4 An Extension of ML with a Dotless Dot Notation

In this chapter, we describe a extension of our language that allows more flexible use of existential types. Following notations used in actual programming languages, this extension assumes the same representation type each time a value of existential type is accessed, provided that each access is through the same identifier. We give a type reconstruction algorithm and show semantic soundness by translating into the language from Chapter 3.

4.1 Introduction

MacQueen [Mac86] observes that the use of existential types in connection with an elimination construct (**`open`**, **`abstype`**, or our **`let`**) is impractical in certain programming situations; often, the scope of the elimination construct has to be made so large that some of the benefits of abstraction are lost. In particular, the lowest-level entities have to be opened at the outermost level; these are the traditional disadvantages of block-structured languages.

We present an extension of ML that provides the same flexibility as the dot notation described in [CL90]. In this extension, abstract types are again modeled by ML datatypes with existentially quantified component types. Values of abstract type are created by applying a datatype constructor to a

value, and they are decomposed in a pattern-matching **let** expression. However, we allow existentially quantified type variables to escape the scope of the identifier in whose type they appear, as long as the expression decomposed is an identifier and the existentially quantified type variables do not escape the scope of that identifier. Each decomposition of an identifier, using the same constructor, produces identical existentially quantified type variables. We call our notation a "dotless" dot notation, since it uses decomposition by pattern-matching instead of record component selection.

4.2 Some Motivating Examples

We assume the type declaration

```
datatype Key = key of 'a * ('a -> int)
```

in the following examples. In the first example,

```
let val x = key(3,fn x => x + 2) in
    (let val key(_,f) = x in f end)
        (let val key(v,_) = x in v end)
end
```

the existential type variable in the type of **f** is the same as the one in the type of **v**, and the function application produces a result of type **int**. This follows from the fact that both **f** and **v** are bound by decomposition of the same[1] identifier, **x**. Consequently, they must hold the same value and the whole expression is type-correct.

[1] We assume the ML scoping discipline, which uses **let** statements as scope boundaries; alternatively, one could require each bound identifier to be unique.

In a language with the traditional dot notation, for example Ada, abstract types can be modeled as packages, and an example corresponding to the previous one would look as follows:

```
package KEY_PKG is
    type KEY is private;
    X : constant KEY;
    function F(X : KEY) return INTEGER;
private
    type KEY is INTEGER;
    X : constant KEY := 3;
end KEY_PKG;

package body KEY_PKG is
    function F(X : KEY) return INTEGER is
    begin
        return X + 2;
    end;
end KEY_PKG;

var Z : INTEGER;
...

Z := KEY_PKG.F(KEY_PKG.X);
```

The components of the abstract type **KEY_PKG** are selected using the dot notation.

The following are examples of incorrect programs. For instance,

```
let val x = key(3,fn x => x + 2) in
    let val key(_,f) = x in
        f
    end
end
```

is not type-correct, since the existential type variable in the type of **f** escapes the scope of **x**. Neither is the following program,

```
let val x = key(3,fn x => x + 2)
    val y = x
in
    (let val key(_,f) = x in f end)
        (let val key(v,_) = y in v end)
end
```

since different identifiers produce different existential type variables, although they hold the same values in this case. As the latter cannot be determined statically, we must assume that the values have different types. Similarly,

```
val z = (3,fn x => x + 2)
let val key(_,f) = key z in
    let val key(v,_) = key z in
        f v
    end
end
```

is not type-correct. Since the expressions that are decomposed are not even identifiers, we cannot assume statically that **f** can be applied to **v**.

4.3 Syntax

4.3.1 Language Syntax

Syntactically, our underlying formal language is almost unchanged, except that pattern-matching **let** expressions only allow an identifier to be decomposed, not a general expression. This is not a significant restriction, since we can always bind the expression in an enclosing **let** before decomposing it. Again, we assume that each identifier bound by a λ or **let** expression is unique.

Identifiers x

Constructors	K	
Type constructors	T	
Expressions	e	::= **()** | **true** | **false** | x | (e_1, e_2) | $e\ e'$ | $\lambda x.e$ | **let** $x = e$ **in** e' | **data** $\forall\alpha_1 \ldots \alpha_n.\chi$ **in** e | K | **is** K | **let** $K\ x = x'$ **in** e'

4.3.2 Type Syntax

Type variables	α	
Skolem functions	κ	
Types	τ	::= *unit* | *bool* | α | $\tau_1 \times \tau_2$ | $\tau \rightarrow \tau'$ | $\kappa_{x,K,i}(\tau_1, \ldots, \tau_n)$ | χ
Recursive types	χ	::= $\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m$ where $K_i \neq K_j$ for $i \neq j$
Existential types	η	::= $\exists\alpha.\eta$ | τ
Type schemes	σ	::= $\forall\alpha.\sigma$ | τ
Assumptions	a	::= σ/x | $\forall\alpha_1 \ldots \alpha_n.\chi$

Our type syntax is almost unchanged. However, Skolem type constructors are now uniquely associated with an identifier x by using the symbol κ, indexed by x, the constructor K used in the decomposition, and the index i of the existentially quantified variable γ_i to be replaced.

4.4 Type Inference

4.4.1 Instantiation and Generalization of Type Schemes

$\forall\alpha_1 \ldots \alpha_n.\tau \geq \tau'$ iff there are types $\tau_1, \ldots \tau_n$ such that

$$\tau' = \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$$

$\exists\alpha_1 \ldots \alpha_n.\tau \leq \tau'$ iff there are types $\tau_1, \ldots \tau_n$ such that

$$\tau' = \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$$

$$gen(A, \tau) = \forall(FV(\tau) \setminus FV(A)).\tau$$

$$skolem^{\cdot}(A, x, K, \exists\gamma_1 \ldots \gamma_n.\tau) =$$

$$\tau[\kappa_{x,K,i}(\alpha_1, \ldots \alpha_k)/\gamma_i] \text{ where}$$

$$\{\alpha_1, \ldots, \alpha_k\} = FV(\exists\gamma_1 \ldots \gamma_n.\tau) \setminus FV(A)$$

Instantiation and generalization are unchanged. The modified function $skolem^{\cdot}$ replaces each existentially quantified variable in a type by a unique type constructor whose actual arguments are those free variables of the type that are not free in the assumption set. Since identifiers are unique, we obtain Skolem constructors uniquely associated with an identifier x by using the symbol κ, indexed by x, the constructor K used in the decomposition, and the index i of the existentially quantified variable γ_i to be replaced. In addition to FV, the set of free type variables in a type scheme or assumption set, we use FS_x, the set of those Skolem type constructors that occur in a type scheme or assumption set and are associated with identifier x.

4.4.2 Inference Rules for Expressions

The first three typing rules are the same as in the original system.

$$\text{(VAR}^{\cdot}\text{)} \quad \frac{A(x) \geq \tau}{A \vdash^{\cdot} x : \tau}$$

$$\text{(PAIR}^{\cdot}\text{)} \quad \frac{A \vdash^{\cdot} e_1 : \tau_1 \qquad A \vdash^{\cdot} e_2 : \tau_2}{A \vdash^{\cdot} (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\text{(APPL}^{\cdot}\text{)} \quad \frac{A \vdash^{\cdot} e : \tau' \to \tau \qquad A \vdash^{\cdot} e' : \tau'}{A \vdash^{\cdot} e\ e' : \tau}$$

The ABS$^{\cdot}$ and LET$^{\cdot}$ rules are modified to prevent Skolem constructors associated with a bound variable to escape the scope of that variable.

$$\text{(ABS}^{\cdot}\text{)} \quad \frac{A\,[\tau'/x] \vdash^{\cdot} e : \tau \qquad FS_x(A) \cup FS_x(\tau) = \varnothing}{A \vdash^{\cdot} \lambda x.e : \tau' \to \tau}$$

$$\text{(LET}^{\cdot}\text{)} \quad \frac{A \vdash^{\cdot} e : \tau \qquad A\,[gen(A, \tau)/x] \vdash^{\cdot} e' : \tau' \qquad FS_x(A) \cup FS_x(\tau') = \varnothing}{A \vdash^{\cdot} \textbf{let}\ \ x = e\ \ \textbf{in}\ \ e' : \tau'}$$

The rules DATA$^{\cdot}$, CONS$^{\cdot}$, TEST$^{\cdot}$ remain unchanged.

$$\text{(DATA}^{\cdot}\text{)} \quad \frac{\sigma = \forall\alpha_1 \ldots \alpha_n.\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m \qquad FV(\sigma) = \varnothing \qquad A\,[\sigma/K_1, \ldots, \sigma/K_m] \vdash^{\cdot} e : \tau}{A \vdash^{\cdot} \textbf{data}\ \ \sigma\ \ \textbf{in}\ \ e : \tau}$$

$$\text{(CONS}^{\cdot}\text{)} \quad \frac{A(K) \geq \mu\beta.\Sigma\,[K\eta] \qquad \eta\,[\mu\beta.\Sigma\,[K\eta]/\beta] \leq \tau}{A \vdash^{\cdot} K : \tau \to \mu\beta.\Sigma\,[K\eta]}$$

$$\text{(TEST}^{\cdot}\text{)} \quad \frac{A(K) \geq \mu\beta.\Sigma[K\eta]}{A \vdash^{\cdot} \mathbf{is}\ \ K : (\mu\beta.\Sigma[K\eta]) \rightarrow bool}$$

$$\text{(PAT}^{\cdot}\text{)} \quad \frac{A(K) \geq \mu\beta.\Sigma[K\eta] \qquad A(x) \geq \mu\beta.\Sigma[K\eta] \qquad A[gen(A, skolem^{\cdot}(A, x, K, \eta[\mu\beta.\Sigma[K\eta]/\beta]))/x'] \vdash^{\cdot} e : \tau}{A \vdash^{\cdot} \mathbf{let}\ \ K\ x' = x\ \ \mathbf{in}\ \ e : \tau}$$

The new PAT· rule does not enforce any restriction on occurrence of Skolem constructors. It only requires that the variable x be of the same type as the result type of the constructor K. The body e is typed under the assumption set extended with an assumption about the bound identifier x'.

4.5 Type Reconstruction

Again, the type reconstruction algorithm is a straightforward translation from the deterministic typing rules.

4.5.1 Auxiliary Functions

While $inst_{\forall}$ and $inst_{\exists}$ are as in the preceding chapter, the other auxiliary functions are the same as in the inference rules.

4.5.2 Algorithm

Our type reconstruction function takes an assumption set and an expression, and it returns a substitution and a type expression. There is one case for each typing rule.

$$TC^{\cdot}(A, x) =$$
$$(Id, inst_{\forall}(A(x)))$$

$TC^{\cdot}(A,\ (e_1, e_2))\ =$

let $(S_1, \tau_1) = TC^{\cdot}(A, e_1)$

$(S_2, \tau_2) = TC^{\cdot}(S_1 A, e_2)$

in $(S_2 S_1, S_2 \tau_1 \times \tau_2)$

$TC^{\cdot}(A, e\ e')\ =$

let $(S, \tau) = TC^{\cdot}(A, e)$

$(S', \tau') = TC^{\cdot}(SA, e')$

β be a fresh type variable

$U = mgu\ (S'\tau, \tau' \rightarrow \beta)$

in $(US'S, U\beta)$

$TC^{\cdot}(A, \lambda x.\, e)\ =$

let β be a fresh type variable

$(S, \tau) = TC^{\cdot}(A\,[\beta / x]\,, e)$

in

if $FS_x(SA) \cup FS_x(\tau) = \varnothing$ **then**

$(S, S\beta \rightarrow \tau)$

$TC^{\cdot}(A,$ **let** $x = e$ **in** $e')\ =$

let $(S, \tau) = TC^{\cdot}(A, e)$

$(S', \tau') = TC^{\cdot}(SA\,[gen\ (SA, \tau) / x]\,, e')$

in

if $FS_x(S'SA) \cup FS_x(\tau') = \varnothing$ **then**

$(S'S, \tau')$

$TC^{\cdot}(A,$ **data** σ **in** $e) =$

let $\forall\alpha_1 \ldots \alpha_n.\, \mu\beta.\, K_1\eta_1 + \ldots + K_m\eta_m = \sigma$ **in**

if $FV(\sigma) = \emptyset$ **then**

$TC^{\cdot}(A\,[\sigma/K_1, \ldots, \sigma/K_m],\, e)$

$TC^{\cdot}(A, K) =$

let $\tau = inst_{\forall}\,(A\,(K))$

$\mu\beta. \ldots + K\eta + \ldots = \tau$

in $(Id,\ (inst_{\exists}\,(\eta\,[\tau/\beta]))) \rightarrow \tau)$

$TC^{\cdot}(A,$ **is** $K) =$

let $\tau = inst_{\forall}\,(A\,(K))$

in $(Id, \tau \rightarrow bool)$

$TC^{\cdot}(A,$ **let** $K\ x' = x$ **in** $e') =$

let $\tau = inst_{\forall}(A(x))$

$U = mgu(\tau, inst_{\forall}(A(K)))$

$\mu\beta. \ldots + K\eta + \ldots = U\tau$

$\tau_{\kappa} = skolem^{\cdot}(UA, x, K,\ (\eta\,[U\tau/\beta]))$

$(S, \tau') = TC^{\cdot}(UA\,[gen\ (UA, \tau_{\kappa})/x'],\, e')$

in

(SU, τ')

4.5.3 Syntactic Soundness and Completeness of Type Reconstruction

Lemma 4.1 [Stability of $\vdash^{\cdot}$] If $A \vdash^{\cdot} e : \tau$ and S is a substitution, then $SA \vdash^{\cdot} e : S\tau$ also holds. Moreover, if there is a proof tree for $A \vdash^{\cdot} e : \tau$ of height n, then there is also a proof tree for $SA \vdash^{\cdot} e : S\tau$ of height less or equal to n.

Theorem 4.2 [Syntactic soundness] If $TC^{\cdot}(A, e) = (S, \tau)$, then $SA \vdash^{\cdot} e : \tau$.

Definition 4.1 [Principal Type] τ is a principal type of expression e under assumption set A if $A \vdash^{\cdot} e : \tau$ and whenever $A \vdash^{\cdot} e : \tau'$ then there is a substitution S such that $S\tau = \tau'$.

Theorem 4.3 [Syntactic completeness] If $\hat{S}A \vdash^{\cdot} e : \hat{\tau}$, then $TC^{\cdot}(A, e) = (S, \tau)$ and there is a substitution R such that $\hat{S}A = RSA$ and $\hat{\tau} = R\tau$.

Corollary 4.4 [Principal type] If $TC^{\cdot}(A, e) = (S, \tau)$, then τ is a principal type for e under A.

Proof: We modify the proofs given in Chapter 3.

4.6 A Translation Semantics

We retain our original semantic interpretation $E[\![\ ]\!]$. Following [CL90], we prove semantic soundness by giving a type- and semantics-preserving translation to our original language. The idea is that we can enclose an expression e with subexpressions of the form **let** $K\ x' = x$ **in** e' by an outer expres-

sion that defines x' and replace **let** $K\ x' = x$ **in** e' by e'. That is, we replace e by

$$\textbf{let}\ K\ x_K = x\ \textbf{in}\ e\,[e'\,[x_K / x'] / \textbf{let}\ K\ x' = x\ \textbf{in}\ e']$$

We chose the enclosing **let** expression defining x' large enough so that no existentially quantified type variables arising through the inner **let** expressions escape this outer definition. Since the ABS$^{\cdot}$ and LET$^{\cdot}$ rules guarantee that no existentially quantified variables emerging from the decomposition of x escape the scope of x, it is safe to enclose the whole body of the λ or **let** expression.

However, we must be careful, since the outer decomposition in the translation might fail, while the inner decomposition in the original expression might not necessarily have been reached; this is possible if the value held by x does not have the constructor tag K. Therefore, we need to replace e by an **if** expression with branches for each constructor tag in the datatype that x has. This is reflected in the definition of the auxiliary translation function $\| \ \|$ below.

4.6.1 Modified Original Language

Type judgments in a modified version of the original language are of the form $A \vdash^{\circ} e : \tau$. We modify the *skolem* function and the PAT rule of our original language:

$$\mathit{skolem}^{\circ}(A, x, \exists\gamma_1 \ldots \gamma_n.\tau) = \tau\,[\kappa_{x,i}(\alpha_1, \ldots \alpha_k) / \gamma_i] \text{ where}$$
$$\{\alpha_1, \ldots, \alpha_k\} = FV(\exists\gamma_1 \ldots \gamma_n.\tau) \setminus FV(A)$$

Unique Skolem type constructors can be generated by using the symbol κ, indexed by the unique name x of the bound identifier and the index i of the existentially quantified type variable γ_i to be replaced.

$$\text{(PAT}^\circ\text{)}\quad \frac{A \vdash^\circ e : \mu\beta.\Sigma\,[K\eta] \qquad FS_x(A) \cup FS_x(\tau') = \varnothing \qquad A\,[gen(A, skolem^\circ(A, x, \eta\,[\mu\beta.\Sigma\,[K\eta]\,/\beta]))/x'] \vdash^\circ e' : \tau'}{A \vdash^\circ \textbf{let}\ \ K\ x = e\ \ \textbf{in}\ \ e' : \tau'}$$

Using this modified *skolem*° function, the PAT° rule can enforce that newly generated Skolem constructors escape their scope by the condition $FS_x(A) \cup FS_x(\tau') = \varnothing$, which expresses that no Skolem constructor associated with x may escape the scope of x.

It is easy to see that this language has the same properties as the original one, in particular, semantic soundness.

4.6.2 Auxiliary Translation Function

The bodies of λ and **let** expressions are translated by the auxiliary function given below. It moves all pattern-matching **let** expressions that decompose the variable bound by the enclosing λ or **let** expression to the outermost level possible.

We use a *conformity check* in form of a nested **if** expression with **is** expressions to determine the constructor tag of the value held by x. This requires us to evaluate[1] x; consequently, the resulting expression is always strict in x. Therefore, this translation is not semantics-preserving if the original expression was non-strict in x. We need to distinguish between the translation of the strict and the non-strict version of our language:

- In the strict language, the expression bound to x is already evaluated at binding time, and evaluating it again leaves the semantics unchanged.
- In the non-strict language, the expression bound to x might not be eval-

[1] It actually suffices to evaluate the argument to *weak head normal form*, so that the top-level constructor of the argument can be inspected; see [PJ87] for details. Nevertheless, the resulting translation is not semantics-preserving.

uated at all; to be semantics-preserving, the translation must not introduce additional evaluations of x.

As described in [PJ87], the only patterns for which a conformity check can be omitted safely are the *irrefutable* patterns involving datatypes with a single constructor. We therefore restrict the *non-strict* version of our language in the following way:

Existentially quantified type variables may occur only in the component types of datatypes with a single constructor.

The auxiliary translation function for the strict version of the language is defined as follows:

$$\|e\|_{x,\, K_1\eta_1 + \ldots + K_n\eta_n} =$$

$$\begin{array}{l}
\textbf{if is } K_1\ x \textbf{ then} \\
\quad \textbf{let } K_1\ x_{K_1} = x \textbf{ in } e\left[\begin{array}{l} e'\,[x_{K_1}/x']/\textbf{let } K_1\ x' = x \textbf{ in } e' \\ \textbf{fail}/\textbf{let } K_{i\neq 1}\ x' = x \textbf{ in } e' \end{array}\right] \\
\textbf{else if is } K_2\ x \textbf{ then} \\
\qquad \ldots \\
\textbf{else if is } K_n\ x \textbf{ then} \\
\quad \textbf{let } K_n\ x_{K_n} = x \textbf{ in } e\left[\begin{array}{l} e'\,[x_{K_n}/x']/\textbf{let } K_n\ x' = x \textbf{ in } e' \\ \textbf{fail}/\textbf{let } K_{i\neq n}\ x' = x \textbf{ in } e' \end{array}\right] \\
\textbf{else} \\
\quad e\,[\textbf{fail}/\textbf{let } K_i\ x' = x \textbf{ in } e']
\end{array}$$

In the non-strict case, there can be only a single constructor with an existential component type, and the auxiliary translation function reduces to:

$$\|e\|_{x,\, K\eta} = \textbf{let } K\ x_K = x \textbf{ in } e\,[e'\,[x_K/x']/\textbf{let } K\ x' = x \textbf{ in } e']$$

$$\|e\|_{x,\, K_1\tau_1 + \ldots + K_n\tau_n} = e$$

4.6.3 Inference-guided Translation

We give a translation guided by the type inference rules, along the lines of [NS91]. Let e_0 be a closed, well-typed term. The translation is defined along with the type inference rules for each subterm of e_0.

$$(\mathrm{VAR}^{\cdot}) \quad \frac{A(x) \geq \tau}{A \vdash^{\cdot} x : \tau \Rightarrow x}$$

$$(\mathrm{PAIR}^{\cdot}) \quad \frac{A \vdash^{\cdot} e_1 : \tau_1 \Rightarrow \tilde{e}_1 \qquad A \vdash^{\cdot} e_2 : \tau_2 \Rightarrow \tilde{e}_2}{A \vdash^{\cdot} (e_1, e_2) : \tau_1 \times \tau_2 \Rightarrow (\tilde{e}_1, \tilde{e}_2)}$$

$$(\mathrm{APPL}^{\cdot}) \quad \frac{A \vdash^{\cdot} e : \tau' \to \tau \Rightarrow \tilde{e} \qquad A \vdash^{\cdot} e' : \tau' \Rightarrow \tilde{e}'}{A \vdash^{\cdot} e\ e' : \tau \Rightarrow \tilde{e}\ \tilde{e}'}$$

$$(\mathrm{ABS}^{\cdot}) \quad \frac{A[\tau'/x] \vdash^{\cdot} e : \tau \Rightarrow \tilde{e} \qquad \tau' \neq \mu\beta.\Sigma[K\eta]}{A \vdash^{\cdot} \lambda x.e : \tau' \to \tau \Rightarrow \lambda x.\tilde{e}}$$

$$(\mathrm{ABS}^{\cdot\prime}) \quad \frac{\begin{array}{c} A[\mu\beta.\Sigma[K\eta]/x] \vdash^{\cdot} e : \tau \qquad FS_x(A) \cup FS_x(\tau) = \varnothing \\ A[\mu\beta.\Sigma[K\eta]/x] \vdash^{\cdot} \|e\|_{x,\Sigma[K\eta]} : \tau \Rightarrow \tilde{e} \end{array}}{A \vdash^{\cdot} \lambda x.e : (\mu\beta.\Sigma[K\eta]) \to \tau \Rightarrow \lambda x.\tilde{e}}$$

$$(\mathrm{LET}^{\cdot}) \quad \frac{\begin{array}{c} A \vdash^{\cdot} e : \tau \Rightarrow \tilde{e} \qquad \tau \neq \mu\beta.\Sigma[K\eta] \\ A[gen(A,\tau)/x] \vdash^{\cdot} e' : \tau' \Rightarrow \tilde{e}' \end{array}}{A \vdash^{\cdot} \mathbf{let}\ x = e\ \mathbf{in}\ e' : \tau' \Rightarrow \mathbf{let}\ x = \tilde{e}\ \mathbf{in}\ \tilde{e}'}$$

$$\text{(LET}^{\cdot\prime}\text{)}\quad \frac{\begin{array}{c} A \vdash^{\cdot} e : \mu\beta.\Sigma[K\eta] \Rightarrow \tilde{e} \qquad FS_x(A) \cup FS_x(\tau') = \varnothing \\ A[gen(A, \mu\beta.\Sigma[K\eta])/x] \vdash^{\cdot} e' : \tau' \\ A[gen(A, \mu\beta.\Sigma[K\eta])/x] \vdash^{\cdot} \|e'\|_{x,\Sigma[K\eta]} : \tau' \Rightarrow \tilde{e}' \end{array}}{A \vdash^{\cdot} \mathbf{let}\ x = e\ \mathbf{in}\ e' : \tau' \Rightarrow \mathbf{let}\ x = \tilde{e}\ \mathbf{in}\ \tilde{e}'}$$

$$\text{(DATA}^{\cdot}\text{)}\quad \frac{\begin{array}{c} \sigma = \forall\alpha_1 \ldots \alpha_n.\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m \\ FV(\sigma) = \varnothing \qquad A[\sigma/K_1, \ldots, \sigma/K_m] \vdash^{\cdot} e : \tau \Rightarrow \tilde{e} \end{array}}{A \vdash^{\cdot} \mathbf{data}\ \sigma\ \mathbf{in}\ e : \tau \Rightarrow \mathbf{data}\ \sigma\ \mathbf{in}\ \tilde{e}}$$

$$\text{(CONS}^{\cdot}\text{)}\quad \frac{A(K) \geq \mu\beta.\Sigma[K\eta] \qquad \eta[\mu\beta.\Sigma[K\eta]/\beta] \leq \tau}{A \vdash^{\cdot} K : \tau \rightarrow \mu\beta.\Sigma[K\eta] \Rightarrow K}$$

$$\text{(TEST}^{\cdot}\text{)}\quad \frac{A(K) \geq \mu\beta.\Sigma[K\eta]}{A \vdash^{\cdot} \mathbf{is}\ K : (\mu\beta.\Sigma[K\eta]) \rightarrow bool \Rightarrow \mathbf{is}\ K}$$

$$\text{(PAT}^{\cdot}\text{)}\quad \frac{\begin{array}{c} A(K) \geq \mu\beta.\Sigma[K\eta] \qquad A(x) \geq \mu\beta.\Sigma[K\eta] \\ A[gen(A, skolem^{\cdot}(A, x, K, \eta[\mu\beta.\Sigma[K\eta]/\beta]))/x'] \vdash^{\cdot} e : \tau \\ \Rightarrow \tilde{e} \end{array}}{A \vdash^{\cdot} \mathbf{let}\ K\ x' = x\ \mathbf{in}\ e : \tau \Rightarrow \mathbf{let}\ K\ x' = x\ \mathbf{in}\ \tilde{e}}$$

4.6.4 Translation of Type Schemes and Assumption Sets

After applying $\| \ \|$ to the body of a λ or **let** expression, the only pattern-matching **let** expressions left in the body are of the form $\mathbf{let}\ K\ x_K = x\ \mathbf{in}\ e$. In the following translation, the Skolem constructors associated with x become associated with x_K. This is reflected by the following translations:

$$\lfloor \sigma \rfloor = \sigma\left[\kappa_{x_K, i}/\kappa_{x, K, i}\right]$$

$$\lfloor A \rfloor \quad = \quad [\lfloor \sigma \rfloor / x \mid A(x) = \sigma]$$

4.6.5 Properties of the Translation

Lemma 4.5 Let $\bar{A} = A\,[\forall\alpha_1 \ldots \alpha_k.\mu\beta.K_1\eta_1 + \ldots + K_n\eta_n / x]$. If $\bar{A} \vdash^{\cdot} e : \tau$, then $\bar{A} \vdash^{\cdot} \|e\|_{x,\, K_1\eta_1 + \ldots + K_n\eta_n} : \tau$.

Proof: In the strict case, let $1 \le i \le n$ be arbitrary. We are free to extend $\bar{A}$, assuming that x_{K_i} is not free in $\bar{A}$, hence

$\bar{A}\left[gen(\bar{A}, skolem^{\cdot}(\bar{A}, x, K_i, \eta_i)) / x_{K_i} \right] \vdash^{\cdot} e : \tau$.

Then, any subexpression **let** $K_i\ x' = x$ **in** e' of e is well-typed and we have a subproof for $A' \vdash^{\cdot}$ **let** $K_i\ x' = x$ **in** $e' : \tau'$, where $A'(x_{K_i}) = gen(\bar{A}, skolem^{\cdot}(\bar{A}, x, K_i, \eta_i))$. A premise of this judgment is $A'\left[gen(A', skolem^{\cdot}(A', x, K_i, \eta_i)) / x_{K_i} \right] \vdash^{\cdot} e' : \tau'$. Therefore, $A' \vdash^{\cdot} e'\,[x_{K_i} / x'] : \tau'$, since we may drop the assumption about x' after substituting x_{K_i} for it.

By replacing the proof tree for the **let** subexpression by this latter one and by observing that **fail** has any type, we can prove

$$\bar{A}\left[gen(\bar{A}, skolem^{+}(\bar{A}, x, K_i, \eta_i)) / x_{K_i} \right] \vdash^{\cdot} e \begin{bmatrix} e'\,[x_{K_i} / x'] \,/\, \textbf{let}\ \ K_i\ x' = x\ \ \textbf{in}\ \ e' \\ \textbf{fail} \,/\, \textbf{let}\ \ K_{j \ne i}\ x' = x\ \ \textbf{in}\ \ e' \end{bmatrix} : \tau \ .$$

Thus,

$$\bar{A} \vdash^{\cdot} \textbf{let}\ \ K_i\ x_{K_i} = x\ \ \textbf{in}\ \ e \begin{bmatrix} e'\,[x_{K_i} / x'] \,/\, \textbf{let}\ \ K_i\ x' = x\ \ \textbf{in}\ \ e' \\ \textbf{fail} \,/\, \textbf{let}\ \ K_{j \ne i}\ x' = x\ \ \textbf{in}\ \ e' \end{bmatrix} : \tau$$

Using a suitable typing for the **if** expressions, we conclude that

$\bar{A} \vdash^{\cdot} \| e \|_{x, K_1\eta_1 + \ldots + K_n\eta_n} : \tau.$

The claim for the non-strict case follows analogously.

■

Theorem 4.6 [Type preservation] If $A \vdash^{\cdot} e : \tau \Rightarrow \tilde{e}$, then $\lfloor A \rfloor \vdash^{\circ} \tilde{e} : \lfloor \tau \rfloor$.

Proof: By structural induction on e. We show the only interesting case, all others are straightforward.

$A \vdash^{\cdot}$ **let** $K\ x' = x$ **in** $e : \tau \Rightarrow$ **let** $K\ x' = x$ **in** $\tilde{e}$

Since our expression is a subexpression of a well-typed expression, x is bound either in a λ or in a **let** expression. Thus, it must be a subexpression of an expression of the form $\| e' \|_{x, \Sigma[K\eta]}$, where

$A'[\forall\alpha_1 \ldots \alpha_k.\mu\beta.\Sigma[K\eta]/x] \vdash^{\cdot} \| e' \|_{x, \Sigma[K\eta]} : \tau'$ and

$FS_x(A') \cup FS_x(\tau') = \varnothing$ for some A' and τ'. By definition of $\| \ \|$, the only subexpressions of $\| e' \|_{x, \Sigma[K\eta]}$ of the form **let** $K\ x' = x$ **in** e are the branches of the **if** expression, each of the form **let** $K\ x_K = x$ **in** e, where

$A'[\forall\alpha_1 \ldots \alpha_k.\mu\beta.\Sigma[K\eta]/x] \vdash^{\cdot}$ **let** $K\ x_K = x$ **in** $e : \tau'$ and

$FS_x(A') \cup FS_x(\tau') = \varnothing$; therefore $\tau = \tau'$ and $A = A'$ in the subproof. As a premise, we have

$$A[gen(A, skolem^{\cdot}(A, x, K, \eta[\mu\beta.\Sigma[K\eta]/\beta]))/x_K] \vdash^{\cdot} e : \tau \Rightarrow \tilde{e};$$

by the inductive assumption,

$$\lfloor A[gen(A, skolem^{\cdot}(A, x, K, \eta[\mu\beta.\Sigma[K\eta]/\beta]))/x_K] \rfloor \vdash^{\circ} \tilde{e} : \lfloor \tau \rfloor,$$

hence

$$\lfloor A \rfloor [gen(A, \lfloor skolem^{+}(A, x, K, \eta [\mu\beta.\Sigma [K\eta] / \beta]) \rfloor) / x_K] \vdash^{\circ} \tilde{e} : \lfloor \tau \rfloor.$$

Since

$$\lfloor skolem^{\cdot}(A, x, K, \eta [\mu\beta.\Sigma [K\eta] / \beta]) \rfloor$$
$$= skolem^{\circ}(A, x_K, \eta [\mu\beta.\Sigma [K\eta] / \beta])$$

we have

$$\lfloor A \rfloor [gen(A, skolem^{\circ}(A, x_K, \eta [\mu\beta.\Sigma [K\eta] / \beta])) / x_K] \vdash^{\circ} \tilde{e} : \lfloor \tau \rfloor.$$

We translate the other two premises and obtain $\lfloor A \rfloor(K) \geq \lfloor \mu\beta.\Sigma [K\eta] \rfloor$ and $\lfloor A \rfloor(x) \geq \lfloor \mu\beta.\Sigma [K\eta] \rfloor$. We further observe that $\lfloor FS_x(A) \rfloor = FS_{x_K}(\lfloor A \rfloor)$ and $\lfloor FS_x(\tau) \rfloor = FS_{x_K}(\lfloor \tau \rfloor)$, thus $FS_{x_K}(\lfloor A \rfloor) \cup FS_{x_K}(\lfloor \tau \rfloor) = \emptyset$.

Finally, we can apply the PAT° rule and conclude that

$$\lfloor A \rfloor \vdash^{\circ} \mathbf{let}\ K\ x_K = x\ \mathbf{in}\ \tilde{e} : \lfloor \tau \rfloor.$$

■

Lemma 4.7 $E[\![e]\!]\ \rho = E[\![\|e\|_{x, K_1\eta_1 + \ldots + K_n\eta_n}]\!]\ \rho$ for arbitrary ρ defined for x.

Proof: By definition of E, any subexpression of e is evaluated in an environment $\rho' \supseteq \rho$, whence $\rho'(x) = \rho(x)$. We identify two cases:

$\rho(x) \in \{K_i\} \times V$ for some i. Then, in the strict case only,

$$E[\![\mathbf{let}\ K_{j \neq i} x' = x\ \mathbf{in}\ e']\!]\ \rho' = \bot$$

and in both cases,

$$E[\![\mathbf{let}\ K_i\ x' = x\ \mathbf{in}\ e']\!]\ \rho' = E[\![e' [x_{K_i} / x']]\!]\ (\rho' [\mathsf{snd}(\rho(x)) / x_{K_i}])$$

Consequently, in the strict case,

$$
\begin{aligned}
& E[\![e]\!]\ \rho \\
&= E[\![e \begin{bmatrix} e'[x_{K_i}/x'] \,/\, \textbf{let}\ \ K_i\ x' = x\ \ \textbf{in}\ \ e' \\ \textbf{fail} \,/\, \textbf{let}\ \ K_{j \neq i}\ x' = x\ \ \textbf{in}\ \ e' \end{bmatrix}]\!]\ \ (\rho\,[\mathsf{snd}(\rho(x))/x_{K_i}]) \\
&= E[\![\textbf{let}\ \ K_i\ x_{K_i} = x\ \ \textbf{in}\ \ e \begin{bmatrix} e'[x_{K_i}/x'] \,/\, \textbf{let}\ \ K_i\ x' = x\ \ \textbf{in}\ \ e' \\ \textbf{fail} \,/\, \textbf{let}\ \ K_{j \neq i}\ x' = x\ \ \textbf{in}\ \ e' \end{bmatrix}]\!]\ \rho \\
&= E[\![\|e\|_{x,\, K_1\eta_1 + \ldots + K_n\eta_n}]\!]\ \rho\ ,
\end{aligned}
$$

since the **if** branch for i gets selected.

In the non-strict case for $i = 1$,

$$
\begin{aligned}
& E[\![e]\!]\ \rho \\
&= E[\![e\,[e'\,[x_K/x'] \,/\, \textbf{let}\ \ K\ x' = x\ \ \textbf{in}\ \ e']]\!]\ \ (\rho\,[\mathsf{snd}(\rho(x))/x_K]) \\
&= E[\![\textbf{let}\ \ K\ x_K = x\ \ \textbf{in}\ \ e\,[e'\,[x_K/x'] \,/\, \textbf{let}\ \ K\ x' = x\ \ \textbf{in}\ \ e']]\!]\ \rho \\
&= E[\![\|e\|_{x,\, K\eta}]\!]\ \rho\ ,
\end{aligned}
$$

and for $i > 1$,

$$E[\![e]\!]\ \rho\ = E[\![\|e\|_{x,\, K_1\tau_1 + \ldots + K_n\tau_n}]\!]\ \rho\ .$$

$\rho(x) \notin \{K_i\} \times V$ for any i.

Then, in the strict case, $E[\![\textbf{let}\ \ K_i\ x' = x\ \ \textbf{in}\ \ e']\!]\ \rho' = \perp$, whence

$$
\begin{aligned}
& E[\![e]\!]\ \rho \\
&= E[\![e\,[\textbf{fail} \,/\, \textbf{let}\ \ K_i\ x' = x\ \ \textbf{in}\ \ e']]\!]\ \rho \\
&= E[\![\|e\|_{x,\, K_1\eta_1 + \ldots + K_n\eta_n}]\!]\ \rho\ ,
\end{aligned}
$$

since the last **else** branch gets selected.

In the non-strict case for $i = 1$,

$$E[\![\textbf{let}\ \ K\ x' = x\ \ \textbf{in}\ \ e']\!]\ \rho'$$

$= E[\![e']\!]\ (\rho'[\bot/x'])$

$= E[\![e'[x_K/x']]\!]\ (\rho'[\bot/x_K])$.

Therefore,

$E[\![e]\!]\ \rho$

$= E[\![e[e'[x_K/x']/\mathbf{let}\ \ K\ x' = x\ \ \mathbf{in}\ \ e']]\!]\ (\rho[\bot/x_K])$

$= E[\![\|e\|_{x, K\eta}]\!]\ \rho$.

■

Theorem 4.8 [Preservation of semantics] If $A \vdash' e : \tau \Rightarrow \tilde{e}$, then $E[\![e]\!]\ \rho\ = E[\![\tilde{e}]\!]\ \rho$ for arbitrary ρ.

Proof: By structural induction on e.

Corollary 4.9 [Semantic soundness] If e is a closed term, $A \vdash' e : \tau$ and $\models_{\rho, \psi} A$ as defined previously, then $E[\![e]\!]\ \rho \in T[\![\tau]\!]\ \psi$.

Proof: Follows immediately from the two theorems, observing that $\lfloor \tau \rfloor = \tau$ and $\lfloor A \rfloor = A$, since neither τ nor A contain any κ's.

■

5 An Extension of Haskell with First-Class Abstract Types

This chapter introduces an extension of the functional language Haskell with existential types. Existential types combine well with the systematic overloading polymorphism provided by Haskell type classes. Briefly, we extend Haskell's **`data`** declaration in a similar way as the ML datatype declaration above. In Haskell, it is possible to specify what type class a (universally quantified) type variable belongs to. In our extension, we can do the same for existentially quantified type variables. This lets us use type classes as signatures of abstract data types; we can then construct heterogeneous aggregates over a given type class. A type reconstruction algorithm is given, and semantic soundness is shown by translating into an extension of the language from Chapter 3.

5.1 Introduction

Haskell [HPJW+92] uses type classes as a systematic approach to ad-hoc polymorphism, otherwise known as overloading. Type classes capture common sets of operations. A particular type may be an instance of a type class, and has an operation corresponding to each operation defined in the type class. Type classes may be arranged hierarchically.

In [WB89], Wadler and Blott called for a closer exploration of the relationship between type classes and abstract data types. After an initial exploration described in [LO91], we now present an extension of Haskell with datatypes whose component types may be existentially quantified.

In Haskell, an algebraic datatype declaration is of the form

$$\textbf{data}\ \ [c \Rightarrow] T\ a_1 \ldots a_n\ \ \textbf{=}\ \ K_1\ t_{11} \ldots t_{1k_1} \mid \ldots \mid K_m\ t_{m1} \ldots t_{mk_m}$$

It introduces a new type constructor T with value constructors $K_1, \ldots, K_m$. The optional context c specifies of which type classes the type variables $a_1, \ldots, a_n$ are instances. The constructors are used in two ways: as functions to construct values, and in patterns to decompose values already constructed. The types of the constructors are universally quantified over the type variables $a_1, \ldots, a_n$; no other type variable may appear free in the component types t_{ij}.

We describe an extension of Haskell analogous to the extension of ML described above. Type variables that appear free in the component types are interpreted as existentially quantified. In addition to the "global" context for the universally quantified parameters of the type constructor, we introduce "local" contexts for each value constructor. The local context specifies of which type classes the existentially quantified type variables in the component types are instances. The extended datatype declaration is of the form

$$\begin{aligned} \textbf{data}\ \ [c \Rightarrow] T\ a_1 \ldots a_n\ \ \textbf{=}\ \ & [c_1 \Rightarrow] K_1\ t_{11} \ldots t_{1k_1} \\ \mid\ & \ldots \\ \mid\ & [c_m \Rightarrow] K_m\ t_{m1} \ldots t_{mk_m} \end{aligned}$$

When constructing a value using a constructor with an existentially quantified component type, the existential type variables instantiate to the actual types of the corresponding function arguments, and we lose any information on the actual types. However, we know that these types are instances of the same type classes as the corresponding existential type variables. This

means that we have types whose identity is unknown but which support the operations specified by their type classes. Therefore we regard type classes as signatures of abstract types.

5.2 Some Motivating Examples

5.2.1 Minimum over a Heterogeneous List

This example is the extended Haskell version of the example given in Section 3.2.1. We first define a type class **Key** defining the operation **whatkey** needed to obtain an integer value from the value to be compared.

```
class Key a where
    whatkey :: a -> Int
```

We now define a datatype **KEY** with a single constructor **key**. The component type of **key** is the type variable **a**, which is existentially quantified and is required to be an instance of type class **Key**.

```
data KEY = (Key a) => key a
```

We further define several instances of **Key** along with their implementations of the function **whatkey**.

```
instance Key Int   where whatkey = id
instance Key Float where whatkey = round
instance Key [a]   where whatkey = length
instance Key Bool  where whatkey =
                              \x -> if x then 1 else 0
```

A heterogeneous list of values of type **KEY** could be defined as follows:

```
hetlist = [key 3,key [1,2,3,4],key 7,
           key True,key 12]
```

The **min** function finds the minimum over a list of **KEY**'s by decomposing the elements of the list and comparing their corresponding integer values obtained by applying **whatkey**.

```
min [x] = x
min ((key v1):xs) =
    case min xs of
        key v2 ->
            if whatkey v1 <= whatkey v2 then
                key v1
            else
                key v2
```

Then **min hetlist** evaluates to **key True**, as this is the element for which **whatkey** returns the smallest number.

5.2.2 Abstract Stack with Multiple Implementations

We also give the extended Haskell version of the stack example from Section 3.2.2. However, these stacks have a fixed element type, since Haskell type classes cannot be parameterized. An extension of Haskell with parameterized type classes is found in [CHO92]; it could in turn be extended with existential types, which would allow us to have polymorphic abstract stacks.

An integer stack is described by the following type class:

```
class Stack a where
    empty   :: a
    push    :: Int -> a -> a
    pop     :: a -> a
    top     :: a -> Int
    isempty :: a -> Bool
```

To achieve abstraction, we define the corresponding datatype of "encapsulated" stacks:

```
data STACK = (Stack a) => Stack a
```

We define two stack implementations, one based on a list of integers:

```
instance Stack [Int] where
    empty   = []
    push    = (:)
    pop     = tail
    top     = head
    isempty = null
```

and one based on an integer array:

```
maxIndex :: Int
maxIndex = 100

data FixedArray = Fixarr Int (Array Int Int)

instance Stack FixedArray where
    empty = Fixarr 0 (listArray(1,maxIndex)[])
    push a (Fixarr i s) =
        if i >= maxIndex then
            error "stack size exceeded"
        else
            Fixarr(i+1)(s // [(i+1) := a])
    pop(Fixarr i s) =
        if i <= 0 then
            error "stack empty"
        else
            Fixarr(i-1) s
    top(Fixarr i s) =
        if i <= 0 then
            error "stack empty"
        else
            s!i
    isempty(Fixarr i s) = i <= 0

arrayStack xs = Stack(Fixarr(length xs)
                      (listArray(1,maxIndex) xs)
```

As we saw in Section 3.2.2, it is convenient to define wrapper functions that apply the functions operating on instances of the type class **`Stack`** to an encapsulated value of type **`STACK`**; these "outer" wrappers open the encapsulated stack, apply the corresponding "inner" operations, and close the stack again. This provides dynamic dispatching of operations across different implementations of **`STACK`**. The wrapper **`wpush`** is defined as follows:

```
wpush a (Stack s) = Stack(push a s)
```

We can define the following list, which is a homogeneous list of two different implementations of **`STACK`**:

```
stackList = [Stack([1,2,3] :: [Int]),
             arrayStack([5,6,7] :: [Int])]
```

Using the wrapper **`wpush`** and the built-in function **`map`**, we can uniformly push an integer onto each element of the list:

```
map (wpush 8) stackList
```

5.3 Syntax

The formal treatment of our extension of Haskell builds on the article [NS91] by Nipkow and Snelting, who are the first to give an accurate treatment of type inference in Haskell. Our language is an extension of theirs with algebraic data types.

5.3.1 Language Syntax

Identifiers x

Constructors K

Type constructors t

Expressions e ::= `()` | **`true`** | **`false`** |
x | (e_1, e_2) | $e\ e'$ | $\lambda x.e$ |

$$\texttt{let}\ \ x = e\ \ \texttt{in}\ \ e' \mid$$
$$K \mid \texttt{is}\ \ K \mid \texttt{let}\ \ K\ x = e\ \ \texttt{in}\ \ e'$$

Declarations d ::= $\texttt{data}\ \ t = \forall\alpha_{\gamma_1} \ldots \alpha_{\gamma_n} . \chi\ \ \texttt{in}\ \ e \mid$

$$\texttt{class}\ \ \gamma \leq \gamma_1, \ldots, \gamma_n\ \ \texttt{where}$$
$$x_1 : \forall\alpha_\gamma . \tau_1, \ldots, x_k : \forall\alpha_\gamma . \tau_k \mid$$
$$\texttt{inst}\ \ t : (\gamma_1, \ldots, \gamma_n)\,\gamma\ \ \texttt{where}$$
$$x_1 = e_1, \ldots, x_k = e_k$$

Programs p ::= $d_1 \ldots d_n e$

5.3.2 Type Syntax

Type variables α

Skolem functions κ

Type constructors t

Types τ ::= $unit \mid bool \mid \alpha_\gamma \mid \tau_1 \times \tau_2 \mid \tau \rightarrow \tau' \mid$
$\kappa \mid t(\tau_1, \ldots, \tau_n) \mid \chi$

Recursive types χ ::= $\mu\beta . K_1\eta_1 + \ldots + K_m\eta_m$ where $K_i \neq K_j$ for $i \neq j$

Existential types η ::= $\exists\alpha_\gamma . \eta \mid \tau$

Type schemes σ ::= $\forall\alpha_\gamma . \sigma \mid \eta \rightarrow \tau \mid \tau$

Assumptions a ::= $\sigma / x \mid \sigma / K$

Our type syntax includes recursive types χ and Skolem type constructors κ; the latter are used to type identifiers bound by a pattern-matching **let**

whose type is existentially quantified. Explicit existential types arise only as domain types of value constructors. Further, let $\Sigma[K\eta]$ stand for sum type contexts such as $K_1\eta_1 + \ldots + K_m\eta_m$, where $K_i = K$ and $\eta_i = \eta$ for some i. Our type syntax also includes explicit type constructors t; this makes it possible to extend the order-sorted signature with arities for user-defined type constructors.

5.4 Type Inference

5.4.1 Instantiation and Generalization of Type Schemes

$\forall\alpha_{\gamma_1}\ldots\alpha_{\gamma_n}.\tau \geq_C \tau'$ iff there are types $\tau_1, \ldots\tau_n$ of sorts $\gamma_1, \ldots, \gamma_n$, respectively, such that $\tau' = \tau\left[\tau_1/\alpha_{\gamma_1}, \ldots, \tau_n/\alpha_{\gamma_n}\right]$

$\exists\alpha_{\gamma_1}\ldots\alpha_{\gamma_n}.\tau \geq_C \tau'$ iff there are types $\tau_1, \ldots\tau_n$ of sorts $\gamma_1, \ldots, \gamma_n$, respectively, such that $\tau' = \tau\left[\tau_1/\alpha_{\gamma_1}, \ldots, \tau_n/\alpha_{\gamma_n}\right]$

In addition to FV, the set of free type variables in a type scheme or assumption set, we use FS, the set of those Skolem type constructors that occur in a type scheme or assumption set, and FT, the set of defined type constructors in a type scheme.

5.4.2 Inference Rules for Expressions

The first five typing rules are the same as in the system described in [NS91].

$$(\mathrm{VAR}^+) \quad \frac{A(x) \geq_C \tau}{(A, C) \vdash^+ x : \tau}$$

$$\text{(PAIR}^+\text{)}\quad \frac{(A, C) \vdash^+ e_1 : \tau_1 \qquad (A, C) \vdash^+ e_2 : \tau_2}{(A, C) \vdash^+ (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\text{(APPL}^+\text{)}\quad \frac{(A, C) \vdash^+ e : \tau' \to \tau \qquad (A, C) \vdash^+ e' : \tau'}{(A, C) \vdash^+ e\ e' : \tau}$$

$$\text{(ABS}^+\text{)}\quad \frac{(A\,[\tau'/x], C) \vdash^+ e : \tau}{(A, C) \vdash^+ \lambda x.e : \tau' \to \tau}$$

$$\text{(LET}^+\text{)}\quad \frac{\begin{array}{c} FV(\tau) \setminus FV(A) = \{\alpha_{\gamma_1}, \ldots, \alpha_{\gamma_n}\} \\ (A, C) \vdash^+ e : \tau \qquad (A\left[\forall\overline{\alpha_{\gamma_n}}.\tau/x\right], C) \vdash^+ e' : \tau' \end{array}}{(A, C) \vdash^+ \mathbf{let}\ \ x = e\ \ \mathbf{in}\ \ e' : \tau'}$$

The new rules CONS$^+$, TEST$^+$, and PAT$^+$ are used to type value constructors, **is** expressions, and pattern-matching **let** expressions, respectively.

$$\text{(CONS}^+\text{)}\quad \frac{A(K) \geq_C \eta \to t(\overline{\tau_n}) \qquad \eta \leq_C \tau}{(A, C) \vdash^+ K : \tau \to t(\overline{\tau_n})}$$

The CONS$^+$ rule observes the fact that existential quantification in argument position means universal quantification over the whole function type; this is expressed by the second premise.

$$\text{(TEST}^+\text{)}\quad \frac{A(K) \geq_C \eta \to t(\overline{\tau_n})}{(A, C) \vdash^+ \mathbf{is}\ \ K : t(\overline{\tau_n}) \to \mathit{bool}}$$

The TEST$^+$ rule ensures that **is** K is applied only to arguments whose type is the same as the result type of constructor K.

$$\text{(PAT}^{+}\text{)}\quad \frac{\begin{array}{c} A(K) \geq_C (\exists \overline{\beta_{\delta_k}}.\tau) \rightarrow t(\overline{\tau_n}) \qquad (A, C) \vdash^{+} e : t(\overline{\tau_n}) \\ \{\kappa_1, \ldots, \kappa_k\} \cap (FS(\tau') \cup FS(A)) = \varnothing \\ \begin{pmatrix} A\left[\tau\left[\kappa_i/\beta_{\delta_i} \middle| i = 1 \ldots k\right]/x\right] \\ C\left[\kappa_i : \delta_i \middle| i = 1 \ldots k\right] \end{pmatrix} \vdash^{+} e' : \tau' \end{array}}{(A, C) \vdash^{+} \textbf{let}\ \ K\ x = e\ \ \textbf{in}\ \ e' : \tau'}$$

The last rule, PAT^{+}, governs the typing of pattern-matching **let** expressions. It requires that the expression e be of the same type as the result type of the constructor K. The body e' is typed under the assumption set extended with an assumption about the bound identifier x. The new Skolem type constructors must not appear in A; this ensures that they do not appear in the type of any identifier free in e' other than x. It is also guaranteed that the Skolem type constructors do not appear in the result type τ'. The Skolem type constructors $\kappa_1, \ldots, \kappa_k$ replace the existentially quantified type variables of sorts $\delta_1, \ldots, \delta_k$. Thus the body of the **let** expression is typed under the extended signature containing appropriate arities for $\kappa_1, \ldots, \kappa_k$. The pattern-matching **let** expression is monomorphic in the sense that the type of the bound variable x is not generalized. This restriction is sufficient to guarantee a type-preserving translation into a target language (see Section 5.6.5). The **case** expression in Haskell syntax corresponds to a nested **if** with an **is** and a pattern-matching **let** expression for each case.

5.4.3 Inference Rules for Declarations and Programs

The rules for class and instance declarations, and programs are the same as in [NS91]. We add the DATA^{+} rule to elaborate a recursive datatype declaration.

$$(\text{CLASS}^+)\ \frac{FT(\tau_1) \cup \ldots \cup FT(\tau_k) \subseteq \mathrm{Dom}(C)}{\begin{array}{c}(A, C)\ \vdash^+ \mathbf{class}\ \ \gamma \leq \gamma_1, \ldots, \gamma_n\ \ \mathbf{where} \\ x_1 : \forall\alpha_\gamma.\tau_1, \ldots, x_k : \forall\alpha_\gamma.\tau_k : \\ (A\,[\forall\alpha_\gamma.\tau_i/x_i \mid i = 1\ldots k], C\,[\gamma \leq \gamma_1, \ldots, \gamma_n])\end{array}}$$

$$(\text{INST}^+)\ \frac{\begin{array}{c}t \in \mathrm{Dom}(C) \qquad A(x_i) = \forall\alpha_\gamma.\tau_i \\ (A, C)\ \vdash^+ e_i : \tau_i\left[t(\overline{\alpha_{\gamma_n}})/\alpha_\gamma\right] \qquad i = 1\ldots k\end{array}}{\begin{array}{c}(A, C)\ \vdash^+ \mathbf{inst}\ \ t : (\overline{\gamma_n})\,\gamma\ \ \mathbf{where}\ \ x_1 = e_1, \ldots, x_k = e_k : \\ (A, C\,[t : (\overline{\gamma_n})\,\gamma])\end{array}}$$

$$(\text{PROG}^+)\ \frac{\begin{array}{c}(A_{i-1}, C_{i-1})\ \vdash^+ d_i : (A_i, C_i) \qquad i = 1\ldots n \\ (A_n, C_n)\ \vdash^+ e : \tau\end{array}}{(A_0, C_0)\ \vdash^+ d_1 \ldots d_n e : \tau}$$

$$(\text{DATA}^+)\ \frac{\begin{array}{c}\sigma = \forall\overline{\alpha_{\gamma_n}}.\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m \\ FT(\sigma) \subseteq \mathrm{Dom}(C) \qquad t \notin \mathrm{Dom}(C)\end{array}}{\begin{array}{c}(A, C)\ \vdash^+ \mathbf{data}\ \ t = \sigma : \\ \left(\begin{array}{c}A\left[\forall\overline{\alpha_{\gamma_n}}.\eta_i\left[t(\overline{\alpha_{\gamma_n}})/\beta\right] \to t(\overline{\alpha_{\gamma_n}})/K_i \mid i = 1\ldots m\right] \\ C\,[t : (\overline{\gamma_n})\,\Omega]\end{array}\right)\end{array}}$$

The DATA$^+$ rule adds assumptions about the value constructors to the assumption set, and extends the signature with an appropriate arity for the new type constructor. Whereas recursive datatypes were anonymous in the two preceding chapters, they are now represented by named type constructors. This is necessary since the order-sorted signature C may contain arity declarations for user-defined type constructors. We avoid using a separate type

constructor environment; therefore, in an assumption σ/K about a value constructor, σ now is the type scheme for K when regarded as a function, as opposed to the type scheme describing the entire recursive datatype K belongs to.

5.4.4 Relation to the Haskell Type Inference System

Theorem 5.1 [Conservative extension] Let Mini-Haskell' be an extension of Mini-Haskell with recursive datatypes and a monomorphic pattern-matching **`let`** expression, but without existential quantification. Then, for any Mini-Haskell' program p, $(A, C) \vdash^{+} p : \tau$ iff $(A, C) \vdash_{\mathrm{MH'}} p : \tau$.

Proof: By structural induction on p.

Corollary 5.2 [Conservative extension] Our type system is a conservative extension of the Mini-Haskell type system described in [NS91], in the following sense: For any Mini-Haskell program p, $(A, C) \vdash^{+} p : \tau$ iff $(A, C) \vdash_{\mathrm{MH}} p : \tau$.

Proof: Follows immediately from Theorem 5.1.

5.5 Type Reconstruction

The type reconstruction algorithm is a translation from the deterministic typing rules, using order-sorted unification [SS85][MGS89] instead of standard unification.

5.5.1 Unitary Signatures for Principal Types

The article [NS91] describes several conditions necessary to guarantee unitary signatures, which are sufficient to guarantee principal types. First, to make a signature C regular and downward compete, we perform the following two steps to obtain a new signature C_R:

- For any two incomparable classes $\gamma_1, \gamma_2 \in \mathrm{Dom}(C)$, we introduce a new class declaration **class** $\gamma \leq \gamma_1, \gamma_2$ with an empty **where** part combining the operations of γ_1 and γ_2.
- Then, for each type constructor with instance declarations

  inst $t : (\overline{\gamma_{1n}})\,\gamma_1$ **where** ...

  inst $t : (\overline{\gamma_{2n}})\,\gamma_2$ **where** ...

  introduce another instance declaration of the form

  inst $t : (\gamma_{11} \wedge \gamma_{21}, \ldots, \gamma_{1n} \wedge \gamma_{2n})\ (\gamma_1 \wedge \gamma_2)$

  where $\gamma \wedge \delta$ is simply the additionally declared class if γ and δ are incomparable, or otherwise the lower one in the class hierarchy.

Note that Haskell uses multiple class assertions for type variables to express this conjunction of classes.

Since regular signatures alone do not guarantee the existence of principal types, we impose the following two conditions on C_R, which are also present in Haskell:

- Injectivity: A type constructor may not be declared as an instance of a particular class more than once in the same scope
- Subsort reflection: If $\gamma_1, \ldots, \gamma_m$ are the immediate superclasses of δ, a declaration **inst** $t : (\overline{\delta_n})\,\delta$ **where** ... must be preceded by declarations **inst** $t : (\overline{\gamma_n^i})\,\gamma_i$ **where** ... such that δ_j is a subclass of γ_j^i for al $i = 1 \ldots m$ and $j = 1 \ldots n$.

As discussed in [NS91], a Haskell signature that satisfies these conditions is unitary.

5.5.2 Auxiliary Functions

In our algorithm, we need to instantiate universally quantified types and generalize existentially quantified types. Both are handled in the same way.

$inst_{\forall}(\forall\alpha_{\gamma_1}\ldots\alpha_{\gamma_n}.\tau) = \tau\left[\beta_{\gamma_1}/\alpha_{\gamma_1}, \ldots, \beta_{\gamma_n}/\alpha_{\gamma_n}\right]$ where $\beta_{\gamma_n}, \ldots, \beta_{\gamma_n}$ are fresh type variables

$inst_{\exists}(\exists\alpha_{\gamma_1}\ldots\alpha_{\gamma_n}.\tau) = \tau\left[\beta_{\gamma_1}/\alpha_{\gamma_1}, \ldots, \beta_{\gamma_n}/\alpha_{\gamma_n}\right]$ where $\beta_{\gamma_n}, \ldots, \beta_{\gamma_n}$ are fresh type variables

$osu_C(\tau, \tau')$ the most general unifier of τ and τ' under order-sorted signature C

5.5.3 Algorithm

Our type reconstruction function takes an assumption set, an order-sorted signature, and an expression, and it returns a substitution and a type expression. There is one case for each typing rule.

$TE(A, C, x) =$

$(Id, inst_{\forall}(A(x)))$

$TE(A, C, (e_1, e_2)) =$

let $(S_1, \tau_1) = TE(A, C, e_1)$

$(S_2, \tau_2) = TE(S_1A, C, e_2)$

in $(S_2S_1, S_2\tau_1 \times \tau_2)$

$TE(A, C, ee') =$

let $(S, \tau) = TE(A, C, e)$
$(S', \tau') = TE(SA, C, e')$
β be a fresh type variable
$U = osu_C(S'\tau, \tau' \to \beta)$
in $(US'S, U\beta)$

$TE(A, C, \lambda x.\, e) =$

let β be a fresh type variable
$(S, \tau) = TE(A\,[\beta/x], C, e)$
in
$(S, S\beta \to \tau)$

$TE(A, C,$ **let** $x = e$ **in** $e') =$

let $(S, \tau) = TE(A, C, e)$
$(S', \tau') = TE(SA\,[gen\,(SA, \tau)/x], C, e')$
in
$(S'S, \tau')$

$TE(A, C, K) =$

let $\eta \to \tau = inst_\forall\,(A\,(K))$
in $(Id, (inst_\exists\,(\eta)) \to \tau)$

$TE(A, C,$ **is** $K) =$

let $\eta \to \tau = inst_\forall\,(A\,(K))$
in $(Id, \tau \to bool)$

$TE(A, C,$ **let** $K\ x = e$ **in** $e') =$

let $(S, \tau) = TE(A, C, e)$

$(\exists \overline{\beta_{\delta_k}}.\tau_0) \rightarrow t(\overline{\alpha_{\gamma_n}}) = inst_{\forall}(A(K))$

$U = osu_C(\tau, t(\overline{\alpha_{\gamma_n}}))$

$\kappa_1, \ldots, \kappa_k$ fresh type constructors

$\tau_\kappa = (U\tau_0)\left[\kappa_i / \beta_{\delta_i} \middle| i = 1 \ldots k\right]$

$C' = C\,[\kappa_i : \delta_i \mid i = 1 \ldots k]$

$(S', \tau') = TE(USA\,[\tau_\kappa / x], C', e')$

in

if $\{\kappa_1, \ldots, \kappa_k\} \cap (FS(S'USA) \cup FS(\tau')) = \varnothing$ **then**

$(S'US, \tau')$

$TD(A, C,$ **data** $t = \sigma) =$

let $\forall \alpha_{\gamma_1} \ldots \alpha_{\gamma_n}.\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m = \sigma$ **in**

if $FV(\sigma) = \varnothing \wedge$

$t \notin \mathrm{Dom}(C) \wedge FT(\sigma) \subseteq \mathrm{Dom}(C)$

then

$$\begin{pmatrix} A\left[\forall\overline{\alpha_{\gamma_n}}.\eta_i\left[t(\overline{\alpha_{\gamma_n}})/\beta\right] \rightarrow t(\overline{\alpha_{\gamma_n}})/K_i \middle| i = 1 \ldots m\right] \\ C\,[t : (\overline{\gamma_n})\,\Omega] \end{pmatrix}$$

$TD(A, C,$ **class** $\gamma \leq \gamma_1, \ldots, \gamma_n$ **where** $x_1 : \forall\alpha_\gamma.\tau_1, \ldots, x_k : \forall\alpha_\gamma.\tau_k) =$

$(A\,[\forall\alpha_\gamma.\tau_i / x_i \mid i = 1 \ldots k], C\,[\gamma \leq \gamma_1, \ldots, \gamma_n])$

$TD(A, C,$ **inst** $t : (\overline{\gamma_n})\,\gamma$ **where** $x_1 = e_1, \ldots, x_k = e_k) =$

$(A, C\,[t : (\overline{\gamma_n})\,\gamma])$

$TD(A, C, d_1 \ldots d_n) =$

let $(A', C') = TD(A, C, d_1)$ **in**
$TD(A, C', d_2 \ldots d_n)$

$TP(A, C, d_1 \ldots d_n e) =$

let $(A', C') = TD(A, C, d_1 \ldots d_n)$ **in**
$TE(A', C', e)$

5.5.4 Syntactic Soundness and Completeness of Type Reconstruction

Lemma 5.3 [Stability of $\vdash^+$] If $(A, C) \vdash^+ e : \tau$ and S is a substitution, then $(SA, C) \vdash^+ e : S\tau$ also holds. Moreover, if there is a proof tree for $(A, C) \vdash^+ e : \tau$ of height n, then there is also a proof tree for $(SA, C) \vdash^+ e : S\tau$ of height less or equal to n.

Theorem 5.4 [Syntactic soundness] If $TC(A, C, e) = (S, \tau)$, then $(SA, C) \vdash^+ e : \tau$.

Definition 5.1 [Principal type] τ is a principal type of expression e under assumption set A and signature C if $(A, C) \vdash^+ e : \tau$ and whenever $(A, C) \vdash^+ e : \tau'$ then there is a substitution S such that $S\tau = \tau'$.

Theorem 5.5 [Syntactic completeness] If $(\hat{S}A, C) \vdash e : \hat{\tau}$, then $TC(A, C, e) = (S, \tau)$ and there is a substitution R such that $\hat{S}A = RSA$ and $\hat{\tau} = R\tau$.

Proof: We extend Nipkow's recent work on type classes and order-sorted unification and extend it with existential types.

We assume that the signature built from the global **`class`** and **`inst`** declarations is unitary. Clearly, the extended signature used to type the body of a pattern-matching **`let`** expression is also unitary, since the Skolem type constructors κ_i are unique, and each κ_i appears in only one arity declaration. The latter trivially guarantees injectivity and subsort reflection.

■

5.6 Semantics

As in [NS91] [WB89], we give an inference-guided translation to the target language, an enhanced version of our extension of ML with existential types described in Chapter 3. Type classes and instances are replaced by *(method) dictionaries*, which contain all the operations associated with a particular instance of a type class. The translation rules are of the form $(A, C) \vdash^{+} e : \tau \Rightarrow \tilde{e}$ and mean "in the context (A, C), e is assigned type τ and translates to $\tilde{e}$."

5.6.1 Target Language

Our extension of Mini-Haskell is translated into an extended version of the language presented in Chapter 3. As a generalization of pair types, the language contains all n-ary product types $\alpha_1 \times \ldots \times \alpha_n$ with expressions $(e_1, \ldots, e_n)$ and projection functions π_i^n of type $\alpha_1 \times \ldots \times \alpha_n \rightarrow \alpha_i$. The PAIR rule is superseded by the TUPLE rule:

$$\text{(TUPLE)}\ \frac{A \vdash e_1 : \tau_1 \qquad \ldots \qquad A \vdash e_n : \tau_n}{A \vdash (e_1, \ldots, e_n) : \tau_1 \times \ldots \times \tau_n}$$

Semantically, expressions of the form

`let` $K(x_1, \ldots, x_n) = e$ **`in`** e'

are regarded as short forms for nested **let** expressions of the form

$$\begin{array}{l}\textbf{let}\ \ K\ z = e\ \ \textbf{in}\\ \quad\textbf{let}\ \ x_1 = \pi_1^n z, \ldots, x_n = \pi_n^n z\ \ \textbf{in}\ \ e'\end{array}$$

and are typed by following PAT'' rule:

$$\text{(PAT'')}\quad \frac{\begin{array}{c}A \vdash e : \mu\beta.\Sigma\,[K\eta] \qquad \eta = \exists\beta_1 \ldots \beta_k.\tau_1 \times \ldots \times \tau_n\\ \tau'_1 \times \ldots \times \tau'_n = \mathit{skolem}(A, \eta) \qquad \mathit{FS}(\tau') \subseteq \mathit{FS}(A)\\ \sigma_1 = \mathit{gen}(A, \tau'_1) \qquad \ldots \qquad \sigma_n = \mathit{gen}(A, \tau'_n)\\ A\,[\sigma_1/x_1, \ldots, \sigma_n/x_n] \vdash^+ e' : \tau'\end{array}}{A \vdash \textbf{let}\ \ K\,(x_1, \ldots, x_n) = e\ \ \textbf{in}\ \ e' : \tau'}$$

This rule is semantically sound, since the translation of the short form to the full form is type-preserving: an application of the PAT'' rule is replaced by an application of the PAT rule followed by n successive applications of the LET rule, using appropriate typings for the tuple projections.

5.6.2 Dictionaries and Translation of Types

We call the translated types "ML-types" to distinguish them from the original ones. ML-types introduce a method dictionary for each sorted type variable in the original type; each sorted type variable is then replaced by an ordinary type variable.

A class declaration

$$(A, C) \vdash^+ \textbf{class}\ \ \gamma \leq \gamma_1, \ldots, \gamma_n\ \ \textbf{where}\ \ x_1 : \forall\alpha_\gamma.\tau_1, \ldots, x_k : \forall\alpha_\gamma.\tau_k$$

introduces a new ML-type for method dictionaries of this class,

$$\gamma(\alpha) = \tau_1\,[\alpha/\alpha_\gamma] \times \ldots \times \tau_k\,[\alpha/\alpha_\gamma] \times \gamma_1(\alpha) \times \ldots \times \gamma_n(\alpha)$$

where the type parameter α stands for the type of the instance. The first k components of $\gamma(\alpha)$ are the operations corresponding to class γ. The next n components are the dictionaries for all immediate superclasses $\gamma_1, \ldots, \gamma_n$ of γ. Note that the dictionary $\Omega(\alpha)$ for the top of the class hierarchy is the empty product type.

Instead of defining $\gamma(\alpha)$ directly, the dictionary type is defined in terms of access functions $x_1, \ldots, x_k$ to extract operations from a dictionary, and access functions $\gamma_{1_\gamma}, \ldots, \gamma_{n_\gamma}$ to extract the dictionaries for the immediate superclasses from a dictionary.

Coercion functions are needed to convert a dictionary α_γ of a class γ into a dictionary of a superclass of γ; they are defined the same way as in [NS91]:

$$cast_C(\alpha_{\gamma'}, \gamma') = \begin{cases} \alpha_\gamma & \text{if } \gamma = \gamma' \\ (\gamma'_\delta \; cast_C(\alpha_{\gamma'}, \delta)) & \text{if } \gamma \leq \delta \wedge \gamma' \in super_C(\delta) \end{cases}$$

If there is more than one path from γ to γ' with respect to $\geq_C$, $cast_C$ chooses an arbitrary fixed path. The immediate superclasses of a class γ are defined as:

$$super_C(\gamma) = \{\gamma' \mid \gamma < \gamma' \wedge \neg\exists\delta.\, \gamma < \delta < \gamma'\}$$

The method dictionary for an instance τ of a class γ within the signature C is defined as

$$dict_C(\alpha_{\gamma'}, \gamma) = cast_C(\alpha_{\gamma'}, \gamma)$$

$$dict_C(t(\tau_1, \ldots, \tau_n), \gamma) = \gamma_t\,(dict_C(\tau_1, \gamma_1)) \ldots (dict_C(\tau_n, \gamma_n)) \text{ if according to } C,\ t(\tau_1, \ldots, \tau_n) \text{ is sort-correct with resulting sort } \gamma$$

Note that on the left hand side, $\alpha_{\gamma'}$ is a type variable, and on the right hand side, an identifier that stands for a dictionary of class γ'. As we will see below, the function γ_t is the dictionary defined for the type constructor t in the translation of the corresponding instance declaration. Its arguments are the dictionaries for the actual type parameters in an application of t.

We define the translation function for type schemes as follows:

$$ML(\forall\overline{\alpha_{\gamma_n}}.\tau) = \forall\overline{\alpha_n}.\gamma_1(\alpha_1) \to \ldots \to \gamma_n(\alpha_n) \to \tau\left[\alpha_1/\alpha_{\gamma_1}, \ldots, \alpha_n/\alpha_{\gamma_n}\right]$$

where each α_{γ_i} uniquely maps to an α_t. For existential component types of user-defined datatypes, the corresponding ML-types need to include the dictionaries for the existentially quantified type variables. This reflects the way operations on existential types are explicitly included in the datatype components in Chapter 3.

$$ML(\exists\overline{\beta_{\delta_k}}.\tau) = \exists\overline{\beta_k}.\tau\left[\beta_1/\beta_{\delta_1}, \ldots, \beta_k/\beta_{\delta_k}\right] \times \delta_1(\beta_1) \times \ldots \times \delta_k(\beta_k)$$

The resulting translation function for user-defined recursive type schemes is

$$ML(\forall\overline{\alpha_{\gamma_n}}.\mu\beta.\Sigma\,[K\eta]) = \forall\overline{\alpha_n}.\mu\beta.\Sigma\,[K\ ML(\eta)]\left[\alpha_1/\alpha_{\gamma_1}, \ldots, \alpha_n/\alpha_{\gamma_n}\right]$$

Note that the dictionaries for the universally quantified type variables are *not* included in the component types, as they are determined by the actual instance types substituted for the type variables. Since user-defined datatypes are anonymous in the target language, the translation function for the type of a value constructor K is given by the entire recursive type scheme to which K belongs:

$$ML(A(K)) = ML(\forall\overline{\alpha_{\gamma_n}}.\mu\beta.\Sigma\,[K\eta]) \text{ where } A(K) = \forall\overline{\alpha_{\gamma_n}}.\eta \to \tau$$

The function ML extends to assumption sets as follows:

$$ML(A) \;=\; \{ML(A(x))/x \mid x \in \mathrm{Dom}A\}$$

5.6.3 Translation Rules for Declarations and Programs

The first three translation rules concern class declarations, instance declarations, and programs. They are the same as in [NS91]. Declarations translate to **let** expressions without bodies.

$$\text{(CLASS}^+\text{)}\ \frac{FT(\tau_1) \cup \ldots \cup FT(\tau_k) \subseteq \mathrm{Dom}(C)}{\begin{array}{l}(A, C) \vdash^+ \textbf{class}\ \ \gamma \le \gamma_1, \ldots, \gamma_n\ \ \textbf{where} \\ \qquad x_1 : \forall\alpha_\gamma.\tau_1, \ldots, x_k : \forall\alpha_\gamma.\tau_k : \\ \qquad (A\,[\forall\alpha_\gamma.\tau_i/x_i \mid i = 1\ldots k],\, C\,[\gamma \le \gamma_1, \ldots, \gamma_n]\,) \Rightarrow \\ \qquad \textbf{let}\ \ x_1 = \pi_1^{k+n}, \ldots, x_k = \pi_k^{k+n}, \gamma_{1_\gamma} = \pi_{k+1}^{k+n}, \ldots, \gamma_{n_\gamma} = \pi_{k+n}^{k+n}\end{array}}$$

$$\text{(INST}^+\text{)}\ \frac{\begin{array}{c}super_C(\gamma) = \{\gamma^1, \ldots, \gamma^s\} \\ t \in \mathrm{Dom}(C) \qquad A(x_i) = \forall\alpha_\gamma.\tau_i \\ (A, C) \vdash^+ e_i : \tau_i\left[t(\overline{\alpha_{\gamma_n}})/\alpha_\gamma\right] \Rightarrow \tilde{e}_i \qquad i = 1\ldots k\end{array}}{\begin{array}{l}(A, C) \vdash^+ \textbf{inst}\ \ t : (\overline{\gamma_n})\,\gamma\ \ \textbf{where}\ \ x_1 = e_1, \ldots, x_k = e_k : \\ \qquad (A, C\,[t : (\overline{\gamma_n})\,\gamma]\,) \Rightarrow \\ \qquad \textbf{let}\ \ \gamma_t = \lambda\overline{\alpha_{\gamma_n}}.(\tilde{e}_1, \ldots, \tilde{e}_k, \\ \qquad\qquad (\gamma_t^1 cast_C(\alpha_{\gamma_1}, \gamma_1^1)\ldots cast_C(\alpha_{\gamma_n}, \gamma_n^1)), \\ \qquad\qquad \ldots \\ \qquad\qquad (\gamma_t^s cast_C(\alpha_{\gamma_1}, \gamma_1^s)\ldots cast_C(\alpha_{\gamma_n}, \gamma_n^s))\,)\end{array}}$$

$$\text{(PROG}^+\text{)}\ \frac{\begin{array}{c}(A_{i-1}, C_{i-1}) \vdash^+ d_i : (A_i, C_i) \Rightarrow \tilde{d}_i \qquad i = 1\ldots n \\ (A_n, C_n) \vdash^+ e : \tau \Rightarrow \tilde{e}\end{array}}{(A_0, C_0) \vdash^+ d_1\ldots d_n e : \tau \Rightarrow \tilde{d}_1\ \ \textbf{in}\ \ \ldots\ \ \textbf{in}\ \ \tilde{d}_n\ \ \textbf{in}\ \ \tilde{e}}$$

$$\sigma = \forall\overline{\alpha_{\gamma_n}}.\,\mu\beta.\,K_1\exists\overline{\beta_{\delta_{1k_1}}}.\,\tau_1 + \ldots + K_m\exists\overline{\beta_{\delta_{mk_m}}}.\,\tau_m$$

$$(\text{DATA}^+)\quad \frac{FT(\sigma) \subseteq \text{Dom}(C) \qquad t \notin \text{Dom}(C)}{(A, C) \vdash^+ \textbf{data}\ \ t = \sigma :}$$

$$\left(\begin{array}{c} A\left[\forall\overline{\alpha_{\gamma_n}}.\,(\exists\overline{\beta_{\delta_{ik_i}}}.\,\tau_1\left[t(\overline{\alpha_{\gamma_n}})/\beta\right]) \rightarrow t(\overline{\alpha_{\gamma_n}})/K_i \,\middle|\, i = 1\ldots m\right] \\ C\,[t :\ (\overline{\gamma_n})\,\Omega] \end{array}\right)$$

$$\Rightarrow \textbf{data}\ \ \forall\overline{\alpha_n}.\,\mu\beta.$$

$$K_1\exists\overline{\beta_{1k_1}}.\,\tau_1\left[\alpha_i/\alpha_{\gamma_i},\, \beta_{1j}/\beta_{\delta_{1j}}\right] \times \delta_{11}(\beta_{11}) \times \ldots \times \delta_{1k_1}(\beta_{1k_1})$$

$$+ \ldots +$$

$$K_m\exists\overline{\beta_{mk_m}}.\,\tau_m\left[\alpha_i/\alpha_{\gamma_i},\, \beta_{mj}/\beta_{\delta_{mj}}\right] \times \delta_{m1}(\beta_{m1}) \times \ldots \times \delta_{mk_m}(\beta_{mk_m})$$

The DATA$^+$ rule translates a **data** declaration with order-sorted type variables to a **data** declaration in the target language. The component types of the translated datatype consist of the original component types together with the dictionaries for the existentially quantified type variables. This is reflected in the CONS$^+$ and PAT$^+$ rules below.

5.6.4 Translation Rules for Expressions

The first five translation rules are identical to the ones in [NS91].

$$(\text{VAR}^+)\quad \frac{A(x) = \forall\overline{\alpha_{\gamma_n}}.\,\tau}{(A, C) \vdash^+ x : \tau\left[\tau_1/\alpha_{\gamma_1}, \ldots, \tau_n/\alpha_{\gamma_n}\right] \Rightarrow (x\ \ dict_C(\tau_1, \gamma_1)\ldots dict_C(\tau_n, \gamma_n))}$$

$$(\text{PAIR}^+)\quad \frac{(A, C) \vdash^+ e_1 : \tau_1 \Rightarrow \tilde{e}_1 \qquad (A, C) \vdash^+ e_2 : \tau_2 \Rightarrow \tilde{e}_2}{(A, C) \vdash^+ (e_1, e_2) : \tau_1 \times \tau_2 \Rightarrow (\tilde{e}_1, \tilde{e}_2)}$$

$$(\text{APPL}^+)\quad \frac{(A, C) \vdash^+ e : \tau' \to \tau \Rightarrow \tilde{e} \qquad (A, C) \vdash^+ e' : \tau' \Rightarrow \tilde{e'}}{(A, C) \vdash^+ e\ e' : \tau \Rightarrow \tilde{e}\ \tilde{e'}}$$

$$(\text{ABS}^+)\quad \frac{(A\,[\tau'/x], C) \vdash^+ e : \tau \Rightarrow \tilde{e}}{(A, C) \vdash^+ \lambda x.e : \tau' \to \tau \Rightarrow \lambda x.\tilde{e}}$$

$$(\text{LET}^+)\quad \frac{\begin{array}{c}(A, C) \vdash^+ e : \tau \Rightarrow \tilde{e} \qquad FV(\tau) \setminus FV(A) = \{\alpha_{\gamma_1}, \ldots, \alpha_{\gamma_n}\} \\ (A\left[\forall\overline{\alpha_{\gamma_n}}.\tau/x\right], C) \vdash^+ e' : \tau' \Rightarrow \tilde{e'}\end{array}}{\begin{array}{l}(A, C) \vdash^+ \textbf{let}\ \ x = e\ \ \textbf{in}\ \ e' : \tau' \Rightarrow \\ \qquad \textbf{let}\ \ x = \lambda\overline{\alpha_{\gamma_n}}.\tilde{e}\ \ \textbf{in}\ \ \tilde{e'}\end{array}}$$

In the LET$^+$ rule, the translation $\tilde{e}$ of the expression to be bound to x may contain free dictionary variables corresponding to the generic type variables in τ; λ-bindings for those dictionaries need to be provided.

New translation rules are added for value constructors, **is** expressions, and pattern-matching **let** expressions.

$$(\text{CONS}^+)\quad \frac{A(K) = \forall\overline{\alpha_{\gamma_n}}.\ (\exists\overline{\beta_{\delta_k}}.\tau) \to t(\overline{\alpha_{\gamma_n}})}{\begin{array}{l}(A, C) \vdash^+ K :\ (\tau \to t(\overline{\alpha_{\gamma_n}}))\left[\tau_i/\alpha_{\gamma_i}, \hat{\tau}_j/\beta_{\delta_j}\right] \Rightarrow \\ \qquad (\lambda x.K\,(x, dict_C(\hat{\tau}_1, \delta_1), \ldots, dict_C(\hat{\tau}_k, \delta_k))\,)\end{array}}$$

The translation of a value constructor is again a value constructor; this translated constructor packs dictionaries for the types substituted for the existentially quantified type variables together with the value.

$$(\text{TEST}^+)\quad \frac{A(K) \geq_C \eta \to t(\overline{\tau_n})}{(A, C) \vdash^+ \mathbf{is}\ \ K : t(\overline{\tau_n}) \to \mathit{bool} \Rightarrow \mathbf{is}\ \ K}$$

The **is** K expression is needed to examine the constructor tag and translates to itself.

$$(\text{PAT}^+)\quad \frac{\begin{array}{c} A(K) \geq_C (\exists \overline{\beta_{\delta_k}}.\tau) \to t(\overline{\tau_n}) \\ (A, C) \vdash^+ e : t(\overline{\tau_n}) \Rightarrow \tilde{e} \\ \{\kappa_1, \ldots, \kappa_k\} \cap (FS(\tau') \cup FS(A)) = \varnothing \\ \begin{pmatrix} A\left[\tau\left[\kappa_i / \beta_{\delta_i} \middle| i = 1 \ldots k\right] / x\right] \\ C\left[\kappa_i : \delta_i \middle| i = 1 \ldots k\right] \end{pmatrix} \vdash^+ e' : \tau' \Rightarrow \tilde{e}' \end{array}}{\begin{array}{c} (A, C) \vdash^+ \mathbf{let}\ \ K\ x = e\ \ \mathbf{in}\ \ e' : \tau' \Rightarrow \\ \mathbf{let}\ \ K(x, \delta_{\kappa_1}, \ldots, \delta_{\kappa_k}) = \tilde{e}\ \ \mathbf{in}\ \ \tilde{e}' \end{array}}$$

To translate a pattern-matching **let** expression, we need to look at the way value constructors are translated. We need to provide a binding for the original bound variable x, which corresponds to the first component of the encapsulated value; we further need to retrieve the dictionaries for the Skolem type constructors from the remaining k components of the encapsulated value and bind them to variables $\delta_{\kappa_1}, \ldots, \delta_{\kappa_k}$. Any of these bound variables may occur in $\tilde{e}'$, the translation of the body of the **let** expression.

Since $\tilde{e}$, the translation of the expression to be bound, is used monomorphically, no λ-bindings for potentially free dictionary variables need to be provided.

5.6.5 Properties of the Translation

As in [NS91], the correctness of our translation scheme depends on the condition that each instance declaration lists exactly the same operations $x_1, \ldots, x_n$ as the corresponding class declaration, and in the same order. This scheme prohibits redefinition of operations listed in superclasses.

Furthermore, when translating a program $d_1 \ldots d_n e$, we require the signature resulting from elaborating $d_1 \ldots d_n$ to satisfy the injectivity and subsort reflection conditions stated in Section 5.5.1.

The next lemma says that the translation can only *introduce* free dictionary variables in an expression and is needed in the main theorem.

Lemma 5.6 [Free variables] If $(A, C) \vdash^{+} e : \tau \Rightarrow \tilde{e}$, then $FV(\tilde{e}) \supseteq FV(e)$, and $FV(\tilde{e}) \setminus FV(e)$ contains only dictionary variables.

Proof: By structural induction on e. Free dictionary variables are explicitly generated in the VAR^{+} and CONS^{+} rules. The variables that are bound in the LET^{+} rule are also dictionary variables and, by definition, are not free in the original expression.

Lemma 5.7 [Free variables] If $(A, C) \vdash^{+} e : \tau \Rightarrow \tilde{e}$ and $(A', C') \vdash^{+} e' : \tau \Rightarrow \tilde{e'}$, then

$$(FV(\tilde{e}) \cup FV(\tilde{e'})) \setminus (FV(e) \cup FV(e')) = (FV(\tilde{e}) \setminus FV(e)) \cup (FV(\tilde{e'}) \setminus FV(e')) \ .$$

Proof: Using Lemma 5.6 and the fact that $FV(e)$ and $FV(e')$ do not contain any dictionary variables.

Lemma 5.8 [Types of dictionaries] Let $d_1 \ldots d_n e$ be a program with translation $\tilde{d}_1$ **in** $\ldots$ **in** $\tilde{d}_n$ **in** $\tilde{e}$. Let C be the unitary signature obtained by elaborating $d_1 \ldots d_n$, and A a superset of the assumption set obtained by elaborating $\tilde{d}_1$ **in** $\ldots$ **in** $\tilde{d}_n$. The following type judgments hold for occurrences of *cast* and *dict* in $\tilde{e}$:

$$A\,[\gamma(\alpha)/\alpha_\gamma] \vdash cast_C(\alpha_{\gamma'}, \gamma') : \gamma'(\alpha)$$

$$A\left[\gamma_i(\alpha_i)/\alpha_{\gamma_i} \,\middle|\, i = 1 \ldots n\right] \vdash dict_C(\tau, \gamma) : \gamma(\tau\left[\alpha_i/\alpha_{\gamma_i} \,\middle|\, i = 1 \ldots n\right])$$

Proof: $\tilde{d}_1$ **in** $\ldots$ **in** $\tilde{d}_n$ **in** is a nested **let** expression without body; it results in assumptions for superclass dictionary access functions γ_{i_γ} and instance dictionaries of t for γ and its superclasses. The claim follows from the definitions of *cast* and *dict* and the conditions on C.

The following, main theorem of this section states that a well-typed expression in the original type system translates to an expression that is well typed in the type system of the target language.

Theorem 5.9 [Type preservation] If $(A, C) \vdash^+ e : \tau \Rightarrow \tilde{e}$ and $(FV(\tilde{e}) \setminus FV(e)) \cup (FV(\tau) \setminus FV(A)) = \{\alpha_{\gamma_1}, \ldots, \alpha_{\gamma_n}\}$, then $ML(A)\left[\gamma_1(\alpha_1)/\alpha_{\gamma_1}, \ldots, \gamma_n(\alpha_n)/\alpha_{\gamma_n}\right] \vdash \tilde{e} : \tau\left[\alpha_1/\alpha_{\gamma_1}, \ldots, \alpha_n/\alpha_{\gamma_n}\right]$, where $\{\alpha_1, \ldots, \alpha_n\} \cap FV(ML(A)) = \emptyset$.

Proof: We first observe that the translation rules from Section 5.6.3 exactly implement the type translation scheme from Section 5.6.2 by producing the corresponding **let** expressions without bodies. We then continue by structural induction on the expression e, going through each case in turn:

$(A, C) \vdash^{+} x : \tau\left[\tau_1/\alpha_{\gamma_1}, \ldots, \tau_n/\alpha_{\gamma_n}\right] \Rightarrow (x\ dict_C(\tau_1, \gamma_1)\ldots dict_C(\tau_n, \gamma_n))$

The premise of this judgment is $A(x) = \forall\overline{\alpha_{\gamma_n}}.\tau$, whence

$(ML(A))(x) = \forall\overline{\alpha_n}.\gamma_1(\alpha_1) \to \ldots \to \gamma_n(\alpha_n) \to \tau\left[\alpha_1/\alpha_{\gamma_1}, \ldots, \alpha_n/\alpha_{\gamma_n}\right]$.

Observing that $FV(dict_C(\tau_i, \gamma_i)) = FV(\tau_i)$, let

$\{\alpha'_{\gamma_1}, \ldots, \alpha'_{\gamma_m}\} = FV(\tau_1) \cup \ldots \cup FV(\tau_n) = FV(x\ dict_C(\tau_1, \gamma_1)\ldots)$.

By Lemma 5.8 and extending the assumption set with assumptions for all of the α'_{γ_j} where necessary, we have for $1 \le i \le n$

$$ML(A)\left[\gamma'_i(\alpha'_i)/\alpha'_{\gamma_i} \mid i = 1\ldots m\right] \vdash dict_C(\tau_i, \gamma_i) : \gamma_i(\tau_i\left[\alpha'_1/\alpha'_{\gamma_1} \mid i = 1\ldots m\right])$$

By extending the assumption set again and using the TAUT rule, we also have

$$ML(A)\left[\gamma'_i(\alpha'_i)/\alpha'_{\gamma_i} \mid i = 1\ldots m\right] \vdash x : (\gamma_1(\alpha_{\gamma_1}) \to \ldots \to \gamma_n(\alpha_{\gamma_n}) \to \tau)\left[\tau_i\left[\alpha'_1/\alpha'_{\gamma_1} \mid i = 1\ldots m\right]/\alpha_{\gamma_i}\right]$$

We apply the APPL rule n times and obtain

$$ML(A)\left[\gamma'_i(\alpha'_i)/\alpha'_{\gamma_i} \mid i = 1\ldots m\right] \vdash (x\ dict_C(\tau_1, \gamma_1)\ldots dict_C(\tau_n, \gamma_n)) : (\tau\left[\tau_1/\alpha_{\gamma_1}, \ldots, \tau_n/\alpha_{\gamma_n}\right])\left[\alpha'_1/\alpha'_{\gamma_1} \mid i = 1\ldots m\right]$$

$(A, C) \vdash^{+} e\ e' : \tau \Rightarrow \tilde{e}\ \tilde{e}'$

The premises of this judgment according to the translation rules are $(A, C) \vdash^{+} e : \tau \to \tau' \Rightarrow \tilde{e}$ and $(A, C) \vdash^{+} e' : \tau' \Rightarrow \tilde{e}'$. Let

$(FV(\tilde{e}\ \tilde{e}') \setminus FV(e\ e')) \cup ((FV(\tau) \cup FV(\tau')) \setminus FV(A)) = \{\alpha_{\gamma_1}, \ldots, \alpha_{\gamma_m}\}$.

Using Lemma 5.7, we choosing a suitable numbering such that

$(FV(\tilde{e}) \setminus FV(e)) \cup (FV(\tau' \to \tau) \setminus FV(A)) = \{\alpha_{\gamma_1}, \ldots, \alpha_{\gamma_l}\}$ and

$(FV(\tilde{e'}) \setminus FV(e')) \cup (FV(\tau') \setminus FV(A)) = \{\alpha_{\gamma_{h+1}}, \ldots, \alpha_{\gamma_m}\}$, where $h \le l$.

By the induction assumption, the following two judgments hold:

$$ML(A)\left[\gamma_i(\alpha_i)/\alpha_{\gamma_i} \middle| i = 1 \ldots l\right] \vdash \tilde{e} : (\tau' \to \tau)\left[\alpha_i/\alpha_{\gamma_i} \middle| i = 1 \ldots l\right]$$

$$ML(A)\left[\gamma_i(\alpha_i)/\alpha_{\gamma_i} \middle| i = h+1 \ldots m\right] \vdash \tilde{e'} : \tau'\left[\alpha_i/\alpha_{\gamma_i} \middle| i = h+1 \ldots m\right].$$

Since we can extend the assumption sets and substitutions in both judgments for variables that do not occur free, we obtain:

$$ML(A)\left[\gamma_i(\alpha_i)/\alpha_{\gamma_i} \middle| i = 1 \ldots m\right] \vdash \tilde{e} : (\tau' \to \tau)\left[\alpha_i/\alpha_{\gamma_i} \middle| i = 1 \ldots m\right]$$

$$ML(A)\left[\gamma_i(\alpha_i)/\alpha_{\gamma_i} \middle| i = 1 \ldots m\right] \vdash \tilde{e'} : \tau'\left[\alpha_i/\alpha_{\gamma_i} \middle| i = 1 \ldots m\right].$$

Our claim follows by applying the APPL rule and eliminating the superfluous variables

$\{\alpha_{\gamma_1}, \ldots, \alpha_{\gamma_m}\} \setminus ((FV(\tilde{e}\ \tilde{e'}) \setminus FV(e\ e')) \cup (FV(\tau) \setminus FV(A)))$ from the assumption set.

$(A, C) \vdash^+ (e_1, e_2) : \tau_1 \times \tau_2 \Rightarrow (\tilde{e}_1, \tilde{e}_2)$

$(A, C) \vdash^+ \lambda x.e : \tau' \to \tau \Rightarrow \lambda x.\tilde{e}$

These two cases are handled in a similar way as the previous one.

$(A, C) \vdash^+ \mathbf{let}\ \ x = e\ \ \mathbf{in}\ \ e' : \tau' \Rightarrow \mathbf{let}\ \ x = \lambda\overline{\alpha_{\gamma_n}}.\tilde{e}\ \ \mathbf{in}\ \ \tilde{e'}$

Let

$(FV(\tilde{e}\ \tilde{e'}) \setminus FV(e\ e')) \cup ((FV(\tau) \cup FV(\tau')) \setminus FV(A)) = \{\alpha_{\gamma_1}, \ldots, \alpha_{\gamma_m}\}.$

Using Lemma 5.7, we choosing a suitable numbering such that

$(FV(\tilde{e}) \setminus FV(e)) \cup (FV(\tau) \setminus FV(A)) = \{\alpha_{\gamma_1}, \ldots, \alpha_{\gamma_l}\}$ and

$(FV(\tilde{e'}) \setminus FV(e')) \cup (FV(\tau') \setminus FV(A)) = \{\alpha_{\gamma_{h+1}}, \ldots, \alpha_{\gamma_m}\}$, where

$n \le h \le l \le m$. By the inductive assumption,

$ML(A)\left[\gamma_i(\alpha_i)/\alpha_{\gamma_i} \mid i = 1 \ldots l\right] \vdash \tilde{e} : \tau\left[\alpha_i/\alpha_{\gamma_i} \mid i = 1 \ldots l\right]$, and after n applications of the ABS rule, we obtain

$$ML(A)\left[\gamma_i(\alpha_i)/\alpha_{\gamma_i} \mid i = n+1 \ldots l\right]$$
$$\vdash \lambda\overline{\alpha_{\gamma_n}}.\tilde{e} : \gamma_1(\alpha_1) \to \ldots \to \gamma_n(\alpha_n) \to \tau\left[\alpha_i/\alpha_{\gamma_i} \mid i = n+1 \ldots l\right]$$

We apply the inductive assumption to the last premise, observing that $FV(A) = FV(A\left[\forall\overline{\alpha_{\gamma_n}}.\tau/x\right])$:

$$ML(A\left[\forall\overline{\alpha_{\gamma_n}}.\tau/x\right])\left[\gamma_i(\alpha_i)/\alpha_{\gamma_i} \mid i = h+1 \ldots m\right]$$
$$\vdash \tilde{e'} : \tau'\left[\alpha_i/\alpha_{\gamma_i} \mid i = h+1 \ldots m\right]$$

Finally, we extend the assumption sets of this and the preceding judgment to include $\alpha_{\gamma_{n+1}}, \ldots, \alpha_{\gamma_m}$ and are ready to apply the LET rule.

$(A, C) \vdash^+ K : (\tau \to t(\overline{\alpha_{\gamma_n}}))\left[\tau_i/\alpha_{\gamma_i}, \hat{\tau}_j/\beta_{\delta_j}\right] \Rightarrow$

$(\lambda x.K(x, dict_C(\hat{\tau}_1, \delta_1), \ldots, dict_C(\hat{\tau}_k, \delta_k)))$

Let $ML(A(K)) = \forall\overline{\alpha_n}.\mu\beta.\Sigma[K\eta] = \forall\overline{\alpha_n}.\chi$, where

$\eta = \exists\overline{\beta_k}.\tau\left[\alpha_i/\alpha_{\gamma_i}, \beta_j/\beta_{\delta_j}\right] \times \delta_1(\beta_1) \times \ldots \times \delta_k(\beta_k)$, and let

$FV(\tau_1) \cup \ldots \cup FV(\tau_n) \cup FV(\hat{\tau}_1) \cup \ldots \cup FV(\hat{\tau}_k) = \{\alpha'_{\gamma_1}, \ldots, \alpha'_{\gamma_m}\}$.

Furthermore, let $\tau' = \tau\left[\tau_i/\alpha_{\gamma_i}, \hat{\tau}_j/\beta_{\delta_j}\right]$.

We first apply the CONS rule to derive

$$ML(A)\left[\gamma'_1(\alpha'_1)/\alpha'_{\gamma_1}, \ldots, \gamma'_m(\alpha'_m)/\alpha'_{\gamma_m}, \tau'\left[\alpha'_i/\alpha'_{\gamma_i}\right]/x\right] \vdash K :$$
$$(\tau' \times \delta_1(\hat{\tau}_1) \times \ldots \times \delta_k(\hat{\tau}_k) \to \chi\,[\tau_i/\alpha_i])\left[\alpha'_i/\alpha'_{\gamma_i}\right]$$

and the TUPLE rule together with Lemma 5.7 to derive

$$ML(A)\left[\gamma'_1(\alpha'_1)/\alpha'_{\gamma_1}, \ldots, \gamma'_m(\alpha'_m)/\alpha'_{\gamma_m}, \tau'\left[\alpha'_i/\alpha'_{\gamma_i}\right]/x\right]$$
$$\vdash (x, dict_C(\hat{\tau}_1, \delta_1), \ldots, dict_C(\hat{\tau}_k, \delta_k)) :$$
$$(\tau' \times \delta_1(\hat{\tau}_1) \times \ldots \times \delta_k(\hat{\tau}_k))\left[\alpha'_i/\alpha'_{\gamma_i}\right]$$

for the argument supplied to K. We then use the APPL rule to derive

$$ML(A)\left[\gamma'_1(\alpha'_1)/\alpha'_{\gamma_1}, \ldots, \gamma'_m(\alpha'_m)/\alpha'_{\gamma_m}, \tau'\left[\alpha'_i/\alpha'_{\gamma_i}\right]/x\right]$$
$$\vdash K\ (x, dict_C(\hat{\tau}_1, \delta_1), \ldots, dict_C(\hat{\tau}_k, \delta_k)) : (\chi\,[\tau_j/\alpha_j])\left[\alpha'_i/\alpha'_{\gamma_i}\right]$$

and finally the ABS rule, which gives us

$$ML(A)\left[\gamma'_1(\alpha'_1)/\alpha'_{\gamma_1}, \ldots, \gamma'_m(\alpha'_m)/\alpha'_{\gamma_m}\right]$$
$$\vdash (\lambda x.K\,(x, dict_C(\hat{\tau}_1, \delta_1), \ldots, dict_C(\hat{\tau}_k, \delta_k))) :$$
$$(\tau' \to \chi\,[\tau_j/\alpha_j])\left[\alpha'_i/\alpha'_{\gamma_i}\right]$$

$(A, C) \vdash^+ \mathbf{is}\ \ K : t(\overline{\tau_n}) \to bool \Rightarrow \mathbf{is}\ \ K$

This case is a straightforward application of the TEST rule in the target language.

$(A, C) \vdash^+ \mathbf{let}\ \ K\ x = e\ \ \mathbf{in}\ \ e' : \tau' \Rightarrow \mathbf{let}\ \ K\,(x, \delta_{\kappa_1}, \ldots, \delta_{\kappa_k}) = \tilde{e}\ \ \mathbf{in}\ \ \tilde{e}'$

Let

$$(FV(\tilde{e}\ \tilde{e}') \setminus FV(e\ e')) \cup ((FV(t(\overline{\tau_n})) \cup FV(\tau')) \setminus FV(A)) = \{\alpha_{\gamma_1}, \ldots, \alpha_{\gamma_m}\}$$

choosing a suitable numbering such that

$(FV(\tilde{e}) \setminus FV(e)) \cup (FV(t(\overline{\tau_n})) \setminus FV(A)) = \{\alpha_{\gamma_1}, \ldots, \alpha_{\gamma_l}\}$ and

$(FV(\tilde{e'}) \setminus FV(e')) \cup (FV(\tau') \setminus FV(A)) = \{\alpha_{\gamma_{h+1}}, \ldots, \alpha_{\gamma_m}\}$, where

$h \le l \le m$.

Let $\forall\overline{\alpha'_n}.\mu\beta.\Sigma[K\exists\overline{\beta_k}.\tilde{\tau}] = ML(\sigma)$, where

$\tilde{\tau} = \tau\left[\beta_1/\beta_{\delta_1}, \ldots, \beta_k/\beta_{\delta_k}\right] \times \delta_1(\beta_1) \times \ldots \times \delta_k(\beta_k)$.

By the induction assumption,

$ML(A)\left[\gamma_i(\alpha_i)/\alpha_{\gamma_i} \middle| i = 1 \ldots l\right] \vdash \tilde{e} : \tilde{t}\left[\alpha_i/\alpha_{\gamma_i} \middle| i = 1 \ldots l\right]$, where

$\tilde{t} = (\mu\beta.\Sigma[K\exists\overline{\beta_k}.\tilde{\tau}])\left[\tau_1/\alpha'_{\gamma_1}, \ldots, \tau_n/\alpha'_{\gamma_n}\right]$. We further apply the induction assumption to the last premise to obtain

$$ML(A)\left[\begin{array}{c} \gamma_i(\alpha_i)/\alpha_{\gamma_i} \middle| i = h+1 \ldots m \\ \tau\left[\kappa_j/\beta_{\delta_j}\right]/x,\ \delta_1(\kappa_1)/\delta_{\kappa_1}, \ldots, \delta_k(\kappa_k)/\delta_{\kappa_k} \end{array}\right]$$
$$\vdash \tilde{e'} : \tau'\left[\alpha_i/\alpha_{\gamma_i} \middle| i = h+1 \ldots m\right]$$

Note that $dict_{C'}(\kappa_j, \delta_j) = \delta_{\kappa_j}$ in $\tilde{e'}$. We now extend the assumption sets of this and the preceding judgment to include $\alpha_{\gamma_1}, \ldots, \alpha_{\gamma_m}$, and apply the PAT'' rule. Our claim follows after restricting the final assumption set to $(FV(\tilde{e}\ \tilde{e'}) \setminus FV(e\ e')) \cup (FV(\tau') \setminus FV(A))$.

■

The following corollary is a deterministic version of the type preservation theorem in [NS91]. It covers the case of unambiguous resulting expressions, that is, expressions whose translations do not contain free dictionary variables not free in their types.

Corollary 5.10 [Type preservation] Let all types in the range of A be closed. If $(A, C) \vdash^{+} e : \tau \Rightarrow \tilde{e}$ and $FV(\tilde{e}) \setminus FV(e) \subseteq FV(\tau)$, then $ML(A) \vdash \lambda\overline{\alpha_{\gamma_n}}.\tilde{e} : \gamma_1(\alpha_1) \to \ldots \to \gamma_n(\alpha_n) \to \tau\left[\alpha_1/\alpha_{\gamma_1}, \ldots, \alpha_n/\alpha_{\gamma_n}\right]$, where $\{\alpha_1, \ldots, \alpha_n\} \cap FV(ML(A)) = \emptyset$.

Proof: We use the preceding theorem and apply the ABS rule from Chapter 3 n successive times.

Corollary 5.11 [Semantic soundness] Let all types in the range of A be closed, and let ψ be a type environment such that for every $\alpha \in \mathrm{Dom}\,\psi$, $\mathsf{wrong} \notin \psi(\alpha)$. If $(A, C) \vdash^{+} e : \tau \Rightarrow \tilde{e}$ and $\models_{\rho, \psi} ML(A)$, then $E[\![\tilde{e}]\!]\,\rho \neq \mathsf{wrong}$.

Proof: By type preservation and semantic soundness of the target language.

6 Related Work, Future Work, and Conclusions

6.1 Related Work

The following table compares our work with other programming languages with similar features or objectives. The design criteria used as a basis for our comparison are taken from Section 1.1:

1. Strong and static typing,
2. type reconstruction,
3. higher-order functions,
4. parametric polymorphism,
5. extensible abstract types with multiple implementations, and
6. first-class abstract types.

In the table, ✔ means the feature is supported, ❍ means it is not fully supported, and a blank entry means it is not supported at all.

	Design Criterion					
Language	1.	2.	3.	4.	5.	6.
Our work	✔	✔	✔	✔	✔	✔
ML/Haskell	✔	✔	✔	✔	❍	❍
SOL	✔		✔	✔	✔	✔
Hope+C	❍	✔	✔	❍	❍	✔
XML+	✔		✔	✔	✔	✔
Dynamics	❍	✔	✔	✔	❍	❍
OOL	❍			❍	❍	✔

6.1.1 SOL

SOL is based on the full second-order polymorphic λ-calculus. It is not known whether there is a type reconstruction algorithm for this language.

6.1.2 Hope+C

The only other work known to us that deals with Damas-Milner-style type reconstruction for existential types is [Per90]. However, the typing rules given there are not sufficient to guarantee the absence of runtime type errors, even though the Hope+C compiler seems to impose sufficient restrictions. The following unsafe program, here given in ML syntax, is well-typed according to the typing rules, but rejected by the compiler:

```
datatype T = K of ''a
fun f x = let val K z = x in z end
f(K 1) = f(K true)
```

In addition, an identifier bound in a pattern-matching **`let`** expression is not polymorphic according to the typing rules. This restriction does not apply to our work.

6.1.3 XML+

The possibility of making ML structures first-class by implicitly hiding their type components is discussed in [MMM91] without addressing the issue of type inference. By hiding the type components of a structure, its type is implicitly coerced from a strong sum type to an existential type. Detailed discussions of sum types can be found in [Mac86] [MH88].

6.1.4 Dynamics in ML

An extension of ML with objects that carry dynamic type information is described in [LM91]. A dynamic is a pair consisting of a value and the type of the value. Such an object is constructed from a value by applying the constructor **`dynamic`**. The object can then be dynamically coerced by pattern matching on both the value and the runtime type. Existential types are used to match dynamic values against dynamic patterns with incomplete type information. Dynamics are useful for typing functions such as **`eval`**. However, they do not provide type abstraction, since they give access to the type of an object at runtime. It seems possible to combine their system with ours, extending their existential patterns to existential types. We are currently investigating this point.

6.1.5 Object-Oriented Languages

Most statically typed object-oriented languages identify subclassing with subtyping (C++ [Str86], Modula-3 [CDG$^+$89]) at the expense of severely restricting the expressive power of the language. Due to the *contravariance rule* for function subtyping, not even simple algebraic structures can be described in C++; this is discussed in detail in [CHC90] [HL91].

Recognizing and attempting to overcome this restriction, other languages sacrifice static typing (Eiffel [Mey92], Ada 9X [dod91]) and rely on run time checks to guarantee compatibility of function arguments.

Furthermore, most object-oriented languages do not support type reconstruction; a recent advance in type reconstruction for a Smalltalk-like language is presented in [PS91].

6.2 Current State of Implementation

We have implemented a Standard ML prototype of an interpreter with type reconstruction for our core language, Mini-ML [CDDK86] extended with recursive datatypes over existentially quantified component types. The ML-style examples from this thesis have been developed and tested using our interpreter.

Technically, the interpreter consists of the following components:

- Lexer and parser were built using the tools ML-Lex [AMT89] and ML-Yacc [TA91], respectively.
- The type reconstruction phase is based on [Han87].
- The evaluator directly implements the denotational semantics presented in Section 3.6.2.

We plan further to develop this prototype towards an interpreter of a full language based on our extension of SML.

The latest releases of the Lazy ML [AJ92] and Haskell B. [Aug92] systems feature datatypes with existentially quantified component types. Both systems were developed at the Chalmers University of Technology; they provide full compilers and interpreters capable of dealing with larger programs. The Haskell examples from this thesis have been tested using the Chalmers Haskell B. interpreter.

6.3 Conclusions

The question we had set out to answer in this dissertation was:

Is it feasible to design a high-level programming language that satisfies criteria 1. *through* 6.*?*

We showed that such a design is feasible from a type-theoretic, a language design, and an implementation perspective:

- *Type-theoretic view*: Static typing and semantic soundness of the type systems hold for all three languages presented. Furthermore, we extended the Damas-Milner type reconstruction algorithm used in ML to cope with our languages.
- *Language design view*: Our examples demonstrated that we gain considerable expressiveness and flexibility by adding first-class abstract types to ML and Haskell while retaining the syntactic and semantic "look and feel" of the original languages.
- *Implementation view*: Our prototype implementation shows that our languages can be implemented using standard techniques as the ones described in [Han87] or used in the Standard ML of New Jersey implementation [AM92]. The Chalmers LML and HBC systems demonstrate that it is feasible to implement our extensions in practical compilers and interpreters.

6.4 Future Work

Our work leads off to a number of future research directions, some of which are discussed below.

6.4.1 Combination of Modules and Existential Quantification in ML

We demonstrated in Chapter 5 how Haskell type classes can be used as signatures of abstract data types. The ML module system also provides signatures, which are strong sum types. One could imagine using these signatures to describe interfaces of abstract types. First-class abstract types could then

be achieved by applying an injection that makes the type components of the signature existentially quantified, along the lines of [MMM91].

6.4.2 A Polymorphic Pattern-Matching `let` Expression

An identifier bound using the pattern-matching **`let`** expression from Chapter 5 is monomorphic, whereas an identifier bound by the corresponding expression from Chapter 3 can be used polymorphically. It would be desirable to overcome this restriction by exploring an extended target language, where a function depending on some method dictionaries can be decomposed before being applied to any arguments. While unsound in the general case, we conjecture that this is sound in our case, since the arguments are the same whenever the bound identifier is used with the same type.

6.4.3 Combination of Parameterized Type Classes and Existential Types in Haskell

Type classes in Haskell are not parameterized, thus we cannot model abstract container classes. This shortcoming was discussed in [LO91] and is also present in our extension of Haskell described in Chapter 5; thus the stack example from Section 5.2.2 is not polymorphic. An extension of Haskell with parameterized type classes was recently presented in [CHO92]; it would be desirable to apply the same extension to our language. We conjecture that parameterized type classes are an orthogonal extension and combine well with existential quantification.

Another interesting extension of Haskell is one with a dotless dot notation analogous to the ML extension from Chapter 4; it appears that such a language could be translated into the language described in Chapter 5.

6.4.4 Existential Types and Mutable State

Since the full ML language also provides polymorphic references, an extension of this language with existential types would depend on the coexistence of existential types and polymorphic references. Similar considerations hold for other forms of mutable state such as linear types [Ode91] [Wad90].

6.4.5 Full Implementation

Whereas implementations of Lazy ML and Haskell B. extended with existential types are now available, further implementation work could be envisioned both at the ML level and at the Haskell level.

At the ML level, the language would be strict and include datatypes with existentially quantified component types, polymorphic references, and possibly modules.

At the Haskell level, the language could be strict or non-strict and include existential quantification over parameterized type classes. Alternative implementation strategies for Haskell or similar languages with type classes could be explored; instead of translating to an ML-like language, type classes could be mapped to C++ templates [Ode92]. A possible starting point for further exploration could be an explicitly typed version of Mini-Haskell in the spirit of [MH88].

Bibliography

[Aba92] M. Abadi. Private communication, June 1992.

[AJ92] L. Augustsson and T. Johnsson. Lazy ML user manual. July 1992.

[AM92] A. Appel and D. MacQueen. Standard ML of New Jersey. In *Proc. 19th ACM Symp. on Principles of Programming Languages (POPL)*, January 1992.

[AMT89] A. Appel, J. Mattson, and D. Tarditi. A lexical analyzer generator for Standard Ml. Part of the Standard ML of NJ Distribution, December 1989.

[Aug92] L. Augustsson. Haskell B. user manual. July 1992.

[Bar84] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[BM92] K. Bruce and J. Mitchell. PER models of subtyping, recursive types and higher-order polymorphism. In *Proc. 19th ACM Symp. on Principles of Programming Languages (POPL)*, pages 316–327, January 1992.

[CDDK86] D. Clement, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *Proc. ACM Conf. Lisp and Functional Programming*, pages 13–27, 1986.

[CDG+89] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. *Modula-3 Report (revised)*, Oct. 1989.

[CHC90] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 125–135, Jan. 1990.

[CHO92] K. Chen, P. Hudak, and M. Odersky. Parametric type classes. In *Proc. ACM Conf. Lisp and Functional Programming*, 1992.

[CL90] L. Cardelli and X. Leroy. Abstract types and the dot notation. In *Proc. IFIP Working Conference on Programming Concepts and Methods*, pages 466–491, Sea of Gallilee, Israel, April 1990.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, Dec. 1985.

[Dam85] L. Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, 1985.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th Annual ACM Symp. on Principles of Programming Languages*, pages 207–212, Jan. 1982.

[dod91] Ada 9x mapping rationale, 1991. United States Department of Defense.

[FGJM85] K. Futatsugi, J. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proc. 12th ACM Symp. on Principles of Programming Languages (POPL)*, pages 52–66, January 1985.

[Han87] P. Hancock. Polymorphic type checking. In S. Peyton-Jones, editor, *The Implementation of Functional Programming Languages*, chapter 8. Prentice-Hall, 1987.

[Har90] R. Harper. Introduction to Standard ML. Technical report, Carnegie Mellon University, September 1990.

[HL91] F. Henglein and K. Läufer. Programming with structures, functions, and objects. In *Proc. XVII Latin American Informatics Conference (PANEL '91)*, pages 333–352. USB, 1991.

[HPJW+92] P. Hudak, S. Peyton-Jones, P. Wadler, et al. Report on the programming language Haskell A non-strict, purely functional language Version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[LM91] X. Leroy and M. Mauny. Dynamics in ML. In *Proc. Functional Programming Languages and Computer Architecture*, pages 406–426. ACM, 1991.

[LO91] K. Läufer and M. Odersky. Type classes are signatures of abstract types. In *Proc. Phoenix Seminar and Workshop on Declarative Programming*, November 1991.

[Mac86] D. MacQueen. Using dependent types to express modular structure. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, pages 277–286. ACM, Jan. 1986.

[Mey92] B. Meyer. *Eiffel The Language*. Prentice-Hall, 1992.

[MGS89] J. Meseguer, J. Goguen, and G. Smolka. Order-sorted unification. *J. Symbolic Computation*, (8):383–413, 1989.

[MH88] J. Mitchell and R. Harper. The essence of ML. In *Proc. Symp. on Principles of Programming Languages*. ACM, Jan. 1988.

[Mil78] R. Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17:348–375, 1978.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, Apr. 1982.

[MMM91] J. Mitchell, S. Meldal, and N. Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Proc. ACM Symp. on Principles of Programming Languages*, Jan. 1991.

[MP88] J. Mitchell and G. Plotkin. Abstract types have existential type. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988.

[MPS86] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71, 1986.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[NS91] T. Nipkow and G. Snelting. Type classes and overloading resolution via order-sorted unification. In *Proc. Functional Programming Languages and Computer Architecture*, pages 1–14. ACM, 1991.

[Ode91] M. Odersky. Objects and subtyping in a functional perspective. IBM Research Report RC 16423, 1991.

[Ode91b] M. Odersky. How to make destructive updates less destructive. In *Proc. 18th ACM Symp. on Principles of Programming Languages (POPL)*, January 1991.

[Ode92] M. Odersky. Translating type classes to C++ templates. Private communication, March 1992.

[Per90] N. Perry. *The Implementation of Practical Functional Programming Languages*. PhD thesis, Imperial College, 1990.

[PJ87] S. Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[Plo83] G. Plotkin. Domains. Course notes, 1983. TeX-ed edition.

[PS91] J. Palsberg and M. Schwartzbach. Object-oriented type inference. In *Proc. ACM Conf. Object-Oriented Programming: Systems, Languages, Applications (OOPSLA)*, pages 146–165, 1991.

[Rob65] J. Robinson. A machine-oriented logic based on the resolution principle. *J. Assoc. Comput. Mach.*, 12(1):23–41, 1965.

[Smy77] M. Smyth. Effectively given domains. *Theoretical Computer Science*, 5:257–274, 1977.

[SS71] D. Scott and Ch. Strachey. Toward a mathematical semantics for computer languages. In *Proc. Symp. on Computers and Automata*, pages 19–46, Brooklyn, New York, 1971. MRI Symp. Proc‘s, Vol. XXI.

[SS85] M. Schmidt-Schauss. A many-sorted calculus with polymorphic functions based on resulution and paramodulation. In *Proc. 9th Int. Joint Conf. Artificial Intelligence*, pages 1162–1168, 1985.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[TA91] D. Tarditi and A. Appel. ML-Yacc Version 2.1. Part of the Standard ML of NJ Distribution, March 1991.

[Wad90] P. Wadler. Linear types can change the world. In *IFIP TC2 Working Conference on Programming Concepts and Methods*, April 1990.

[WB89] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. 16th Annual ACM Symp. on Principles of Programming Languages*, pages 60–76. ACM, Jan. 1989.