Dictionary Passing for Polytypic Polymorphism

Juan Chen and Andrew W. Appel
Princeton University

March 21, 2001

Abstract

Polytypic functions are defined by induction on the structure of types. These functions are not parametrically polymorphic in the conventional sense, but they are not entirely ad hoc either. An example is the "polymorphic" equality function of Standard ML. Current implementations of polytypic polymorphism either inhibit efficient data representations, need the programmer to supply specialized functions for concrete types, or don't generalize to the ML module system or quantified type systems. We implement polytypic polymorphism by dictionary passing. In our approach the compiler generates and passes dictionaries automatically; the kind of a type decides the shape of its dictionary. The type theory and dictionary transformations are not new, but we show that among the current tag-free approaches that implement polytypic polymorphism dictionary passing has the best integration with an existing production compiler. We show the type theory, describe an implementation, and explain how the transformation interacts with the compiler.

1 Introduction

Standard ML provides the programmer with a useful facility, polymorphic equality. The programmer can use the polymorphic equality operator = to test any two values for equality (two integers, two booleans, etc.) as long as those two values are of the same equality type. This operator has type $\forall\alpha.\alpha \to \alpha \to \text{bool}$, where α must be instantiated to an equality type. When applied to two integers, the operator = tests if the two integers are the same; when applied to two records, it tests if the records are componentwise the same. The behavior of this operator depends on the type of its arguments, in an inductive fashion.

Many other primitives – such as pretty printing, copy in garbage collection, pickling – are also parameterized by types. But they are not parametrically polymorphic – their behavior is not completely oblivious of the arguments' types, but is defined inductively based on the structure of these types. We call these operators polytypic primitives [8]. This paper focuses on implementations of these primitives.

There are several ways to compile polytypic polymorphism:

Specialization: Specialize a polymorphic primitive to its corresponding monomorphic function at each applied occurrence of the primitive. This is simple, but may cause exponential code duplication as programs must be completely monomorphized. It may not terminate in a system with polymorphic recursion. Also, specialization is not compatible with separate compilation, where a function is defined in one module and used in another.

Boxing and tagging: Make each object have a uniform representation (usually a word). Any object that does not fit in the uniform representation is boxed (a block is allocated in the heap for it and it is referred to by a pointer pointing to the block). Tag bits are used to indicate types. Standard ML of New Jersey (SML/NJ) boxes values, and uses a tag bit to distinguish pointers from non-pointers. It has a "magic" polymorphic equality function which can be applied to any two values of an equality type. If the two values are nonpointers, the magic

function compares them bitwise, otherwise it follows pointers. The disadvantages of this approach are: first, tag bits and pointers cause both space and time inefficiency; second, it inhibits efficient (tagless) data representations and interoperability with other languages that use natural representations; finally, the magic function is unsafe in that it cannot be type-checked.

Type-analyzing system: Pass types at run time, and dispatch to different code according to runtime types. This needs a runtime dispatch facility both at the term level and the type level. Type analyzingcan be divided into two categories: type passing (pass and analyze types, as in λ_i^{ML} calculus [6]), and type erasure (pass and analyze term-level type representations, as in λ_R calculus [1]). We can write a polymorphic function **peq** using the runtime code dispatch construct **typecase** in λ_i^{ML}. Peq takes a type argument and selects a branch to execute according to the type at run time.

$$\begin{array}{lrl}
\text{fun peq} = \Lambda\alpha.\text{typecase } \alpha \text{ of} & & \\
\quad \text{int} & \Rightarrow & \text{IntEq} \\
\quad \text{bool} & \Rightarrow & \text{BoolEq} \\
\quad t_1 * t_2 & \Rightarrow & \lambda x{:}t_1 * t_2.\lambda y{:}t_1 * t_2. \\
& & \quad \text{peq}[t_1]\,(\pi_1 x)(\pi_1 y) \text{ and} \\
& & \quad \text{peq}[t_2]\,(\pi_2 x)(\pi_2 y) \\
\quad \ldots & &
\end{array}$$

Dictionary passing[18]: Represent types by dictionaries. A dictionary for a type is a set of functions, much like the method suite of an object. For example, the dictionary for the integer type contains primitive functions on integers:

$$d_{int} = \{\text{eq} = \text{IntEq}, \text{pp} = \text{IntTostring}, \ldots\}$$

For each type argument of a polymorphic function, a dictionary value is passed at run time. Polytypic primitives such as polymorphic equality are replaced by specialized functions in dictionaries. A polymorphic function $f(x,y) = (x = y)$ now has one more argument, d, which is the dictionary corresponding to f's type argument. The eq function in d is substituted for the operator = in the body of f. When f is applied to two integers, the dictionary for the integer type is passed:

$$\text{fun} f(d,x,y) = d.\text{eq}(x,y)$$

$$f(d_{int}, 1, 2)$$

The disadvantages of specialization and tagging force us to resort to type analyzingor dictionary passing. λ_i^{ML}, λ_R, and dictionary passing are all variants of *intensional type analysis*, support efficient data representations, and can implement runtime services with provable safety. But they have several differences:

- Naive type analyzingin λ_i^{ML} interacts in a complicated way with closure conversion [6], but λ_R [1] uses a dependent-type calculus to solve this problem. Our dictionary passing interacts gracefully with closure conversion.
- It is difficult in λ_i^{ML} and λ_R to support quantified types (it requires a really complex type system [16, 11]); our dictionary-passing system needs no extra type theory to support quantified types.
- To restrict computations to certain types (for instance, equality can only be applied to values of equality types), type analyzinguses the type-level dispatch construct **Typerec** [6], whereas dictionary passing uses kinds instead.
- λ_i^{ML} breaks parametricity by analyzing types. λ_R restores abstraction by not providing type representations to analyze. But in order to implement polytypic primitives for abstract types (e.g., pretty-printing an abstract value without the representation of the hidden type), λ_R has to use dictionaries. Dictionary passing preserves abstraction naturally.
- Type analyzingand dictionary passing are modular in orthogonal directions; adding a new polytypic primitive (such as **toString**) is easy in type analyzing, but requires a new method in every dictionary; adding a new type constructor is easy in dictionary passing, but requires a new case-branch in every **typecase** of a type-analyzing system.
- It is easier in type analyzingto make the behavior of a polytypic function depend on the types of

two or more type arguments simultaneously; dictionaries might require "multimethods" to do this. Most common polytypic primitives have only one type argument (even if they have several term arguments), so dictionary passing can still be used in many areas.

We were first motivated to study polytypic polymorphism in order to implement tag-free polymorphic equality in SML/NJ. This production SML compiler uses a typed intermediate language (a variant of F_ω) [12], a powerful module system, but no type passing at run time or dependent types. All these features are common in compilers of functional programming languages with polymorphism. Dictionary passing is a simpler and cleaner way than other approaches in such circumstances. We will show in detail that in a production compiler that does not pass types at run time, dictionary passing combined with automatic dictionary generation and kind refinement [3] provides a full implementation of polytypic primitives, and imposes the least restrictions on the compiler. In our experience, dictionary passing can be implemented easily with almost no extra type theory and only minor changes to existing phases in the compiler.

2 Related Work

Crary et al's calculus λ_R [1] is the one most closely related to ours, where runtime type information is represented by special *R*-terms. This approach duplicates types at the term level, so that after erasing types, it still can analyze term-level type representations. The special terms are primitive, and have singleton types. Dictionaries in our system are concrete term representations of types. We adopted the rules of *R*-term generation and transformation in λ_R to construct and pass dictionaries. Details of how the implementation of λ_R differs from dictionary passing will be presented in section 3.4.

Dictionary passing is used in Haskell to implement type classes [18]. Hinze[7] proposed generic (that is, polytypic) programming which extends type class mechanism . The Haskell user defines a generic function by specifying cases for primitive types such as Int, and for primitive constructors such as '$\times$' and '$+$'. A source-to-source translator generates instances at each desired type by mimicking the structure of the type at the term level. Since generic programming aims at user-level facility, it doesn't consider universal or existential types. Our work is at the intermediate language level; it can be applied to universal and existential types, and runtime services such as copy in garbage collection.

Elsman[4] also passed dictionaries to get tag-free polymorphic equality, but his paper described only a small calculus (based on system F) with primitive types, product types, and sum types, and it imposed constraints on structures to support separate compilation. We provide a much richer calculus (based on F_ω), and our work can be easily extended to the module system. Elsman's work deals with polymorphic equality only, so it uses equality type variables to decide when to pass equality functions; our implementation supports general polytypic primitives, not just polymorphic equality, so separating equality type variables is not enough. In our system we use kinds to specify type properties.

Duggan[3] and Ohori[10] used kinds to constrain computations statically. Duggan allowed user-defined marshalling by dynamic type dispatch, as in type-analyzing. But he used kind refinement (kinds classify certain types) instead of type-level dispatch. We don't have dynamic type dispatch, and use only very simple refinement on kinds (just a new kind, no kind operations such as union and recursion in Duggan's calculus). Ohori translated polymorphic record operations to index operations, whereas these record operations can be implemented by dictionaries without restrictions on record representations.

3 Implementation

We implement dictionary passing in the production compiler SML/NJ. The SML/NJ front end translates ML source code to the typed intermediate language FLINT [12]. The "middle end" does optimizations on this intermediate format. The back end translates FLINT to continuation passing style (CPS), does closure conversion, and translates CPS to machine code. FLINT is a predicative variant of F_ω, can be type-checked. Although the CPS is untyped, an experimental version of the compiler at YALE University uses a typed low-level

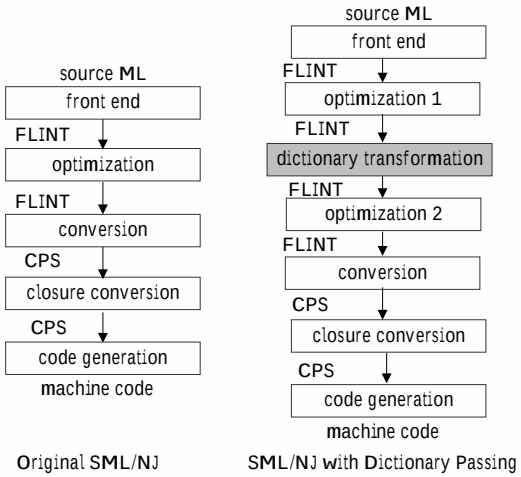

Figure 1: Pipelines of SML/NJ 110.29 without/with dictionary passing

FLINT instead, and our dictionary-passing transformation makes no new demands on this low-level system.

Our implementation of dictionary passing has added only a new kind in the type system, and a new compilation phase (dictionary transformation) in the middle end of SML/NJ. This new phase consists of about 155 lines of SML code for program transformation, 500 lines for dictionary generation, 40 lines for constructing types of dictionaries, and 25 lines for type translation. Figure 1 shows the pipelines of SML/NJ before and after adding dictionary transformation phase.

3.1 Type System

Our type system is standard. The only difference between FLINT and this calculus is that this calculus has simple kind refinement.

The intermediate language is a predicative variant of F_ω, with four syntactic classes: kinds κ, types τ, type schemes σ, and terms e, as in figure 2. Kinds classify types, and type schemes classify terms. Figure 3 shows type formation rules; figure 4 shows selected term formation rules.

The only thing new is the kind Ω_s, where s is a set of methods. Any type of this kind supports all the methods

$$
\begin{array}{lcl}
\kappa & ::= & \Omega_s \mid \kappa \to \kappa' \\
\tau & ::= & \mathrm{int} \mid \mathrm{bool} \mid \alpha \mid \tau \to \tau' \mid \lambda\alpha{:}\kappa.\,\tau \mid \tau\,\tau' \\
 & & \mid \tau_1 \times \ldots \times \tau_n \mid \tau_1 + \ldots + \tau_n \mid \mu\tau \\
\sigma & ::= & \tau \mid \sigma \to \sigma' \mid \sigma_1 \times \ldots \times \sigma_n \\
 & & \mid \forall\alpha_1 : \kappa_1, \ldots, \alpha_n : \kappa_n.\,\sigma \\
 & & \mid \{l_1 : \sigma_1, \ldots, l_n : \sigma_n\} \\
e & ::= & n \mid \mathrm{true} \mid \mathrm{false} \mid x \mid\ = \\
 & & \mid \lambda x{:}\sigma.\,e \mid e_1\,e_2 \mid \mathrm{fix}\,x : \sigma.e \\
 & & \mid \Lambda\alpha_1 : \kappa_1, \ldots, \alpha_n : \kappa_n.\,e \mid e[\tau_1, \ldots, \tau_n] \\
 & & \mid (e_1, \ldots, e_n) \mid \pi_i\,e \\
 & & \mid \mathrm{fold}[\sigma]e \mid \mathrm{unfold}[\sigma]e \\
 & & \mid \{l_1 = e_1, \ldots, l_n = e_n\} \mid e.l_i \\
 & & \mid \mathrm{inj}_i^\sigma\,e \mid \mathrm{switch}\,e\,\mathrm{of}\,(e_1, \ldots, e_n)
\end{array}
$$

Figure 2: Syntax of IL

in s. To simplify the calculus, we discuss only pretty-printing and equality methods, and use Ω as an abbreviation for $\Omega_{\{\mathrm{pp}\}}$, and Ω_{eq} for $\Omega_{\{\mathrm{pp,eq}\}}$.

We have kind Ω for monotypes such as integer and function types, and function kind $\kappa \to \kappa'$ for type constructors such as *list*. Standard ML divides types into two categories, those which admit equality (equality types) and those which might not. We use Ω_{eq} for equality types and Ω for general types; thus the subscript (method suite) corresponds to the equality property of the type. In our type system, a type t admits equality if it is of kind Ω_{eq}, or of kind $\kappa \to \kappa'$ if $t\,t'$ admits equality for any type t' of kind κ.

The term-reduction rules of our calculus are entirely standard: beta-reduction, type-beta-reduction, tuple or record projection, and case analysis of sum types (figure 5).

Lemma 1 (Value Substitution) *Typing judgements are preserved under substitution:*

$$\frac{\Delta; \Gamma, x : \sigma' \vdash e : \sigma \quad \Delta; \Gamma \vdash e' : \sigma'}{\Delta; \Gamma \vdash e[e'/x] : \sigma}$$

Proof: by induction over the structure of expressions.

Lemma 2 (Type Substitution) *Kinding judgements are preserved under type substitution:*

$$\frac{\Delta, \alpha : \kappa' \vdash \tau : \kappa \quad \Delta \vdash \tau' : \kappa'}{\Delta \vdash \tau[\tau'/\alpha] : \kappa}$$

$$\frac{}{\vdash \text{int} : \Omega_{\text{eq}}} \quad \frac{}{\vdash \text{bool} : \Omega_{\text{eq}}} \qquad \frac{}{\Delta \vdash \alpha : \Delta(\alpha)} \quad \frac{\Delta \vdash \tau : \Omega_{\text{eq}}}{\Delta \vdash \tau : \Omega}$$

$$\frac{\Delta, \alpha{:}\kappa \vdash \tau : \kappa'}{\Delta \vdash \lambda\alpha{:}\kappa.\tau : \kappa \to \kappa'} \qquad \frac{\Delta \vdash \tau : \kappa \to \kappa' \quad \Delta \vdash \tau' : \kappa}{\Delta \vdash \tau\tau' : \kappa'}$$

$$\frac{\Delta \vdash \tau_1 : \Omega_s, \ldots, \Delta \vdash \tau_n : \Omega_s}{\Delta \vdash \tau_1 \times \ldots \times \tau_n : \Omega_s} \qquad \frac{\Delta \vdash \tau_1 : \Omega_s, \ldots, \Delta \vdash \tau_n : \Omega_s}{\Delta \vdash \tau_1 + \ldots + \tau_n : \Omega_s}$$

$$\frac{\Delta \vdash \tau : \Omega \quad \Delta \vdash \tau' : \Omega}{\Delta \vdash \tau \to \tau' : \Omega} \qquad \frac{\Delta \vdash \tau : \Omega_s \to \Omega_s}{\Delta \vdash \mu\tau : \Omega_s}$$

Figure 3: Type Formation: $\Delta \vdash \tau : \kappa$

$$\frac{}{\vdash n : \text{int}} \quad \frac{}{\Delta; \Gamma \vdash x : \Gamma(x)} \quad \frac{}{\vdash \text{true} : \text{bool}} \quad \frac{}{\vdash \text{false} : \text{bool}}$$

$$\frac{}{\vdash = : \forall\alpha : \Omega_{\text{eq}}.\alpha \to \alpha \to \text{bool}} \qquad \frac{\Delta; \Gamma, x{:}\sigma \vdash e : \sigma}{\Delta; \Gamma \vdash \text{fix}\, x : \sigma.e : \sigma}$$

$$\frac{\Delta, \alpha_1 : \kappa_1, \ldots, \alpha_n : \kappa_n; \Gamma \vdash e : \sigma}{\Delta; \Gamma \vdash \Lambda\alpha_1 : \kappa_1, \ldots, \alpha_n : \kappa_n.e : \forall\alpha_1 : \kappa_1, \ldots, \alpha_n : \kappa_n.\sigma}$$

$$\frac{\Delta; \Gamma \vdash e : \forall\alpha_1 : \kappa_1, \ldots, \alpha_n : \kappa_n.\sigma \qquad \Delta \vdash \tau_1 : \kappa_1, \ldots, \Delta \vdash \tau_n : \kappa_n}{\Delta; \Gamma \vdash e[\tau_1, \ldots, \tau_n] : \sigma[\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]}$$

Figure 4: Selected Rules of Expression Formation: $\Delta; \Gamma \vdash e : \sigma$

$$\begin{array}{lcl}
(\lambda x{:}\sigma.e)\, e' & \mapsto & e[e'/x] \\
(\text{fix}\, x : \sigma.e)\, e' & \mapsto & e[(\text{fix}\, x : \sigma.e)/x]\, e' \\
(\Lambda\alpha_1 : \kappa_1, \ldots, \alpha_n : \kappa_n.e)[\tau_1, \ldots, \tau_n] & \mapsto & e[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n] \\
\pi_i\,(e_1, \ldots, e_n) & \mapsto & e_i \\
\text{switch}\,(\text{inj}_i^{\sigma}\, e)\,\text{of}\,(e_1, \ldots, e_n) & \mapsto & e_i\, e \\
\{l_1 = e_1, \ldots, l_n = e_n\}.l_i & \mapsto & e_i
\end{array}$$

Figure 5: Term Reduction Rules

$$
\begin{array}{lcl}
T(\alpha : \Omega_{\{pp\}}) & = & \{pp : \alpha \to string\} \\
T(\alpha : \Omega_{\{pp,eq\}}) & = & \{pp : \alpha \to string, \\
 & & \quad eq : \alpha \to \alpha \to bool\} \\
T(\alpha : \kappa \to \kappa') & = & \forall \beta : \kappa . T(\beta : \kappa) \to T(\alpha\beta : \kappa')
\end{array}
$$

Figure 6: Type of Dictionaries

Proof: by induction over the structure of types.

Theorem 3 (Preservation)

$$\frac{\Delta; \Gamma \vdash e : \sigma \quad e \mapsto e'}{\Delta; \Gamma \vdash e' : \sigma}$$

Proof: by induction over term reduction rules.

3.2 Automatic Dictionary Generation

Each type has a dictionary of its own, which is a collection of functions for this type. The compiler constructs dictionaries for all types automatically.

The shape of a dictionary depends only on the corresponding type. For example, functions types do not admit equality, so their dictionaries should not contain equality functions.

3.2.1 Shape of Dictionaries

The criterion that decides the shape of a dictionary is the kind of its corresponding type. For types of kind $\Omega_{\{pp\}}$, dictionaries contain pretty-printing functions but no equality functions. For types of kind $\Omega_{\{pp,eq\}}$, dictionaries include both. For a type function F of kind $\kappa \to \kappa'$, its dictionary is a polymorphic function, which takes any type t of kind κ and t's dictionary d, returns the dictionary for type $F\,t$.

Figure 6 shows how to decide the shape of a dictionary based on the kind. The dictionary for type α at kind κ has type $T(\alpha : \kappa)$. In our system, a type can have several kinds – for example, int has kinds $\Omega_{\{pp\}}$ and $\Omega_{\{pp,eq\}}$ – and the choice of kind (thus the choice of dictionary) depends on the context.

3.2.2 Dictionary Generation

The compiler generates dictionaries automatically. Figure 7 shows some rules of dictionary generation. $\Delta; \Theta \vdash_d \tau : \kappa \mapsto d$ means that under kind environment Δ and dictionary environment Θ, type τ of kind κ has dictionary d. Δ maps type variables to kinds, and Θ maps type variables to dictionary variables (term variables representing dictionaries). For each kind Ω_s, the primitive dictionaries for int and bool type are $DInt_s$ and $DBool_s$, of type $T(\text{int} : \Omega_s)$ and $T(\text{bool} : \Omega_s)$ respectively. $Prod_s^n$ and Sum_s^n are dictionary generators for product types and sum types of n elements. Applying generator $Prod_s^n$ to n types $\tau_1, \ldots, \tau_n$ of kind Ω_s and the corresponding dictionaries $d_1, \ldots, d_n$, we get the dictionary for $\tau_1 \times \ldots \times \tau_n$.

$$d_{(\tau_1 \times \ldots \times \tau_n : \Omega_s)} = Prod_s^n[\tau_1, \ldots, \tau_n]\, d_1 \ldots d_n$$

These generators construct dictionaries from element dictionaries in the most natural way. To get the equality function for $\tau_1 \times \ldots \times \tau_n$, $Prod_{\{pp,eq\}}^n$ uses the first dictionary d_1 to compare the first element, the second dictionary d_2 for the second element, and so on.

$$
\begin{array}{l}
Prod_{\{pp,eq\}}^n = \\
\quad \Lambda\alpha_1 : \Omega_{\{pp,eq\}}, \ldots, \alpha_n : \Omega_{\{pp,eq\}} . \\
\quad \lambda d_1 : T(\alpha_1 : \Omega_{\{pp,eq\}}) . \cdots \lambda d_n : T(\alpha_n : \Omega_{\{pp,eq\}}) . \\
\quad \{eq = \lambda x : \sigma_1 \times \ldots \times \sigma_n . \lambda y : \sigma_1 \times \ldots \times \sigma_n . \\
\quad\quad d_1.eq\,(\pi_1 x)(\pi_1 y) \text{ and } \ldots \text{ and } d_n.eq\,(\pi_n x)(\pi_n y), \\
\quad\quad pp = \ldots\}
\end{array}
$$

Define $|\Theta|$ as $|\Theta|(d) = T(\alpha : \kappa)$ if $\Theta(\alpha : \kappa) = d$, i.e., $|\Theta|$ maps dictionary variables to their types. It is easy to prove by induction that:

Lemma 4 (Type of Dictionaries) *The generated dictionaries are correctly typed.*

$$\frac{\Delta; \Theta \vdash_d \tau : \kappa \mapsto d}{\Delta; |\Theta| \vdash d : T(\tau : \kappa)}$$

3.2.3 Pretty Printing

Our dictionary-passing system is a general framework within the compiler in which many polytypic primitives can be implemented automatically. Here we use the examples of polymorphic equality (limited to types of kind

$$\frac{}{\vdash_{\mathrm{d}} \mathrm{int} : \Omega_s \mapsto \mathrm{DInt}_s} \qquad \frac{}{\Delta; \Theta \vdash_{\mathrm{d}} \alpha : \kappa \mapsto \Theta(\alpha)}$$

$$\frac{\Delta; \Theta \vdash_{\mathrm{d}} \tau : \Omega_s \to \Omega_s \mapsto d_\tau}{\Delta; \Theta \vdash_{\mathrm{d}} \mu\tau : \Omega_s \mapsto \mathrm{fix}\, x : T(\mu\tau : \Omega_s).d_\tau[\mu\tau]\, x}$$

$$\frac{\Delta, \alpha : \kappa; \Theta, \alpha : d_\alpha \vdash_{\mathrm{d}} \tau : \kappa' \mapsto d_\tau}{\Delta; \Theta \vdash_{\mathrm{d}} \lambda\alpha{:}\kappa.\tau : \kappa \to \kappa' \mapsto \Lambda\alpha{:}\kappa.\lambda d_\alpha{:}T(\alpha : \kappa).d_\tau}$$

$$\frac{\Delta; \Theta \vdash_{\mathrm{d}} \tau : \kappa \to \kappa' \mapsto d_\tau \quad \Delta; \Theta \vdash_{\mathrm{d}} \tau' : \kappa \mapsto d'_\tau}{\Delta; \Theta \vdash_{\mathrm{d}} \tau\tau' : \kappa' \mapsto d_\tau[\tau']\, d'_\tau}$$

$$\frac{\Delta; \Theta \vdash_{\mathrm{d}} \tau_1 : \Omega_s \mapsto d_1, \ldots, \Delta; \Theta \vdash_{\mathrm{d}} \tau_n : \Omega_s \mapsto d_n}{\Delta; \Theta \vdash_{\mathrm{d}} \tau_1 \times \ldots \times \tau_n : \Omega_s \mapsto \mathrm{Prod}^n_s[\tau_1, \ldots, \tau_n]\, d_1 \ldots d_n}$$

Figure 7: Selected Rules of Dictionary Generation: $\Delta; \Theta \vdash_{\mathrm{d}} \tau : \kappa \mapsto d$

$\Omega_{\{\mathrm{pp,eq}\}}$) and pretty printing (available at all types, i.e., $\Omega_{\{\mathrm{pp}\}}$). The pretty-printing primitive *toString* has type $\forall\alpha : \Omega_{\{\mathrm{pp}\}}.\alpha \to \mathrm{string}$. Pretty-printing functions are generated just like equality functions:

$$\begin{array}{l}
\mathrm{Prod}^n_{\{\mathrm{pp}\}} = \\
\quad \Lambda\alpha_1 : \Omega_{\{\mathrm{pp}\}}, \ldots, \alpha_n : \Omega_{\{\mathrm{pp}\}}. \\
\quad\ \lambda d_1{:}T(\alpha_1 : \Omega_{\{\mathrm{pp}\}}).\cdots\lambda d_n{:}T(\alpha_n : \Omega_{\{\mathrm{pp}\}}). \\
\quad\quad \{\mathrm{pp} = \lambda x{:}\sigma_1 \times \ldots \times \sigma_n. \\
\quad\quad\quad \text{"}\langle\text{"} \mathbin{^\wedge} (d_1.\mathrm{pp}\ (\pi_1 x)) \mathbin{^\wedge} \text{","} \mathbin{^\wedge} \ldots \mathbin{^\wedge} \text{","} \\
\quad\quad\quad\quad \mathbin{^\wedge} (d_n.\mathrm{pp}\ (\pi_n x)) \mathbin{^\wedge} \text{"}\rangle\text{"}\}
\end{array}$$

3.3 Dictionary Transformation

Dictionary passing adds a new compilation phase, dictionary transformation, in the middle end of SML/NJ. The new phase is almost independent of other phases in the compiler.

We implement dictionary transformation as a type-directed and type-preserving program transformation at the intermediate language level. The main idea is to pass one additional value argument (the dictionary) per type argument to polymorphic function definitions and applications, and replace the polytypic primitives with corresponding dictionary functions (see figure 8; $\vec{\alpha} : \vec{\kappa}$ is a shortcut for $\alpha_1 : \kappa_1, \ldots, \alpha_n : \kappa_n$). The type scheme of a polymorphic function is changed because of these extra arguments, so we translate the type scheme as well (see figure 9; $|\sigma| = \sigma'$ means type scheme σ is translated to σ').

In figure 8, $\Delta;\Gamma;\Theta \vdash e : \sigma \Rightarrow e'$ means that under kind environment Δ (which maps type variables to kinds), type environment Γ (which maps term variables to types), and dictionary environment Θ (which maps type variables to term variables representing their dictionaries), expression e of type scheme σ is translated to expression e'.

Define $|\Gamma|$ as $|\Gamma|(x) = |\Gamma(x)|$. The type environment $|\Gamma|$ maps term variables to their translated types; as defined in section 3.2.2, $|\Theta|$ maps term variables representing dictionaries to their types.

We define environment union $\Gamma_1 + \Gamma_2$ as:

$$(\Gamma_1 + \Gamma_2)(x) = \begin{cases} \Gamma_1(x) & \text{if } x \text{ in domain}(\Gamma_1) \\ \Gamma_2(x) & \text{otherwise} \end{cases}$$

It is easy to show that

Lemma 5 $|\Gamma, x : \sigma| + |\Theta| = |\Gamma| + |\Theta|, x : |\sigma|$.

Proof: by the definitions of $|\Gamma|$, $|\Theta|$, and $\Gamma_1 + \Gamma_2$.

Lemma 6 $|\Gamma| + |\Theta, \alpha : d| = (|\Gamma| + |\Theta|), d : T(\alpha : \kappa)$ *if* d *does not occur in* Γ, *and* α *is of kind* κ.

Proof: by the definitions of $|\Gamma|$, $|\Theta|$, and $\Gamma_1 + \Gamma_2$.

$$\frac{}{\vdash n : \text{int} \Rightarrow n}\ (Int) \qquad \frac{}{\vdash =: \forall\alpha : \Omega_{\text{eq}}.\alpha \to \alpha \to \text{bool} \Rightarrow \Lambda\alpha{:}\Omega_{\text{eq}}.\lambda d{:}T(\alpha : \Omega_{\text{eq}}).d.\text{eq}}\ (Peq)$$

$$\frac{}{\Delta;\Gamma;\Theta \vdash x : \sigma \Rightarrow x}\ (Var) \qquad \frac{\Delta;\Gamma,x:\sigma;\Theta \vdash e : \sigma' \Rightarrow e'}{\Delta;\Gamma;\Theta \vdash \lambda x{:}\sigma.e : \sigma \to \sigma' \Rightarrow \lambda x{:}|\sigma|.e'}\ (Abs)$$

$$\frac{\Delta;\Gamma;\Theta \vdash e_1 : \sigma \to \sigma' \Rightarrow e_1' \quad \Delta;\Gamma;\Theta \vdash e_2 : \sigma \Rightarrow e_2'}{\Delta;\Gamma;\Theta \vdash e_1\,e_2 : \sigma' \Rightarrow e_1'\,e_2'}\ (App)$$

$$\frac{\Delta,\vec{\alpha} : \vec{\kappa};\Gamma;\Theta,\alpha_1 : d_1,\ldots,\alpha_n : d_n \vdash e : \sigma \Rightarrow e' \quad d_1,\ldots,d_n \text{ are new variables}}{\Delta;\Gamma;\Theta \vdash \Lambda\vec{\alpha} : \vec{\kappa}.e : \forall\vec{\alpha} : \vec{\kappa}.\sigma \Rightarrow \Lambda\vec{\alpha} : \vec{\kappa}.\lambda d_1{:}T(\alpha_1 : \kappa_1).\ldots\lambda d_n{:}T(\alpha_n : \kappa_n).e'}\ (Tabs)$$

$$\frac{\Delta;\Gamma;\Theta \vdash e : \forall\vec{\alpha} : \vec{\kappa}.\sigma \Rightarrow e' \quad \Delta;\Theta \vdash_{\text{d}} \tau_1 : \kappa_1 \mapsto d_1 \ldots \Delta;\Theta \vdash_{\text{d}} \tau_n : \kappa_n \mapsto d_n}{\Delta;\Gamma;\Theta \vdash e[\tau_1,\ldots,\tau_n] : \sigma \Rightarrow e'[\tau_1,\ldots,\tau_n]\,d_1 \ldots d_n}\ (Tapp)$$

Figure 8: Selected Rules of Expression Translation: $\Delta;\Gamma;\Theta \vdash e : \sigma \Rightarrow e'$

$$\begin{array}{lcl}
|\tau| & = & \tau \\
|\sigma \to \sigma'| & = & |\sigma| \to |\sigma'| \\
|\sigma_1 \times \ldots \times \sigma_n| & = & |\sigma_1| \times \ldots \times |\sigma_n| \\
|\forall\alpha_1 : \kappa_1,\ldots,\alpha_n : \kappa_n.\sigma| & = & \forall\alpha_1 : \kappa_1,\ldots,\alpha_n : \kappa_n.T(\alpha_1 : \kappa_1) \to \ldots \to T(\alpha_n : \kappa_n) \to |\sigma| \\
|\{l_1 : \sigma_1,\ldots,l_n : \sigma_n\}| & = & \{l_1 : |\sigma_1|,\ldots,l_n : |\sigma_n|\}
\end{array}$$

Figure 9: Type Translation

Theorem 7 (Translation Preservation) *Dictionary passing preserves types.*

$$\frac{\Delta;\Gamma;\Theta \vdash e : \sigma \Rightarrow e'}{\Delta;|\Gamma| + |\Theta| \vdash e' : |\sigma|}$$

Proof: by induction over expressions.

3.4 Integration

We have seen that dictionary passing requires only a new kind in the type system and an independent phase in SML/NJ. In this section we show that dictionary passing has better integration with SML/NJ than λ_i^{ML} and λ_R. Actually, Dictionary passing can be easily implemented in any compiler that uses $F_\omega$like typed intermediate language and has standard optimizations, but no runtime type passing nor dependent types.

Quantified types. Dictionary passing has better support for quantified types than λ_i^{ML} and λ_R.

In dictionary passing, recursive types have recursive functions in their dictionaries. No extra type construct is needed. But the type-level analyzing construct **Typerec** in λ_i^{ML} or λ_R may fail to terminate; this makes typechecking undecidable. To force the iteration to terminate, fully reflexive intensional type analysis [16] adds new facilities to the type system.

In an impredicative calculus, dictionary passing can also be applied to universal/existential types. Hiding the identity of types can be done either by parametric polymorphism or by abstract types. To preserve abstraction, we cannot analyze the hidden types or their representations because the analysis exposes their identity. Therefore the hidden types must export additional information (for example, their pretty-printing functions), that is, they must pass dictionaries. Using dictionaries is the

only way to preserve abstraction. Fully intensional type analysis [16, 11] analyzes hidden types, thus breaks the abstraction.

Comparing dictionaries with other type descriptors. Representing types by dictionaries has very little effect on the existing compiler. Phases after dictionary transformation are unchanged.

Passing types at run time as in λ_i^{ML} requires a type system much different from the one used by the compiler, which will dramatically change the compiler back end, and make closure conversion and code generation more complex.

Using special terms as type representations as in λ_R will not have so much effect on the compiler as λ_i^{ML}, but it still needs some support:

First, many new constructs and corresponding rules are needed in the type system (FLINT and the lower-level language): new R terms (representations of run-time types), new types $R(\alpha)$ (types of R terms), and the type-level dispatch construct **Typerec** which requires big changes in type checking and has the non-termination problem.

Second, a polymorphic function for each polytypic primitive needs to be defined. For example, polymorphic equality needs a definition:

$$
\begin{array}{l}
\mathrm{fix}\,\mathrm{peq} : \forall\alpha:\Omega.R(\alpha) \to \mathrm{Eq}[\alpha] \to \mathrm{Eq}[\alpha] \to \mathrm{bool}. \\
\quad \Lambda\alpha{:}\Omega_{\mathrm{eq}}.\,\lambda x_\alpha{:}R(\alpha).\ \mathrm{typecase}\ x_\alpha\ \mathrm{of} \\
\quad\quad R_{\mathrm{int}} \Rightarrow \mathrm{IntEq} \\
\quad\quad R_{\to}(r_\beta, r_\gamma)\ \mathrm{as}\ \beta \to \gamma \Rightarrow \lambda x{:}\mathrm{void}.\,\lambda y{:}\mathrm{void}.\,\mathrm{false} \\
\quad\quad R_{\times}(r_\beta, r_\gamma)\ \mathrm{as}\ \beta \times \gamma \Rightarrow \\
\quad\quad\quad \lambda x{:}\mathrm{Eq}[\beta] \times \mathrm{Eq}[\gamma].\,\lambda y{:}\mathrm{Eq}[\beta] \times \mathrm{Eq}[\gamma]. \\
\quad\quad\quad\quad \mathrm{peq}[\beta]\, r_\beta(\pi_1 x)(\pi_1 y) \\
\quad\quad\quad\quad\ \ \mathrm{and}\ \ \mathrm{peq}[\gamma]\, r_\gamma(\pi_2 x)(\pi_2 y) \\
\quad\quad R_{\mathrm{void}} \Rightarrow \lambda x{:}\mathrm{void}.\,\lambda y{:}\mathrm{void}.\,\mathrm{false}
\end{array}
$$

and $\mathrm{Eq}[\alpha]$ is defined by type-level dispatch construct **Typerec** (it is shown in patter-matching style):

$$
\begin{array}{l}
\mathrm{Eq}[\mathrm{int}] = \mathrm{int} \\
\mathrm{Eq}[\tau_1 \to \tau_2] = \mathrm{void} \\
\mathrm{Eq}[\tau_1 \times \tau_2] = \mathrm{Eq}[\tau]_1 \times \mathrm{Eq}[\tau]_2 \\
\mathrm{Eq}[\mathrm{void}] = \mathrm{void}
\end{array}
$$

Third, when the compiler has several levels of typed intermediate language, types that are passed and analyzed will be transformed in the same way as other types. It is hard to preserve algorithm structure of type analysis. The LX system [2] uses a very rich kind and type language to solve this problem.

Finally, the dependency between R-terms and the types they represent requires more powerful lower-level languages and complicates the back end.

Interaction with optimizations. How does dictionary passing interact with the optimizations in the existing compiler? We tried several implementations:

- Dictionary passing before optimization. In this implementation, the benchmark "life" ran three times slower than when compiled without dictionary passing (figure 10). Other benchmarks such as boyer and mlyacc ran 6 - 7% slower. Part of the inefficiency came from constructing and passing unnecessary dictionary arguments.

- Type specialization [13] is a standard optimization that reduces polymorphic code to monomorphic, where possible. When we moved the dictionary-passing translation to after type specialization, we avoided constructing dictionaries for functions that will be made monomorphic. This significantly improved "life", but it still ran slower than when compiled by the original compiler, because additional polymorphic code (such as $\mathrm{Prod}^n_{\{\mathrm{pp,eq}\}}$) is introduced by constructing dictionaries.

- Dictionary passing between two phases of type specialization. We added one more type specialization phase after dictionary passing, to remove (if possible) the polymorphic code introduced by dictionaries. This gained some speedup (7%) without sacrificing compilation time.

We show the performance of these three implementations in figure 10. This figure indicates that dictionary passing can improve performance if properly organized.

Module system and separate compilation. Dictionary passing fits in without difficulty in the SML module system.

Standard ML uses structure, signature, and functor constructs to organize programs. The SML/NJ front end

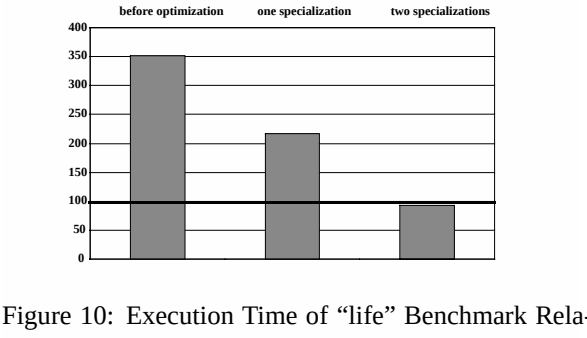

Figure 10: Execution Time of "life" Benchmark Relative to SML/NJ 110.29

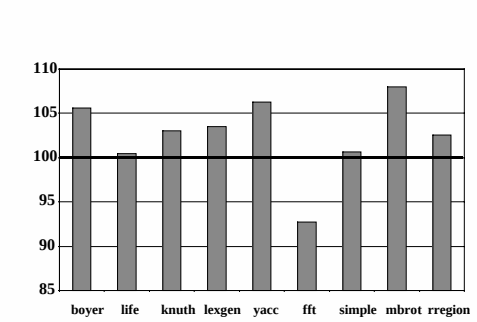

Figure 11: Compilation Time Relative to SML/NJ 110.29

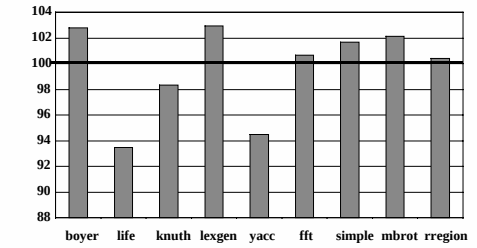

Figure 12: Execution Time Relative to SML/NJ 110.29

translates structures to records, and functors to (polymorphic) functions, so dictionary transformation is not affected by these constructs.

SML/NJ supports separate compilation. A global environment keeps the type information for values exported by any module. In our implementation of dictionary passing, type checking at the ML source level is entirely unchanged. In the type-directed translation after dictionary transformation at the FLINT intermediate language, however, whenever an extern (defined in another module) polymorphic function is used, we translate its type according to the rules of figure 9, so that each intermediate program can be type-checked.

3.5 Performance Measurement

Since constructing and passing dictionaries at run time can be costly, we measure the performance of several benchmarks to determine the overhead of dictionary passing. Based on these benchmarks, the overhead of dictionary passing is low: it does not slow down the compilation and execution phases much (at most 8% slower in compilation and 3% slower in execution); we never measured more than 2% increase of program size in practice. Figure 11 shows the comparison of compilation time of several benchmarks in two compilers (the new compiler with dictionary passing and the original one with no dictionary passing but the "magic" equality function). Figure 12 shows execution time comparison. The bar heights represent the ratios (in percentage) of compilation (or execution) time in the new compiler to compilation (or execution) time in the original compiler.

We use optimizations and dictionary sharing to achieve the results in figures 11 and 12.

It is expensive to construct dictionaries, especially for large types, so it is worth sharing dictionaries. Several ways are possible:

- Dictionaries for primitive types. Dictionaries such as $\mathrm{DInt}_{\{pp,eq\}}$ are generated no more than once per compilation unit.
- Dictionary generators. The dictionary constructor for a product type does not depend on the element types of the product type. The dictionary generator Prod_s^n is implemented as a polymorphic function which takes n dictionaries for n element types and returns the dictionary for the product type consisting of these n element types. For each n and s, this function is generated no more than once per compilation unit.
- Sharing across different modules. If a type t is defined in module A and used in both module A and B, a compiler could construct t's dictionary d only once. Then the interface between two modules would be not only type t, but dictionary d.

Since we did not want to modify module interfaces, we did not implement cross-module sharing. Even so, we never observed more than 2% code blowup in any benchmark.

4 Applications

Our first application of polytypy – the only polytypic function built into the Standard ML – was equality. In this paper we have also used pretty printing as an example application. In each case, automated translation of polytypic functions to ordinary functional code enables us to eliminate an ad-hoc, "magical" function from the runtime system: respectively, the tag-based polymorphic equality function and the unsafe-cast-based type-directed pretty-printer. In the modern context of typed-intermediate-language compilers (which need not be trusted because their low-level output can be type-checked), moving functionality from the runtime system (which must be trusted) to the compiler is very desirable.

There are many other polytypic functions to which our methods can be applied. Recent work in garbage collection [15, 19, 9] has shown how intensional type analysis can be used to construct provably safe, tag-free garbage-collection copy functions. One challenge is how to copy closures. From the discussion in section 3.4, dictionary passing can solve this problem by including copy functions in dictionaries for closures. It is worth studying the details of implementing garbage collection by dictionaries.

Marshalling and unmarshalling of data in a distributed system have also been done with compiler or language support [17].

Implementing debuggers has been difficult in the world of type-directed compilation, because the same data type can be represented in different ways in different contexts. Thus, a machine-language-level implementation of debug breakpoints can be very complex. We plan to explore the implementation of debug breakpoints as polytypic functions using dictionary passing. In the style of Tolmach [14], insertion of debug breakpoints will be a type-safe source-to-source (actually FLINT-to-FLINT) transformation.

5 Conclusion and Future Work

We have implemented dictionary passing in SML/NJ to support polytypic polymorphism. Dictionaries are generated and passed to polymorphic functions automatically by the compiler, and substituted for polytypic primitives. We introduce new kinds in the intermediate language to describe type properties and decide the shape of dictionaries.

We show that dictionary passing provides a simple and general framework for polytypic primitives, and can be better integrated into in an existing compiler with $F_\omega$like intermediate language than type-analyzing. It needs only simple kind refinement in the type system, and an independent program transformation in the compiler. Existing phases in the compiler are almost untouched. Yet the approach is powerful enough to express many polytypic primitives, and support quantified types without breaking parametricity. It can be used to implement tag-free, safe runtime services such as polymorphic equality and garbage collection, and to display values in a debugger without runtime type reconstruction.

In the future we will explore different applications of this approach, and study maximum dictionary sharing and its effects on performance.

References

[1] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional Polymorphism in Type-Erasure Semantics. In 1998 ACM International Conference on Functional Programming, page 301-312, Baltimore, September 1998.

[2] Karl Crary, Stephanie Weirich. Flexible Type Analysis. In ACM SIGPLAN International Conference on Functional Programming Languages, pages 233-248, September 1999.

[3] Dominic Duggan. A Type-Based Semantics for User-Defined Marshalling in Polymorphic Languages. In Second Workshop on Types in Compilation, Japan, April 1998.

[4] Martin Elsman. Polymorphic Equality - No Tags Required. In Second International Workshop on Types in Compilation. Kyoto, Japan, March 1998.

[5] Robert Harper, John C. Mitchell. On the Type Structure of Standard ML. In ACM Transactions on Programming Languages and Systems, 15(2), pages 211-253, April 1993.

[6] Robert Harper, Greg Morrisett. Compiling Polymorphism Using Intensional Type Analysis. In Twenty-second ACM Symposium on Principles of Programming Languages, pages 130-141, San Francisco, January 1995.

[7] Ralf Hinze. A New Approach to Generic Functional Programming. In Twenty-seventh ACM Symposium on Principles of Programming Languages, pages 119-132, Boston, January 2000.

[8] Patrik Jansson, Johan Jeuring. PolyP - a Polytypic Programming Language extension. In Twenty-fourth ACM Symposium on Principles of Programming Languages, pages 470-482, Paris, France, 1997.

[9] Stefan Monnier, Bratin Saha, Zhong Shao. Principled Scavenging. Yale University TR-1205. November 2000.

[10] Atsushi Ohori. A polymorphic record calculus and its compilation. ACM Transactions on Programming Languages and Systems, 17(6), pages 844-895, 1995.

[11] Bratin Saha, Valery Trifonov, and Zhong Shao. Fully Reflexive Intensional Type Analysis in Type Erasure Semantics. In Third ACM SIGPLAN Workshop on Types in Compilation, Montreal, September 2000.

[12] Zhong Shao. An Overview of the FLINT/ML Compiler. In Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation, June 1997.

[13] Zhong Shao. Typed Common Intermediate Format. In 1997 USENIX Conference on Domain Specific Languages, Santa Barbara, October 1997.

[14] Andrew P. Tolmach, Andrew W. Appel. Debugger Standard ML without Reverse Engineering. In Proc. 1990 ACM SIG Conference on Lisp and Functional Programming, pages 1-12, June 1990.

[15] Andrew P. Tolmach. Tag-free Garbage Collection Using Explicit Type Parameters. In Proc. 1994 Conference on Lisp and Functional programming, pages 1-11, 1994.

[16] Valery Trifonov, Bratin Saha, and Zhong Shao. Fully Reflexive Intensional Type Analysis. In Proc. 2000 ACM SIGPLAN International Conference on Functional Programming, 2000.

[17] Sun Microsystems. Java Object Serialization Specification—JDK 1.2. November, 1998.

[18] Philip Wadler, Stephen Boltt. How to Make ad-hoc Polymorphism Less ad hoc. In Sixteenth ACM Symposium on Principles of Programming Languages, pages 60-76, Austin, Texas, January 1989.

[19] Daniel C. Wang, Andrew W. Appel. Type-Preserving Garbage Collection. In Twenty-eighth ACM Symposium on Principles of Programming Languages, pages 166-178, January 2001.

A Complete Rules

We show all the rules of typing, dictionary generation and expression translation here.

$$\frac{}{\vdash n : \text{int}} \quad \frac{}{\Delta; \Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\vdash \text{true} : \text{bool}} \quad \frac{}{\vdash \text{false} : \text{bool}}$$

$$\frac{}{\vdash =: \forall \alpha : \Omega_{\text{eq}}.\alpha \to \alpha \to \text{bool}} \qquad \frac{\Delta; \Gamma, x{:}\sigma \vdash e : \sigma}{\Delta; \Gamma \vdash \text{fix}\, x : \sigma.e : \sigma}$$

$$\frac{\Delta; \Gamma, x{:}\sigma \vdash e : \sigma'}{\Delta; \Gamma \vdash \lambda x{:}\sigma.e : \sigma \to \sigma'} \qquad \frac{\Delta; \Gamma \vdash e : \sigma \to \sigma' \quad \Delta; \Gamma \vdash e' : \sigma}{\Delta; \Gamma \vdash e\, e' : \sigma'}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \sigma_1 \ldots \Delta; \Gamma \vdash e_n : \sigma_n}{\Delta; \Gamma \vdash (e_1, \ldots, e_n) : \sigma_1 \times \ldots \times \sigma_n} \qquad \frac{\Delta; \Gamma \vdash e : \sigma_1 \times \ldots \times \sigma_n \quad 0 < i \leq n}{\Delta; \Gamma \vdash \pi_i\, e : \sigma_i}$$

$$\frac{\Delta; \Gamma \vdash e : \sigma\mu\sigma}{\Delta; \Gamma \vdash \text{fold}[\sigma] e : \mu\sigma} \qquad \frac{\Delta; \Gamma \vdash e : \mu\sigma}{\Delta; \Gamma \vdash \text{unfold}[\sigma] e : \sigma\mu\sigma}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \sigma_1, \ldots \Delta; \Gamma \vdash e_n : \sigma_n}{\Delta; \Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : \sigma_1, \ldots, l_n : \sigma_n\}}$$

$$\frac{\Delta; \Gamma \vdash e : \{l_1 : \sigma_1, \ldots, l_n : \sigma_n\} \quad 0 < i \leq n}{\Delta; \Gamma \vdash e.l_i : \sigma_i}$$

$$\frac{\sigma = \sigma_1 + \ldots + \sigma_n \quad \Delta; \Gamma \vdash e : \sigma_i \quad 0 < i \leq n}{\Delta; \Gamma \vdash \text{inj}_i^{\sigma}\, e : \sigma}$$

$$\frac{\Delta; \Gamma \vdash e : \sigma_1 + \ldots + \sigma_n \quad \begin{array}{c} \Delta; \Gamma \vdash e_1 : \sigma_1 \to \sigma' \\ \vdots \\ \Delta; \Gamma \vdash e_n : \sigma_n \to \sigma' \end{array}}{\Delta; \Gamma \vdash \text{switch}\, e\, \text{of}\, (e_1, \ldots, e_n) : \sigma'}$$

$$\frac{\Delta, \alpha_1 : \kappa_1, \ldots, \alpha_n : \kappa_n; \Gamma \vdash e : \sigma}{\Delta; \Gamma \vdash \Lambda\alpha_1 : \kappa_1, \ldots, \alpha_n : \kappa_n.e : \forall \alpha_1 : \kappa_1, \ldots, \alpha_n : \kappa_n.\sigma}$$

$$\frac{\Delta; \Gamma \vdash e : \forall \alpha_1 : \kappa_1, \ldots, \alpha_n : \kappa_n.\sigma \qquad \Delta \vdash \tau_1 : \kappa_1, \ldots, \Delta \vdash \tau_n : \kappa_n}{\Delta; \Gamma \vdash e[\tau_1, \ldots, \tau_n] : \sigma[\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]}$$

Figure 13: Complete Rules of Expression Formation

$$\frac{}{\vdash_{\mathrm{d}} \mathrm{int} : \Omega_s \mapsto \mathrm{DInt}_s} \qquad \frac{}{\vdash_{\mathrm{d}} \mathrm{bool} : \Omega_s \mapsto \mathrm{DBool}_s}$$

$$\frac{}{\Delta; \Theta \vdash_{\mathrm{d}} \alpha : \kappa \mapsto \Theta(\alpha)} \qquad \frac{}{\Delta; \Theta \vdash_{\mathrm{d}} \tau \to \tau' : \Omega \mapsto \mathrm{DFun}}$$

$$\frac{\Delta; \Theta \vdash_{\mathrm{d}} \tau : \Omega_s \to \Omega_s \mapsto d_\tau}{\Delta; \Theta \vdash_{\mathrm{d}} \mu\tau : \Omega_s \mapsto \mathrm{fix}\, x : T(\mu\tau : \Omega_s).d_\tau[\mu\tau]\, x}$$

$$\frac{\Delta, \alpha : \kappa; \Theta, \alpha : d_\alpha \vdash_{\mathrm{d}} \tau : \kappa' \mapsto d_\tau}{\Delta; \Theta \vdash_{\mathrm{d}} \lambda\alpha{:}\kappa.\, \tau : \kappa \to \kappa' \mapsto \Lambda\alpha{:}\kappa.\, \lambda d_\alpha{:}T(\alpha : \kappa).\, d_\tau}$$

$$\frac{\Delta; \Theta \vdash_{\mathrm{d}} \tau : \kappa \to \kappa' \mapsto d_\tau \quad \Delta; \Theta \vdash_{\mathrm{d}} \tau' : \kappa \mapsto d'_\tau}{\Delta; \Theta \vdash_{\mathrm{d}} \tau\tau' : \kappa' \mapsto d_\tau[\tau']\, d'_\tau}$$

$$\frac{\Delta; \Theta \vdash_{\mathrm{d}} \tau_1 : \Omega_s \mapsto d_1, \ldots, \Delta; \Theta \vdash_{\mathrm{d}} \tau_n : \Omega_s \mapsto d_n}{\Delta; \Theta \vdash_{\mathrm{d}} \tau_1 \times \ldots \times \tau_n : \Omega_s \mapsto \mathrm{Prod}^n_s[\tau_1, \ldots, \tau_n]\, d_1 \ldots d_n}$$

$$\frac{\Delta; \Theta \vdash_{\mathrm{d}} \tau_1 : \Omega_s \mapsto d_1, \ldots, \Delta; \Theta \vdash_{\mathrm{d}} \tau_n : \Omega_s \mapsto d_n}{\Delta; \Theta \vdash_{\mathrm{d}} \tau_1 + \ldots + \tau_n : \Omega_s \mapsto \mathrm{Sum}^n_s[\tau_1, \ldots, \tau_n]\, d_1 \ldots d_n}$$

Figure 14: Complete Rules of Dictionary Generation

$$\frac{}{\vdash n : \text{int} \Rightarrow n}\ (Int) \qquad \frac{}{\Delta;\Gamma;\Theta \vdash x : \sigma \Rightarrow x}\ (Var) \qquad \frac{}{\vdash \text{true:bool} \Rightarrow \text{true}}\ (True) \qquad \frac{}{\vdash \text{false:bool} \Rightarrow \text{false}}\ (False)$$

$$\frac{\Delta;\Gamma;\Theta \vdash e : \sigma\,\mu\sigma \Rightarrow e'}{\Delta;\Gamma;\Theta \vdash \text{fold}[\sigma]e : \mu\sigma \Rightarrow \text{fold}[\sigma]e'}\ (Fold) \qquad \frac{\Delta;\Gamma;\Theta \vdash e : \mu\sigma \Rightarrow e'}{\Delta;\Gamma;\Theta \vdash \text{unfold}[\sigma]e : \sigma\,\mu\sigma \Rightarrow \text{unfold}[\sigma]e'}\ (Unfold)$$

$$\frac{\Delta;\Gamma;\Theta \vdash e : \sigma_i \Rightarrow e' \quad \sigma = \sigma_1 + \ldots + \sigma_n}{\Delta;\Gamma;\Theta \vdash \text{inj}_i^{\sigma} e : \sigma \Rightarrow \text{inj}_i^{|\sigma|} e'}\ (Inj) \qquad \frac{\Delta;\Gamma,x:\sigma;\Theta \vdash e : \sigma \Rightarrow e'}{\Delta;\Gamma;\Theta \vdash \text{fix}\,x:\sigma.e : \sigma \Rightarrow \text{fix}\,x:|\sigma|.e'}\ (Fix)$$

$$\frac{\Delta;\Gamma;\Theta \vdash e : \sigma_1 \times \ldots \times \sigma_n \Rightarrow e'}{\Delta;\Gamma;\Theta \vdash \pi_i\, e : \sigma_i \Rightarrow \pi_i\, e'}\ (SelP) \qquad \frac{\Delta;\Gamma;\Theta \vdash e : \{l_1 : \sigma_1, \ldots, l_n : \sigma_n\} \Rightarrow e'}{\Delta;\Gamma;\Theta \vdash e.l_i : \sigma_i \Rightarrow e'.l_i}\ (SelR)$$

$$\frac{\Delta;\Gamma,x:\sigma;\Theta \vdash e : \sigma' \Rightarrow e'}{\Delta;\Gamma;\Theta \vdash \lambda x{:}\sigma.e : \sigma \to \sigma' \Rightarrow \lambda x{:}|\sigma|.e'}\ (Abs) \qquad \frac{\Delta;\Gamma;\Theta \vdash e_1 : \sigma \to \sigma' \Rightarrow e_1' \quad \Delta;\Gamma;\Theta \vdash e_2 : \sigma \Rightarrow e_2'}{\Delta;\Gamma;\Theta \vdash e_1\, e_2 : \sigma' \Rightarrow e_1'\, e_2'}\ (App)$$

$$\frac{\Delta;\Gamma;\Theta \vdash e_1 : \sigma_1 \Rightarrow e_1' \ldots \Delta;\Gamma;\Theta \vdash e_n : \sigma_n \Rightarrow e_n'}{\Delta;\Gamma;\Theta \vdash (e_1,\ldots,e_n) : \sigma_1 \times \ldots \times \sigma_n \Rightarrow (e_1',\ldots,e_n')}\ (Prod)$$

$$\frac{\Delta;\Gamma;\Theta \vdash e_1 : \sigma_1 \Rightarrow e_1' \ldots \Delta;\Gamma;\Theta \vdash e_n : \sigma_n \Rightarrow e_n'}{\Delta;\Gamma;\Theta \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : \sigma_1, \ldots, l_n : \sigma_n\} \Rightarrow \{l_1 = e_1', \ldots, l_n = e_n'\}}\ (Rec)$$

$$\frac{\begin{array}{c}\Delta;\Gamma;\Theta \vdash e_1 : \sigma_1 \to \sigma' \Rightarrow e_1' \\ \vdots \\ \Delta;\Gamma;\Theta \vdash e : \sigma_1 + \ldots + \sigma_n \Rightarrow e' \quad \Delta;\Gamma;\Theta \vdash e_n : \sigma_n \to \sigma' \Rightarrow e_n'\end{array}}{\Delta;\Gamma;\Theta \vdash \text{switch}\,e\,\text{of}\,(e_1,\ldots,e_n) : \sigma' \Rightarrow \text{switch}\,e'\,\text{of}\,(e_1',\ldots,e_n')}\ (Switch)$$

$$\frac{\Delta,\alpha_1:\kappa_1,\ldots,\alpha_n:\kappa_n;\Gamma;\Theta,\alpha_1:d_1,\ldots,\alpha_n:d_n \vdash e : \sigma \Rightarrow e' \quad d_1,\ldots,d_n \text{ are new variables}}{\Delta;\Gamma;\Theta \vdash \Lambda\vec{\alpha}:\vec{\kappa}.e : \forall\vec{\alpha}:\vec{\kappa}.\sigma \Rightarrow \Lambda\vec{\alpha}:\vec{\kappa}.\lambda d_1{:}T(\alpha_1:\kappa_1)\ldots.\lambda d_n{:}T(\alpha_n:\kappa_n).e'}\ (Tabs)$$

$$\frac{\Delta;\Gamma;\Theta \vdash e : \forall\vec{\alpha}:\vec{\kappa}.\sigma \Rightarrow e' \quad \Delta;\Theta \vdash_{\text{d}} \tau_1 : \kappa_1 \mapsto d_1 \ldots \Delta;\Theta \vdash_{\text{d}} \tau_n : \kappa_n \mapsto d_n}{\Delta;\Gamma;\Theta \vdash e[\tau_1,\ldots,\tau_n] : \sigma \Rightarrow e'[\tau_1,\ldots,\tau_n]\,d_1 \ldots d_n}\ (Tapp)$$

$$\frac{}{\vdash =: \forall\alpha : \Omega_{\text{eq}}.\alpha \to \alpha \to \text{bool} \Rightarrow \Lambda\alpha{:}\Omega_{\text{eq}}.\lambda d{:}T(\alpha:\Omega_{\text{eq}}).d.\text{eq}}\ (Peq)$$

Figure 15: Complete Rules of Expression Translation