype are involved in pattern matching. This is also in line with the design methodology behind DML: the programmer should not pay for what is not used. Nonetheless, we emphasize that the programmer can always choose to resolve sequentiality when it is unclear whether this is needed. In Figure 7, the programmer uses the syntax to indicate that only the last clause needs to be expanded into a sequence of clauses for resolving sequentiality in pattern matching while there is no need to do so for the first four clauses. In this case, the last clause expands into 36 clauses of the form

$$\textit{restore}(t \ \ \textbf{as} \ \ (p_1, \_, p_2)) = B(t),$$

where p_1 and p_2 range over the following 6 patterns: E, $B(\_)$, $R(E, \_, E)$, $R(E, \_, B(\_))$, $R(B(\_), \_, E)$, and $R(B(\_), \_, B(\_))$. If the programmer is required to resolve sequentiality in pattern matching manually, it is not only error-prone but can also make a program less readable and probably cause a compiler to produce inferior code.

The approach to resolving sequentiality in pattern matching is also useful for detecting the exhaustiveness of a sequence of patterns with respect to a given type. For instance, the following code implements a function that zips together two lists of *the same* length.

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

With the presented approach, we can find the patterns $(\textit{cons}(\_), \textit{nil})$ and $(\textit{nil}, \textit{cons}(\_))$ such that any pair of lists matches one of them if and only if the pair matches neither $(\textit{nil}, \textit{nil})$ nor $(\textit{cons}(x, \textit{xs}), \textit{cons}(y, \textit{ys}))$. However, neither $(\textit{cons}(\_), \textit{nil})$ nor $(\textit{nil}, \textit{cons}(\_))$ can have a type of the form $(\tau_1)\textit{list}(n) * (\tau_2)\textit{list}(n)$ for any natural number n. Therefore, we can simply conclude that the pattern matching clauses in the definition of *zip* are exhaustive. Please see (Xi 1999a) for more details. As it is frequent to encounter pattern matching failure in practice, it is often a standard feature in many compilers to perform pattern matching exhaustiveness detection. Therefore, we expect that this feature in the presence of dependent datatypes can be of greater value for catching program errors at compile-time since data structures can now be modeled with more accuracy.

4 Tag Check Elimination

Pattern matching compilation is essentially a process to transform pattern matching into a sequence of elementary *if-then-else* checks on tags.

4.1 The Approach

We present some basics on tag check elimination in the presence of dependent types. An expression in ML_0 is internally represented as a syntax tree and a position in a syntax tree is of the form $\circ.i_1.\ldots.i_n$, where $\circ$ stands for the root position and each i_j indicates that we take the i_jth branch of the current node (branch numbering starts from 0). Given an expression e, the subexpression of e at position pos is the expression represented by the subtree of the syntax tree of e at position pos.

For instance, we can generate the tag test tree in Figure 8 (with both node 5 and node 6 uncrossed at first) for compiling the function *zip*. Given a value v, we can use the tree to perform pattern matching as follows. At the root, we assume that the value matches the pattern $(\bullet, \bullet)$, that is, the value is a pair. We then check the tag of the subexpression of v at position $\circ.0$, that is, the left component of the pair, reaching either node 2 or 3 depending on whether the tag stands for nil or $cons$. The rest of the nodes can be explained similarly. In general, every node in a test tree contains a pattern which the value must match when tag checking reaches that node, and every inner node also contains a position that indicates the subexpression on which the subsequent tag check should be performed.

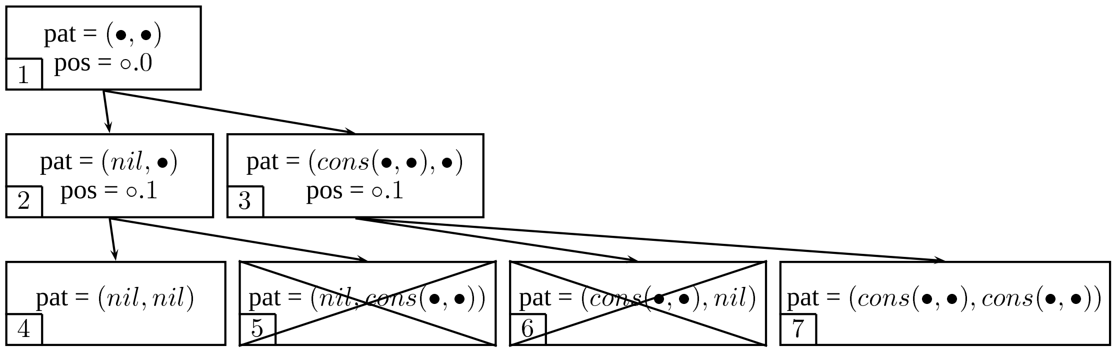
pat = $(\bullet, \bullet)$
pos = $\circ.0$
1
pat = $(nil, \bullet)$
pos = $\circ.1$
2
pat = $(cons(\bullet, \bullet), \bullet)$
pos = $\circ.1$
3
pat = (nil, nil)
4
pat = $(nil, cons(\bullet, \bullet))$
5
pat = $(cons(\bullet, \bullet), nil)$
6
pat = $(cons(\bullet, \bullet), cons(\bullet, \bullet))$
7

Figure 8: A test tree for pattern matching compilation

We now use dependent types to prune the test tree in Figure 8. Note that the test tree is generated for values of type $\tau = ((\alpha)\textit{list}(n), (\alpha)\textit{list}(n))$, where n is an index variable of sort nat. Therefore, a leaf node is reachable only if the pattern attached to it can be assigned the type τ. For $p = (nil, cons(\bullet, \bullet))$, we can derive $p \Downarrow \tau \rhd (\phi; \cdot)$ with the rules in Figure 4, where ϕ is $a : nat, a + 1 = n, 0 = n$. Clearly, ϕ is a contradictory context since no number n can be both 0 and $a + 1$ for some natural number a. This implies that no pair of lists of equal length can match p as is proven in (Xi 1999a). We thus cross out leaf node 5 since it is unreachable. Similarly, we can cross out leaf node 6. Since there is only one node coming out of the node 2, there is no tag check necessary for that node. Thus, we replace node 2 with node 4. Similarly, we replace node 3 with node 7. The final test tree is given in Figure 9, which means that we need only one tag check on the left component of a pair of lists to determine whether the pair matches (nil, nil) or $(cons(\bullet, \bullet), cons(\bullet, \bullet))$.

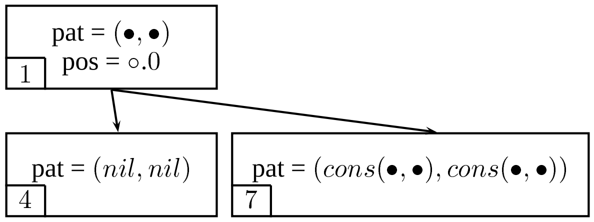
pat = $(\bullet, \bullet)$
pos = $\circ.0$
1
pat = (nil, nil)
4
pat = $(cons(\bullet, \bullet), cons(\bullet, \bullet))$
7

Figure 9: The final test tree

```
fun evaluate e = eval(e, [])

and eval (Zero(e), env) = let
  val ValInt i = eval (e, env)
in ValBool (i = 0) end
... ...
```

Figure 10: Code fragment for an interpreter

The general strategy for tag check elimination can be described as follows. Let e be a case-expression **case** e_0 **of** $p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n$ in a well-typed program P in DML. In the typing derivation that establishes the well-typedness of P, we can find typing derivations of $\phi; \Gamma \vdash e_0 : \tau_0$ and $\phi; \Gamma \vdash p_i \rhd e_i : \tau_0 \to \tau$ $(1 \leq i \leq n)$ for some index variable context ϕ, expression variable context Γ and types τ_0 and τ. Given patterns $p_1, \ldots, p_n$, we construct a test tree for patterns $p_1, \ldots, p_n$ as is described above.[5] For each leaf node, we check the attached pattern p against τ_0, deriving a judgment of the form $p \Downarrow \tau_0 \rhd (\phi_0, \Gamma_0)$. If the context ϕ, ϕ_0 is contradictory, we cross out the leaf node. We then cross out an inner node if all branches coming out of it are crossed out. Finally, if there is only one branch coming out of an inner node, we replace the inner node with the branch. The resulting test tree is then to be used for compiling the case-expression.

4.2 An Example

In the implementation of an interpreter for a programming language, it is often necessary to use tags to distinguish values, namely, evaluation results, of different types.

Let us declare two datatypes *exp* and *value* representing expressions (in the object language) that are to be evaluated and results that are to be returned from evaluation, respectively. For instance, we use *Zero*(e) for the zero test on expression e and *ValInt*(i) for an integer result.

```
datatype exp = ... | Zero of exp | ...
datatype value =
  ... | ValBool of bool | ValInt of int | ...
```

In Figure 10, the code fragment illustrates that the evaluation function *evaluate* calls function *eval*, which takes an expression and an environment (represented as a list of values) binding the free variables in the expression to values, and returns a value. The code also shows the evaluation of an expression of the form *Zero*(e), that is, the zero test on e; when *eval*$(e,$ *env*$)$ returns, we need to check the tag of the returned value to determine whether it

[5]There are various methods for doing this such as the ones in (MacQueen and Baudinet 1985; Leroy 1990) or Wadler's Chapter in (Peyton Jones 1987)

	last	nth	zip	evaluate
sml/nj	4.00/3.36 (16%)	2.03/1.34 (34%)	0.70/0.49 (30%)	3.02/2.91 (4%)
ocaml	12.11/11.45 (5%)	18.0/14.9 (21%)	2.11/2.05 (3%)	9.84/9.63 (2%)
ocamlopt	2.35/2.34(< 1%)	1.15/1.14 (< 1%)	1.40/1.40 (0%)	1.35/1.20 (11%)

Figure 11: Some experiment results on tag check elimination

represents *ValInt*; if it is, we extract out the integer result i and return either *ValBool*$($*true*$)$ or *ValBool*$($*false*$)$ depending on whether i is 0. There are usually a vast number of such tag checks during evaluation.

Now suppose that the interpreter is written for some typed programming language $\mathbf{Lam}$ and can only be applied to an expression that represents a well-typed program in $\mathbf{Lam}$. This means that e should always stand for an integer expression in $\mathbf{Lam}$ when *Zero*(e) is formed and *eval*$(e,$ *env*$)$ should return a value representing an integer, namely, a value that matches the pattern *ValInt*(i). Thus, we should be able to eliminate the tag check when compiling the following line in the definition of *eval*.

```
val ValInt i = eval (e, env)
```

We can indeed use the type system of DML to capture the above reasoning. The basic idea is to refine the datatype *exp* into *exp*(t, c) such that each expression of type *exp*(t, c) stands for a term in $\mathbf{Lam}$ that is of type t under context c, where a context is represented as a list of types in $\mathbf{Lam}$. Similarly, we refine *value* into *value*(t). We can then assign *evaluate* the following type,

```
{t:typ} exp(t, Empty) -> value(t)
```

which states that *evaluate* returns a result representing a value of type t in $\mathbf{Lam}$ when given an expression representing a closed term of type t in $\mathbf{Lam}$ (note that *Empty* stands for an empty context). We remark that `typ` is a sort (not a type) in DML for type index expression representing types in $\mathbf{Lam}$.

5 Experimentation

The approach to resolving sequentiality in pattern matching in Section 3 has already been implemented in DML and it is frequently used in practice.

The method in Section 4 for eliminating tag checks during pattern compilation has yet to be implemented. Currently, the dependent types in a DML program are erased after type-checking and this erasure makes the DML program a well-typed ML program, which can then be compiled using an existing ML compiler. Unfortunately, dependent types are needed to be present for tag check elimination. Thus, it is currently difficult to adopt the method into an existing ML compiler, though the implementation of the method itself seems straightforward.

There are nonetheless some unsafe features in SML/NJ (Unsafe.cast) and Objective Caml (Obj.magic) allowing us to experiment with tag check elimination and measure the potential performance gains.

All of our experiments are performed on a machine with a Pentium 550 MHz CPU running Linux Redhat (version 5.2). The three sets of data in Figure 11 are collected using the SML/NJ compiler (version 110.0.3), the OCAML bytecode compiler (version 2.02) and the OCAML native code compiler (version 2.02). We give some brief description on the tested programs.

last We locate the last element of an integer list of length $10,000$ repeatedly for $10,000$ times.

nth We find the nth element in an integer list of length $10,000$ for $n = 0, 1, \ldots, 9999$.

zip We zip together two integer lists of length $10,000$ repeatedly for 100 times.

evaluate We use an interpreter for Lam to evaluate the 30th Fibonacci number.

rbtree We form a red-black tree containing $100,000$ distinct natural numbers chosen randomly. The implementation of red-black trees is largely adopted from (Okasaki 1998).

We use the format t_1/t_2 $(n\%)$ in Figure 11 to indicate that it takes t_1 (t_2) seconds to run the experiment without (with) tag check elimination and t_2 is $n\%$ less than t_1. Garbage collection time is excluded when SML/NJ is used and included otherwise.

Tag check elimination leads to virtually no gain in the case of *last*, *nth* and *zip* when the OCAML native compiler is used. This is not surprising since what is eliminated is simply a conditional test and no memory instruction is involved. On the other hand, the significant gains in these cases when the SML/NJ compiler is used seem to indicate that tag check elimination may have interacted with other compiler optimizations such as loop unrolling.

Though the gains are marginal at best in some of the presented cases, we feel that tag check elimination is justified as (a) the main machinery for tag check elimination is already set up during type-checking, (b) tag check elimination *always* removes dead code and thus leads to more efficient code that is of smaller size, and (c) tag check elimination becomes necessary if we intend to build a certifying compiler that can certify that no match failure can result from the code generated from a set of exhaustive pattern matching clauses. Note that (b) is a strong point as it clearly separates tag check elimination from various heuristic compiler optimizations that make some programs run faster but slow others down.

6 Related Work

A study on pattern matching compilation in ML can be found in (MacQueen and Baudinet 1985), where heuristics are presented for arranging tag checks so as to minimize the size of generated test trees. Also pattern matching compilation for lazy evaluation is studied in (Augustsson 1985; Laville 1988; Puel and Suárez 1993). Clearly, we can always use the methods presented in these studies to generate a test tree and then further prune the test tree with the method given in Section 4.

Pattern matching with dependent types in Martin-Löf's type theory (Nordström, Petersson, and Smith 1990) is studied in (Coquand 1992). There, some sufficient conditions are presented to ensure the correctness of a function definition given through pattern matching. In this respect, the work is casually related to ours.

A type-preserving interpreter for a language like $\mathbf{Lam}$ is presented in (Augustsson and Carlsson 1999), The implementation, to which our implementation in DML bears certain similarity, is written in Cayenne, a functional programming language that extends Haskell with dependent types (Augustsson 1998). The implementation is reported to be considerably faster than a corresponding one in Haskell.

Dependent datatypes are most closely related to indexed types developed in the context of lazy functional programming (Zenger 1998) (an earlier version can be found in (Zenger 1997)), and a brief comparison between indexed types and the dependent types in DML is given in (Xi and Pfenning 1999). Also, the use of dependent types in array bound check elimination is studied in (Xi and Pfenning 1998).

7 Conclusion

The primary motivation for the introduction of dependent datatypes is to allow the programmer to express more program properties through types and thus capture more program errors at compile-time.

In DML, nondeterministic pattern matching is assumed in static semantics while sequential pattern matching is adopted in dynamic semantics. This gap makes the typing rules for pattern matching in DML too conservative as is demonstrated in many examples. We have presented and implemented an approach that can bridge the gap by resolving the sequentiality in pattern matching. We have also proven the optimality of this approach according to a reasonable criterion.

In (Xi and Pfenning 1998), it is demonstrated that run-time array bound checking in realistic programs can be effectively eliminated with the use of dependent types. In this paper, the application of dependent types in DML to compiler optimization is further demonstrated as it is shown with experimental results that dependent types can also help eliminate run-time tag checking.

In general, we are interested in the use of formal methods in language design and implementation that can lead to not only more robust but also more efficient programs. The use of a restricted form of dependent types in DML has exhibited some promising results in this direction.

8 Acknowledgment

I thank Chad Brown for proofreading a draft of this paper and providing me with some valuable comments.

References

Augustsson, L. (1985). Compiling Pattern Matching. In J.-P. Jouannaud (Ed.), *Functional Programming Languages and Computer Architecture*, Berlin, pp. 368–381. Springer-Verlag LNCS 201.

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

Augustsson, L. and M. Carlsson (1999). An excercise in dependent types: A well-typed interpreter. Available as `http://www.cs.chalmers.se/~augustss/cayenne/interp.ps`.

Clément, D., J. Despeyroux, T. Despeyroux, and G. Kahn (1986). A simple applicative language: Mini-ML. In *Proceedings of 1986 Conference on LISP and Functional Programming*, pp. 13–27.

Coquand, T. (1992, June). Pattern Matching with Dependent Types. In *Proceedings of Logical Framework Workshop at Baastad*.

Kahn, G. (1987). Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pp. 22–39. Springer-Verlag LNCS 247.

Laville, A. (1988). Implementation of Lazy Pattern Matching Algorithms. In *Proceedings of European Symposium on Programming*, pp. 298–316. Springer-Verlag LNCS 1422.

Laville, A. (1990). A Comparison of Priority Rules in Pattern Matching and Term Rewriting. *Journal of Symbolic Computation 11*(4), 321–347.

Leroy, X. (1990, Feburary). The ZINC Experiment: An Economical Implementation of the ML Language. Technical Report No. 117, INRIA.

MacQueen, D. and M. Baudinet (1985, December). Tree Pattern matching for ML. Unpublished manuscript.

Maranget, L. (1994, October). Two Techiques for Compiling Lazy Pattern Matching. Technical Report No. 2385, INRIA.

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

Nordström, B., K. Petersson, and J. M. Smith (1990). *Programming in Martin-Löf's Type Theory*, Volume 7 of *International Series of Monographs on Computer Science*. Oxford: Clarendon Press.

Okasaki, C. (1998). *Purely Functional Data Structures*. Cambridge University Press.

Peyton Jones, S. et al. (1999, February). Haskell 98 – A non-strict, purely functional language. Available at`http://www.haskell.org/onlinereport/`.

Peyton Jones, S. L. (1987). *The Implementation of Functional Programming Languages*. London: Prentice-Hall International.

Puel, L. and A. Suárez (1993). Compiling Pattern Matching by Term Decomposition. *Journal of Symbolic Computation 15*(1), 1–26.

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

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

Xi, H. (1999b, September). Dependently Typed Data Structures. In *Proceedings of Workshop on Algorithmic Aspects of Advanced Programming Languages*, Paris, France, pp. 17–33.

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

Xi, H. and F. Pfenning (1999, January). Dependent types in practical programming. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, San Antonio, Texas, pp. 214–227.

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

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