Semantics and Coherence for Parametric Type Classes

Kung Chen

Research Report YALEU/DCS/RR-1003
May 1994

Semantics and Coherence for Parametric Type Classes

Kung Chen

Department of Computer Science
Yale University
Box 8285 Yale Station
New Haven, CT 06520

June 2, 1994

Abstract

In [COH92], we have described a type system for extending Haskell with parametric type classes. In particular, two type inference systems and a type reconstruction algorithm are presented. In this report we turn to the problem of associating meaning with well-typed expressions. One feature of the Haskell-style overloading is that it is possible at compile-time, based on typing derivations, to translate any program usinging overloaded operators to an equivalent program that does not. Since our main concern is resolving overloaded expressions, such a translation scheme can be seen as a semantic specifiction for the source language. Our proprosed parametric extension maintains this feature, and it requires the same mechanism to realize the translation.

The report consists of two parts. The first part develops a translation semantics for systems with parametric type classes while the second one addresses the problem of ambiguity, a problem inherent in overloading. Ambiguity arises when the compiler does not have sufficient type information to determine the suitable implementation for a particular occurrence of an overloaded operator. The problem manifests in our translation semantics when an expression has many different typing derivations, which in turn yield semantically distinct translations; choosing one would give different results than the other. Nevertheless, like Haskell, there exist simple syntactic conditions to detect such undesirable expressions. More specifically, we prove that if an expression's principal type is unambiguous in a specific sense, it is guaranteed to have a well-defined meaning.

Introduction

In [COH92], we have described a type system for extending Haskell with parametric type classes. A small example lamguage Mini–Haskell$^+$ is used to illustrate the main ideas. In this report we turn to the problem of associating meaning with well-typed Mini–Haskell$^+$ expressions. One feature of the Haskell-style overloading is that it is possible at compile-time, based on typing derivations, to translate a program using overloaded operators to an equivalent program that does not. Since our main concern is overloading resolution, such a translation scheme can be seen as a semantic specifiction for the source language. Our proprosed parametric extension maintains this feature, and it requires the same mechanisms to realize the translation.

The target language of the translation semantics is a version of the polymorphic λ–calculus called CP that includes explicit constructs to handle overloading. Except those constructs, CP is very similar to Mini–Haskell$^+$. One of the main uses of CP is to assign meaning to Mini–Haskell$^+$ expressions by translating them into CP expressions where overloading is made explicit. The translation is based on a mapping between typing derivations in Mini–Haskell$^+$ and CP. More specifically, we show that each typing derivation for a Mini–Haskell$^+$ expression corresponds to a typing derivation for a CP expression with explicit overloading, which thereby serves as a translation for the given expression. In other words, every well-typed Mini–Haskell$^+$ expression has a translation and all the translations obtained in this way are well-typed in CP.

Any approach to overloading must address the possiblity of ambiguity, a problem inherent in overloading. Ambiguity arises when the compiler does not have sufficient type information to determine the suitable implementation for a particular occurrence of an overloaded operator. The problem manifests in our translation semantics when an expression has many different typing derivations, which in turn yield semantically distinct translations; choosing one would give different results than the other.

Due to the existence of such expressions in Haskell as well as our parametric extension, we cannot hope to establish a general *coherence* theorem [BCGS89] (a property referring to translation semantics in which all derivations of a given typing judgement yield the same meaning). Instead, we derive some simple syntactic conditions that are sufficient to exclude those expressions and thus identify a collection of expressions for which the coherence property can be established. Like Haskell, the conditions are based on the principal type computed by our type inferencer for any given expression. Essentially, if an expression's principal type is *unambiguous* in a specific sense, all its translations will be coherent.

This report consists of two parts. The first part describes a translation semantics for Mini–Haskell$^+$ while the second one addresses the problem of ambiguity and presents a conditional coherence result. Certain knowledge about parametric type classes is assumed; readers are referred to our previous report this information. Proofs for results of this report can be found in the author's forthcoming Ph.D. thesis [Che94].

Part I: Translation Semantics

This part begins with a short introduction to the translation scheme and then leads to the development of a formal translation semantics for Mini–Haskell$^+$. For ease of reference, the abstract syntax of Mini–Haskell$^+$ is shown again in Figure 1.

Type variables	α		
Type constructors	κ		
Types	τ	$::=$	$\alpha \mid () \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2 \mid \kappa\,\tau$
Type schemes	σ	$::=$	$\tau \mid \forall\alpha{::}\Gamma\,.\,\sigma$
Class constructors	c		
Type classes	γ	$::=$	$c\,\tau$
Class sets	Γ	$::=$	$\{c_1\,\tau_1, ..., c_n\,\tau_n\}$ $(n \geq 0,\ c_i$ pairwise disjoint$)$
Programs	p	$::=$	`class` $\alpha{::}\gamma$ `where` $x{:}\sigma$ `in` p
		$\mid$	`inst` $C \Rightarrow \tau{::}\gamma$ `where` $x = e$ `in` p
		$\mid$	e
Expressions	e	$::=$	$x \mid e_1\,e_2 \mid \lambda x.e \mid$ `let` $x = e_1$ `in` e_2
Contexts	C	$::=$	$\{\alpha_1{::}\Gamma_1, \ldots, \alpha_n{::}\Gamma_n\}$ $(n \geq 0)$

Figure 1: Abstract Syntax of Mini–Haskell$^+$

1 An Informal Presentation

It is instructive to describe the translation scheme informally before we proceed to the formal semantics. The complex number example given in [Che94], included in Figure 2, will be used to illustrate the main ideas.

Class/Instance Declarations

Following [WB89], type classes and their instances are replaced by so called *(method) dictionaries* which contain all the functions associated with a class. In particular, each instance declaration generates an appropriate definition of a dictionary that contains methods for all the (overloaded) operators associated with a class at a given type. For example, corresponding to the `Int::Eq` and `Int::Ord` instances, we have the dictionaries:

```
DEqInt    =  <primEqInt>
DOrdInt   =  <primLeInt, primLeqInt>
```

Here $\langle e_1, \ldots, e_n \rangle$ builds a dictionary. The dictionary for `Eq` contains only the equality method while the one for `Ord` has two methods for `(<)` and `(<=)`.

If an instance declaration has a context, then it translates into a definition of a *dictionary constructor* whose parameters correspond to the dictionaries required by the context. For example, the declaration:

```
inst  a::Num => Rect a :: Complex a  where ...
```

```
class  c::Complex a  where
   real-part,
   imag-part,
   magnitude,
   angle         :   c -> a

--Two representations of complex numbers

data  Rect a    =  MkRect a a

data  Polar a   =  MkPolar a a

instance  a::Num => Rect a  :: Complex a  where
   real-part (MkRect x y)   =  x
   imag-part (MkRect x y)   =  y
   magnitude (MkRect x y)   =  sqrt (square x  +  square y)
   angle (MkRect x y)       =  atan y x

instance  a::Num => Polar a :: Complex a  where
   real-part (MkPolar r t)  =  r * cos t
   imag-part (MkPolar r t)  =  r * sin t
   magnitude (MkPolar r t)  =  r
   angle     (MkPolar r t)  =  t

-- Arithmetic operations on complex numbers

cAdd z1 z2  =  MkRect (real-part z1  +  real-part z2)
                      (imag-part z1  +  imag-part z2)

cSub z1 z2  =  MkRect (real-part z1  -  real-part z2)
                      (imag-part z1  -  imag-part z2)

cMul z1 z2  =  MkPolar (magnitude z1  *  magnitude z2)
                       (angle z1  +  angle z2)

cDiv z1 z2  =  MkPolar (magnitude z1  /  magnitude z2)
                       (angle z1  -  angle1 z2)
```

Figure 2: Complex-number Arithmetic

generates the definition of the unary dictionary constructor DComRect:

```
            DComRect dNum    =    <...>
```

When given a dictionary for a::Num, this dictionary constructor yields a dictionary for instance (Rect a :: Complex a). Hence (DComRect DNumFloat) generates a dictionary for Complex with rectangular form of floating-point numbers.

For each operation in a class, there is an *extractor* to select the appropriate method from the corresponding dictionary. Hence for the Complex class, we generate the following selectors:

```
            real-part  <e1, e2, e3, e4>   =  e1
            imag-part  <e1, e2, e3, e4>   =  e2
            magnitude  <e1, e2, e3, e4>   =  e1
            angle      <e1, e2, e3, e4>   =  e1
```

Overloaded Expressions

Having introduced how class/instance declarations are handled, we now turn to overloaded expressions. Each reference to an overloaded operator is translated into an extraction from some dictionary variable, which will either be resolved to a concrete dictionary or remain unknown and become a parameter to the whole expression. For example, the cAdd function given in Figure 2:

```
      cAdd z1 z2  =  MkRect (real-part z1  +  real-part z2)
                            (imag-part z1  +  imag-part z2)
```

is translated as follows:

```
      cAdd dNum dCom1 dCom2 z1 z2  =
                     MkRect ( (+ dNum) ((real-part dCom1) z1)
                                       ((real-part dCom2) z2) )
                            ( (+ dNum) ((imag-part dCom1) z1)
                                       ((imag-part dCom2) z2) )
```

Three additional parameters, dNum, dCom1, and dCom2, are generated, corresponding to the required dictionaries. In general, these dictionary parameters witness the class constraints in the type of an overloaded function, as demonstrated by the type of cAdd:

$$\forall \mathtt{a} :: \mathtt{Num}. \forall \mathtt{c1} :: \mathtt{Complex\ a}. \forall \mathtt{c2} :: \mathtt{Complex\ a}.\ \mathtt{c1} \rightarrow \mathtt{c2} \rightarrow \mathtt{Rect\ a}$$

Finally, each call of an overloaded function supplies the appropriate dictionary arguments. Thus the application cAdd (MkRect 1.5 2.5) (MkPolar 4.0 3.5) translates to

```
       cAdd DNumFloat (DComRect DNumFloat) (DComPolar DNumFloat)
            (MkRect 1.5 2.5) (MkPolar 4.0 3.5)
```

Type variables	α			
Type constructors	κ			
Types	σ	$::=$	$\alpha \mid () \mid \sigma_1 \times \sigma_2 \mid \sigma_1 \rightarrow \sigma_2 \mid \kappa\,\sigma \mid \forall\alpha{::}\Gamma\,.\,\sigma$	
Class lists	Γ	$::=$	$\gamma_1, \ldots, \gamma_n$	
Expressions	e	$::=$	x	term variables
		$\mid$	$e_1\, e_2$	applications
		$\mid$	$\lambda x.e$	abstractions
		$\mid$	$e\,\mathtt{d}$	dictionary applications
		$\mid$	$\lambda\mathtt{v}.e$	dictionary abstractions
		$\mid$	$\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$	local definitions
Dictionary constructors	χ			
Dictionaries	$\mathtt{d}$	$::=$	$\mathtt{v}$	dictionary variables
		$\mid$	$\chi\ \mathtt{d}_1 \ldots \mathtt{d}_n$	dictionary construction

Figure 3: Abstract Syntax of CP

2 CP: The Target Language

This section decribes the target language of our translation semantics, a version of polymorphic λ–calculus that include explicit constructs for handling dictionary expressions. We called the system CP, intended as a mnemonic for 'Constrained Polymorphic λ–calculus'. The surface syntax of CP is designed to be very similar to that of Mini–Haskell$^+$, though, semantically, Mini–Haskell$^+$ is a proper sublanguage of CP.

2.1 Syntax of Types and Expressions

Figure 3 presents CP's syntax of types and expressions. Compared to Mini–Haskell$^+$, there are two major differences. First, CP has a more expressive set of types. Following Mini–Haskell$^+$, quantified type variables may be associated with class constraints Γ, but there is no distinction of simple types and type schemes, since quantification of type variables is no longer restricted to be outermost. Thus types such as $(\forall\mathtt{a}{::}\mathtt{Eq.a} \rightarrow \mathtt{a}) \rightarrow \mathtt{Int}$ are valid in CP. In other words, functions may take aruguments of polymorphic and/or overloaded types. Section 6 explores this feature to relate a group of CP expressions whose types are ordered by the instantiation ordering ($\preceq_C$) defined in [COH92].

Second, CP has additional abstraction and application constructs for dictionary expressions. Overloaded Mini–Haskell$^+$ functions will be translated to dictionary abstractions in CP while applications of overloaded Mini–Haskell$^+$ functions will become dictionary applications in CP. With such explicit use of dictionary expressions, the order of class constraints on a type variable in the type of an object (and hence the order of dictionary parameters taken by an overloaded value) can no longer be ignored. Indeed, expressions such as $(e\,\mathtt{d}_1)\mathtt{d}_2$ may become meaningless if written as

$(e\,\mathrm{d}_2)\mathrm{d}_1$. Thus all notions defined in previous report citeptclass:yale using sets must be replaced by those using *lists*, e.g., lists of class, lists of instance assumptions (contexts).

The set of free term and dictionary variables in an expression e are defined in an obvious manner and denoted by $\mathrm{fv}(e)$ and $\mathrm{dv}(e)$ respectively. As usual, we use the notation $[e'/x]e$ to represent substitution of the free occurrences of a variable x in an expression e by another expression e', and assume the standard renaming convention for avoiding variable capturing.

2.2 Dictionary Expressions

A dictionary expression is either a dictionary variable or a construction obtained from an instance declaration similar to that of Mini–Haskell$^+$. After introducing some notational conventions for writing objects that involve dictionary expressions, this section describes CP's program declarations and how dictionaries are synthesized through a straightforward extension of the instance entailment system given in [COH92].

Notations

Since we will mostly deal with multiple dictionary expressions grouped into separate lists, it is useful to introduce some notations for writing lists of dictionary expressions. In particular, fonts are used to distinguish dictionary expressions of different units, as detailed by the following table:

Object	expression	abbreviation
List of dictionary variables	$\mathrm{v}_1, \ldots, \mathrm{v}_n$	v
List of list of dictionary variables	$v_1, \ldots, v_m$	$\mathbf{v}$
List of dictionary expressions	$\mathrm{d}_1, \ldots, \mathrm{d}_n$	d
List of list of dictionary expressions	$d_1, \ldots, d_m$	$\mathbf{d}$
Dictionary abstraction	$\lambda \mathrm{v}_1 . \ldots \lambda \mathrm{v}_m . e$	$\lambda v . e$
Dictionary application	$(\ldots(e\,\mathrm{d}_1)\ldots)\mathrm{d}_n$	$e \cdot d$

We also use u and w to denote dictionary variables. Note that we only deal with lists without repetitive elements since each dictionary corresponds to a class constraint. Concatenation of lists is simply expressed by juxtaposition, e.g., $\mathbf{vw}$. In some situations, to emphasize that we are concatenating two disjoint lists, we use $\oplus$ as the concatenating operator, e.g., $\mathbf{v} \oplus \mathbf{w}$.

The empty list is denoted by ϵ. The *length* of a list is denoted by $|l|$. We use $\downarrow k$ for indexing the kth element in a list object, e.g., $v{\downarrow}k = \mathrm{v}_k$. A list l is a *sublist* of another list l', written $l \sqsubseteq l'$, iff $\forall i \in 1..|l|, l{\downarrow}i = l'{\downarrow}k$ for some $1 \leq k \leq |l'|$. Two list l and l' are isomorphic, written $l \cong l'$, iff $l \sqsubseteq l'$ and $l' \sqsubseteq l$.

In addition, to describe the dictionary construction scheme, we introduce some notations for combiningg instance judgements with dictionary expressions. Given the setting:

Γ_i	$=$	$\gamma_{i,1}, \ldots, \gamma_{i,n_i}$	Lists of type classes
v_i	$=$	$\mathrm{v}_{i,1}, \ldots, \mathrm{v}_{i,n_i}$	Lists of dictionary variables
$\mathbf{v}$	$=$	$v_1, \ldots, v_m$	List of list of dictionary variables
C	$=$	$\alpha_1{::}\Gamma_1, \ldots, \alpha_m{::}\Gamma_m$	Context—List of instance assumptions

we define *dictionary-augmented contexts* as follows:

$$\mathbf{v}:C \overset{def}{=} v_1:(\alpha_1::\Gamma_1),\ldots,v_m:(\alpha_m::\Gamma_m)$$

In other words, we pair off lists of dictionary variables v_i and instance assumptions $\alpha_i::\Gamma_i$. The notation

$$v:(\alpha::\Gamma)$$

expresses that, for all i, $1 \leq i \leq |\Gamma|$, dictionary variable $v{\downarrow}i$ is associated with the instance assumption $\alpha::\Gamma{\downarrow}i$—namely, one dictionary for one class constraint. And we often overload the operations involving simple contexts to operate on augmented ones, e.g., membership test $v:(\alpha::\Gamma) \in \mathbf{v}:C$ and context restriction $(\mathbf{v}:C)\backslash_{v:(\alpha::\Gamma)}$. When there is no particular need to explicitly mention the dictionary variables associated with an augmented context we will often write just C in place of $\mathbf{v}:C$.

Similarly, we define dictionary-augmented instance predicates as follows:

$$\mathbf{d}:P \overset{def}{=} d_1:(\tau_1::\Gamma_1),\ \ldots\ ,d_m:(\tau_m::\Gamma_m)$$

where

d_i	$=$	$\mathtt{d}_{i,1},\ldots,\mathtt{d}_{i,n_i}$	Lists of dictionaries
$\mathbf{d}$	$=$	$d_1,\ldots,d_m$	List of list of dictionaries
P	$=$	$\tau_1::\Gamma_1,\ \ldots\ ,\tau_m::\Gamma_m$	List of instance predicates

Program Declarations

Like Mini–Haskell$^+$, a CP program consists of a sequence of declarations followed by an expression. As illustrated in the informal presentation, these declarations introduce overloaded operators (as extractors) and their dictionary implementations to be used in the main expression. To simplify the presentation of the translation, however, we make the syantx of CP declarations very similar to that of Mini–Haskell$^+$. In particular, the declarations are "sugared" as class and instance declarations that declare overloaded operators and define their implementations respectively, as detailed below:

$$\begin{array}{lrl} \text{Programs}\ p & ::= & \mathtt{class}\ \alpha::\gamma\ \mathtt{where}\ x_1:\sigma_1,\ldots,x_n:\sigma_n\ \mathtt{in}\ p \\ & | & \chi\mathtt{:\ inst}\ \mathbf{v}:C \Rightarrow \tau::\gamma\ \mathtt{where}\ x_1 = e_1,\ldots,x_n = e_n\ \mathtt{in}\ p \\ & | & e \end{array}$$

Indeed, the syntax is closely modeled on that of Mini–Haskell$^+$. What is new here is the additional dictionary "annotations": Each instance declaration is assigned a dictionary constructor χ, and the context C is paired with matching dictionary variables $\mathbf{v}$, which may occur in the method expressions e_i.

The informal meaning of these declarations is as follows: a class declaration introduces an extractor definition and an instance declaration defines a dictionary using the specified dictionary constructor. In the simplified case where a class introduces only one operator, every overloaded operator is simply the identity function and a dictionary is just a function. Essentially, these declarations are merely syntatic sugars for global definitions. For example, an instance declaration stands for a binding of a dictionary:

$$\mathtt{let}\ \chi\,\mathbf{v}\ =\ \langle e_1,\ldots,e_n\rangle\ \mathtt{in}\ p$$

$$\Gamma)\quad \frac{\forall i \in 1..|\Gamma|\ (\mathbf{v}\!:\!C \Vdash d{\downarrow}i\!:\!(\tau::\Gamma{\downarrow}i))}{\mathbf{v}\!:\!C \Vdash d\!:\!(\tau::\Gamma)}$$

$$\alpha)\quad \frac{v\!:\!(\alpha::\Gamma) \in \mathbf{v}\!:\!C}{\mathbf{v}\!:\!C \Vdash v{\downarrow}i : (\alpha::\Gamma{\downarrow}i)} \quad (i = 1,\ldots,|\Gamma|)$$

$$\tau)\quad \frac{\mathbf{v}\!:\!C \Vdash \mathbf{d} : \langle\tau_i::\Gamma_i\rangle}{\mathbf{v}\!:\!C \Vdash \chi\,\mathbf{d} : (\tau::\gamma)} \quad (\,\chi : \ulcorner \texttt{inst } \mathbf{v}'\!:\!\langle\tau_i::\Gamma_i\rangle \Rightarrow \tau::\gamma\,\urcorner \in \Sigma\,)$$

Figure 4: Augmented Instance Entailment System

However, by making their surface syntax similar to that of Mini-Haskell$^+$, we get a simpler description of the translation, as will be demonstrated in the subsequent sections.

Instance Entailment and Dictionary Construction

An important step of our translation scheme is dictionary construction whereby overloading is made explicit. In CP, as an advantage of the syntax similarity discussed above, the set of instance declarations Σ in a program forms the basis for both instance constraint inference and dictionary construction. This is easily achieved by extending the inference rules for Mini-Haskell$^+$ instance entailment given in [COH92] such that the satisfication of an instance predicate is witnessed by a list of matching dictionaries. More specifically, given Σ, the extended inference rules presented in Figure 4 allow us to deduce augmented instance judgements of the form

$$\mathbf{v}\!:\!C \Vdash d : (\tau::\Gamma),$$

which assert that from the augmented context $\mathbf{v} : C$ it follows that type τ is an instance of class list Γ as witnessed by dictionary list d. If $\mathbf{d}$ is a list of dictionaries of proper length for a list of instance predicates $\langle\tau_i::\Gamma_i\rangle_{i=1,n}$, then we use the notation

$$\mathbf{v}\!:\!C \Vdash \mathbf{d} : \langle\tau_i::\Gamma_i\rangle$$

to represent

$$\forall i \in 1..n\ (\,\mathbf{v}\!:\!C \Vdash \mathbf{d}{\downarrow}i\!:\!(\tau_i::\Gamma_i)\,)$$

This notation is used in the rule (τ) above to synthesize parameterized dictionaries.

The augmented instance entailment system forms the complete engine of overloading resolution: Any references to an overloaded operator in an expression is checked by the system and, in addition, if validated, a proper implementation is supplied by the system in forms of dictionaries.

The following properties of the augmented system are easily established.

- If $\mathbf{v}\!:\!C \Vdash d\!:\!(\tau::\Gamma)$, then $\mathrm{dv}(d) \subseteq \mathrm{dv}(\mathbf{v})$.

- If $\mathbf{v}:C \Vdash d:(\tau::\Gamma)$, then $\mathbf{v}:C \oplus \mathbf{v}':C' \Vdash d:(\tau::\Gamma)$.
- If $\mathbf{v}_1:C_1 \oplus v:(\alpha::\Gamma') \oplus \mathbf{v}_2:C_2 \Vdash d:(\tau::\Gamma)$ and $\alpha \notin \mathrm{tv}(\tau) \cup \mathrm{tv}(\Gamma)$, then $\mathbf{v}_1:C_1 \oplus \mathbf{v}_2:C_2 \Vdash d:(\tau::\Gamma)$.
- If $\mathbf{v}_1:C_1 \oplus \mathbf{v}_2:C_2 \oplus \mathbf{v}_3:C_3 \Vdash d:(\tau::\Gamma)$, then $\mathbf{v}_2:C_2 \oplus \mathbf{v}_1:C_1 \oplus \mathbf{v}_3:C_3 \Vdash d:(\tau::\Gamma)$.

The lemma of transitivity under substitution presented in [COH92] extends naturally to the augmented system.

Lemma 2.1 For any substitution S, augmented contexts $\mathbf{v}:C$ and $\mathbf{v}':C'$, if $\mathbf{v}:C \Vdash d:(\tau::\Gamma)$ and $\mathbf{v}':C' \Vdash \mathbf{d}':SC$, then $\mathbf{v}':C' \Vdash [\mathbf{d}'/\mathbf{v}]d : S(\tau::\Gamma)$.

Uniqueness of Dictionary Construction

Since a dictionary constructed from the instance declarations implements certain overloaded operators, we need to impose, as before, certain restrictions on instance declarations to avoid ambiguity. In fact, the CP programs we will consider are the translations of valid Mini–Haskell$^+$ programs, and hence maintain all the restrictions stated in [COH92] for Mini–Haskell$^+$. In particular, for every pair of type and class constructor (κ, c), there is at most one instance declaration of the form χ : `inst` $C \Rightarrow \kappa\,\tau'::c\,\tau$ `where` .... Furthermore, we require that the dictionary constructor associated with an instance declaration be unique within a program. It is then easy to show that there is at most one dictionary that makes a type an instance of a particular class within a program:

Lemma 2.2 When the set of instance declarations Σ in a program satisfies the two restrictions discussed above, then the augmented instance entailment system admits unique construction of dictionaries. In other words, if $\mathbf{v}:C \Vdash d:(\tau::\Gamma)$ and $\mathbf{v}:C \Vdash d':(\tau::\Gamma)$, then $d \equiv d'$.

2.3 Typing Rules for CP

The typing rules for CP expressions and declarations are given in Figure 5 and Figure 6 respectively. Similar to the case of instance entailment, these rules are derived from those of Mini–Haskell$^+$ ([COH92]) by extending them with support for dictionary expressions. Indeed, if we remove all the dictionary augmentations, we obtain the same sets of typing rules. Note that, as mentioned earlier, we omit the list of dictionary variables from the augmented context when they are not explicitly referenced.

3 Translating Mini–Haskell$^+$ to CP

This section presents the formal definition of the translation scheme. We show that every well-typed Mini–Haskell$^+$ program has a CP translation and all translations obtained in this way are well-typed in CP.

The translation scheme is based on the similarities between the typing derivations in Mini–Haskell$^+$ and CP. As mentioned in the previous section, every Mini–Haskell$^+$ type can be treated as a CP type. Moreover, the typing rules of Mini–Haskell$^+$ are just a restricted version of the rules for

$$(var)\quad \frac{A(x) = \sigma}{A,\ C\ \vdash\ x : \sigma}$$

$$(\forall - E)\quad \frac{A,\ C\ \vdash\ e : \forall\alpha{::}\Gamma.\sigma \qquad C \Vdash d{:}(\tau{::}\Gamma)}{A,\ C\ \vdash\ e{\cdot}d : [\tau/\alpha]\,\sigma}$$

$$(\forall - I)\quad \frac{A,\ C_1 \oplus v{:}(\alpha{::}\Gamma) \oplus C_2\ \vdash\ e : \sigma}{A,\ C_1 C_2\ \vdash\ \lambda v.e : \forall\alpha{::}\Gamma.\sigma} \qquad \alpha \notin \mathrm{fv}(A) \cup \mathrm{reg}(C_1 C_2)$$

$$(\lambda - E)\quad \frac{A,\ C\ \vdash\ e_1 : \sigma' \rightarrow \sigma \qquad A,\ C\ \vdash\ e_2 : \sigma'}{A,\ C\ \vdash\ e_1\ e_2 : \sigma}$$

$$(\lambda - I)\quad \frac{A.x{:}\sigma',\ C\ \vdash\ e : \sigma}{A,\ C\ \vdash\ \lambda x.e : \sigma' \rightarrow \sigma}$$

$$(let)\quad \frac{A,\ C\ \vdash\ e_1 : \sigma' \qquad A.x{:}\sigma',\ C\ \vdash\ e_2 : \sigma}{A,\ C\ \vdash\ (\texttt{let}\ x = e_1\ \texttt{in}\ e_2) : \sigma}$$

Figure 5: Typing Rules for CP Expressions

CP, except that typing derivations in the latter involve augmented contexts rather than simple contexts and require explicit dictionary application and abstraction in the rules ($\forall - elim$) and ($\forall - intro$) respectively. Based on these observations, we can easily establish a correspondence between Mini–Haskell$^+$ and CP typing derivations using two auxiliary functions: The first function *Cxt* maps an augmented context to the corresponding simple context:

$$\begin{array}{lcl} Cxt(\mathrm{v}_1 : C \oplus \mathrm{v}' : C') & = & Cxt(\mathrm{v} : C) \cup Cxt(\mathrm{v}' : C') \\ Cxt(v{:}(\alpha{::}\Gamma)) & = & \{\alpha{::}\Gamma\} \\ Cxt\ \emptyset & = & \emptyset \end{array}$$

The second function *Erase* maps CP expressions with explicit overloading to Mini–Haskell$^+$ expressions by eliminating all occurrences of dictionary expressions:

$$\begin{array}{lcl} Erase(x) & = & x \\ Erase(e_1\ e_2) & = & Erase(e_1)\ Erase(e_2) \\ Erase(\lambda x.e) & = & \lambda x.Erase(e) \\ Erase(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) & = & \texttt{let}\ x = Erase(e_1)\ \texttt{in}\ Erase(e_2) \\ Erase(e\ \mathrm{d}) & = & Erase(e) \\ Erase(\lambda \mathrm{v}.e) & = & Erase(e) \end{array}$$

The correspondence can then be formally described by the following theorem:

$$(\mathit{class})\qquad \frac{A.x:\forall_{\mathrm{tv}(\gamma)}\forall\alpha::\{\gamma\}.\sigma,\ C\ \vdash\ p:\sigma'}{A,\ C\ \vdash\ (\texttt{class}\ \alpha::\gamma\ \texttt{where}\ x:\sigma\ \texttt{in}\ p):\sigma'}$$

$$(\mathit{inst})\qquad \frac{A,\ C\ \vdash\ x:\forall\alpha::\{\gamma\}.\sigma \quad A,\ \mathrm{v}:C\oplus\mathrm{v}':C'\ \vdash\ e:[\tau/\alpha]\sigma \quad A,\ C\ \vdash\ p:\sigma'}{A,\ C\ \vdash\ (\chi:\ \texttt{inst}\ \mathrm{v}':C'\Rightarrow\tau::\gamma\ \texttt{where}\ x=e\ \texttt{in}\ p):\sigma'}$$

Figure 6: Typing Rules for CP Declarations

Theorem 3.1 If $A,\ C\ \vdash\ e:\sigma$ in Mini–Haskell$^+$, then there is a CP expression e' and an augmented context $\mathrm{v}':C'$ such that $C = \mathit{Cxt}(\mathrm{v}':C')$, $e = \mathit{Erase}(e')$ and $A,\ \mathrm{v}':C'\ \vdash\ e':\sigma$ using a derivation of the same structure.

In other words, every well-typed Mini–Haskell$^+$ expression has a well-typed CP translation that can be obtained from its typing derivation.

The proof is straightforward, using induction on the length of $A,\ C\ \vdash\ e:\sigma$. We thereby define the expression e' in the statement of the theorem to be a *translation* of e and use the notation $A, C\ \vdash\ e \rightsquigarrow e':\sigma$ to refer to a translation of an expression in a specific setting. Note that, in general, a Mini–Haskell$^+$ expression will have many distinct translations in any given setting, each corresponding to a different derivation of $A, C\ \vdash\ e:\sigma$ in Mini–Haskell$^+$. The issue of well-definedness will be addressed in Part II.

This theorem also suggests a more succinct way to describe the translations of Mini–Haskell$^+$ expressions—i.e., by combining the typing rules of Mini–Haskell$^+$ and CP expressions, as illustrated in Figure 7. It is easy to show that $A, C\ \vdash\ e \rightsquigarrow e':\sigma$ according to the original definition of translations above if, and only if, the same judgement can be derived from these rules.

Since the method defined in an instance declaration is also an expression, we can easily generalize the correspondence result to program declarations. Specifically, we extend the definition of *Erase* to declarations as follows:

$$\begin{aligned}
&\mathit{Erase}(\texttt{class}\ \alpha::\gamma\ \texttt{where}\ x:\sigma\ \texttt{in}\ p) = \texttt{class}\ \alpha::\gamma\ \texttt{where}\ x:\sigma\ \texttt{in}\ \mathit{Erase}(p)\\
&\mathit{Erase}(\chi:\ \texttt{inst}\ \mathrm{v}':C'\Rightarrow\tau'::\gamma\ \texttt{where}\ x=e\ \texttt{in}\ p) =\\
&\qquad\qquad\qquad \texttt{inst}\ \mathit{Cxt}(\mathrm{v}':C')\Rightarrow\tau'::\gamma\ \texttt{where}\ x=\mathit{Erase}(e)\ \texttt{in}\ \mathit{Erase}(p)
\end{aligned}$$

Then the previous theorem naturally extends to the following result.

Theorem 3.2 If $A,\ C\ \vdash\ p:\sigma$ in Mini–Haskell$^+$, then there is a CP program p' and an augmented context $\mathrm{v}':C'$ such that $C = \mathit{Cxt}(\mathrm{v}':C')$, $p = \mathit{Erase}(p')$ and $A,\ \mathrm{v}':C'\ \vdash\ p':\sigma$ using a derivation of the same structure.

Such a correspondence effectively specifies a translation sematics for Mini–Haskell$^+$. In the next part, we will extend the syntax-directed system and the type reconstruction algorithm given in

$$(var) \quad \frac{A(x) = \sigma}{A,\ C \vdash x \leadsto x\ : \sigma}$$

$$(\forall - E) \quad \frac{A,\ C \vdash e \leadsto e'\ : \forall\alpha{::}\Gamma.\sigma \qquad C \Vdash d{:}(\tau{::}\Gamma)}{A,\ C \vdash e \leadsto e'{\cdot}d\ : [\tau/\alpha]\,\sigma}$$

$$(\forall - I) \quad \frac{A,\ C_1 \oplus v{:}(\alpha{::}\Gamma) \oplus C_2 \vdash e \leadsto e'\ : \sigma}{A,\ C_1C_2 \vdash e \leadsto \lambda v.e'\ : \forall\alpha{::}\Gamma.\sigma} \quad \alpha \notin \mathrm{fv}(A) \cup \mathit{reg}(C_1C_2)$$

$$(\lambda - E) \quad \frac{A,\ C \vdash e_1 \leadsto e'_1\ : \tau' \to \tau \qquad A,\ C \vdash e_2 \leadsto e'_2\ : \tau'}{A,\ C \vdash e_1\ e_2 \leadsto e'_1\ e'_2\ : \tau}$$

$$(\lambda - I) \quad \frac{A.x{:}\tau',\ C \vdash e \leadsto e'\ : \tau}{A,\ C \vdash \lambda x.e \leadsto \lambda x.e'\ : \tau' \to \tau}$$

$$(let) \quad \frac{A,\ C \vdash e_1 \leadsto e'_1\ : \sigma \qquad A.x{:}\sigma,\ C \vdash e_2 \leadsto e'_2\ : \tau}{A,\ C \vdash (\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2) \leadsto (\mathtt{let}\ x = e'_1\ \mathtt{in}\ e'_2)\ : \tau}$$

Figure 7: Typing & Translation Rules for Mini–Haskell$^+$ Expressions

earlier chapters to include the calculation of translation, and address the problem of multiple translations for a single expression.

Part II. Ambiguity and Coherence

In this part we address the problem of overloaded-operator ambiguity. To begin with, it is necessary to specify when two translations are equivalent. After illustrating the incoherence problem, we develop a typed equational theory for CP expressions whereby we can formally establish the equivalence between translations. Then, motivated by [Jon92], we define *conversions* to relate different translations of an overloaded expresssion based on their types. In particular, such conversions are CP expressions derived from a type scheme and its generic instances in a way that, when applied to a translation of an expression, they repackage the dictionaries involved to yield another translation whose type is less general.

Based on the notion of conversions, we generalize the definition of principal types to that of *principal translations.* We extend our type reconstruction algorithm to include the calculation of translations and show that, analogous to the principal type property, the extended type inferencer computes the most general translation for any well-typed expressions. In other words, any translation of an expression can be obtained by applying a conversion to its principal translation. Consequently, the equivalence of two translations at a given type is determined by the equivalence of the conversions from which they are derived. We show that when an expression's principal type is unambiguous, the conversions that can be derived from the type and any of its generic instances are all equivalent, thereby establishing the conditional coherence result.

4 The Coherence Problem

This section motivates the problem of overloaded-operator ambiguity and identifies it with the incoherence of our translation semantics.

Usually compilers rely on type information to resolve overloaded operators. For polymorphic language such as Haskell overloading resolution is complicated by the presence of type variables. As described in the preceding part, constarints on type variables and explicit dictionary abstractions are used to handle unresolved overloaded operators. When demanded, these operators are properly resolved according to the types at which they are used. As long as the type that instantiates a type variable satisfies the associated constraints, unique resolution is guaranteed by the restrictions imposed on instance delcrations (Lemma 2.2).

There are, however, situations under which the type inferencer cannot determine the suitable type to instantiate a constrained type variable and is therefore unable to supply the proper dictionary to resolve a particular occurrence of an overloaded operator. Arbitrary instantations of such type variables may lead to inconsistent resolutions and thus the ambiguity problem.

As an example, consider the following class `Parsable` that declares two overloaded operations `parse` and `unparse`, which convert strings to/from values of a certain type:

```
class  a::Parsable  where
   parse   : String -> a
   unparse : a -> String
```

Now supposing that just `Int` and `Float` are instances of `Parsable`, then the expression `unparse` (`parse` "123") is ambiguous: The composition of `parse` and `unparse` creates an intermediate value whose type, a type variable, is constrained by class `Parsable`, but does not appear in the type of whole expression, `String`. As a result, the type inferencer is not able to determine the intermediate type via unification; instantiating it to `Int` or `Float` will give different results: "123" and "123.0" respectively.

Our translation semantics exploits an expression's typing derivations to resolve overloaded operators; it maps Mini–Haskell$^+$ typing derivations into CP typing derivations whereby overloading is made explicit. As such, ambiguity occurs when there are many different derivations for a single typing judgement, which in turn yield many semantically distinct translations. Indeed, the ambiguity in the preceding example can be described in this manner: There are two ways to derive the typing judgement $\vdash$ `unparse` (`parse` "123") : `Bool`, one using integer parse/unparse functions and one floating-point parse/unparse functions, and consequently two translations which are clearly not equivalent:

`intUnparse` (`intParse` "123") and `floatUnparse` (`floatParse` "123")

In general, ambiguity arises in our translation semantics when it is possible to get, from derivations $A,\ C\ \vdash\ e \leadsto e_1 : \sigma$ and $A,\ C\ \vdash\ e \leadsto e_2 : \sigma$, translations e_1 and e_2 of a Mini–Haskell$^+$ expression e that are not equivalent.

The existence of ambiguous expressions indicates that our translation semantics is not coherent, i.e., the meaning associated with a typed expression depends on the way that its type is derived. This also means that the mapping from expressions to translations is not well-defined. Indeed, ambiguous expressions do not have well-defined semantics and thus must be eliminated. Proposal to make the translation semantics coherent by strictly restricting the type class mechanism has been made [Wad90], but further study is needed to assess this proposal. Instead, follwing Haskell, we choose to develop conditions that are sufficient to exclude those expressions and thus ensure that the semantics of an expression is well-defined.

5 Equality of Translations

As a first step towards the conditions sufficient to guarantee coherence, we need to specify formally what it means for two translations to be equivalent. This section defines a *typed equational theory* for CP expessions; two translations of an overloaded expression are said to be equivalent if they are provably equal within the theory.

The theory comprises equational judgements of the form:

$$A,\, C \;\vdash\; e = e' : \sigma$$

where we assume that e and e' have type σ in the setting determined by type assumptions A and instance assumptions C. Intuitively, the judgement $A,\, C \;\vdash\; e = e' : \sigma$ asserts that expressions e and e' denote the same element of type σ in environments that satisfy A and C. The implicit side-condition that both $A,\, C \;\vdash\; e : \sigma$ and $A,\, C \;\vdash\; e' : \sigma$ is necessary since only typed expressions and their translations are considered meaningful.

The axioms and inference rules of the theory are given in Figure 8 and Figure 9. The axioms in Figure 8 contain the familar definitions of α-conversion and β-conversion and their extensions to dictionary-augumented expressions. Also included is a rule of η-conversion for removing unnecessary dictionary abstractions.

These axioms, except α-conversion, are often formulated as *reduction rules* by orienting them from left to right. As such they have simpler side-condition of well-typedness since it can be shown that if the expression on the left has type σ then the reduced expression on the right also has type σ. This is a consequence of the *subject reduction property*—reduction preserves typing—which can be proved using standard techniques as in [HS86].

The second group of axioms and inference rules in Figure 9 make the typed equality an equivalence relation and a congruence with respect to the expression formation operation. To make the equality an equivalence relation, we have included the symmetry rule (*sym*) and the transativity rule (*trans*). On the other hand, there is no need to include the reflexivity rule since it is a direct consequence of the other rules, which are closely modeled on the original typing rules of Figure 5 to make the equality a congruence by allowing the equivalence of sub-expressions within a given expression.

One may observe from rule ($app-d$) that the congruence property does not include dictionaries. This is a consequence of Lemma Lemma 2.2. As discussed in Section 2.2, dictionary construction is unique within a program when the set of instance declarations in a program satisfies the restrictions mentioned therein. Hence there is no induced equivalence relation over dictionaries to be included in rule ($app-d$).

The following lemma states some useful properties of `let` –expressions that follow directly from rule ($\beta-let$).

Lemma 5.1 For any CP expressions e_1, e_2 and e' and distinct term variable x and y such that

$$(\alpha) \qquad \frac{y \notin \mathrm{fv}(\lambda x.e)}{A,\ C\ \vdash\ (\lambda x.e)\ =\ (\lambda y.[y/x]e)\ :\sigma}$$

$$(\alpha_d) \qquad \frac{\mathtt{w} \notin \mathrm{fv}(\lambda \mathtt{v}.e)}{A,\ C\ \vdash\ (\lambda \mathtt{v}.e)\ =\ (\lambda \mathtt{w}.[\mathtt{w}/\mathtt{v}]e)\ :\sigma}$$

$$(\alpha-let) \qquad \frac{y\ \notin\ \mathrm{fv}(e) \cup \mathrm{fv}(e')}{A,\ C\ \vdash\ (\mathtt{let}\ x = e'\ \mathtt{in}\ e)\ =\ (\mathtt{let}\ y = e'\ \mathtt{in}\ [y/x]e)\ :\sigma}$$

$$(\beta) \qquad A,\ C\ \vdash\ (\lambda x.e)e'\ =\ [e'/x]e\ :\sigma$$

$$(\beta_d) \qquad A,\ C\ \vdash\ (\lambda v.e)d\ =\ [d/v]e\ :\sigma$$

$$(\beta-let) \qquad A,\ C\ \vdash\ (\mathtt{let}\ x = e'\ \mathtt{in}\ e)\ =\ [e'/x]e\ :\sigma$$

$$(\eta_d) \qquad \frac{v\ \notin\ \mathrm{dv}(e)}{A,\ C\ \vdash\ (\lambda v.e \cdot v)\ =\ e\ :\sigma}$$

Figure 8: Equation rules for CP expressions, I

$y \notin \mathrm{fv}(e_1)$:

1. $A,\ C\ \vdash\ (\mathtt{let}\ x = e_1\ \mathtt{in}\ [e'/x]e_2)\ =\ (\mathtt{let}\ x = [e_1/x]e'\ \mathtt{in}\ e_2)\ :\sigma$
2. $A,\ C\ \vdash\ \lambda y.(\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2)\ =\ (\mathtt{let}\ x = e_1\ \mathtt{in}\ \lambda y.\ e_2)\ :\sigma$
3. $A,\ C\ \vdash\ e'(\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2)\ =\ (\mathtt{let}\ x = e_1\ \mathtt{in}\ e'\ e_2)\ :\sigma$
4. $A,\ C\ \vdash\ (\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2)\ e'\ =\ (\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2\ e')\ :\sigma$

To give a flavor of typed equational reasoning in our system, we include here the proof of the first property of this lemma. In particular, we lay out the equational deduction as follows:

$$\begin{array}{llll} A,\ C\ \vdash\ (\mathtt{let}\ x = e_1\ \mathtt{in}\ [e'/x]e_2) & = & [e_1/x]([e'/x]e_2) & (\beta-let) \\ & = & [[e_1/x]e'/x]e_2 & (substitution) \\ & = & (\mathtt{let}\ x = [e_1/x]e'\ \mathtt{in}\ e_2)\ :\sigma & (\beta-let) \end{array}$$

Note that we have also used rules (*sym*) and (*trans*) in the deduction. Furthermore, the required side-condition on types is preserved by the intermediate steps: From the given hypothesis $A,\ C\ \vdash\ (\mathtt{let}\ x\ =\ e_1\ \mathtt{in}\ [e'/x]e_2)\ :\ \sigma$ and the subject reduction property it follows that $A,\ C\ \vdash\ [e_1/x]([e'/x]e_2) : \sigma$. Hence the first application of $(\beta-let)$ is justified. Similarly, another hypothesis $A,\ C\ \vdash\ (\mathtt{let}\ x = [e_1/x]e'\ \mathtt{in}\ e_2) : \sigma$ justifies the second application of $(\beta-let)$.

$$(sym) \qquad \frac{A,\ C \vdash e = e'}{A,\ C \vdash e' = e\ : \sigma}$$

$$(trans) \qquad \frac{A,\ C \vdash e = e'\ : \sigma \qquad A,\ C \vdash e' = e''\ : \sigma}{A,\ C \vdash e = e''\ : \sigma}$$

$$(var) \qquad \frac{A(x)\ =\ \sigma}{A,\ C \vdash x\ =\ x\ : \sigma}$$

$$(app-d) \qquad \frac{A,\ C \vdash e\ =\ e'\ : \forall\alpha{::}\Gamma.\sigma \qquad C \Vdash d{:}(\tau{::}\Gamma)}{A,\ C \vdash e{\cdot}d\ =\ e'{\cdot}d\ : [\tau/\alpha]\,\sigma}$$

$$(abs-d) \qquad \frac{A,\ C_1 \oplus v{:}(\alpha{::}\Gamma) \oplus C_2\ \vdash\ e\ =\ e'\ : \sigma}{A,\ C_1C_2\ \vdash\ \lambda v.e\ =\ \lambda v.e'\ : \forall\alpha{::}\Gamma.\sigma} \qquad \alpha \not\in \mathrm{fv}(A)\ \cup\ \mathrm{reg}(C)$$

$$(\mu) \qquad \frac{A,\ C \vdash e_1\ =\ e_1'\ : \sigma' \to \sigma \qquad A,\ C \vdash e_2\ =\ e_2'\ : \sigma'}{A,\ C \vdash e_1\, e_2\ =\ e_1'\, e_2'\ : \sigma}$$

$$(\xi) \qquad \frac{A.x{:}\sigma',\ C \vdash e\ =\ e'\ : \sigma}{A,\ C \vdash \lambda x.e\ =\ \lambda x.e'\ : \sigma' \to \sigma}$$

$$(cong-let) \qquad \frac{A,\ C \vdash e_1\ =\ {e'}_1\ : \sigma \qquad A.x{:}\sigma,\ C \vdash e_2\ =\ {e'}_2\ : \tau}{A,\ C \vdash (\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2)\ =\ (\mathtt{let}\ x = {e'}_1\ \mathtt{in}\ {e'}_2)\ : \tau}$$

Figure 9: Equation rules for CP expressions, II

Given the typed equality over CP expressions, our task in the subsequent sections is to derive conditions sufficient to guarantee that:

$$\text{If } A, \mathbf{v}{:}C \vdash e \rightsquigarrow e_1 : \sigma \text{ and } A, \mathbf{v}{:}C \vdash e \rightsquigarrow e_2 : \sigma, \text{ then } A, \mathbf{v}{:}C \vdash e_1 = e_2 : \sigma.$$

6 Ordering and Conversion Functions

This section explores the generic instance ordering between type schemes ($\preceq_C$) to relate the translations of an overloaded expression. Following [Jon92], we show that any two translations of an expression can be related by certain functions if their types are related by the ordering. Such functions are called *conversions* since they convert one translation to another, whose type is less general. Furthermore, the notion of conversions between translations naturally leads us to define *principal translations* along the lines of principal types. Later in this part we will show that the principal translation property holds for our translation system. As part of the technical development, we extend the definition of conversions to type asumption sets.

6.1 Conversions and Principal Translations

Given that each typing derivation for a Mini–Haskell$^+$ expression yields a type as well as a translation, and all the types that we can associate with an expression are generic instances of its principal type, it is conceivable to consider the relation between the translation obtained from the principal-type derivation and those from other derivations based on the relation between their types. Indeed, as we will show immediately, a functional relation between these translations can be established through a semantic interpretation of the ordering between their types.

Conversions are functions we use to relate translations; they convert a translation of a more general type to another translation of the same expression whose type is less general. Furthermore, they can be expressed in our system as CP functions. More formally, given a Mini–Haskell$^+$ expression e and two of its translations e_1 and e_2 obtained from the typing derivations $A,\ C \vdash e \rightsquigarrow e_1 : \sigma$ and $A,\ C \vdash e \rightsquigarrow e_2 : \sigma'$ with $\sigma' \preceq_C \sigma$, we are interested in functions K such that $A,\ C \vdash Ke_1 = e_2 : \sigma'$.

We have, therefore, the following characterizations of the conversions. First, it is clear that the type for such expressions is $\sigma \rightarrow \sigma'$, under A and C. Note that this type, in general, cannot be expressed as a Mini–Haskell$^+$ type scheme since it uses the richer structure of CP types. Second, from the translation semantics we know that $\mathit{Erase}(e_1) = e$ and $\mathit{Erase}(e_2) = e$. Now since $\mathit{Erase}(Ke_1) = \mathit{Erase}(K)\mathit{Erase}(e_1)$ and $Ke_1 = e_2$, we need to ensure that $\mathit{Erase}(K)\mathit{Erase}(e_1) = \mathit{Erase}(e_1)$. An obvious choice is to require that $\mathit{Erase}(K)$ be equivalent to the identity function $\mathit{id} = \lambda x.x$.

The insight of [Jon92] is that we can derive from the definition of generic instance ordering a "canonical" conversion that suits our purpose. Such conversions, when applied to a translation of an overloaded expression, repackage the dictionaries involved to yield another translation whose type is less general. In our system, the idea is embodied in the following definition of conversions:

Definition. (Conversions) Given type assumption set A and context $\mathbf{v}:C$. Suppose that $\sigma' = \forall\langle\alpha'_j{::}\Gamma'_j\rangle.\tau'$ and $\sigma = \forall\langle\alpha_i{::}\Gamma_i\rangle.\tau$ and that none of the α'_j occurs free in σ or C. A CP expression K of type $\sigma \rightarrow \sigma'$ under A and $\mathbf{v}:C$ such that $\mathit{Erase}(K) = \mathit{id}$ is called a conversion from σ to

σ' under A and $\mathbf{v} : C$, written $K : \sigma' \preceq_{A,\mathbf{v}:C} \sigma$, if there are types τ_i, dictionary variables $\mathbf{w}$, and dictionary expressions $\mathbf{d}$ such that

- $\tau' = [\tau_i/\alpha_i]\,\tau$
- $\mathbf{v}{:}C \oplus \mathbf{w}{:}\langle\alpha'_j{::}\Gamma'_j\rangle \Vdash \mathbf{d}{:}\,[\tau_i/\alpha_i]\,\langle\alpha_i{::}\Gamma_i\rangle$ and
- $A,\ \mathbf{v}{:}C \vdash K = \lambda x.\lambda\mathbf{w}.\,x{\cdot}\mathbf{d}$

Since conversions contain no free term variables and hence only context $\mathbf{v}{:}C$ is significant in the setting of A and $\mathbf{v}{:}\,C$, we will drop the type assumption set A from the subscript of $\preceq_{A,\mathbf{v}:C}$. We will also omit the dictionary application symbol and write $x\,\mathbf{d}$ for $x{\cdot}\mathbf{d}$ in what follows.

It is straightforward to verify that $\lambda x.\lambda\mathbf{w}.\,x\,\mathbf{d}$ is itself a conversion from σ to σ' under A and $\mathbf{v}{:}\,C$; clearly *Erase*($\lambda x.\lambda\mathbf{w}.\,x\,\mathbf{d}) = \lambda x.\,x$ and the following derivation establishes the required typing:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A.x{:}\sigma,\ \mathbf{v}{:}C \vdash x : \sigma}{A.x{:}\sigma,\ \mathbf{v}{:}C \vdash x : \forall\langle\alpha_i{::}\Gamma_i\rangle.\tau}\ \sigma = \forall\langle\alpha_i{::}\Gamma_i\rangle.\tau}{A.x{:}\sigma,\ \mathbf{v}{:}C \oplus \mathbf{w}{:}\langle\alpha'_j{::}\Gamma'_j\rangle \vdash x\,\mathbf{d} : [\tau_i/\alpha_i]\,\tau}\ (\forall-\mathit{elim})}{A.x{:}\sigma,\ \mathbf{v}{:}C \oplus \mathbf{w}{:}\langle\alpha'_j{::}\Gamma'_j\rangle \vdash x\,\mathbf{d} : \tau'}\ \tau' = [\tau_i/\alpha_i]\,\tau}{A.x{:}\sigma,\ \mathbf{v}{:}C \vdash \lambda\mathbf{w}.\,x\,\mathbf{d} : \sigma'}\ (\forall-\mathit{intro}),\ \sigma' = \forall\langle\alpha'_j{::}\Gamma'_j\rangle.\tau'}{A,\ \mathbf{v}{:}C \vdash \lambda x.\lambda\mathbf{w}.\,x\,\mathbf{d} : \sigma \to \sigma'}\ (\lambda-\mathit{intro})$$

This canonical conversion works by repackaging the dictionaries involved. Also, as noted earlier, the types of conversion functions utilize the more expressive polymorphism provided by CP.

The following two lemmas state some properties of conversions that will be useful in subsequent work. The first one is pretty straighforward.

Lemma 6.1 If $K : \sigma' \preceq_{\mathbf{v}:C} \sigma$ and $\mathbf{v}{:}C \sqsubseteq \mathbf{u}{:}C'$, then $K : \sigma' \preceq_{\mathbf{u}:C'} \sigma$.

The second lemma shows when two conversions can be meaningfully composed.

Lemma 6.2 If $K' : \sigma'' \preceq_{\mathbf{v}:C} \sigma'$ and $K : \sigma' \preceq_{\mathbf{v}:C} \sigma$ then $(K' \circ K) : \sigma'' \preceq_{\mathbf{v}:C} \sigma$.

The introduction of conversions is a key step towards our goal. With the notion of conversion we can generalize the definition of principal types to that of *principal translations*, which are translations obtained from the principal-type derivations and from which all other translations can be derived. If the principal translation property holds for our system, our task of determining the equivalence of two translations is reduced to that of the two conversions derived between the principal translation and the respective translations, which is simpler since conversions are also CP expressions but with regular structures. The following definition formalizes the notion of principal translations:

Definition. (Principal translations) Given A, $\mathbf{v}{:}\,C$, and e, we call $e'{:}\sigma$ a *principal translation* for e under A and $\mathbf{v}{:}\,C$ iff $A, \mathbf{v}{:}\,C \vdash e \rightsquigarrow e' : \sigma$, and for every σ', if $A, \mathbf{v}{:}\,C \vdash e \rightsquigarrow e'' : \sigma'$ and $K : \sigma' \preceq_{\mathbf{v}:C} \sigma$, then $A, \mathbf{v}{:}\,C \vdash K\,e' = e'' : \sigma'$.

6.2 Conversions Between Type Assumption Sets

We have seen in [COH92] the extension of the generic instance to an ordering between type assumption sets. Similarly, we can extend the definition of conversions to type assumption sets. The additional complexity here is that we need to express multiple conversions, one for each pair of types associated with a term variable in the respective type assumption sets. Since each of these conversions maps a term variable to an expression, we use *expression substitutions* to define the conversions between type assumption sets, as suggested by [Jon92].

As a simple example, consider the following two type assumption sets:

$$\begin{aligned} A' &= \{ (==) : \forall \text{a::Eq}.\, \texttt{List a} \to \texttt{List a} \to \texttt{Bool} \} \\ \text{and} \quad A &= \{ (==) : \forall \text{a::Eq}.\, \texttt{a} \to \texttt{a} \to \texttt{Bool} \} \end{aligned}$$

Assuming that $\texttt{w}:(\text{a::Eq}) \Vdash (\texttt{DEqList w}) : (\texttt{List a} :: \text{Eq})$, one possible conversion between A and A' would be a substitution that maps (==) to $\lambda\texttt{w}.(==)(\texttt{DEqList w})$, but leaves other variables unchanged.

The following definition formalizes the idea:

Definition. (Conversions between type assumption sets) A substitution K is a conversion from a type assumption set A to another type assumption set A' under context $\mathbf{v} : C$, written $K : A' \preceq_{\mathbf{v}:C} A$, if

- dom A = dom A' and
- For each $x \in$ dom A there is a conversion $\lambda x.K(x) : A'(x) \preceq_{\mathbf{v}:C} A(x)$. On the other hand, if $x \notin$ *dom* A, then $K(x) = x$.

Note that since every conversion is a CP expression without any free term variables, it follows that the only free term variable that appears free in the expression of the form $K(x)$ is the variable x itself. Based on this observation, we can easily establish the following results:

Lemma 6.3 If $K : A' \preceq_{\mathbf{v}:C} A$, then

1. $K(\lambda x.\, e) = \lambda x.\, K_x e$,
2. $K(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) = (\texttt{let}\ x = Ke_1\ \texttt{in}\ K_x e_2)$,
3. $K : (A.x : \sigma) \preceq_{\mathbf{v}:C} (A'.x : \sigma)$ for any σ, and
4. $[e_1/x](K_x e_2) = (K[e_1/x])e_2$ for any e_1 and e_2

where K_x stands for the substitution such that $K_x(x) \equiv x$ and $K_x e \equiv Ke$ for any expression e that $x \notin \text{fv}(e)$.

The following lemma is a direct consequenc of the definition.

Lemma 6.4 If $K : A' \preceq_{\mathbf{v}:C} A$ and $\mathbf{v} : C \sqsubseteq \mathbf{u} : C'$, then $K : A' \preceq_{\mathbf{u}:C'} A$

7 Syntax-directed Translation

The next three sections follow the developments of [COH92] to extend our type inferencer to include the calculuation of translations for any given expression. To begin with, we extend the syntax-directed typing rules given in [COH92] to include the construction of translations. Figure 10 gives the extended inference rules.

Note that although the structure of a derivation $A,\ C\ \vdash'\ \ e \rightsquigarrow e'\ :\tau$ is uniquely determined by the syntactic structure of the expression e, it need not be the case for the translation e'. The reason is that in rule (*var'*), the dictionary expressions $\mathbf{d}$ are determined by the types τ_i we choose to instantiate the quantified type variables, and there may be distinct choices for such types and consequently distinct dictionary expressions. This is exactly where incoherence may occur in the translation semantics of Mini–Haskell$^+$.

$$(var')\quad \frac{A(x)\ =\ \forall\langle\alpha_i{::}\Gamma_i\rangle.\tau \qquad C \Vdash \mathbf{d} : ([\tau_i/\alpha_i]\,\langle\alpha_i{::}\Gamma_i\rangle)}{A,\ C\ \vdash'\ x \rightsquigarrow x\mathbf{d}\ :[\tau_i/\alpha_i]\tau}$$

$$(\lambda-E')\quad \frac{A,\ C\ \vdash'\ e_1 \rightsquigarrow {e'}_1\ :\tau' \to \tau \qquad A,\ C\ \vdash'\ e_2 \rightsquigarrow {e'}_2\ :\tau'}{A,\ C\ \vdash'\ e_1\ e_2 \rightsquigarrow {e'}_1\ {e'}_2\ :\tau}$$

$$(\lambda-I')\quad \frac{A.x:\tau',\ C\ \vdash'\ e \rightsquigarrow e'\ :\tau}{A,\ C\ \vdash'\ \lambda x.e \rightsquigarrow \lambda x.e'\ :\tau' \to \tau}$$

$$(let')\quad \frac{A,\ \mathbf{v}':C'\ \vdash'\ e_1 \rightsquigarrow {e'}_1\ :\tau_1 \qquad A.x:\sigma,\ \mathbf{v}:C\ \vdash'\ e_2 \rightsquigarrow {e'}_2\ :\tau_2}{A,\ C\ \vdash'\ (\texttt{let}\ x = e_1\ \texttt{in}\ e_2) \rightsquigarrow (\texttt{let}\ x = \lambda\mathbf{w}.{e'}_1\ \texttt{in}\ {e'}_2)\ :\tau_2}$$
$$\text{where } (\sigma,\ \mathbf{v}'':C'',\ \mathbf{w}) = gen(\tau_1,\ A,\ \mathbf{v}':C',\epsilon) \text{ and } \mathbf{v}'':C'' \sqsubseteq \mathbf{v}:C$$

Figure 10: Syntax-directed Translation Rules

To accommodate dictionary abstractions in translating `let`–expressions, we extend the definition of function *gen* as follows:

$$\begin{aligned} gen\,(\sigma,\ A, \mathbf{v}:C, \mathbf{w}) \ &=\ \textbf{if}\ \exists(v:(\alpha{::}\Gamma)) \in \mathbf{v}:C\ \textbf{and}\ \alpha \notin (\,\mathrm{fv}(A) \cup \mathrm{reg}(C)\,) \\ &\quad\ \textbf{then}\ gen\,(\forall\alpha{::}\Gamma.\sigma,\ A,\ (\mathbf{v}:C)\backslash_{v:(\alpha::\Gamma)},\ v\mathbf{w}) \\ &\quad\ \textbf{else}\ (\sigma,\ \mathbf{v}:C,\ \mathbf{w}) \end{aligned}$$

In other words, now *gen* not only extracts generic type variables from the context but also accumulates the dictionary variables associated with those type variables.

The following three lemmas about the extended *gen* function can be easily established.

Lemma 7.1 If $gen(\tau, A, \mathbf{v}: C, \epsilon) = (\sigma, \mathbf{v}': C', \mathbf{w})$ then $\sigma = \langle \alpha_i :: C\alpha_i \rangle_1^n . \tau$, for some $n \geq 0$ such that $\langle \alpha_i \rangle \oplus C^*(\text{fv } A) = \text{dom}(C)$ and $\mathbf{v} \cong \mathbf{w} \oplus \mathbf{v}'$.

Lemma 7.2 If $A, \mathbf{v}: C \ \vdash \ e \rightsquigarrow e' : \tau$ and $(\sigma, \mathbf{v}': C', \mathbf{w}) = gen(\tau, A, \mathbf{v}: C, \epsilon)$, then $A, \mathbf{v}': C' \ \vdash \ e \rightsquigarrow \lambda \mathbf{w}.\, e' : \sigma$.

Lemma 7.3 Let $(\sigma, C_1, \mathbf{w}) = gen(\tau, A, C, \epsilon)$ and $(\sigma', C_1, \mathbf{v}') = gen(\tau, A, \mathbf{v}: C \oplus \mathbf{u}: D, \epsilon)$. Then $\mathbf{v}' = \mathbf{uw}$, $\lambda x. \lambda \mathbf{u}.\, x : \sigma' \preceq \sigma$ and $\lambda \mathbf{u} \lambda x.\, x\mathbf{u} : \sigma \preceq_{\mathbf{u}:D} \sigma'$.

Our goal in the remainder of this section is to show that the set of syntax-directed translation rules is equivalent to the original set of translation rules given in Figure 7. By a straightforward induction, we can show that the syntax-directed system is sound with respect to the original one:

Theorem 7.4 If $A, \mathbf{v}: C \ \vdash' \ e \rightsquigarrow e' : \tau$ then $A, \mathbf{v}: C \ \vdash \ e \rightsquigarrow e' : \tau$.

To show that the syntax-directed system is also as general as the original one, we need to develop a series of lemmas about the syntax-directed system. We begin with the following two lemmas that describe the interaction between the *gen* function and type substitutions under the common scenario $A, \mathbf{v}: C \ \vdash' \ e \rightsquigarrow e' : \tau$.

Lemma 7.5 Let $(\sigma, C', \mathbf{w}) = gen(\tau, A, \mathbf{v}: C, \epsilon)$ and $(\sigma', \mathbf{u}': D', \mathbf{w}') = gen(S\tau, SA, \mathbf{u}: D, \epsilon)$. If $\sigma = \forall \langle \alpha_i :: \Gamma_i \rangle . \tau$ and there exist dictionary expressions $\mathbf{d}$ such that $\mathbf{u}: D \Vdash \mathbf{d} : \langle \alpha_i :: \Gamma_i \rangle$, then $\lambda x. \lambda \mathbf{u}'.\, \mathbf{w}' x\, \mathbf{d} : \sigma' \preceq_{\mathbf{u}': D'} S\sigma$.

Lemma 7.6 Given $gen(\tau, A, \mathbf{v}: C, \epsilon) = (\sigma, C_0, \mathbf{w})$. If $\mathbf{v}': C' \Vdash \mathbf{d} : SC_0$ then there is a substitution R, a context $\mathbf{u}: D$ and dictionary expressions $\mathbf{d}'$ such that

$$RA = SA, \quad \mathbf{u}: D \Vdash \mathbf{d}' : RC \quad \text{and} \quad \mathbf{d}' \cong \mathbf{wd}.$$

Furthermore, if $gen(R\tau, RA, \mathbf{u}: D, \epsilon) = (\sigma', D', \mathbf{w}')$ then

$$S\sigma = \sigma', \quad D' \sqsubseteq C' \quad \text{and} \quad \mathbf{w}' \equiv \mathbf{w}.$$

The next group of lemmas state the properties of the syntax-directed translation; except the first one, they are all extended from the properties of the syntax-directed type system in [COH92] by including the calculation of translations.

Lemma 7.7 If $A, \mathbf{v}: C \ \vdash' \ e \rightsquigarrow e' : \tau$ then $\text{dv}(e') \subseteq \mathbf{v}$

Lemma 7.8 If $A, \mathbf{v}: C \ \vdash' \ e \rightsquigarrow e' : \tau$ and $\mathbf{v}: C \sqsubseteq \mathbf{v}': C'$, then $A, \mathbf{v}': C' \ \vdash' \ e \rightsquigarrow e' : \tau$.

Lemma 7.9 If $A, \mathbf{v}: C \ \vdash' \ e \rightsquigarrow e' : \tau$ and $\mathbf{v}': C' \Vdash \mathbf{d} : SC$, then $SA, \mathbf{v}: C' \ \vdash' \ e \rightsquigarrow [\mathbf{d}/\mathbf{v}]e' : \tau$.

Lemma 7.10 If $A', \mathbf{v}: C \ \vdash' \ e \rightsquigarrow e' : \tau$ and $K : A' \preceq_{\mathbf{v}:C} A$, then $A, \mathbf{v}: C \ \vdash' \ e \rightsquigarrow e'' : \tau$ and $A, \mathbf{v}: C \ \vdash \ Ke' = e'' : \tau$.

Finally, using these lemmas, we can show that the syntax-directed translation system is also complete with respect to the original one in the following sense:

Theorem 7.11 If $A, \mathbf{v}\!:\!C \vdash e \leadsto e' : \sigma$ then there is a context $\mathbf{v}'\!:\!C'$, a type τ' and ax expression e'' such that $\mathbf{v}\!:\!C \sqsubseteq \mathbf{v}'\!:\!C'$, $A, \mathbf{v}'\!:\!C' \vdash' e \leadsto e'' : \tau'$ and $A, \mathbf{v}\!:\!C \vdash K\,(\lambda \mathbf{w}.e'') = e' : \sigma$ where $K : \sigma \preceq_{\mathbf{v}:C} \sigma'$ and $(\sigma', C'', \mathbf{w}) = gen(\tau', A, \mathbf{v}'\!:\!C', \epsilon)$.

Therefore, any translation derived from the roiginal set of translation rules can also be obtained by using the set of syntax-directed translation rules.

8 Unification and Dictionary Construction

Before we can extend our type reconstruction algorithm to include the calculation of translations, we need to develop some mechanisms to synthesize dictionaries that are essentail to the translation scheme. This section extends the unification algorithm given in citeptclass:yale to incorporate dictionary construction. As discussed therein, unification of types is associated with a context normalization process to ensure that the underlying context is properly preserved. This normalization sub-algorithm can be viewed as an implementation of the instance entailment system of [COH92]. We have also shown in Section 2.2 that we can easily extend the instance entailment system to include dictionary construction using augmented judgements of the form $\mathbf{v}\!:\!C \Vdash \mathbf{d} : (\tau{::}\Gamma)$. Therefore, our main task here is to extend the normalization algorithm to implement the augmented instance entailment system.

Augmented Constrained Substitutions

We extend constrained substitutions defined in [COH92] with dictionary substitutions to handle dictionary construction. Similar to the substitutions of type variables by types, a dictionary substitution is a map from dictionary variables to dictionaries. We use dictionary substitutions to keep track of the dictionaries that are constructed during type reconstruction to synthesize the translation. The initial dictionary substitution maps all the dictionary variables to themselves; as types get unified, along with the associated context normalization process, dictionaries will be constructed to replace dictionary variables involved in the translation being synthesized, thereby yielding more refinied dictionary substitutions. At the end, we obtain the complete translation by applying the resulting dictionary substitution.

Using Θ to denote dictionary substitutions, we extend the definiton of constrained substitutions as follows:

Definition. An *augmented constrained substitution* is a triple $(S, \mathbf{v}\!:\!C, \Theta)$ where S is a substitution, $\mathbf{v}\!:\!C$ a augmented context, and Θ a dictionary substitution such that $C = SC$ and $\mathbf{v} = \Theta\mathbf{v}$.

Consequently, definitions derived from constrained substitutions have to be extended to include dictionary substitutions. The following definition extends the notion of context preserving to augmented constrained substitutions.

Definition. An augmented constrained substitution $(S, \mathbf{v} : C, \Theta)$ *preserves* another augmented constrained substitution $(S_0, \mathbf{v}_0 \!: C_0, \Theta_0)$ if there is a substitution S' and a dictionary substitution Θ' such that $S = S' \circ S_0$, $\Theta = \Theta' \circ \Theta_0$, and $\mathbf{v} : C \Vdash \Theta'\mathbf{v}_0 : S'C_0$. We write in this case $(S, \mathbf{v} \!: C, \Theta) \preceq (S_0, \mathbf{v}_0 \!: C_0, \Theta_0)$.

The augmented context-preserving unifiers and normalizers are similarly defined..

Definition. An augmented constrained substitution $(S, \mathbf{v} \!: C, \Theta)$ is a

(a) $(S_0, \mathbf{v}_0 \!: C_0, \Theta_0)$-preserving *unifier* of the type expressions τ and τ' if $S\tau = S\tau'$ and $(S, \mathbf{v} \!: C, \Theta) \preceq (S_0, \mathbf{v}_0 \!: C_0, \Theta_0)$.

(b) $(S_0, \mathbf{v}_0 \!: C_0, \Theta_0)$-preserving *normalizer* of an instance predicate set P if there exist dictionary expressions $\mathbf{d}$ such that $\mathbf{v} \!: C \Vdash \mathbf{d} : SP$ and $(S, \mathbf{v} \!: C, \Theta) \preceq (S_0, \mathbf{v}_0 \!: C_0, \Theta_0)$.

Algorithm

Given the notion of augmented constrained substitutions, we can now extend our unification algorithm to include dictionary construction. Figure 11 presents the augmented algorithms. A few words on our notations are helpful here. As before, we have used simple contexts where augmented contexts are meant. Thus expression $C\alpha$ yields the class list Γ associated with α as well as the matching list of dictionary variables v. Such augmented class lists are denoted by $v : \Gamma$, but we may simply write Γ if there is no need to mention v.

The augmented algorithms retain the basic structure of the original algorithms. There are still four mutually recursive functions: *mgu*, *mgu'*, *mgn*, and *mgn'*; all follow the same calling patterns of their predecessors. On the other hand, two major changes are made to incorporate dictionary construction. First, the common state thread becomes an augmented constrained substitution. Second, the normalization functions *mgn* and *mgn'* are threaded with an additional argument d to accumulate the dictionaries constructed in checking the satisfiability of the given instance predicate.

More specifically, the context in the augmented constrained substitution keeps track of the class constraints on the underlying type variables and their associated dictionary variables as well. In the meantime, any change to those dictionary variables will be recorded in the dictionaty substitution of the augmented constrained substitution. As in the original algorithm, the call *mgu'* $\alpha\ \tau\ (S, C, \Theta)$ will in turn invoke *mgn* to check the satisfiability of $\tau{::}C\alpha$; but, in addition, *mgn* will return a list of dictionaries as a witness to the satisfication of this instance predicate and as a source for updating the dictionary substitution. These dictionaries are individually constructed by function *mgn'* using the augmented context and the given set of instance declarations.

By some straightforward manipulation, we can extend the properties of the original algorithms to include the construction of dictionaries. Precisely, the following lemmas can be established by similar induction proofs.

Lemma 8.1 (Soundness of *mgu* and *mgn*)

1. If *mgu* $\tau_1\ \tau_2\ (S, \mathbf{v} \!: C, \Theta) = (S', \mathbf{v}' \!: C', \Theta')$, then $S'\tau_1 = S'\tau_2$ and $(S', \mathbf{v}' \!: C', \Theta') \preceq (S, \mathbf{v} \!: C, \Theta)$.

2. If *mgn* $\tau\ \Gamma\ (S, \mathbf{v} \!: C, \Theta,\ d) = (S', \mathbf{v}' \!: C', \Theta', d_1 d)$, then $\mathbf{v}' \!: C' \Vdash d_1 : S'(\tau{::}\Gamma)$ and $(S', \mathbf{v}' \!: C', \Theta') \preceq (S, \mathbf{v} \!: C, \Theta)$.

$$\begin{aligned}
&\mathit{mgu} : \tau \to \tau \to S \times C \times \Theta \to S \times C \times \Theta \\
&\mathit{mgn} : \tau \to \Gamma \to S \times C \times \Theta \times d \to S \times C \times \Theta \times d
\end{aligned}$$

$$\begin{aligned}
\mathit{mgu}\ \tau_1\ \tau_2\ (S, C, \Theta) &= \mathit{mgu}'\ (S\tau_1)\ (S\tau_2)\ (S, C, \Theta) \\
\mathit{mgu}'\ \alpha\ \alpha &= \mathit{id}_{S \times C \times \Theta} \\
\mathit{mgu}'\ \alpha\ \tau\ (S, C, \Theta) \mid \alpha \notin \mathrm{fv}(\tau),\ v{:}(\alpha{::}\Gamma) \in C &= \\
&\mathbf{let}\ \ (S', C', \Theta', d) = \mathit{mgn}\ \tau\ C\alpha\ ([\tau/\alpha]{\circ}S,\ [\tau/\alpha]C\backslash_\alpha,\ \Theta,\ \epsilon) \\
&\mathbf{in}\ \ (S', C', [d/v]{\circ}\Theta) \\
\mathit{mgu}'\ \tau\ \alpha\ (S, C, \Theta) &= \mathit{mgu}\ \alpha\ \tau\ (S, C, \Theta) \\
\mathit{mgu}'\ ()\ () &= \mathit{id}_{S \times C \times \Theta} \\
\mathit{mgu}'\ \kappa\ \tau\ \kappa\ \tau' &= \mathit{mgu}\ \tau\ \tau'\ (S, C, \Theta) \\
\mathit{mgu}'\ (\tau_1 \times \tau_2)\ (\tau_1' \times \tau_2') &= (\mathit{mgu}\ \tau_1\ \tau_1') \circ (\mathit{mgu}\ \tau_2\ \tau_2') \\
\mathit{mgu}'\ (\tau_1 \to \tau_2)\ (\tau_1' \to \tau_2') &= (\mathit{mgu}\ \tau_1\ \tau_1') \circ (\mathit{mgu}\ \tau_2\ \tau_2')
\end{aligned}$$

$$\begin{aligned}
\mathit{mgn}\ \tau\ \{\} &= \mathit{id}_{S \times C \times \Theta \times d} \\
\mathit{mgn}\ \tau\ \mathbf{v}{:}\langle\gamma\rangle\ (S, C, \Theta, d) &= \mathit{mgn}'\ (S\tau)\ \mathbf{v}{:}(S\gamma)\ (S, C, \Theta, d) \\
\mathit{mgn}\ \tau\ (v_1{:}\Gamma_1 \oplus v_2{:}\Gamma_2) &= (\mathit{mgn}\ \tau\ v_1{:}\Gamma_1) \circ (\mathit{mgn}\ \tau\ v_2{:}\Gamma_2) \\
\mathit{mgn}'\ \alpha\ \mathbf{v}{:}(c\ \tau)\ (S, C, \Theta, d) &= \mathbf{if}\ \exists\tau'.\,\mathtt{w}{:}(c\ \tau') \in C\alpha \\
&\ \mathbf{then\ let}\ \ (S', C', \Theta') = \mathit{mgu}\ \tau\ \tau'\ (S, C, \Theta) \\
&\ \qquad\quad \mathbf{in}\ \ (S', C', \Theta', \mathtt{w}d) \\
&\ \mathbf{else}\ \ (S, C[C\alpha \oplus \mathbf{v}{:}(c\ \tau)/\alpha], \Theta, \mathbf{v}d)
\end{aligned}$$

$$\begin{aligned}
&\mathit{mgn}'\ \kappa\,\tau'\ c\,\tau\ (S, C, \Theta, d) \mid \exists \ulcorner \chi : \mathtt{inst}\ C' \Rightarrow \kappa\ \tilde{\tau}'{::}c\ \tilde{\tau} \urcorner\ \mathtt{in}\ \Sigma \\
&\qquad \mathbf{let}\ \ S' = \mathit{match}\ \ \tilde{\tau}'\ (\kappa\,\tau') \\
&\qquad\qquad (S'', C'', \Theta'') = \mathit{mgu}\ \tau\ (S'\tilde{\tau})\ (S, C, \Theta) \\
&\qquad\qquad \langle \tau_1{::}\Gamma_1, \ldots, \tau_n{::}\Gamma_n \rangle = S'C' \\
&\qquad\qquad (S_1, C_1, \Theta_1, \mathbf{d}_1 d) = \\
&\qquad\qquad\qquad (\mathit{mgn}\ \tau_1\ \Gamma_1\ (\ \ldots\ (\mathit{mgn}\ \tau_n\ \Gamma_n\ (S'', C'', \Theta'', d)))) \\
&\qquad \mathbf{in}\ \ (S_1, C_1, \Theta_1, (\chi\,\mathbf{d}_1)d)
\end{aligned}$$

(and similarly for $\to$, $\times$, $()$)

Figure 11: Augmented Unification and Normalization Algorithms

Lemma 8.2 (Completeness of *mgu* and *mgn*)

1. Suppose that $(S', \mathbf{v}' : C', \Theta') \preceq (S_0, \mathbf{v}_0 : C_0, \Theta_0)$ and $S'\tau_1 = S'\tau_2$. Then $mgu\ \tau_1\ \tau_2\ (S_0, \mathbf{v}_0 : C_0, \Theta_0) = (S, \mathbf{v} : C, \Theta)$, with $S\tau_1 = S\tau_2$ and $(S, \mathbf{v} : C, \Theta) \preceq (S', \mathbf{v}' : C', \Theta') \preceq (S_0, \mathbf{v}_0 : C_0, \Theta_0)$.

2. Suppose that $(S', \mathbf{v}' : C', \Theta') \preceq (S_0, \mathbf{v}_0 : C_0, \Theta_0)$ and there exist dictionary expressions d' such that $\mathbf{v}' : C' \Vdash d' : S'(\tau :: \Gamma)$. Then $mgn\ \tau\ \Gamma\ (S_0, \mathbf{v}_0 : C_0, \Theta_0, d) = (S, \mathbf{v} : C, \Theta, d_1 d)$ with $\mathbf{v} : C \Vdash d_1 : S'(\tau :: \Gamma)$ and $(S, \mathbf{v} : C, \Theta) \preceq (S', \mathbf{v}' : C', \Theta') \preceq (S_0, \mathbf{v}_0 : C_0, \Theta_0)$.

Moreover, since the recursive calling patterns are unchanged, the termination property of the original algorithms carries over to the augmented ones.

Lemma 8.3 (Termination of *mgu*) For any augmented constrained substitution (S, C, Θ) and types τ_1, τ_2, the invocation $mgu\ \tau_1\ \tau_2\ (S, C, \Theta)$ either fails or terminates.

Based on these results, we can easily established the following theorem for the augmented unification algorithm.

Theorem 8.4 Given a constrained substitution $(S_0, \mathbf{v}_0 : C_0, \Theta_0)$ and types τ_1, τ_2, if there is a (S_0, C_0, Θ_0)-preserving unifier of τ_1 and τ_2 then $mgu\ \tau_1\ \tau_2\ (S_0, \mathbf{v}_0 : C_0, \Theta_0)$ returns a most general such unifier. If there is no such unifier then $mgu\ \tau_1\ \tau_2\ (S_0, \mathbf{v}_0 : C_0, \Theta_0)$ fails in a finite number of steps.

9 Type Reconstruction and Translation

As the last step towards the coherence result, this section extends our type reconstruction algorithm to include the calculation of translations, and shows that the augmented algorithm computes the most general teranslation for any given expressions.

Figure 12 gives the augmented type inferencer. As before, function *tp* proceeds by cases dispatching on the form of the input expression, but it yields a translation in addition to a type. More precisely, if $tp(e,\ A,\ S,\ \mathbf{v} : C,\ \Theta) = (\tau,\ e',\ S',\ \mathbf{v}' : C',\ \Theta')$, $\Theta' e'$ would be the translation of e at type τ. The dictionary bindings maintained in the dictionary substitution Θ' are acquired through calls to the augmented unfication algorithm presented in the preceding section.

In the remainder of this section, we will establish the principal translation property for our translation semantics. We begin with the key property of *tp* that the augmented constrained substitution produced by *tp* preserves the input one, as formalized by the following lemma.

Lemma 9.1 If $tp(e, A, S, \mathbf{v} : C, \Theta) = (\tau,\ e',\ S',\ \mathbf{v}' : C',\ \Theta')$, then $(S', \mathbf{v}' : C', \Theta') \preceq (S, \mathbf{v} : C, \Theta)$.

The following theorem states that any typing and translation obtained by *tp* can also be derived using the rules for the syntax-directed system described in Section 7.

Theorem 9.2 If $tp(e, A, S, \mathbf{v} : C, \Theta) = (\tau,\ e',\ S',\ \mathbf{v}' : C',\ \Theta')$, then $S'A, \mathbf{v}' : C' \vdash'\ e \leadsto \Theta' e' : \tau$.

Combining this result with Theorem Theorem 7.4, we obtain the soundness property of *tp*:

```
$tp(e, A, S, \mathrm{v}\!:\!C, \Theta)$ = case $e$ of

  $x$ :              $inst\,(S(Ax), x, S, C, \Theta)$

  $e_1\ e_2$ :         let  $(\tau_1, e'_1, S_1, \mathrm{v}_1\!:\!C_1, \Theta_1) = tp(e_1, A, S, \mathrm{v}\!:\!C, \Theta)$
                          $(\tau_2, e'_2, S_2, \mathrm{v}_2\!:\!C_2, \Theta_2) = tp(e_2, A, S_1, \mathrm{v}_1\!:\!C_1, \Theta_1)$
                          $\alpha$ be a new type variable
                          $(S_3, \mathrm{v}_3\!:\!C_3, \Theta_3) = mgu\ \tau_1\ (\tau_2 \to \alpha)\ (S_2, C_2 \oplus (\alpha{::}\langle\rangle), \Theta_2)$
                     in   $(S_3\alpha,\ (e'_1 e'_2),\ S_3,\ C_3, \Theta_3)$

  $\lambda x.e$ :          let  $\alpha$ be a new type variable
                          $(\tau_1, e'_1, S_1, C_1, \Theta_1) = tp\,(e_1, A.x{:}\alpha, S, C \oplus (\alpha{::}\langle\rangle), \Theta)$
                     in   $(S_1\alpha \to \tau_1, (\lambda x.e'_1),\ S_1, C_1, \Theta_1)$

  let $x = e_1$ in $e_2$ : let  $\mathrm{v}'\!:\!C' = \langle\ C\alpha\ \mid\ \alpha \in C^*(\mathrm{fv}\ SA)\ \rangle$
                          $\mathrm{u}\!:\!D = (\mathrm{v}\!:\!C)\backslash(\mathrm{v}'\!:\!C')$
                          $(\tau_1, e'_1, S_1, \mathrm{v}_1\!:\!C_1, \Theta_1) = tp\,(e_1, A, S, \mathrm{v}'\!:\!C', \Theta)$
                          $(\sigma, \mathrm{v}_2\!:\!C_2, \mathrm{w}) = gen\,(\tau_1, S_1A, \mathrm{v}_1\!:\!C_1, \epsilon)$
                          $(\tau, e'_2, S_2, \mathrm{v}_3\!:\!C_3, \Theta_2) = tp\,(e_2, A.x{:}\sigma, S_1, C_2, \Theta_1)$
                     in   $(\tau,\ (\mathtt{let}\ x = \lambda\mathrm{w}.\Theta_1 e'_1\ \mathtt{in}\ e'_2),\ S_2, \mathrm{v}_3\!:\!C_3 \oplus \mathrm{u}\!:\!D, \Theta_2)$
where

  $inst\,(\forall\alpha{::}\Gamma.\sigma, e', S, \mathrm{v}\!:\!C, \Theta)$ = let $\beta$, $u$ be new variables
                                       in $inst\,([\beta/\alpha]\sigma,\ (e'\ u),\ S, \mathrm{v}\!:\!C \oplus u{:}(\beta{::}\Gamma),\ \Theta)$
  $inst\,(\tau, e', S, C, \Theta)$            = $(\tau, e', S, C, \Theta)$
```

Figure 12: Type Reconstruction & Translation Algorithm

Theorem 9.3 If $tp(e, A, S, \mathbf{v}\!:C, \Theta) = (\tau,\ e',\ S',\ \mathbf{v}'\!:C',\ \Theta')$, then $S'A,\ \mathbf{v}'\!:C' \ \vdash\ e \leadsto \Theta'e'\ :\tau$.

Furthermore, any translation derived from the syntax-directed system can be expressed in terms of the translation synthesized by *tp*.

Theorem 9.4 Suppose that $S'A,\ \mathbf{v}'\!:C' \ \vdash'\ e \leadsto e'\ :\tau'$ and $(S', \mathbf{v}'\!:C', \Theta') \preceq (S_0, \mathbf{v}_0\!:C_0, \Theta_0)$. Then $tp(e, A, S_0, \mathbf{v}_0\!:C_0, \Theta_0)$ succeeds with $(\tau,\ e'',\ S,\ \mathbf{v}\!:C,\ \Theta)$, and there is a substitution R and dictionary expressions $\mathbf{d}$ such that

1. $S' = RS$, except on new type variables of $tp(e, A, S_0, \mathbf{v}_0\!:C_0, \Theta_0)$,
2. $\tau' = R\tau$,
3. $\mathbf{v}'\!:C' \Vdash \mathbf{d} : RC$,
4. $S'A,\ \mathbf{v}'\!:C' \ \vdash\ e' = [\mathbf{d}/\mathbf{v}]\Theta e''\ :\tau'$.

Combining this result with Theorem Theorem 7.11, we obtain the completeness property of *tp*:

Corollary 9.5 Suppose that $S'A,\ \mathbf{v}'\!:C' \ \vdash\ e \leadsto e'\ :\sigma'$ and $(S', \mathbf{v}'\!:C', \Theta') \preceq (S_0, \mathbf{v}_0\!:C_0, \Theta_0)$. Then $tp(e, A, S_0, \mathbf{v}_0 : C_0, \Theta_0)$ succeeds with $(\tau,\ e'',\ S,\ \mathbf{v} : C,\ \Theta)$, and there is a substitution R, conversion K and dictionary expressions $\mathbf{d}$ such that

1. $S' = RS$, except on new type variables of $tp(e, A, S_0, \mathbf{v}_0\!:C_0, \Theta_0)$,
2. $K : \sigma' \preceq_{\mathbf{v}':C'} R\sigma$,
3. $\mathbf{v}'\!:C' \Vdash \mathbf{d} : RC''$ and
4. $S'A,\ \mathbf{v}'\!:C' \ \vdash\ K\,(\lambda \mathbf{w}.\,[\mathbf{d}/\mathbf{u}]\Theta e'') = e'\ :\sigma'$

where $(\sigma, \mathbf{u}\!:C'', \mathbf{w}) = gen(\tau, SA, \mathbf{v}\!:C, \epsilon)$.

As a corollary, we obtain the principal translation result:

Corollary 9.6 (Principal translations) Suppose that $\mathrm{dom}(C_0) = (C_0)^*(\mathrm{tv}\ S_0A)$ and $tp(e, A, S_0, \mathbf{v}_0\!: C_0, \Theta_0) = (\tau,\ e',\ S,\ \mathbf{v}\!:C,\ \Theta)$. Then $\lambda\mathbf{w}.\,\Theta e'\!:\sigma$ is a *principal translation* for e under SA and $\mathbf{v}'\!:C'$ where $(\sigma, \mathbf{v}'\!:C', \mathbf{w}) = gen(\tau,\ SA,\ \mathbf{v}\!:C,\ \epsilon)$

10 The Coherence Result

Having established the principal translation property, we can now proceed to develop the conditions that are sufficient to ensure coherent translation. This section defines the notion of ambiguous types and shows that unambiguous principal types entail coherent translations.

The notion of ambiguous types, as first described in the Haskell Report [HJW90], has a rather intuitive interpretation. For example, the following type is ambiguous:

$$\forall \mathtt{a}::\mathtt{Parsable}.\mathtt{String}$$

In this type scheme, the quantified type variable a is constrained by the class Parsable, but does not appear in the type proper String. Given such a type scheme during type reconstruction, unification is not able to determine which instance type of Parsable to instantiate a since only type proper of a type scheme is used in unification. indeed, as discussed in Section 4, overloaded expressions of this type may have several distinct translations and hence should be rejected.

Our definition of ambiguous types generalizes that of Haskell to include parameterized classes. Before presenting the formal definitions, it is instructive to consider some examples: The following type scheme is ambiguous for reasons similar to those of the preceding example on the quantified type variable k.

$$\forall \texttt{a}::\{\texttt{Eq}\}.\forall \texttt{k}::\{\texttt{Collection a}\}.\texttt{a} \rightarrow \texttt{Bool}$$

However, the following one is not consiodered ambiguous:

$$\forall \texttt{a}::\{\texttt{Eq}\}.\forall \texttt{k}::\{\texttt{Collection a}\}.\texttt{k} \rightarrow \texttt{k}$$

In this type scheme, although the quantified type variable a is constrained by class Eq and does not appear in the type proper, it is, through the constraint Collection a, a dependent of another type variable k, which does appear in the type proper. Once k is instantiated to some type τ through unification, we can obtain a's value as a consequence of sovling the instance predicate τ::Collection a.

As illustrated in the examples, quantified type variables manifest the potential ambiguity of a type scheme. In general, a type scheme is not ambiguous if its quantified type variables that are constrained by classes are also depended upon, directly or indirectly, by its type proper. This dependence relation can be expressed through the context closure operation C^* defined in [COH92], which computes, for a given set of type variables Δ, the set of type variables that are related, directly or indirectly, to those in Δ through the class constraints in C. The following definition of ambiguous type variables formalizes the idea.

Definition. (Ambiguous type variables) A quantified type variable α in a type scheme $\sigma = \forall\langle\alpha_i::\Gamma_i\rangle.\tau$ is *ambiguous* if $C_\sigma(\alpha) \neq \emptyset$ and $\alpha \notin C^*_\sigma(\text{tv } \tau)$ where C_σ stands for the generic context, $\langle\alpha_i::\Gamma_i\rangle$, of σ.

Clearly, a type scheme is ambiguous if it contains ambiguous type variables. Note that in Haskell, $C^*_\sigma(\text{tv } \tau) = \text{tv } \tau$; therefore, this definition generalizes the notion of ambiguous types described in the Haskell Report.

For the coherence result, we are more interested in unambiguous types.

Definition. (Unambihuous type schemes) A type scheme $\sigma = \forall\langle\alpha_i::\Gamma_i\rangle.\tau$ is *unambiguous* if none of the α_i is ambiguous.

We are now ready to illustrate why unambiguous types entail coherent translation. From Corollary 9.6 we know that any translation of a Mini-Haskell$^+$ expression e in a particeular setting can be written in the form $K e'$ where e' is e's principal translation and K is some suitable conversion. Now suppose that we have two arbitrary derivations $A, \mathbf{v} : C \ \vdash \ e \rightsquigarrow e'_1 \ : \sigma'$ and $A, \mathbf{v}:C \vdash e \rightsquigarrow e'_2 : \sigma'$. It then follows that:

$$A, \mathbf{v}:C \vdash e'_1 = K_1\, e' : \sigma' \text{ and } A, \mathbf{v}:C \vdash e'_2 = K_2\, e' : \sigma'$$

where K_1 and K_2 are conversions from e's principal type σ to σ' under $\mathbf{v} : C$. Clearly, the two translations would be equivalent if $\mathbf{v}:C \vdash K_1 = K_2$.

Assume that $\sigma' = \forall\langle\alpha'_j{::}\Gamma'_j\rangle.\nu'$ and $\sigma = \forall\langle\alpha_i{::}\Gamma_i\rangle.\nu$ and that none of the α'_j occurs free in σ or C. It follows from the definition of conversions that

$$[\tau_i/\alpha_i]\nu = \nu' \quad \text{and} \quad \mathbf{v}{:}C \oplus \mathbf{w}{:}\langle\alpha'_j{::}\Gamma'_j\rangle \Vdash \mathbf{d}_1 : [\tau_i/\alpha_i]\langle\alpha_i{::}\Gamma_i\rangle$$

for some types τ_i and that $\mathbf{v}{:}C \ \vdash \ K_1 = \lambda x.\lambda \mathbf{w}.\, x\, \mathbf{d}_1$. Similarly for K_2 there are types τ'_i such that

$$[\tau'_i/\alpha_i]\nu = \nu' \quad \text{and} \quad \mathbf{v}{:}C \oplus \mathbf{w}{:}\langle\alpha'_j{::}\Gamma'_j\rangle \Vdash \mathbf{d}_2 : [\tau'_i/\alpha_i]\langle\alpha_i{::}\Gamma_i\rangle$$

and $\mathbf{v}{:}C \ \vdash \ K_2 = \lambda x.\lambda \mathbf{w}.\, x\, \mathbf{d}_2$. Obviously, it is sufficient to show that $\mathbf{d}_1 = \mathbf{d}_2$ to prove that these two conversions are equivalent. This in turn depends on whether the two instance predicate lists $[\tau_i/\alpha_i]\langle\alpha_i{::}\Gamma_i\rangle$ and $[\tau'_i/\alpha_i]\langle\alpha_i{::}\Gamma_i\rangle$ are identical since the dictionaries are uniquely determined by the instance predicates. But in general this is not true due to the differences between the types τ_i and τ'_i.

On the other hand, since $[\tau_i/\alpha_i]\nu = \nu' = [\tau'_i/\alpha_i]\nu$, it follows that $\tau_i = \tau'_i$ for all $\alpha_i \in \mathrm{tv}(\nu)$. This result can be fruther extended to those type variables' dependents. Recall the consistency requirement for parameterized classes:

$$\tau :: c\ \tau_1 \ \text{ and } \ \tau :: c\ \tau_2 \ \text{ implies } \ \tau_1 = \tau_2.$$

The requirement enables us to equate more types between τ_i and τ'. Indeed, a straightforward inductive reasoning gives $\tau_i = \tau'_i$ for all $\alpha_i \in C^*_\sigma(\mathrm{tv}\ \nu)$.

Now, if σ is unambiguous, for all α_i, either $C_\sigma\alpha_i = \emptyset$ or $\alpha_i \in C^*_\sigma(\alpha_i)$. The former case needs no dictionary; the latter one yields the same dictionary since $\tau_i = \tau'_i$. Consequently, all conversions from σ to any of its instances are equivalent:

Lemma 10.1 If $K_1, K_2 : \sigma' \preceq_{\mathbf{v}:C} \sigma$ are conversions and σ is an unambiguous type scheme then $\mathbf{v}{:}C \ \vdash \ K_1 = K_2$.

As a corollary, it follows that unambiguous principal type entails coherent translations.

Theorem 10.2 (Coherence) If $A, \mathbf{v}:C \ \vdash \ e \leadsto e'_1 : \sigma$ and $A, \mathbf{v}:C \ \vdash \ e \leadsto e'_2 : \sigma$ and the principal type of e under A and C is unambiguous, then $A, \mathbf{v}{:}C \ \vdash \ e'_1 = e'_2 : \sigma$.

The practical significance of this technical result is clear. If the principal type, computed by the type inferencer, of the given expression is not unambiguous, we cannot guarantee a well-defined semantics for the expression and hence must reject it. Otherwise, we are sure that the translation calculated is well-defined.

References

[BCGS89] V. Breazu, T. Coquand, C. Gunter, and A. Scedrov. Inheritance and explicit coercion. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 112–129, June 1989.

[Che94] Kung Chen. *A Parametric Extension of Haskell's Type Classes.* PhD thesis, Yale University, New Haven, Connecticut, September 1994.

[COH92] Kung Chen, Martin Odersky, and Paul Hudak. Type inference for parametric type classes. Technical Report YALEU/DCS/RR-900, Dept. of Computer Science, Yale University, New Haven, Conn., June 1992.

[HJW90] Paul Hudak, Simon Peyton Jones, and Philip L. Wadler. Report on the programming language Haskell: a non-strict, purely functional language, version 1.0. Technical Report YALEU/DCS/RR-777, Dept. of Computer Science, Yale University, New Haven, Conn., April 1990. Currert version 1.2, March 1992.

[HS86] R. Hindley and J. Seldin. *Introduction to Combinators and λ-Calculus*, volume 1 of *London Mathematical Society Student Texts.* Cambridge University Press, 1986.

[Jon92] Mark P. Jones. *Qualified types: theory and practice.* PhD thesis, Oxford University, Oxford, UK, July 1992.

[Wad90] Philip Wadler. Simplified overloading for haskell. Note sent to the Haskell mailing list, October 1990.

[WB89] Phil Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Sixteenth Annual ACM Symp. on Principles of Programming Languages*, pages 60–76. ACM, 1989.