Essential Language Support for Generic Programming: Formalization Part 1 Technical Report 605

Jeremy Siek and Andrew Lumsdaine

December 21, 2004

Abstract

"Concepts" are an essential language feature needed to support generic programming in the large. Concepts allow for succinct expression of bounds on type parameters of generic algorithms, enable systematic organization of problem domain abstractions, and make generic algorithms easier to use. In this paper we formalize the design of a type system and semantics for concepts that is suitable for non-type-inferencing languages. Our design shares much in common with the type classes of Haskell, though our primary influence is from best practices in the C++ community, where concepts are used to document type requirements for templates in generic libraries. The technical development in this paper defines an extension to System F and a type-directed translation from the extension back to System F. The translation is proved sound; the proof is written in the human readable but machine checkable Isar language and has been automatically verified by the Isabelle proof assistant. This document was generated directly from the Isar theory files using Isabelle's support for literate proofs.

Contents

1 Introduction

Generic programming is an effective methodology for developing reusable software libraries. Musser and Stepanov developed the methodology in the late 1980's [32, 33] and applied it to the construction of sequence and graph algorithms in Scheme, Ada, and C. In the early 1990's they shifted focus to C++ and took advantage of templates [46] to construct the Standard Template Library [45] (STL). The STL became part of the C++ Standard, which brought generic programming into the mainstream. Since then, generic programming has been successfully applied in the creation of generic libraries for numerous problem domains [4, 24, 38, 41, 43, 48, 50].

A distinguishing characteristic of generic programming is that generic algorithms are expressed in terms of properties of types, rather than in terms of any particular type. A generic algorithms can be used (more importantly, reused) with any type that has the necessary properties. (Support for generic programming in a statically typed language thus requires type parameterization.)

A fundamental issue in providing language support for generic programming is how to express the set of admissible types for a given algorithm, or equivalently, how to design a type system that can check calls to a generic (type-parameterized) algorithm and separately check the implementation of the algorithm. An important complementary issue is providing the run-time mechanism by which user-defined operations, such as multiplication for a BigInt type, are connected with uses of operations inside a generic algorithm, such as a call to $x * x$ in an algorithm parameterized on the number type. In today's programming languages there are three common approaches to addressing these issues: subtype bounds, type classes, and by-name operation lookup. We briefly describe each of these approaches below and show examples in Figure 1.

Subtype Bounds (Figure 1 (a)) In object-oriented languages, bounds on type parameters are typically expressed via subtyping [7, 8, 37]. When a generic function constrains a type parameter to be a subtype of an interface, objects passed to the generic function must carry along the necessary operations. This approach is used in Eiffel [28] and in the generics extensions to Java [6] and C# [23, 29].

Type Classes (Figure 1 (b)) In Haskell, type classes are used to describe the set of admissible types to a generic function [49]. A type class contains a list of required operations, and a type is declared to belong to a type class through an instance declaration that provides implementations of the required operations. If a type parameter to a generic function is constrained to be an instance of a type class, operations from the appropriate instance declaration are implicitly passed into the generic function. A type class is similar to an object-oriented interface in that it specifies a set of required operations. However, unlike interfaces, type classes are not themselves types (e.g., one cannot declare a variable with a type class as its type).

By-Name Operation Lookup (Figure 1 (c)) In CLU [26] and Cforall [11], a generic function declares the name and signature of all the operations it needs. Then at a call to the generic function, the enclosing scope must contain definitions of functions with the appropriate names and signatures. These functions are then passed implicitly into the generic function. The approach used in C++ is similar in that individual operations are found based on their names. However, a generic function does not explicitly declare which operations it needs. Instead, name resolution in the body of the function is performed after instantiation, using argument-dependent lookup [16].

In [12] we implemented a generic graph library (based on the Boost Graph Library [42])

```
public interface Number<U> {
  public U mult(U other);
}
public class BigInt implements Number<BigInt> {
  public BigInt mult(BigInt x) { ... }
  ...
}
public class Square {
  <T extends Number<T>>
  T square(T x) { return x.mult(x); }

  public static void main(String[] args) {
    square(BigInt(4));
  }
}
```

(a) Subtyping: parameter T must extend the Number interface.

```
class Number a where
    mult :: a →a →a

instance Number Int where
   mult = (∗)

square :: Number a ⇒a →a
square x = mult x x

main = square (4::Int)
```

(b) Type classes: parameter "a" must be an instance of the Number type class.

```
template <class Number>
Number square(Number x) {
  return mult(x, x);
}

int mult(int x, int y) { return x ∗ y; }

int main() {
  return square(4);
}
```

(c) By-name operation lookup: a function with the name "mult" is found for type int.

Figure 1: Common approaches to realizing generic programming.

using programming languages in each of the above three categories. We carefully evaluated each language with respect to support for generic programming and found that although these approaches were able to support generic programming to varying degrees, none was ideal. The primary limitation was that existing languages do not fully capture the essential feature of generic programming, namely, *concepts*.

In the parlance of generic programming, concepts are used to express sets of admissible types to an algorithm. More specifically, a concept is defined as a collection of abstractions, membership in which is defined by a list of requirements. Concepts as specifications were formalized in the generic programming literature [21, 22, 51], but are more widely known through their use in the documentation of C++ template libraries [5, 44].

Contributions. The current practice of generic programming is impeded because no existing language provides all the features and abstractions needed to support generic programming. In this paper we capture the essence of the necessary language abstractions in a small formal system. Our primary contribution is System F^G, a simple language based on System F [13, 40] that explicitly includes concepts. Our design of F^G reflects a decade of experience in generic library construction in C++. Technically, System F^G is unique because 1) it provides scoped concept and model declarations, 2) concepts integrate nested types and type sharing in a type class-like feature, and 3) it explores the design space of type classes for non-type-inferencing languages. The formal developments in this paper were carried out using the Isabelle/Isar proof assistant [34, 35]. We define System F^G and a translation from F^G to F and prove that the translation is sound. The proof is expressed in the Isar proof language, a language that is both human readable and machine checkable, and the proofs have been verified in Isabelle. This document was generated directly from the Isar theory files.

Road map. Concepts have a number of similarities to the type classes of Haskell [15, 49] and F^G has a number of similarities (and differences) with existing work, which we discuss in Section 2. In Section 3 we provide a brief introduction to Isabelle and Isar. In Section 4 we review System F, formalize its type system in Isabelle, and prove a few properties that are necessary for our proof that the translation from F^G to F is sound. In Section 5 we introduce the syntax of F^G and present some examples that demonstrate generic programming in F^G. We define both the type system and dynamic semantics of F^G in terms of a type-directed translation to System F (similar to the translation of type classes to System F in [15]). We present an informal description of the translation in Section 6 and the Isabelle formalization in Section 7. We prove that the translation is sound in Section 8. Section 9 discusses future work and concludes.

2 Related Work

Of existing languages, Haskell's type classes are the most similar to concepts. They are based purely on parametric polymorphism, as are concepts. A fundamental difference between our approach and that of type classes is that we are targeting languages without Hindley-Milner style type inference. This gives our design more freedom in other aspects. For example, in F^G two concepts may share the same member name (as do classes in object-oriented languages) whereas in Haskell two type classes in the same module may not share the same member name. In addition, our design is based on experience in the field of generic library construction. One of the primary lessons learned from that experience is the need for modularity, especially for good scoping rules. As a result, concepts and models in F^G are expressions, not declarations (as are type classes and instances in Haskell), and they obey the usual lexical scoping rules. The advantages of lexically scoped concepts and models are discussed in Section 5.

Another lesson we learned is that support for associated types is important. In our study [12] we found that without associated types, interfaces of generic algorithms become cluttered with extra type parameters to the point of causing scalability problems, and internal helper types of abstract data types must be exposed, thereby breaking encapsulation. In response to our study, Chakravarty *et al* proposed an extension to Haskell for associating algebraic data types with concepts [9]. Our work differs from that in [9] in three ways. First, our associated types are not algebraic data types but simply requirements for a type definition; all that is necessary for generic algorithms. The second difference is that we include same-type constraints, which are vital for generic algorithms that use associated types. Associated types and same-type constraints will be treated in Part 2 of the technical report. Third, we include concept inheritance (refinement) in our formalism. Earlier extensions to Haskell [10, 19] address some of the same issues solved by associated types, but they did not address the problems of clutter and encapsulation.

In Standard ML [30], a rough analogy can be made between ML signatures and F^G concepts, and between ML structures and F^G models. However, there are significant differences. Fist, functors are module-level constructs and therefore provide a more coarse-grained mechanism for parameterization than do generic functions. More importantly, functors require explicit instantiation with a structure, thereby making their use more heavyweight than generic functions in F^G or Haskell, which perform automatic lookup of the required structure. The associated types and same-type constraints of F^G are roughly equivalent to types nested in ML signatures and to type sharing. We reuse some implementation techniques from ML such as a union/find based algorithm for deciding type equality [27]. There are numerous other languages with parameterized modules [1, 14, 39] that also require explicit instantiation with a structure.

As discussed in the introduction, many object-oriented languages choose to express bounds on type parameters via subtyping [6, 23, 28, 29]. For a detailed account of the problems we encountered with the subtype-based approach we refer the reader to our study [12].

In some sense, our work combines some of the best features of Haskell and ML relative to generic programming. However, there are non-trivial details to combining these features and these details are discussed in detail in this paper.

3 Introduction to Isabelle and Isar

Isabelle is a generic proof assistant, and Isabelle/HOL is the version of Isabelle that supports reasoning in higher-order logic. The Isar proof language is a front end to Isabelle that provides both a human readable presentation and a machine checkable formalism. We provide a short introduction to Isabelle and Isar here, which we hope is enough to enable the reader to understand this paper. For a more detailed introduction we refer to the reader to [34, 35].

The following is an example proof in Isar. The lemma proves that the length of two lists appended is the sum of the length of the two lists. The label *length-append* has been given to the lemma so that we can use it in other proofs. Like most proofs in this document, this proof is by induction. The induction is on the list *ls1*. Isabelle encompasses an ML-like functional language, complete with support for data types. Since there are two constructors for the list data type, there will be two cases for the induction. A long dash indicates the start of a comment.

lemma *length-append*: $\forall$ *ls2*. *length* (*ls1*@*ls2*) = *length ls1* + *length ls2*
proof (*induct ls1*)
— The first case is for the empty list. The keyword "show" indicates that a subgoal of the lemma is to be proved. The phrase "by simp" indicates that the statement will be proved using Isabelle's simplifier, which expands definitions, in this case length and append, and performs some simple arithmetic and logic.
show $\forall$ *ls2*. *length* ([] @ *ls2*) = *length* [] + *length ls2* **by** *simp*
next — The second case is for when *ls1* = *x*#*xs*. The keyword "fix" introduces fresh variables.
fix *x xs* — The keyword "assume" introduces one or more premises. We often use the label IH for an induction hypothesis.
assume *IH*: $\forall$ *ls2*. *length* (*xs* @ *ls2*) = *length xs* + *length ls2*
show $\forall$ *ls2*. *length* ((*x*#*xs*) @ *ls2*) = *length* (*x*#*xs*) + *length ls2*
proof *clarify* — "clarify" decomposes logical constructs such as $\forall$ and $\longrightarrow$.
fix *ls2* — The "have" below states an intermediate result.
have *length* ((*x*#*xs*) @ *ls2*) = *length* (*x*#(*xs*@*ls2*)) **by** *simp*
— The keyword "also" indicates equational reasoning. The ellipses stand for the previous right-hand side.
also have ... = *1* + *length* (*xs*@*ls2*) **by** *simp*
— Previously proven statements can be used via the "from" keyword followed by the labels for the statements.
also from *IH* **have** ... = *1* + *length xs* + *length ls2* **by** *simp*
— The keyword "ultimately" indicates we are finished with the equational reasoning and have the first left-hand side equal to the last right-hand side
ultimately have *length* ((*x*#*xs*) @ *ls2*) = *1* + *length xs* + *length ls2* **by** *simp*
— "thus" is like "show", but uses the previous statement.
thus *length* ((*x*#*xs*) @ *ls2*) = *length* (*x*#*xs*) + *length ls2* **by** *simp*
qed
qed

The following *tree* type is an example of Isabelle's facility for defining algebraic data types.

datatype *'a tree* = *Leaf 'a* | *Node 'a tree 'a tree*

Isabelle provides two facilities for the definition of recursive functions. The first restricts definitions to primitive recursive functions, but automatically ensures termination. There must be a pattern match against the input data type, which decomposes the data into its parts. Then a recursive call must refer to one of the parts. The type constructor ⇒ is for (total) functions.

consts *height* :: *'a tree* ⇒ *nat*
primrec
height (*Leaf x*) = *0*
height (*Node a b*) = *1* + *max* (*height a*) (*height b*)

The second facility allows for the definition of total recursive functions, but the user must provide a measure function that decreases with each recursive call. Isabelle will attempt to automatically prove that the measure decreases. If Isabelle fails, the user must provide the appropriate lemmas to allow the termination proof to succeed. Below is a version of quick sort for lists. A lemma concerning the length of a filtered list is needed to prove termination. *Suc* is the constructor for natural numbers that adds one.

lemma *filter-length*: *length* (*filter f xs*) < *Suc* (*length xs*)
 by (*simp add*: *less-Suc-eq-le*)

consts *quicksort* :: *nat list* ⇒ *nat list*
recdef *quicksort measure length*
 quicksort [] = []
 quicksort (*x#xs*) = *quicksort*(*filter* (λ *y*. *y*≤*x*) *xs*) @ [*x*] @ *quicksort*(*filter* (λ *y*. *x*<*y*) *xs*)
(**hints** *recdef-simp*: *filter-length*)

Another important feature of Isabelle is the inductive definition of sets, which will be used in this paper to define judgments of various forms, especially typing judgments. The well typed terms of the simply-typed λ-calculus serves as an example of an inductively defined set. The following data types represent the types and terms of the simply-typed λ-calculus. Nice syntax for the data type constructors is defined in the parentheses.

datatype *stlc-type* = *Fun stlc-type stlc-type* (**infixl** → *100*) | *Bot* (⊥ *100*)
datatype *stlc-term* = *Vrbl nat* (‘-) | *Apply stlc-term stlc-term* (-·-) | *Abs nat stlc-term* (λ -. -)

The set of well typed terms is actually a triple, consisting of a type assignment, a term, and its type. Several labeled introduction rules are defined for the set.

consts *well-typed* :: ((*nat* ⇒ *stlc-type*) × *stlc-term* × *stlc-type*) *set*
inductive *well-typed* **intros**
 stlc-var: (Γ, ‘*x*, Γ *x*) ∈ *well-typed*
 stlc-app: ⟦ (Γ, *e1*, τ→τ′) ∈ *well-typed*; (Γ, *e2*, τ) ∈ *well-typed* ⟧
 ⟹ (Γ, *e1* · *e2*, τ′) ∈ *well-typed*
 stlc-abs: (Γ(*x*:=τ), *e*, τ′) ∈ *well-typed* ⟹ (Γ, λ *x*. *e*, τ→τ′) ∈ *well-typed*

The double arrow ⟹ is Isabelle's meta-level implication, and ⟦ P; Q ⟧ ⟹ R is an abbreviation for $P \Longrightarrow Q \Longrightarrow R$. The notation $\Gamma(x{:=}\tau)$ stands for function update:

$f(a := b) \equiv \lambda x.\ \mathit{if}\ x = a\ \mathit{then}\ b\ \mathit{else}\ f\ x$

Figure 2: Types and Terms of System F

$$
\begin{array}{lll}
s, t & \in & \text{Type Variables} \\
x, y, d & \in & \text{Term Variables} \\
n & \in & \mathbb{N} \\
\sigma, \tau, \nu & ::= & t \mid \mathsf{fn}\ \overline{\tau} \to \tau \mid \tau \times \cdots \times \tau \mid \forall \bar{t}.\ \tau \\
f & ::= & x \mid f(\overline{f}) \mid \lambda \overline{y : \tau}.\ f \mid \Lambda \bar{t}.\ f \mid f[\overline{\tau}] \\
& \mid & \mathsf{let}\ x = f\ \mathsf{in}\ f \mid \langle f, \ldots, f \rangle \mid \mathsf{nth}\ f\ n
\end{array}
$$

The following creates nice syntax for membership in the inductively defined set.

syntax *well-typed* :: [*nat* ⇒ *stlc-type*, *stlc-term*, *stlc-type*] ⇒ *bool* (- ⊢ - : - [*52*,*52*,*52*] *51*)
translations $\Gamma \vdash e : \tau \rightleftharpoons (\Gamma, e, \tau) \in$ *well-typed*

Isabelle has a facility for typesetting any implication as an inference rule with a horizontal bar, which will be used throughout this paper for the introduction rules of inductively defined sets.

$$
\frac{\Gamma \vdash e1 : \tau \to \tau' \qquad \Gamma \vdash e2 : \tau}{\Gamma \vdash e1 \cdot e2 : \tau'}(\text{STLC-APP}) \qquad \frac{\Gamma(x := \tau) \vdash e : \tau'}{\Gamma \vdash \lambda\, x.\, e : \tau \to \tau'}(\text{STLC-ABS})
$$

4 System F

System F, the polymorphic lambda calculus, is the prototypical tool for studying type parameterization [13, 40]. Figure 2 presents the abstract syntax for the types and terms of System F. Type abstractions and functions have multiple parameters, instead of the more standard single parameter, to facilitate the translation from F^G to F. Tuples are included in the language to serve as the runtime representation of models, and a let expression serves to further simplify the translation. Several constants not included here will be used in the examples, such as fix (for recursion), but these are not included in the formalization because they are trivial to add.

It is possible to write generic algorithms in System F, as demonstrated in Figure 3, with a polymorphic sum function. The function is written in the higher-order style, passing the type-specific add and zero as parameters. However, this approach does not scale: practical algorithms typically require dozens of type-specific operations.

The following data types are used to represent types and terms of System F in Isabelle. Shorthand syntax for the data type constructors is given in the parentheses next to each constructor. Dashes in the syntax are place-holders for arguments.

types *var* = *nat*
datatype *ty* = *VarT var* (‘-) | *ArrowT ty list ty* (*fn* - → -) | *AllT var list ty* (∀ -. -)
  | *TupleT ty list* (⟨-⟩) | *BoolT* | *IntT*

Figure 3: Higher Order Sum in System F

```
let sum =
  (Λ t.
    fix (λ sum : fn(list t, fn(t,t)→t, t)→t.
         λls : list t, add : fn(t,t)→t, zero : t.
         if null[t](ls) then zero
         else add(car[t](ls), sum(cdr[t](ls), add, zero)))) in

let ls = cons[int](1, cons[int](2, nil[int])) in
sum[int](ls, iadd, 0)
```

datatype *trm* = *Var var* (‘-) | *App trm trm list* (**infixl ·**)
  | *Lam var list ty list trm* (λ -:-. -) | *LetTrm var trm trm* (*let* - := - *in* -)
  | *Forall var list trm* (Λ -. -) | *Inst trm ty list* (-[-])
  | *Tuple trm list* (⟨-⟩) | *Nth trm nat* | *Boolean bool* | *Integer int*

4.1 Type Substitution

The process of instantiating a type abstraction substitutes types for occurrences of the parameters in the body of the abstraction. For example, take the identify function $id = \Lambda t.\lambda x{:}t.\ x$ whose type is $\forall\, t.t{\to}t$. Instantiating the identity function $id\ [int]$ substitutes int for t, resulting in $\lambda x{:}int.x$ which has the type $int{\to}int$.

As defined here, type abstractions have multiple parameters, so a list of types will be simultaneously substituted for a list of parameters. The following auxiliary function will be used to search through a list of variables and a corresponding list of types to find the type for a variable (and the position of the variable in the list).

consts *lookup* :: [*var*, *var list*, *'t list*, *nat*] ⇒ (*'t* × *nat*) *option*
primrec
 lookup x [] *vs i* = *None*
 lookup x (*k#ks*) *vs i* =
  (*case vs of* [] ⇒ *None* | *v#vs'* ⇒ *if k* = *x then Some* (*v*,*i*) *else lookup x ks vs'* (*Suc i*))

There are several ways to define substitution. The standard definition is used here and the variable convention is relied on to assure that free variables are not captured during substitution [3]. The recursive function below implements substitution. The nested list in the *ty* datatype prevents the use of Isabelle's **primrec** facility, so **recdef** is used to define substitution. The following two lemmas are needed to prove termination. The first states that if x is in ss, then $size\ x$ is less than $size\ (fn\ ss \to t)$. The second states that if if x is in τs, then $size\ x$ is less than $size\ \langle\tau s\rangle$.

lemma *ty-list-tc1*: $x \in set\ ss \longrightarrow size\ x < Suc\ (ty\text{-}list\text{-}size1\ ss + size\ t)$
 by (*induct ss rule*: *list.induct*, *auto*)

lemma *ty-list-tc2*: $x \in set\ \tau s \longrightarrow size\ x < Suc\ (ty\text{-}list\text{-}size2\ \tau s)$
by (*induct* τs *rule*: *list.induct*, *auto*)

consts *sub-ty* :: $(var\ list \times ty\ list \times ty) \Rightarrow ty$
recdef *sub-ty measure* $(\lambda\ p.\ size\ (snd\ (snd\ p)))$
$sub\text{-}ty(ts,\ \tau s,\ `t) = (case\ (lookup\ t\ ts\ \tau s\ 0)\ of\ None \Rightarrow `t \mid Some\ (\tau,i) \Rightarrow \tau)$
$sub\text{-}ty(ts,\ \tau s,\ fn\ \sigma s \rightarrow \tau) = fn\ (map\ (\lambda\ \sigma.\ sub\text{-}ty(ts,\tau s,\sigma))\ \sigma s) \rightarrow sub\text{-}ty(ts,\tau s,\tau)$
$sub\text{-}ty(ts,\ \tau s,\ \forall\ ss.\ \tau) = (\forall\ ss.\ sub\text{-}ty(ts,\tau s,\tau))$
$sub\text{-}ty(ts,\ \tau s,\ \langle \sigma s\rangle) = \langle map\ (\lambda\ \sigma.\ sub\text{-}ty(ts,\tau s,\sigma))\ \sigma s\rangle$
$sub\text{-}ty(ts,\ \tau s,\ BoolT) = BoolT$
$sub\text{-}ty(ts,\ \tau s,\ IntT) = IntT$
(**hints** *recdef-simp*: *ty-list-tc1 ty-list-tc2*)

The following abbreviations are used for substitution. The notation for substitution on a list of types is slightly different to decrease Isabelle's parsing time. (It increases greatly when there is ambiguity).

$[ts \mapsto \tau s]\tau \equiv sub\text{-}ty\ (ts,\ \tau s,\ \tau)$
$\{ts \mapsto \tau s\}\sigma s \equiv map\ (\lambda\sigma.\ sub\text{-}ty\ (ts,\ \tau s,\ \sigma))\ \sigma s$

4.2 Type Equality

The presence of universal types complicates type equality, since the types $\forall\, t.t \rightarrow t$ and $\forall\, s.s \rightarrow s$ should be equal even though they are syntactically different. Two types are equal when a renaming of bound variables (α conversion) can make them syntactically equal. A renaming will be represented as a function from variables to variables. The following function updates a renaming with a series of variable bindings.

consts *extend* :: $['a\ list,\ 'a\ list,\ 'a \Rightarrow 'a] \Rightarrow ('a \Rightarrow 'a)$
primrec
$extend\ []\ vs\ T = T$
$extend\ (k\#ks)\ vs\ T = (case\ vs\ of\ [] \Rightarrow T \mid v\#vs \Rightarrow T(k{:=}v))$

Figure 4 defines the type equality judgment.

4.3 Type Rules for System F

The type rules will refer to a typing environment that map each λ-bound variable to its type.

types $Tenv = (var \times ty)\ set$

The following notation is used to insert a binding into the environment.

$\Gamma,x{:}\tau \equiv \{(x,\ \tau)\} \cup \Gamma$

The following function adds a list of bindings to the environment.

consts *pushs-env* :: $('k \times 'v)\ set \Rightarrow 'k\ list \Rightarrow 'v\ list \Rightarrow ('k \times 'v)\ set$ (-,-:-)

Figure 4: Equality of types in System F up to the renaming of bound type variables.

$$\frac{t = T\ s}{T \vdash_F \text{‘}s = \text{‘}t}\ (\text{F-EQV}) \qquad \frac{T \models_F \tau s = \tau s' \qquad T \vdash_F \tau = \tau'}{T \vdash_F fn\ \tau s \to \tau = fn\ \tau s' \to \tau'}(\text{F-EQF})$$

$$\frac{extend\ ts\ ts'\ T \vdash_F \tau = \tau'}{T \vdash_F \forall\ ts.\ \tau = \forall\ ts'.\ \tau'}(\text{F-EQA}) \qquad \frac{T \models_F \tau s = \tau s'}{T \vdash_F \langle \tau s\rangle = \langle \tau s'\rangle}(\text{F-EQT})$$

$$T \vdash_F BoolT = BoolT\ \ (\text{F-EQB}) \qquad T \vdash_F IntT = IntT\ \ (\text{F-EQI})$$

$$T \models_F [] = []\ \ (\text{F-EQN}) \qquad \frac{T \vdash_F \tau = \tau' \qquad T \models_F \tau s = \tau s'}{T \models_F \tau{\cdot}\tau s = \tau'{\cdot}\tau s'}\ (\text{F-EQC})$$

primrec
 $\Gamma,[]{:}\tau s = (\Gamma{::}('k \times 'v)\ set)$
 $\Gamma,(x\#xs){:}\tau s = (case\ \tau s\ of\ [] \Rightarrow \Gamma \mid \tau\#\tau s \Rightarrow (\Gamma,xs{:}\tau s),x{:}\tau)$

The domain of an environment is defined as follows.

$dom\ \Gamma \equiv \{x \mid \exists\,\tau.\ (x,\,\tau) \in \Gamma\}$

The type rules for System F also keep track of which type variables are in scope, to ensure that the parameters of a type abstraction are disjoint with all other type parameters in scope and thereby maintain the variable convention. Thus the environment includes both the typing environment for term variables and a set of type variables.

record *Fenv* =
 tys :: *Tenv*
 tvars :: *var set*

The type rules must also ensure that λ-bound variables do not appear as free variables in the environment. The *ftv* function computes the free type variables of a type, and *btv* the bound type variables.

consts *ftv* :: *ty* ⇒ *nat set*
recdef *ftv measure size*
 $ftv\ (\text{‘}t) = \{t\}$
 $ftv\ (fn\ \tau s \to \tau) = \bigcup\,(map\ ftv\ \tau s) \cup ftv\ \tau$
 $ftv\ (\forall\ ts.\ \tau) = ftv\ \tau - set\ ts$
 $ftv\ (\langle \tau s\rangle) = \bigcup\,(map\ ftv\ \tau s)$
 $ftv\ BoolT = \{\}$
 $ftv\ IntT = \{\}$
(**hints** *recdef-simp*: *ty-list-tc1 ty-list-tc2*)
consts *btv* :: *ty* ⇒ *nat set*
recdef *btv measure size*
 $btv\ (\text{‘}t) = \{\}$

$btv\ (fn\ \tau s \to \tau) = \bigcup (map\ btv\ \tau s) \cup btv\ \tau$
$btv\ (\forall\ ts.\ \tau) = btv\ \tau \cup set\ ts$
$btv\ (\langle \tau s \rangle) = \bigcup (map\ btv\ \tau s)$
$btv\ BoolT = \{\}$
$btv\ IntT = \{\}$
(**hints** *recdef-simp*: *ty-list-tc1 ty-list-tc2*)

where we have overloaded $\bigcup$ for a list of sets as defined below. *foldr* is used instead of *foldl* because *foldr* follows the natural structure of a list, which makes it easier to work with when performing induction on lists.

$\bigcup\ ls \equiv foldr\ op \cup ls\ \emptyset$

ftv is extended to typing environments with the following definition.

$FTV\ \Gamma \equiv \bigcup \{V \mid \exists x\ \tau.\ (x, \tau) \in \Gamma \wedge V = ftv\ \tau\}$

The type rules for System F are presented in Figure 5.

4.4 Properties of System F

In this section, some basic properties of System F will be proved, properties concerning substitution, environments, and well typing that are needed later in the report.

A few facts about the lookup function are needed. The first lemma states that lookup fails when the item does not appear in the list of keys. The "is" keyword introduces an abbreviation for the proposition to be proved. The keyword *?thesis* refers to the current subgoal.

```
lemma lookup-fails: ∀ x (vs::'v list) i. x ∉ set ks ⟶ lookup x ks vs i = None (is ?P ks)
proof (induct ks) show ?P [] by simp
next fix k ks assume IH: ?P ks show ?P (k#ks)
 proof clarify fix x and vs::'v list and i assume xmem: x ∉ set (k#ks)
  show lookup x (k#ks) vs i = None
  proof (cases vs) assume vs = [] thus ?thesis by simp
  next fix v vs' assume vs: vs = v#vs' from vs xmem IH show ?thesis by auto
  qed
 qed
qed
```

The next lemma characterizes the pre and post-conditions for a successful lookup. The use of "obtain" corresponds to the elimination of an existential.

```
lemma lookup-succeeds:
 ∀ t (τs::'v list). t ∈ set ts ∧ length ts = length τs
 ⟶ (∀ i. (∃ j. i ≤ j ∧ (j − i) < length ts ∧ ts!(j−i) = t ∧ lookup t ts τs i = Some (τs!(j−i),j)))
 (is ?P ts)
proof (induct ts) show ?P [] by simp
next fix k ks assume IH: ?P ks show ?P (k#ks)
 proof clarify fix t and τs::'v list and i
  assume M: t ∈ set(k#ks) and L: length (k#ks) = length τs
```

Figure 5: Type Rules for System F

$$\frac{(x,\tau) \in \mathit{tys}\ \Gamma}{\Gamma \vdash_F \text{‘}x : \tau}\text{(WT-F-VAR)}$$

$$\frac{\Gamma \vdash_F e : \mathit{fn}\ \sigma s \to \tau \qquad \Gamma \models_F es : \sigma s' \qquad id \models_F \sigma s = \sigma s'}{\Gamma \vdash_F e \bullet es : \tau}\text{(WT-F-APP)}$$

$$\frac{\Gamma(\!|\mathit{tys} := \mathit{tys}\ \Gamma, xs{:}\sigma s|\!) \vdash_F e : \tau \qquad \mathit{set}\ xs \cap \mathit{dom}\ \mathit{tys}\ \Gamma = \emptyset \qquad |xs| = |\sigma s|}{\Gamma \vdash_F \lambda\, xs{:}\sigma s.\ e : \mathit{fn}\ \sigma s \to \tau}\text{(WT-F-ABS)}$$

$$\frac{\Gamma \vdash_F e : \forall\ ts.\ \sigma \qquad |ts| = |\tau s|}{\Gamma \vdash_F e[\tau s] : [ts \mapsto \tau s]\sigma}\text{(WT-F-TAPP)}$$

$$\frac{\begin{array}{c}\Gamma(\!|\mathit{tvars} := \mathit{tvars}\ \Gamma \cup \mathit{set}\ ts|\!) \vdash_F e : \sigma \\ \mathit{set}\ ts \cap \mathit{tvars}\ \Gamma = \emptyset \qquad \mathit{set}\ ts \cap \mathit{FTV}\ (\mathit{tys}\ \Gamma) = \emptyset \qquad \mathit{distinct}\ ts\end{array}}{\Gamma \vdash_F \Lambda\ ts.\ e : \forall\ ts.\ \sigma}\text{(WT-F-TABS)}$$

$$\frac{\Gamma \vdash_F e : \sigma \qquad \Gamma(\!|\mathit{tys} := \mathit{tys}\ \Gamma, x{:}\sigma|\!) \vdash_F e' : \tau \qquad x \notin \mathit{dom}\ \mathit{tys}\ \Gamma}{\Gamma \vdash_F \mathit{let}\ x := e\ \mathit{in}\ e' : \tau}\text{(WT-F-LET)}$$

$$\frac{\Gamma \models_F es : \tau s}{\Gamma \vdash_F \langle es \rangle : \langle \tau s \rangle}\text{(WT-F-TUPLE)} \qquad \frac{\Gamma \vdash_F e : \langle \tau s \rangle \qquad \tau s_{[i]} = \tau}{\Gamma \vdash_F \mathit{Nth}\ e\ i : \tau}\text{(WT-F-NTH)}$$

$$\Gamma \vdash_F \mathit{Boolean}\ b : \mathit{BoolT}\ \text{(WT-F-BOOL)} \qquad \Gamma \vdash_F \mathit{Integer}\ b : \mathit{IntT}\ \text{(WT-F-INT)}$$

$$\Gamma \models_F [] : []\ \text{(WT-F-NIL)} \qquad \frac{\Gamma \vdash_F e : \tau \qquad \Gamma \models_F es : \tau s}{\Gamma \models_F e{\cdot}es : \tau{\cdot}\tau s}\text{(WT-F-CONS)}$$

from L **obtain** $\tau\ \tau s'$ **where** ts: $\tau s = \tau\#\tau s'$ **by** $(induct\ \tau s\ rule: list.induct, auto)$
show $\exists j.\ i \le j \wedge (j{-}i) < length\ (k\#ks) \wedge (k\#ks)!(j{-}i) = t$
$\wedge\ lookup\ t\ (k\#ks)\ \tau s\ i = Some\ (\tau s!(j{-}i), j)$
proof $(cases\ t = k)$ **assume** ta: $t = k$ **from** $ta\ ts$ **show** $?thesis$ **by** $auto$
next assume ta: $t \ne k$
from $ta\ M\ L\ ts\ IH$ **obtain** $j\ \tau'$ **where** I: $Suc\ i \le j$ **and** $jilk$: $(j - Suc\ i) < length\ ks$
and $ksji$: $ks\ !\ (j - Suc\ i) = t$ **and** tsi: $\tau s'!(j - Suc\ i) = \tau'$
and lts: $lookup\ t\ ks\ \tau s'\ (Suc\ i) = Some\ (\tau', j)$ **by** $(auto, blast)$
from I **have** $I2$: $i \le j$ **by** $simp$
from I **have** ij: $Suc\ (j - Suc\ i) = j - i$ **by** $arith$
from $ksji\ tsi$ **have** $(k\#ks)!(Suc\ (j - Suc\ i)) = t \wedge (\tau\#\tau s')!(Suc\ (j - Suc\ i)) = \tau'$ **by** $simp$
with $ij\ ts$ **have** A: $(k\#ks)!(j - i) = t \wedge \tau s!(j{-}i) = \tau'$ **by** $simp$
from $jilk$ **have** B: $(j - i) < length\ (k\#ks)$ **by** $(simp, arith)$
from $lts\ ts\ ta\ A$ **have** C: $lookup\ t\ (k\#ks)\ \tau s\ i = Some\ (\tau s!(j{-}i), j)$ **by** $simp$
from $I2\ A\ B\ C$ **show** $?thesis$ **by** $simp$
qed
qed
qed

Next some basic facts about substitution are proved. Substitution on a list of types commutes with append. Substitution also commutes with the nth function, which is derived directly from the fact that the map function commutes with nth. Substitution does not change the length of a list of types.

lemma *subst-append*: $\forall\ ts\ \tau s\ \sigma s'.\ \{ts{\mapsto}\tau s\}(\sigma s @ \sigma s') = \{ts{\mapsto}\tau s\}\sigma s\ @\ \{ts{\mapsto}\tau s\}\sigma s'$
by $(induct\ \sigma s\ rule: list.induct, auto)$

lemma *subst-nth*: $\forall\ i\ ts\ \sigma s.\ i < length\ \tau s \longrightarrow (\{ts{\mapsto}\sigma s\}\tau s)!i = [ts{\mapsto}\sigma s](\tau s!i)$
using *nth-map* **by** *simp*

lemma *subst-length*: $\forall\ ts\ \sigma s.\ length\ \tau s = length\ (\{ts{\mapsto}\sigma s\}\tau s)$
by $(induct\ \tau s\ rule: list.induct, auto)$

If the variables to be substituted do not occur in the type, then substitution does not change the type. Before proving this, the following function is needed to formalize the notion of occurring type variables.

consts $otv :: ty \Rightarrow nat\ set$
recdef *otv measure size*
$otv\ ('t) = \{t\}$
$otv\ (fn\ \tau s \to \tau) = \bigcup (map\ otv\ \tau s) \cup otv\ \tau$
$otv\ (\forall\ ts.\ \tau) = otv\ \tau \cup set\ ts$
$otv\ (\langle \tau s \rangle) = \bigcup (map\ otv\ \tau s)$
$otv\ BoolT = \{\}$
$otv\ IntT = \{\}$
(**hints** *recdef-simp*: *ty-list-tc1 ty-list-tc2*)

The proof is by induction on the structure of types. The induction rule that Isabelle has generated based on the datatype definition is a mutual induction with three parts. The first part is for types and the second and third parts are for lists of types.

lemma *no-otv-subst-is-id-mutual*:
$(\forall\ ts\ \varrho s.\ set\ ts \cap otv\ \tau = \{\} \longrightarrow [ts{\mapsto}\varrho s]\tau = \tau)$
$\wedge\ (\forall\ ts\ \varrho s.\ set\ ts \cap \bigcup (map\ otv\ \tau s) = \{\} \longrightarrow \{ts{\mapsto}\varrho s\}\tau s = \tau s)$
$\wedge\ (\forall\ ts\ \varrho s.\ set\ ts \cap \bigcup (map\ otv\ \tau s) = \{\} \longrightarrow \{ts{\mapsto}\varrho s\}\tau s = \tau s)$
by (*induct rule*: *ty.induct*, *simp add*: *lookup-fails*, *auto*)

corollary *no-otv-subst-ty-is-id*: $\forall\ ts\ \varrho s.\ set\ ts \cap otv\ \tau = \{\} \longrightarrow [ts{\mapsto}\varrho s]\tau = \tau$
using *no-otv-subst-is-id-mutual* **by** *simp*

The next proof is a standard result called the Substitution Lemma [3]. Again the proof is by induction on types. The following two abbreviations will be used for the propositions to be proved.

constdefs *sub-lemma-ty* :: $ty \Rightarrow bool$
$sub\text{-}lemma\text{-}ty\ M \equiv (\forall\ xs\ ys\ Ls\ Ns.\ set\ xs \cap set\ ys = \{\} \wedge set\ xs \cap \bigcup (map\ otv\ Ls) = \{\}$
$\wedge\ length\ xs = length\ Ns \wedge length\ ys = length\ Ls \wedge distinct\ xs$
$\longrightarrow [ys{\mapsto}Ls]([xs{\mapsto}Ns]M) = [xs{\mapsto}\{ys{\mapsto}Ls\}Ns]([ys{\mapsto}Ls]M))$
constdefs *sub-lemma-tys* :: $ty\ list \Rightarrow bool$
$sub\text{-}lemma\text{-}tys\ Ms \equiv (\forall\ xs\ ys\ Ls\ Ns.\ set\ xs \cap set\ ys = \{\} \wedge set\ xs \cap \bigcup (map\ otv\ Ls) = \{\}$
$\wedge\ length\ xs = length\ Ns \wedge length\ ys = length\ Ls \wedge distinct\ xs$
$\longrightarrow \{ys{\mapsto}Ls\}(\{xs{\mapsto}Ns\}Ms) = \{xs{\mapsto}\{ys{\mapsto}Ls\}Ns\}(\{ys{\mapsto}Ls\}Ms))$

The lemma as normally stated would require that

$set\ xs \cap \bigcup (map\ ftv\ Ls) = \{\}$

however, by the variable convention we also have

$set\ xs \cap \bigcup (map\ btv\ Ls) = \{\}$

Thus we make the variable convention explicit, and include the premise

$set\ xs \cap \bigcup (map\ otv\ Ls) = \{\}$

The following fact about the union of a list of sets will be needed in the proof.

lemma *union-list-elem-subset*: $\forall\ i.\ i < length\ ls \longrightarrow ls!i \subseteq \bigcup ls$
by (*induct ls*, *simp*, *clarify*, *case-tac i*, *auto*)

The case for $M \equiv$ ‘t is the non-trivial part of the lemma. The rest of the cases are either immediate or are proved directly from their induction hypotheses.

lemma *substitution-lemma-var*: *sub-lemma-ty* (‘t)
proof (*simp only*: *sub-lemma-ty-def*, *clarify*)
fix *xs ys* **and** *Ls*::*ty list* **and** *Ns*::*ty list*
assume *disj-xs*: $set\ xs \cap set\ ys = \{\}$ **and** *disj-xl*: $set\ xs \cap \bigcup (map\ otv\ Ls) = \{\}$
and *lxn*: $length\ xs = length\ Ns$ **and** *lyl*: $length\ ys = length\ Ls$ **and** *dxs*: *distinct xs*
let $?P = [ys{\mapsto}Ls]([xs{\mapsto}Ns](‘t)) = [xs{\mapsto}\{ys{\mapsto}Ls\}Ns]([ys{\mapsto}Ls](‘t))$
have $t \in set\ xs \vee t \notin set\ xs$ **by** *simp*
moreover { **assume** *txs*: $t \in set\ xs$ **from** *disj-xs txs* **have** *tys*: $t \notin set\ ys$ **by** *auto*
from *txs lxn* **obtain** *i* **where** *ixs*: $i < length\ xs$ **and** *xsi*: $xs!i = t$
and *ltn*: $lookup\ t\ xs\ Ns\ 0 = Some\ (Ns!i,i)$
using *lookup-succeeds*[*of t xs Ns 0*] **by** *auto*
from *ltn* **have** $[ys{\mapsto}Ls]([xs{\mapsto}Ns](‘t)) = [ys{\mapsto}Ls](Ns!i)$ **by** *simp*

```
    also have ... = [xs↦{ys↦Ls}Ns](‘t)
    proof –
      from txs lxn obtain j where jxs: j < length xs and xsj: xs!j = t
        and ltnp: lookup t xs ({ys↦Ls}Ns) 0 = Some ({ys↦Ls}Ns!j, j)
        using lookup-succeeds[of t xs {ys↦Ls}Ns 0] by auto
      from dxs ixs jxs xsi xsj have ij: i = j using distinct-conv-nth by auto
      from ij jxs lxn   have [ys↦Ls](Ns!i) = {ys↦Ls}Ns!i using subst-nth  by simp
      also from ij ltnp have ... = [xs↦{ys↦Ls}Ns](‘t) by simp
      ultimately show ?thesis by simp
    qed
    also from tys have ... = [xs↦{ys↦Ls}Ns]([ys↦Ls](‘t)) by (simp add: lookup-fails)
    finally have ?P by simp
  } moreover { assume txs: t ∉ set xs
    have t ∈ set ys ∨ t ∉ set ys by simp
    moreover { assume tys: t ∈ set ys
      from tys lyl obtain i where iys: i < length ys and ysi: ys!i = t
        and ltl: lookup t ys Ls 0 = Some (Ls!i,i) using lookup-succeeds[of t ys Ls 0] by auto
      from txs ltl have [ys↦Ls]([xs↦Ns](‘t)) = Ls!i by (simp add: lookup-fails)
      also have ... = [xs↦{ys↦Ls}Ns](Ls!i)
      proof –
        from lyl iys have (map otv Ls)!i ⊆ ⋃(map otv Ls)
          using union-list-elem-subset[of i map otv Ls] by simp
        with lyl iys disj-xl have set xs ∩ otv (Ls!i) = {} by auto
        thus ?thesis using no-otv-subst-ty-is-id by auto
      qed
      also from ltl have ... = [xs↦{ys↦Ls}Ns]([ys↦Ls](‘t)) by simp
      finally have ?P by simp
    } moreover { assume tys: t ∉ set ys
      from tys txs have [ys↦Ls]([xs↦Ns](‘t)) = ‘t by (simp add: lookup-fails)
      also from tys txs have ... = [xs↦{ys↦Ls}Ns]([ys↦Ls](‘t)) by (simp add: lookup-fails)
      finally have ?P by simp
    } ultimately have ?P by blast
  } ultimately show ?P by blast
qed

lemma substitution-lemma-mutual: sub-lemma-ty M ∧ sub-lemma-tys Ms ∧ sub-lemma-tys Ms
  by (induct rule: ty.induct, simp only: substitution-lemma-var, simp-all)

corollary substitution-lemma: set xs ∩ set ys = {} ∧ set xs ∩ ⋃(map otv Ls) = {}
   ∧ length xs = length Ns ∧ length ys = length Ls ∧ distinct xs
   ⟶ [ys↦Ls]([xs↦Ns]M) = [xs↦{ys↦Ls}Ns]([ys↦Ls]M)
  using substitution-lemma-mutual by simp
```

If the variables in *ys* do not occur in *Ms* then the Substitution Lemma can be simplified to the following.

```
corollary substitution-lemma2:
  assumes xsys: set xs ∩ set ys = {} and xsls: set xs ∩ ⋃(map otv Ls) = {}
  and ysM: set ys ∩ otv M = {} and xsNs: length xs = length Ns
  and ysls: length ys = length Ls and dxs: distinct xs
```

shows $[ys{\mapsto}Ls]([xs{\mapsto}Ns]M) = [xs{\mapsto}\{ys{\mapsto}Ls\}Ns]M$
proof −
from *xsys xsls ysM xsNs ysls dxs*
have $[ys{\mapsto}Ls]([xs{\mapsto}Ns]M) = [xs{\mapsto}\{ys{\mapsto}Ls\}Ns]([ys{\mapsto}Ls]M)$
using *substitution-lemma* **apply** *blast* **done**
also from *ysM* **have** $\ldots = [xs{\mapsto}\{ys{\mapsto}Ls\}Ns]M$
using *no-otv-subst-ty-is-id* **by** *simp*
finally show *?thesis* **by** *simp*
qed

A couple facts concerning type environments will be needed. The first fact is a kind of associativity and the second fact is that pushing bindings on the environment commutes with set union.

lemma *pushs-env-assoc*:
$\forall\ dts.\ (S{,}d{:}dt){,}ds{:}dts = S{,}(d\#ds){:}(dt\#dts)$
apply (*induct-tac ds*) **apply** *simp* **apply** *clarify* **apply** (*case-tac dts*) **by** *auto*

lemma *push-union-commute*:
$\forall\ S\ S'\ dts.\ (S{,}ds{:}dts) \cup S' = ((S{::}Tenv) \cup S'){,}ds{:}(dts{::}ty\ list)$
apply (*induct-tac ds*) **apply** *simp* **apply** *clarify* **apply** (*case-tac dts*) **apply** *simp*
proof −
fix *a list* $S\ S'$ **and** *dts*::*ty list* **and** *aa lista*
assume *IH*: $\forall (S{::}Tenv)\ S'\ (dts{::}ty\ list).\ S{,}list{:}dts \cup S' = (S \cup S'){,}list{:}dts$
and *dts*: $dts = aa \# lista$
from *dts* **have** $(S{,}a \# list{:}dts) \cup S' = insert\ (a{,}aa)\ (S{,}list{:}lista \cup S')$ **by** *simp*
also from *IH* **have** $\ldots = insert\ (a{,}aa)\ ((S \cup S'){,}list{:}lista)$ **by** *auto*
also from *dts* **have** $\ldots = (S \cup S'){,}a \# list{:}dts$ **by** *simp*
finally show $S{,}a \# list{:}dts \cup S' = (S \cup S'){,}a \# list{:}dts$ **by** *blast*
qed

Type equality is reflexive.

lemma *extend-refl-id*: $(\lambda u.\ u) = extend\ ls\ ls\ (\lambda u.\ u)$ **by** (*induct ls*, *auto*)

lemma *f-equal-refl-mutual*: $(id \vdash_F \tau = \tau) \wedge (id \models_F \sigma s = \sigma s) \wedge (id \models_F \sigma s = \sigma s)$
apply (*induct rule*: *ty.induct*) **apply** *auto*
proof (*rule f-eqa*)
fix *list*::*var list* **and** *ty* **assume** *E*: $(\lambda u.\ u) \vdash_F ty = ty$
have $(\lambda u.\ u) = extend\ list\ list\ (\lambda u.\ u)$ **by** (*simp add*: *extend-refl-id*)
with *E* **show** $(extend\ list\ list\ (\lambda u.\ u)) \vdash_F ty = ty$ **by** *simp*
qed

corollary *f-eq-refl*: $id \vdash_F \sigma = \sigma$ **by** (*simp add*: *f-equal-refl-mutual*)
corollary *f-eqs-refl*: $id \models_F \sigma s = \sigma s$ **by** (*simp add*: *f-equal-refl-mutual*)

Type equality is also symmetric and the following lemma extends symmetry to lists of types.

lemma *f-eqs-symm*: $\bigwedge \sigma s'.\ T \models_F \sigma s = \sigma s' \Longrightarrow T \models_F \sigma s' = \sigma s$
apply (*induct* σs *rule*: *list.induct*) **apply** (*ind-cases* $T \models_F [] = \sigma s'$, *simp*)

apply (*case-tac* $\sigma s'$) **apply** *simp* **apply** (*ind-cases* $T \models_F a\#list = []$, *simp*)
proof *auto*
fix *a list aa lista*
assume *IH*: $\bigwedge \sigma s'. T \models_F list = \sigma s' \Longrightarrow T \models_F \sigma s' = list$
and *E*: $T \models_F a \# list = aa \# lista$
from *E* **have** $T \models_F list = lista$ **by** (*rule inv-f-eqc*, *simp*)
with *IH* **have** *ls*: $T \models_F lista = list$ **by** *simp*
from *E* **have** $T \vdash_F a = aa$ **by** (*rule inv-f-eqc*, *simp*)
hence *a*: $T \vdash_F aa = a$ **by** (*rule f-eq-symm*)
from *a ls* **show** $T \models_F aa\#lista = a\#list$ **by** *simp*
qed

If two lists of terms are well typed, then appending the lists results in a well typed list of terms.

lemma *wt-f-append*: $\forall\ S\ \tau s\ fs'\ \tau s'.\ S \models_F fs : \tau s \wedge S \models_F fs' : \tau s' \longrightarrow S \models_F fs@fs' : \tau s@\tau s'$
by (*induct fs rule*: *list.induct*, *auto*, *rule inv-wt-f-nil*, *auto*,
rule inv-wt-f-cons, *auto*, *rule wt-f-cons*, *auto*)

Alpha-conversion on types should not affect well typing. This trivial fact requires a fair amount of work to prove, so we simply state the following as axioms for now.

axioms
equal-preserves-wt: $[\![S \vdash_F e : \tau; id \vdash_F \tau = \tau']\!] \Longrightarrow S \vdash_F e : \tau'$
equal-preserves-wts: $[\![S \models_F es : \tau s; id \models_F \tau s = \tau s']\!] \Longrightarrow S \models_F es : \tau s'$

The variables occurring in a type are free or bound.

lemma *otv-ftv-btv*: $(otv\ \tau = ftv\ \tau \cup btv\ \tau)$
$\wedge (\bigcup (map\ otv\ \tau s) = \bigcup (map\ ftv\ \tau s) \cup \bigcup (map\ btv\ \tau s))$
$\wedge (\bigcup (map\ otv\ \tau s) = \bigcup (map\ ftv\ \tau s) \cup \bigcup (map\ btv\ \tau s))$
by (*induct rule*: *ty.induct*, *auto*)

5 Introduction to System F^{G}

The syntax for types and terms of F^{G} is presented in Figure 6. Type abstractions in F^{G} have a where clause that requires certain types to model certain concepts. There is a corresponding where clause in the universal type constructor. The terms of F^{G} also include concept and model declarations, and model member access expressions.

To illustrate the features of F^{G}, we evolve the sum function from Figure 3. To be generic, the sum function should work for any element type that supports addition, so we will capture this requirement in a concept. Mathematicians already have a name for a slightly more generalized concept: a Semigroup is some type together with an associative binary operation (such as addition or multiplication). In F^{G}, the Semigroup concept is defined as follows.

```
concept Semigroup(t) {
  binary_op : fn(t,t)→t;
```

Figure 6: Types and Terms of F^{G}

$$
\begin{array}{lcl}
c & \in & \text{Concept Names} \\
s, t & \in & \text{Type Variables} \\
x, y, z & \in & \text{Term Variables} \\
\rho, \sigma, \tau & ::= & t \mid \mathsf{fn}\ (\overline{\tau}) -> \tau \mid \forall \overline{t}\ \mathsf{where}\ \overline{\overline{\sigma}\ \mathsf{models}\ c}.\ \tau \\
e & ::= & x \mid e(\overline{e}) \mid \lambda y : \tau.\ e \\
 & \mid & \Lambda \overline{t}\ \mathsf{where}\ \overline{\overline{\sigma}\ \mathsf{models}\ c}.\ e \mid e[\overline{\tau}] \\
 & \mid & \mathsf{concept}\ c(\overline{t})\{\mathsf{refines}\ \overline{c(\overline{\sigma})};\ \overline{x : \tau};\}\ \mathsf{in}\ e \\
 & \mid & \mathsf{model}\ c(\overline{\tau})\ \{\overline{x = e};\}\ \mathsf{in}\ e \\
 & \mid & <c(\overline{\tau})>.x
\end{array}
$$

```
}
```

The generic sum function requires more than just addition; it also requires a zero element of the appropriate type. Again, mathematicians have a name for this concept: a Monoid, which is a Semigroup with an identity element. In generic programming terminology, we say that Monoid is a *refinement* of Semigroup and define Monoid in F^{G} accordingly.

```
concept Monoid(t) {
  refines Semigroup(t);
  identity_elt : t;
}
```

To completely reflect the mathematical definition of a monoid, the identity_elt must satisfy the following axioms for any object x of type t. Unfortunately, expressing this requirement is outside the scope of the F^{G} type system.

```
binary_op(identity_elt, x) = x = binary_op(x, identity_elt)
```

A particular type, such as int, is said to *model* a concept if it satisfies all of the requirements in the concept. In F^{G}, an explicit declaration is used to introduce a model of a concept (corresponding to an instance declaration in Haskell). The following declares int to be a model of Semigroup and Monoid, using integer addition for the binary operation and 0 for the identity element. The type system checks the body of the model against the concept definition to ensure all required operations are provided and that there are model declarations in scope for each refinement.

```
model Semigroup(int) {
  binary_op = iadd;
}
model Monoid(int) {
  identity_elt = 0;
}
```

A model can be found via the concept name and type, and members of the model can be extracted with the dot operator. For example, the following would return the iadd function.

```
<Monoid(int)>.binary_op
```

With the Monoid concept defined, we are ready to write a generic sum function. Since the function has been generalized to work with any type that has an associative binary operation with an identity element (no longer necessarily addition), a more appropriate name for this function is accumulate. As in System F, type parameterization in F^G is provided by the Λ expression. However, F^G adds a where clause to the Λ expression for listing requirements on the type parameters.

```
let accumulate = (Λ t where t models Monoid. /*body*/)
```

The concepts, models, and where clauses collaborate to provide a mechanism for implicitly passing operations into a generic function. As in System F, a generic function is instantiated by providing type arguments for each type parameter.

```
accumulate[int]
```

In System F, instantiation substitutes int for t in the body of the Λ expression. In F^G, instantiation also involves the following steps:

1. int is substituted for t in the where clause.
2. For each required model in the where clause, the lexical scope of the instantiation is searched for a matching model declaration.
3. The models are implicitly passed into the generic function.

Now consider the body of the accumulate function. The model requirements in the where clause serve as proxies for actual model declarations. Thus, the body of accumulate is type-checked as if there were a model declaration model Monoid(t) in the enclosing scope. The <> notation is used inside the body to access the binary operator and identity element of the Monoid.

```
let accumulate =
  (Λ t where t models Monoid.
    fix (λ accum : fn(list t)→ t.
          λls : list t.
          let binary_op = <Monoid(t)>.binary_op in
          let identity_elt = <Monoid(t)>.identity_elt in
          if null[t](ls) then identity_elt
          else binary_op(car[t](ls), accum(cdr[t](ls)))))
```

It would be more convenient to write binary_op instead of the explicit member access: <Monoid(t)>.binary_op. However, such a statement would be ambiguous without the incorporation of overloading into the language. For example, suppose that a generic function has two type parameters, s and t, and requires each to be a Monoid. Then a call

Figure 7: Generic Accumulate

```
concept Semigroup(t) {
  binary_op : fn(t,t)→t;
} in
concept Monoid(t) {
  refines Semigroup(t);
  identity_elt : t;
} in

let accumulate =
  (Λ t where t models Monoid.
    fix (λ accum : fn(list t)→ t.
          λls : list t.
          let binary_op = <Monoid(t)>.binary_op in
          let identity_elt = <Monoid(t)>.identity_elt in
          if null[t](ls) then identity_elt
          else binary_op(car[t](ls), accum(cdr[t](ls))))) in

model Semigroup(int) {
  binary_op = iadd;
} in
model Monoid(int) {
  identity_elt = 0;
} in

let ls = cons[int](1, cons[int](2, nil[int])) in
accumulate[int](ls)
```

to binary_op might refer to either <Monoid(s)>.binary_op or <Monoid(t)>.binary_op. The addition of function overloading to F^G is future work.

The complete program for this example is in Figure 7. As with System F, F^G is an expression-oriented programming language. The concept and models declarations are like let; they extend the lexical environment for the enclosed expression (after the in).

The lexical scoping of models declarations is an important feature of F^G, and one that distinguishes it from Haskell. We illustrate lexical scoping of models with an example. The mathematical definition of monoid is quite general—it only requires a binary operation and an identity element with respect to that operation. That operation need not be addition and the identity element need not be zero. The integers with multiplication as the binary operation and unity as the identity element also form a monoid. This Monoid is expressed in F^G as follows.

```
model Semigroup(int) {
  binary_op = imult;
```

Figure 8: Intentionally Overlapping Models

```
let sum =
  model Semigroup(int) {
    binary_op = iadd;
  } in
  model Monoid(int) {
    identity_elt = 0;
  } in accumulate[int] in

let product =
  model Semigroup(int) {
    binary_op = imult;
  } in
  model Monoid(int) {
    identity_elt = 1;
  } in accumulate[int] in

let ls = cons[int](1, cons[int](2, nil[int])) in
(sum(ls), product(ls))
```

```
}
model Monoid(int) {
  identity_elt = 1;
}
```

Borrowing from Haskell terminology, this second definition of Semigroup and Monoid creates overlapping model declarations, since there are now two models declarations for Semigroup(int) and Monoid(int). Overlapping model declarations are problematic since they introduce ambiguity: when accumulate is instantiated, which model (with its corresponding binary operation and identity element) should be used?

In F^G, overlapping models declarations can coexist so long as they appear in separate lexical scopes. In Figure 8 we create sum and product functions by instantiating accumulate in the presence of different models declarations. This example would not type check in Haskell even if the two instance declarations were to be placed in different modules, because instance declarations implicitly leak out of a module when anything in the module is used by another module.

6 Informal Description of the Translation

We describe a translation from F^G to System F that is similar to the type-directed translation of Haskell type classes presented in [15]. The translation described here is intentionally naive, since its main purpose is to communicate the semantics of F^G. There is extensive literature on techniques for producing more optimized results [2, 18]. The main idea behind the translation is to represent models with dictionaries that map member names to values, and to pass these dictionaries as extra arguments to generic functions. Here tuples represent dictionaries, so the model declarations for Semigroup(int) and Monoid(int) translate to a pair of let expressions that bind freshly generated dictionary names to the tuples for the models.

```
model Semigroup(int) {
  binary_op = iadd;
} in
model Monoid(int) {
  identity_elt = 0;
} in /* rest */
⟹
let Semigroup_61 = (iadd) in
let Monoid_67 = (Semigroup_61,0) in /* rest */
```

The accumulate function is translated by removing the where clause and wrapping the body in a λ expression with a parameter for each model requirement in the where clause.

```
let accumulate = (Λ t where t models Monoid. /*body*/)
⟹
let accumulate =
  (Λ t. (λ Monoid_18:(fn(t,t)→t)*t. /* body */)
```

The accumulate function is now curried, first taking a dictionary argument and then taking the normal arguments.

```
accumulate[int](ls)
⟹
((accumulate[int])(Monoid_67))(ls)
```

In the body of accumulate there are model member accesses. These are translated into tuple member accesses.

```
let binary_op = <Monoid(t)>.binary_op in
let identity_elt = <Monoid(t)>.identity_elt in
⟹
let binary_op = (nth (nth Monoid_18 0) 0) in
let identity_elt = (nth Monoid_18 1) in
```

<Monoid(t)>.binary_op could also have been written <Semigroup(t)>.binary_op, with the same result. As mentioned before, the where clause introduces proxy model declarations for each type requirement. In addition, the where clause introduces proxies for all refinements. This enables the use of Semigroup, since Monoid refines Semigroup. Note that only a single dictionary is passed into accumulate, and that the dictionary for Semigroup is found inside the dictionary for Monoid, as shown in Figure 9. During translation a table is used to map a concept and type, such as Semigroup(t), to a dictionary name and a dictionary path. In this example, the dictionary name for Semigroup(t)

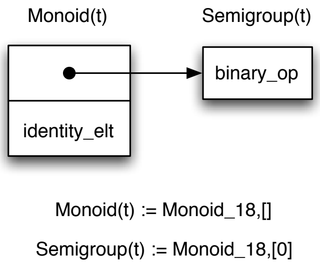

Figure 9: Dictionary representations for the models Monoid(t) and Semigroup(t). Also shown is the model environment, which maps a model to its dictionary name and dictionary path.

is Monoid_18, and the dictionary path is $[0]$, since the Semigroup dictionary is in the first slot of the Monoid dictionary.

The translation for the entire accumulate example is show in Figure 10.

7 Formal Semantics of F^G

This section describes the Isabelle/Isar formalization of a semantics for F^G via a type-directed translation to System F. The types and terms of F^G are represented with the following data types.

datatype *tyg* = *VarTG var* (‘-) | *ArrowG tyg list tyg* (*fn* - $\rightarrow$ -)
| *AllG var list* (*var* $\times$ (*tyg list*)) *list tyg* ($\forall$ - *where* -. -)
| *BoolG* | *IntG*
types *where-clause* = (*var* $\times$ (*tyg list*)) *list*
types *refinements* = (*var* $\times$ (*tyg list*)) *list*
datatype *trmg* = *VarG var* (‘-) | *AppG trmg trmg list* (**infixl** ·)
| *LamG var list tyg list trmg* (λ -:-. -) | *LetTrmG var trmg trmg* (*let* - := - *in* -)
| *ForallG var list where-clause trmg* (Λ - *where* -. -) | *InstG trmg tyg list* (-[-])
| *BooleanG bool* | *IntegerG int*
| *ConceptG var var list refinements var list tyg list trmg*
(*concept* - (-) { *refines* -; - : -; } *in* -)
| *ModelG var tyg list var list trmg list trmg* (*model* - (-) { - = -; } *in* -)
| *ModelMemG var tyg list var* ($\langle$-(-)$\rangle$.-)

7.1 Type Substitution

The definition of simultaneous substitution on types in F^G is given below, again using Isabelle's **recdef** facility. The following lemmas are needed to prove termination. The presence of the where clause in type applications slightly complicates the proof.

lemma *tyg-list-tc1*: $\sigma \in set\ \sigma s \longrightarrow size\ \sigma < Suc\ (tyg\text{-}list\text{-}size1\ \sigma s + size\ \tau)$

Figure 10: Translation of the Accumulate Example

```
let accumulate =
  (Λ t.
   λMonoid_18:(fn(t,t)→t)∗t.
   fix (λ accum:(fn(list t)→t).
       (λ ls:list t.
        let binary_op = (nth (nth Monoid_18 0) 0) in
        let identity_elt = (nth Monoid_18 1) in
        if null[t](ls) then identity_elt
        else binary_op(car[t](ls),accum(cdr[t](ls)))))) in

let Semigroup_61 = (iadd) in
let Monoid_67 = (Semigroup_61,0) in

let ls = cons[int](1,cons[int](2,nil[int])) in
(accumulate[int](Monoid_67))(ls)
```

by (*induct* σs *rule*: *list*.*induct*, *auto*)

lemma *tyg-list-size2-elt*: $\sigma \in set\ \sigma s \longrightarrow size\ \sigma < Suc\ (tyg\text{-}list\text{-}size2\ \sigma s)$
by (*induct* σs *rule*: *list*.*induct*, *auto*)

lemma *where-list-tc*: $[\![\ \sigma \in set\ \sigma s;\ (c,\ \sigma s) \in set\ ws\ ]\!]$
$\Longrightarrow size\ \sigma < Suc\ (nat\text{-}tyg\text{-}list\text{-}x\text{-}list\text{-}size\ ws + size\ \tau)$
apply (*induct ws rule*: *list*.*induct*) **apply** *simp*
proof *clarify*
fix *a b list*
assume *IH*: $[\![\ \sigma \in set\ \sigma s;\ (c,\ \sigma s) \in set\ list\ ]\!]$
$\Longrightarrow size\ \sigma < Suc\ (nat\text{-}tyg\text{-}list\text{-}x\text{-}list\text{-}size\ list + size\ \tau)$
and *sss*: $\sigma \in set\ \sigma s$ **and** *css*: $(c,\sigma s) \in set\ ((a,b)\#list)$
show $size\ \sigma < Suc\ (nat\text{-}tyg\text{-}list\text{-}x\text{-}list\text{-}size\ ((a,\ b)\ \#\ list) + size\ \tau)$
proof (*cases* $(c,\sigma s) = (a,b)$)
assume *eq*: $(c,\sigma s) = (a,b)$
from *sss* **have** $size\ \sigma < Suc\ (tyg\text{-}list\text{-}size2\ \sigma s)$ **by** (*simp add*: *tyg-list-size2-elt*)
with *eq* **show** *?thesis* **by** *simp*
next assume *neq*: $(c,\sigma s) \neq (a,b)$
from *neq css* **have** *css2*: $(c,\sigma s) \in set\ list$ **by** *auto*
from *sss css2 IH* **show** *?thesis* **by** *simp*
qed
qed

consts *sub-tyg* :: $(var\ list \times tyg\ list \times tyg) \Rightarrow tyg$
recdef *sub-tyg measure* $(\lambda\ p.\ size\ (snd\ (snd\ p)))$
$sub\text{-}tyg(ts,\ \tau s,\ \text{‘}t) = (case\ (lookup\ t\ ts\ \tau s\ 0)\ of\ None \Rightarrow \text{‘}t \mid Some\ (\tau,i) \Rightarrow \tau)$

$sub\text{-}tyg(ts,\ \tau s,\ fn\ \sigma s \rightarrow \tau) = fn\ (map\ (\lambda\ \sigma.\ sub\text{-}tyg(ts,\tau s,\sigma))\ \sigma s) \rightarrow sub\text{-}tyg(ts,\tau s,\tau)$
$sub\text{-}tyg(ts,\ \tau s,\ \forall\ ss\ where\ ws.\ \tau) =$
$(\forall\ ss\ where\ (map\ (\lambda\ w.\ (fst\ w,\ map\ (\lambda\ \sigma.\ sub\text{-}tyg(ts,\tau s,\sigma))\ (snd\ w)))\ ws).$
$sub\text{-}tyg(ts,\tau s,\tau))$
$sub\text{-}tyg(ts,\ \tau s,\ BoolG) = BoolG$
$sub\text{-}tyg(ts,\ \tau s,\ IntG) = IntG$
(**hints** *recdef-simp*: *tyg-list-tc1 where-list-tc*)

The following notation is reused for substitution on F^G types and lists of types. New notation is introduced for applying a substitution to the requirements in a where clause.

$[ts{\mapsto}\tau s]\tau \equiv sub\text{-}tyg\ (ts,\ \tau s,\ \tau)$
$\{ts{\mapsto}\tau s\}\sigma s \equiv map\ (\lambda\sigma.\ sub\text{-}tyg\ (ts,\ \tau s,\ \sigma))\ \sigma s$
$\{\!|ts{\mapsto}\tau s|\!\}ws \equiv map\ (\lambda w.\ (fst\ w,\ map\ (\lambda\sigma.\ sub\text{-}tyg\ (ts,\ \tau s,\ \sigma))\ (snd\ w)))\ ws$

The list nth function commutes with substitution, and the length of a list of types is invariant under substitution.

lemma *substg-nth*: $\forall\ i\ \tau\ ts\ \sigma s.\ (\tau s{::}tyg\ list)!i = (\tau{::}tyg) \wedge i < length\ \tau s$
$\longrightarrow (\{ts{\mapsto}\sigma s\}\tau s)!i = [ts{\mapsto}\sigma s]\tau$ **using** *nth-map* **by** *simp*

lemma *substg-length*: $\forall\ ts\ \sigma s.\ length\ (\tau s{::}tyg\ list) = length\ (\{ts{\mapsto}\sigma s\}\tau s)$
by (*induct* τs *rule*: *list.induct*, *auto*)

7.2 Type Equality

Type equality for F^G, shown in Figure 11, is nearly the same as that for F. The difference is that there is a new judgment $T \models_r ws = ws'$ for comparing two where clauses.

7.3 Concept Environments and Translation of Types

The typing context for F^G includes information about concepts and models. The concept environment is a set that maps concept names to the following record of information.

record *concept-info* =
params :: *var list*
rfn :: *refinements*
mem-nms ::*var list*
mem-tys :: *tyg list*
types *Cenv* = (*var* × *concept-info*) *set*

Since type annotations appear in the syntax of System F and F^G our translation must also convert types. The main goal of the type translation is to remove the where clause associated with ∀ ’s and replace it with a function type whose parameters are the types of the dictionaries. The judgment $C \vdash \tau \rightsquigarrow \tau$ translates an F^G type to an F type in the context of concept environment C. This judgment also plays the role of defining well-formed F^G types (just ignore the parts after the $\rightsquigarrow$). The judgment $C \models \tau s \rightsquigarrow \tau s'$

Figure 11: Equality of types in F^G up to the renaming of bound type variables.

$$T \vdash \text{`}s = \text{`}T\,s \;\; (\text{FG-EQV}) \qquad \frac{T \models \tau s = \tau s' \qquad T \vdash \tau = \tau'}{T \vdash \mathit{fn}\ \tau s \to \tau = \mathit{fn}\ \tau s' \to \tau'} (\text{FG-EQF})$$

$$\frac{\mathit{extend}\ ts\ ts'\ T \vdash \tau = \tau' \qquad \mathit{extend}\ ts\ ts'\ T \models_r ws = ws'}{T \vdash \forall\ ts\ \mathit{where}\ ws.\ \tau = \forall\ ts'\ \mathit{where}\ ws'.\ \tau'} (\text{FG-EQA})$$

$$T \vdash \mathit{BoolG} = \mathit{BoolG} \;\; (\text{FG-EQB}) \qquad T \vdash \mathit{IntG} = \mathit{IntG} \;\; (\text{FG-EQI})$$

$$T \models [] = [] \;\; (\text{FG-EQN}) \qquad \frac{T \vdash \tau = \tau' \qquad T \models \tau s = \tau s'}{T \models \tau \cdot \tau s = \tau' \cdot \tau s'} \;\; (\text{FG-EQC})$$

$$T \models_r [] = [] \;\; (\text{FG-EQRN}) \qquad \frac{T \models \varrho s = \varrho s' \qquad T \models_r rs = rs'}{T \models_r (c,\ \varrho s) \cdot rs = (c,\ \varrho s') \cdot rs'} \;\; (\text{FG-EQRC})$$

Figure 12: The translation of types from F^G to F. The judgment for well-formed types of F^G can be obtain by ignoring the parts after $\rightsquigarrow$.

$$C \vdash \text{`}t \rightsquigarrow \text{`}t\ (\text{TRANS-VAR})$$

$$\frac{C \models \tau s \rightsquigarrow \tau s' \qquad C \vdash \tau \rightsquigarrow \tau'}{C \vdash \mathit{fn}\ \tau s \to \tau \rightsquigarrow \mathit{fn}\ \tau s' \to \tau'} \;\; (\text{TRANS-FUN})$$

$$\frac{C \models_d ws \rightsquigarrow \delta s \qquad C \vdash \tau \rightsquigarrow \tau' \qquad \mathit{distinct}\ ts}{C \vdash \forall\ ts\ \mathit{where}\ ws.\ \tau \rightsquigarrow \forall\ ts.\ \mathit{fn}\ \delta s \to \tau'} \;\; (\text{TRANS-ALL})$$

$$C \vdash \mathit{BoolG} \rightsquigarrow \mathit{BoolT}\ (\text{TRANS-BOOL}) \qquad C \vdash \mathit{IntG} \rightsquigarrow \mathit{IntT}\ (\text{TRANS-INT})$$

$$C \models [] \rightsquigarrow []\ (\text{TRANS-NIL}) \qquad \frac{C \vdash \tau \rightsquigarrow \tau' \qquad C \models \tau s \rightsquigarrow \tau s'}{C \models \tau \cdot \tau s \rightsquigarrow \tau' \cdot \tau s'} \;\; (\text{TRANS-CONS})$$

$$\frac{\begin{array}{c}(c,\ ci) \in C \\ C \models \tau s \rightsquigarrow \tau s' \qquad C \models_d \mathit{rfn}\ ci \rightsquigarrow \delta s \qquad C \models \mathit{mem\text{-}tys}\ ci \rightsquigarrow \sigma s \qquad |\tau s| = |\mathit{params}\ ci|\end{array}}{C \vdash_d c\ \tau s \rightsquigarrow [\mathit{params}\ ci \mapsto \tau s'](\langle \delta s\ @\ \sigma s \rangle)} \;\; (\text{R-D})$$

$$C \models_d [] \rightsquigarrow []\ (\text{RS-DS-NIL})$$

$$\frac{C \vdash_d c\ \tau s \rightsquigarrow \delta \qquad C \models_d rs \rightsquigarrow \delta s}{C \models_d (c,\ \tau s) \cdot rs \rightsquigarrow \delta \cdot \delta s} \;\; (\text{RS-DS-CONS})$$

translates a list of types. The judgment $C \vdash_d c\ \varrho s \rightsquigarrow \tau$ specifies the construction of a dictionary type τ from a concept c instantiated with type arguments ϱs. The judgment $C \models_d rs \rightsquigarrow \tau s$ finds dictionary types for each requirement in a where clause, or for a list of refinements in a concept definition. Figure 12 presents the definitions of these judgments.

Adding entries to the concept environment does not affect type and dictionary translation. This is proved by a straightforward induction on the translation judgments.

lemma *grow-env-pres-trans*:
$(C \vdash \tau \rightsquigarrow \tau' \longrightarrow (\forall\ C'.\ C \subseteq C' \longrightarrow C' \vdash \tau \rightsquigarrow \tau'))$
$\wedge\ (C \models \tau s \rightsquigarrow \tau s' \longrightarrow (\forall\ C'.\ C \subseteq C' \longrightarrow C' \models \tau s \rightsquigarrow \tau s'))$
$\wedge\ (C \vdash_d c\ \varrho s \rightsquigarrow \tau' \longrightarrow (\forall\ C'.\ C \subseteq C' \longrightarrow C' \vdash_d c\ \varrho s \rightsquigarrow \tau'))$
$\wedge\ (C \models_d rs \rightsquigarrow \tau s' \longrightarrow (\forall\ C'.\ C \subseteq C' \longrightarrow C' \models_d rs \rightsquigarrow \tau s'))$
apply (*induct rule*: *trans-ty-trans-tys-req-dict-reqs-dicts.induct*)
apply *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp*
apply *simp* **prefer** *2* **apply** *simp* **prefer** *2* **apply** *simp*
proof *clarify*
fix $C\ \delta s\ \sigma s\ \tau s\ \tau s'\ c\ ci\ C'$
assume *cC*: $(c, ci) \in C$ **and** *IH1*: $\forall C'.\ C \subseteq C' \longrightarrow C' \models \tau s \rightsquigarrow \tau s'$
and *IH2*: $\forall C'.\ C \subseteq C' \longrightarrow C' \models_d rfn\ ci \rightsquigarrow \delta s$
and *IH3*: $\forall C'.\ C \subseteq C' \longrightarrow C' \models mem\text{-}tys\ ci \rightsquigarrow \sigma s$
and *L*: $length\ \tau s = length\ (params\ ci)$ **and** *CCp*: $C \subseteq C'$
from *CCp cC* **have** *cCp*: $(c,ci) \in C'$ **by** *auto*
from *cCp CCp IH1 IH2 IH3 L r-d* **show** $C' \vdash_d c\ \tau s \rightsquigarrow [params\ ci \mapsto \tau s'](\langle \delta s\ @\ \sigma s \rangle)$ **by** *simp*
qed

7.4 Model Environments

The model environment contains information about the model declarations that are in scope and plays an important role in the translation from F^G to F. Each model will be translated to a dictionary (represented with a tuple) containing member operations of the model and nested tuples for each refined concept. Each model declaration is translated to a let expression that binds the tuple-creation expression to a fresh variable that will serve as the name of the dictionary.

types $model\text{-}info = var \times tyg\ list \times var \times (nat\ list)$
types $Menv = model\text{-}info\ set$

The model environment stores, for each model, the name of the concept being modeled, the type arguments for the type parameters of the concept, a dictionary name, and a sequence of natural numbers. This sequence gives the path from the top level of the dictionary down to the sub-dictionary for the model. In the typing rule for type abstraction, models are added to the model environment for each requirement in the where clause. In addition, models for all inherited concepts are added to the model environment. The paths in the model environment for these "super" models will point to the appropriate place in the dictionary of the "derived" model that was required in

Figure 13: The addition of models to the environment according to the requirements in a where clause.

$$\frac{(c, ci) \in C \qquad \neg\ \mathit{model\text{-}defined}\ c\ \tau s\ M \qquad M' = \{(c, \tau s, d, ns)\} \cup M \qquad C \models_\flat |\mathit{rfn}\ ci|\ \{\!|\mathit{params}\ ci \mapsto \tau s|\!\}\mathit{rfn}\ ci\ d\ ns\ M' \Rightarrow M''}{C \vdash_\flat c\ \tau s\ d\ ns\ M \Rightarrow M''}\ (\text{FLAT-M-I})$$

$$C \models_\flat 0\ rs\ d\ ns\ M \Rightarrow M\ (\text{FLAT-MS-ZERO})$$

$$\frac{rs_{[i]} = (c', \tau s') \qquad C \vdash_\flat c'\ \tau s'\ d\ ns\ @\ [i]\ M \Rightarrow M' \qquad C \models_\flat i\ rs\ d\ ns\ M' \Rightarrow M''}{C \models_\flat \mathit{Suc}\ i\ rs\ d\ ns\ M \Rightarrow M''}\ (\text{FLAT-MS-SUC})$$

$$C \vdash []\ []\ M \Rightarrow M\ (\text{ADD-MODELS-NIL})$$

$$\frac{C \vdash_\flat c\ \varrho s\ d\ []\ M \Rightarrow M' \qquad C \vdash ws\ ds\ M' \Rightarrow M''}{C \vdash (c, \varrho s)\cdot ws\ d\cdot ds\ M \Rightarrow M''}\ (\text{ADD-MODELS-CONS})$$

the where clause. The addition of models to the environment is formalized with the three judgments defined in Figure 13.

The judgment $C \vdash ws\ ds\ M \Rightarrow M'$ adds models to model environment M for the where clause ws, resulting in M'. The judgment $C \vdash_\flat c\ \tau s\ d\ ns\ M \Rightarrow M'$ processes a single requirement and $C \models_\flat i\ rs\ d\ ns\ M \Rightarrow M'$ is for processing refinements. It would have been preferable to encode these judgments as functions, but they are not primitive recursive, and Isabelle does not support general recursive functions that are mutually recursive. The *model-defined* function used in Figure 13 is defined as follows.

$\mathit{model\text{-}defined}\ c\ \tau s\ M \equiv \exists\, dns.\ (c, \tau s, dns) \in M$

7.5 Model Member Lookup and Access

The translation of model member access expressions, such as <Monoid(s)>.binary_op, requires that we find the type for binary_op and the path to binary_op through the dictionary. The judgments in Figure 14 map a member name, concept, and type arguments to the type of the member and its dictionary path.

In the translation of a model member access expression, a series of tuple access expressions is produced. The access follows a specified path through the dictionary (as in Figure 9), and is accomplished by the *mk-nth* function.

consts
mk-nth :: [*trm*, *nat list*] ⇒ *trm*
primrec

Figure 14: Look up the member of a model and return the type of the member and the dictionary path to the member.

$$\frac{(c, ci) \in C \qquad \textit{lookup } x \ (\textit{mem-nms } ci) \ (\textit{mem-tys } ci) \ 0 = \textit{Some } (\tau, i)}{C \vdash^{\flat} x \ c \ \tau s \ ns \Rightarrow [\textit{params } ci \mapsto \tau s]\tau \ ns \ @ \ [|\textit{rfn } ci| + i]} (\text{LM-M})$$

$$\frac{\begin{array}{c}(c, ci) \in C \\ \textit{lookup } x \ (\textit{mem-nms } ci) \ (\textit{mem-tys } ci) \ 0 = \textit{None} \qquad C \models^{\flat} x \ |\textit{rfn } ci| \ c \ \tau s \ ns \Rightarrow \tau \ ns'\end{array}}{C \vdash^{\flat} x \ c \ \tau s \ ns \Rightarrow \tau \ ns'} (\text{LM-R})$$

$$\frac{(c, ci) \in C \qquad (\textit{rfn } ci)_{[i]} = (c', \tau s') \qquad C \vdash^{\flat} x \ c' \ \{\textit{params } ci \mapsto \tau s\}\tau s' \ ns \ @ \ [i] \Rightarrow \tau \ ns'}{C \models^{\flat} x \ \textit{Suc } i \ c \ \tau s \ ns \Rightarrow \tau \ ns'} (\text{LM-RS1})$$

$$\frac{C \models^{\flat} x \ i \ c \ \tau s \ ns \Rightarrow \tau \ ns'}{C \models^{\flat} x \ \textit{Suc } i \ c \ \tau s \ ns \Rightarrow \tau \ ns'} (\text{LM-RS2})$$

mk-nth-nil: $\textit{mk-nth } d \ [] = d$
mk-nth-cons: $\textit{mk-nth } d \ (n\#ns) = \textit{mk-nth } (\textit{Nth } d \ n) \ ns$

In the translation of type application expressions, the type abstraction, which has been translated into a normal function, is applied to the dictionaries that satisfy its where clause. Since the dictionaries may be nested inside the dictionary of a more refined model, a series of tuple accesses is produced to obtain the right dictionary, again using *mk-nth*. The *mk-nths* function processes a list of dictionaries and paths, invoking *mk-nth* for each dictionary and path.

consts
$\textit{mk-nths} :: [\textit{nat list}, \textit{nat list list}] \Rightarrow \textit{trm list}$
primrec
$\textit{mk-nths} \ [] \ \textit{nns} = []$
$\textit{mk-nths} \ (d\#ds) \ \textit{nss} = (\textit{case nss of } [] \Rightarrow [] \mid (ns\#nss) \Rightarrow (\textit{mk-nth } (`d) \ ns)\#(\textit{mk-nths ds nss}))$

7.6 Translation from F^G to F

The rules defining the translation from F^G to F are presented in Figure 15. The type system for F^G can be obtained from the translation by ignoring what appears after the $\rightsquigarrow$. As mentioned before, the typing environment includes a concept and model environment in addition to the usual type assignments for variables, which are bundled into the following record.

types $\textit{TGenv} = (\textit{var} \times \textit{tyg}) \ \textit{set}$
record $\textit{FGenv} =$
$\textit{tyvars} :: \textit{var set}$

vars :: *TGenv*
concepts :: *Cenv*
models :: *Menv*

The following convenience functions are for manipulating the environment.

$\Gamma, xs{:}\tau s \equiv \Gamma(\!|\ vars := (vars\ \Gamma), xs{:}\tau s |\!)$
$\Gamma, concept\ c\ ci \equiv \Gamma(\!|\ concepts := insert\ (c,ci)\ (concepts\ \Gamma) |\!)$
$\Gamma, model\ mi \equiv \Gamma(\!|\ models := insert\ mi\ (models\ \Gamma) |\!)$

The typing rule for concept declarations requires that the concept being declared must not appear in the type of the body. The following formalizes what it means for a concept name to appear in a type.

$$\frac{c\ occurs\ in\ types\ \tau s \lor c\ occurs\ in\ type\ \tau}{c\ occurs\ in\ type\ fn\ \tau s \to \tau} \qquad \frac{c\ occurs\ in\ ws \lor c\ occurs\ in\ type\ \tau}{c\ occurs\ in\ type\ \forall\ ts\ where\ ws.\ \tau}$$

$$\frac{c\ occurs\ in\ type\ \tau \lor c\ occurs\ in\ types\ \tau s}{c\ occurs\ in\ types\ \tau{\cdot}\tau s}$$

$$c\ occurs\ in\ (c,\ \tau s){\cdot}ws \qquad \frac{c\ occurs\ in\ ws}{c\ occurs\ in\ (c',\ \tau s){\cdot}ws}$$

As in System F, the rule for type abstraction refers to the free type variables in the environment, which in turn refers to the free type variables in a type. We define the following recursive function to compute the free type variables in a type. The pattern of the recursion is the same as for substitution, so we reuse the termination lemmas.

consts *ftvg* :: *tyg* ⇒ *nat set*
recdef *ftvg measure size*
$ftvg\ (`t) = \{t\}$
$ftvg\ (fn\ \tau s \to \tau) = \bigcup (map\ ftvg\ \tau s) \cup ftvg\ \tau$
$ftvg\ (\forall\ ts\ where\ ws.\ \tau) = (\bigcup (map\ (\lambda\ p.\ \bigcup (map\ ftvg\ (snd\ p)))\ ws) \cup ftvg\ \tau) - set\ ts$
$ftvg\ BoolG = \{\}$
$ftvg\ IntG = \{\}$
(**hints** *recdef-simp*: *tyg-list-tc1 where-list-tc*)
consts *btvg* :: *tyg* ⇒ *nat set*
recdef *btvg measure size*
$btvg\ (`t) = \{\}$
$btvg\ (fn\ \tau s \to \tau) = \bigcup (map\ btvg\ \tau s) \cup btvg\ \tau$
$btvg\ (\forall\ ts\ where\ ws.\ \tau) = (\bigcup (map\ (\lambda\ p.\ \bigcup (map\ btvg\ (snd\ p)))\ ws) \cup btvg\ \tau) \cup set\ ts$
$btvg\ BoolG = \{\}$
$btvg\ IntG = \{\}$
(**hints** *recdef-simp*: *tyg-list-tc1 where-list-tc*)

constdefs *btv-cpt* :: *concept-info* ⇒ *var set*
$btv\text{-}cpt\ c \equiv set\ (params\ c) \cup \bigcup (map\ (\lambda\ p.\ \bigcup (map\ btvg\ (snd\ p)))(rfn\ c)) \cup \bigcup (map\ btvg\ (mem\text{-}tys\ c))$

constdefs *btvc* :: *Cenv* ⇒ *var set*
$btvc\ C \equiv \bigcup \{ V.\ (\exists\ c\ cd.\ (c,cd) \in C$
$\quad \wedge V = set\ (params\ cd) \cup \bigcup (map\ (\lambda\ p.\ \bigcup (map\ btvg\ (snd\ p)))\ (rfn\ cd))$
$\qquad\qquad \cup \bigcup (map\ btvg\ (mem\text{-}tys\ cd)))\ \}$

The free type variables in a typing environment is then defined as follows.

$FTVg\ \Gamma \equiv \bigcup \{V \mid \exists x\ \tau.\ (x, \tau) \in \Gamma \wedge V = ftvg\ \tau\}$

8 The Translation is Sound

The main theorem of this paper is that the translation from F^G to F defined in Figure 15 is sound. That is, the output terms are well-typed in System F. The proof is by induction on the derivation of the translation. There are two extra conditions that are needed for the induction: the concept environment must be "sane" and there must be a System F typing environment that corresponds to the F^G typing environment.

8.1 Concept Environment Sanity Conditions

Figure 16 formalizes the following sanity conditions on the concept environment.

1. Concept definitions are unique.
2. The type parameters for a concept are distinct.
3. All types that appear in a concept definition must be well-formed (and thereby have a corresponding System F type).
4. When a concept refines another concept, the other concept must already be defined.
5. The type variables occuring in the body of a concept are a subset of the type parameters of the concept.

8.2 Environment Correspondence

Figure 17 defines the correspondence between the typing environment for F^G and the typing environment for the translated terms of System F. We write $\Gamma \rightsquigarrow S$ to mean the F^G environment Γ is in correspondence with the System F environment S. The correspondence for normal variables is straightforward. If (x, τ) is in *vars* Γ, then there must be a τ' such that *concepts* $\Gamma \vdash \tau \rightsquigarrow \tau'$ and (x,τ') is in S.

The correspondence for the model environment is more involved. If model $(c,\tau s,d,ns)$ is in *models* Γ and if the path $ns = []$, then the dictionary variable d for that model

Figure 15: Translation from F^{G} to F

$$\frac{\begin{array}{c}\Gamma(\!|\mathit{models} := M,\ \mathit{tyvars} := \mathit{tyvars}\ \Gamma \cup \mathit{set\ ts}|\!) \vdash e : \sigma \rightsquigarrow f \\ \mathit{set\ ts} \cap \mathit{tyvars}\ \Gamma = \emptyset \qquad \mathit{set\ ts} \cap \mathit{FTVg}\ (\mathit{vars}\ \Gamma) = \emptyset \\ \mathit{distinct\ ts} \qquad \mathit{concepts}\ \Gamma \models_d \mathit{ws} \rightsquigarrow \tau s \qquad \mathit{concepts}\ \Gamma \vdash \mathit{ws\ ds\ models}\ \Gamma \Rightarrow M\end{array}}{\Gamma \vdash \Lambda\ \mathit{ts\ where\ ws}.\ e : \forall\ \mathit{ts\ where\ ws}.\ \sigma \rightsquigarrow \Lambda\ \mathit{ts}.\ (\lambda\ \mathit{ds}{:}\tau s.\ f)}\ (\textsc{fg-tabs})$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \forall\ \mathit{ts\ where\ ws}.\ \sigma \rightsquigarrow f \\ |\mathit{ts}| = |\tau s| \qquad \mathit{models}\ \Gamma \models \{\!|\mathit{ts}{\mapsto}\tau s|\!\}\mathit{ws} \rightsquigarrow \mathit{ds},\mathit{nns} \qquad \mathit{concepts}\ \Gamma \models \tau s \rightsquigarrow \tau s'\end{array}}{\Gamma \vdash e[\tau s] : [\mathit{ts}{\mapsto}\tau s]\sigma \rightsquigarrow f[\tau s'] \cdot \mathit{mk\text{-}nths\ ds\ nns}}\ (\textsc{fg-tapp})$$

$$\frac{\begin{array}{c}c \notin \mathit{dom\ concepts}\ \Gamma \qquad \mathit{concepts}\ \Gamma \models_d \mathit{rs} \rightsquigarrow \tau s \qquad \mathit{concepts}\ \Gamma \models \sigma s \rightsquigarrow \sigma s' \\ \Gamma,\mathit{concept}\ c\ \mathit{ci} \vdash e : \tau \rightsquigarrow f \qquad \mathit{ci} = (\!|\mathit{params} = \mathit{ts},\ \mathit{rfn} = \mathit{rs},\ \mathit{mem\text{-}nms} = \mathit{xs},\ \mathit{mem\text{-}tys} = \sigma s|\!) \\ \mathit{distinct\ ts} \qquad |\mathit{xs}| = |\sigma s| \qquad \bigcup\ (\mathit{map}\ (\lambda p.\ \bigcup\ (\mathit{map\ ftvg}\ (\mathit{snd}\ p)))\ \mathit{rs}) \subseteq \mathit{set\ ts} \\ \bigcup\ (\mathit{map\ ftvg}\ \sigma s) \subseteq \mathit{set\ ts} \qquad (c, \tau) \notin \mathit{c\text{-}occurs\text{-}ty}\end{array}}{\Gamma \vdash (\mathit{concept}\ c\ \mathit{ts}\ \{\ \mathit{refines\ rs};\ \mathit{xs} : \sigma s;\ \}\ \mathit{in}\ e) : \tau \rightsquigarrow f}\ (\textsc{fg-cpt})$$

$$\frac{\begin{array}{c}\neg\ \mathit{model\text{-}defined}\ c\ \varrho s\ (\mathit{models}\ \Gamma) \qquad (c, \mathit{ci}) \in \mathit{concepts}\ \Gamma \qquad \mathit{concepts}\ \Gamma \models \varrho s \rightsquigarrow \varrho s' \\ \mathit{xs} = \mathit{mem\text{-}nms\ ci} \qquad \Gamma \models \mathit{es} : \sigma s \rightsquigarrow \mathit{fs} \qquad \sigma s = \{\mathit{params\ ci}{\mapsto}\varrho s\}\mathit{mem\text{-}tys\ ci} \\ \mathit{concepts}\ \Gamma \models_d \mathit{rfn\ ci} \rightsquigarrow \mathit{dts} \qquad \mathit{models}\ \Gamma \models \{\!|\mathit{params\ ci}{\mapsto}\varrho s|\!\}\mathit{rfn\ ci} \rightsquigarrow \mathit{ds},\mathit{ns} \\ \mathit{de} = \langle \mathit{mk\text{-}nths\ ds\ ns}\ @\ \mathit{fs}\rangle \qquad |\mathit{params\ ci}| = |\varrho s| \qquad \Gamma,\mathit{model}\ (c, \varrho s, d, []) \vdash e : \tau \rightsquigarrow f\end{array}}{\Gamma \vdash (\mathit{model}\ c\ \varrho s\ \{\ \mathit{xs} = \mathit{es};\ \}\ \mathit{in}\ e) : \tau \rightsquigarrow \mathit{let}\ d := \mathit{de}\ \mathit{in}\ f}\ (\textsc{fg-mdl})$$

$$\frac{(c, \tau s, d, \mathit{ns}) \in \mathit{models}\ \Gamma \qquad \mathit{concepts}\ \Gamma \vdash^{\flat} x\ c\ \tau s\ \mathit{ns} \Rightarrow \tau\ \mathit{ns}'}{\Gamma \vdash (\langle c\tau s\rangle.x) : \tau \rightsquigarrow \mathit{mk\text{-}nth}\ (\text{‘}d)\ \mathit{ns}'}\ (\textsc{fg-mem})$$

$$\frac{(x, \tau) \in \mathit{vars}\ \Gamma}{\Gamma \vdash \text{‘}x : \tau \rightsquigarrow \text{‘}x}\ (\textsc{fg-var})$$

$$\frac{\Gamma \vdash e : \mathit{fn}\ \sigma s \to \tau \rightsquigarrow f \qquad \Gamma \models \mathit{es} : \sigma s' \rightsquigarrow \mathit{fs} \qquad \mathit{id} \models \sigma s = \sigma s'}{\Gamma \vdash e \cdot \mathit{es} : \tau \rightsquigarrow f \cdot \mathit{fs}}\ (\textsc{fg-app})$$

$$\frac{\Gamma,\mathit{xs}{:}\sigma s \vdash e : \tau \rightsquigarrow f \qquad \mathit{concepts}\ \Gamma \models \sigma s \rightsquigarrow \sigma s' \qquad |\mathit{xs}| = |\sigma s|}{\Gamma \vdash \lambda\ \mathit{xs}{:}\sigma s.\ e : \mathit{fn}\ \sigma s \to \tau \rightsquigarrow \lambda\ \mathit{xs}{:}\sigma s'.\ f}\ (\textsc{fg-abs})$$

$$\Gamma \vdash \mathit{BooleanG}\ b : \mathit{BoolG} \rightsquigarrow \mathit{Boolean}\ b\ (\textsc{fg-bool})$$

$$\Gamma \vdash \mathit{IntegerG}\ i : \mathit{IntG} \rightsquigarrow \mathit{Integer}\ i\ (\textsc{fg-int})$$

$$\Gamma \models [] : [] \rightsquigarrow [] \qquad \frac{\Gamma \vdash e : \tau \rightsquigarrow f \qquad \Gamma \models \mathit{es} : \tau s \rightsquigarrow \mathit{fs}}{\Gamma \models e{\cdot}\mathit{es} : \tau{\cdot}\tau s \rightsquigarrow f{\cdot}\mathit{fs}}$$

$$\Gamma \models [] \rightsquigarrow [],[] \qquad \frac{(c, \tau s, d, \mathit{ns}) \in M \qquad M \models \mathit{ws} \rightsquigarrow \mathit{ds},\mathit{nns}}{M \models (c, \tau s){\cdot}\mathit{ws} \rightsquigarrow d{\cdot}\mathit{ds},\mathit{ns}{\cdot}\mathit{nns}}$$

Figure 16: Concept Environment Sanity

$$\frac{\begin{array}{c} C \models_d \mathit{rfn}\ c \rightsquigarrow \tau s \\ C \models \mathit{mem\text{-}tys}\ c \rightsquigarrow \sigma s \qquad \mathit{distinct}\ (\mathit{params}\ c) \qquad |\mathit{mem\text{-}nms}\ c| = |\mathit{mem\text{-}tys}\ c| \\ \bigcup\ (\mathit{map}\ (\lambda p. \bigcup\ (\mathit{map}\ \mathit{ftvg}\ (\mathit{snd}\ p)))\ (\mathit{rfn}\ c)) \subseteq \mathit{set}\ (\mathit{params}\ c) \\ \bigcup\ (\mathit{map}\ \mathit{ftvg}\ (\mathit{mem\text{-}tys}\ c)) \subseteq \mathit{set}\ (\mathit{params}\ c) \end{array}}{C \vdash c\ \mathit{ok}} (\text{WF-C})$$

$$\emptyset\ \mathit{ok}\ (\text{WF-CS-NIL})$$

$$\frac{n \notin \mathit{dom}\ C \qquad C \vdash c\ \mathit{ok} \qquad C\ \mathit{ok}}{\{(n, c)\} \cup C\ \mathit{ok}} (\text{WF-CS-CONS})$$

must be bound in S to the dictionary type τ for that model. If the path $ns \neq []$, then the dictionary variable d must be bound to some dictionary type τ in S and following the path ns from τ yields the sub-dictionary type τ' for this model. The following is the inductive definition for following a path through a dictionary type.

$$\tau {-}[]{\rightarrow} \tau\ (\text{P-NIL}) \qquad \frac{\tau s_{[n]} {-} ns {\rightarrow} \tau'}{\langle \tau s \rangle {-} n{\cdot}ns {\rightarrow} \tau'} (\text{P-CONS})$$

The environment correspondence is used in four cases of the main theorem. The *fg-var* case uses the correspondence to obtain the System F type for the variable. The *fg-tapp*, *fg-mdl*, and *fg-mem* cases use the correspondence to show that their use of dictionaries is well typed.

8.3 Properties of Sane Concept Environments

This section collects a few properties of sane concept environments.

1. For a given concept name there is at most one concept definition.
2. Adding to the concept environment does not affect concept sanity judgements.
3. All concepts in a sane concept environment are sane.

The first lemma and its corollary prove that each concept has a unique definition.

lemma *unique-concept-mutual*:
$(C \vdash cd\ ok \longrightarrow True) \wedge (C\ ok \longrightarrow (c,cd) \in C \wedge (c,cd') \in C \longrightarrow cd = cd')$
by (*induct rule*: *wf-concept-wf-concept-env.induct*, *auto*)

Figure 17: Correspondence between the F^G typeing environment and the System F environment needed to type the output of the translation. This correspondence is an invariant that is maintained by the translation.

$$\Gamma \rightsquigarrow S \equiv \exists Sv\ Sm.\ concepts\ \Gamma \vdash_v vars\ \Gamma \rightsquigarrow Sv \wedge concepts\ \Gamma \vdash_m models\ \Gamma \rightsquigarrow Sm \wedge tvars\ S = tyvars\ \Gamma \wedge tys\ S = Sm \cup Sv$$

$$C \vdash_v \emptyset \rightsquigarrow \emptyset \ (\textsc{cv-nil})$$

$$\frac{C \vdash_v V \rightsquigarrow S \qquad C \vdash \tau \rightsquigarrow \tau'}{C \vdash_v V{,}x{:}\tau \rightsquigarrow S{,}x{:}\tau'}(\textsc{cv-cons})$$

$$C \vdash_m \emptyset \rightsquigarrow \emptyset \ (\textsc{cm-nil})$$

$$\frac{C \vdash_m M \rightsquigarrow S \qquad C \vdash_d c\ \tau s \rightsquigarrow \tau}{C \vdash_m \{(c,\ \tau s,\ d,\ [])\} \cup M \rightsquigarrow S{,}d{:}\tau}(\textsc{cm-cons})$$

$$\frac{C \vdash_m M \rightsquigarrow S \qquad ns \neq [] \qquad (d,\ \tau) \in S \qquad C \vdash_d c\ \tau s \rightsquigarrow \tau' \qquad \tau{-}ns{\rightarrow}\tau'}{C \vdash_m \{(c,\ \tau s,\ d,\ ns)\} \cup M \rightsquigarrow S}(\textsc{cm-drop})$$

corollary *unique-concept*: $[\![\ C\ ok;\ (c,cd) \in C;\ (c,cd') \in C\ ]\!] \Longrightarrow cd = cd'$
using *unique-concept-mutual* **by** *blast*

The next properties is that "weakening" the environment by adding more concept definition does not affect judgements about a concept definition's sanity.

lemma *grow-env-pres-wf-concepts*: $(C \vdash cd\ ok \longrightarrow$
$(\forall\ C'.\ C \subseteq C' \wedge C'\ ok \longrightarrow C' \vdash cd\ ok)) \wedge (C\ ok \longrightarrow True)$
apply (*induct rule*: *wf-concept-wf-concept-env.induct*)
prefer *2* **apply** *simp* **prefer** *2* **apply** *simp*
proof *clarify*
fix $C\ \sigma s\ \tau s$ **and** *c*::*concept-info* **and** C'
assume *rs*: $C \models_d rfn\ c \rightsquigarrow \tau s$ **and** *ms*: $C \models mem\text{-}tys\ c \rightsquigarrow \sigma s$
and *dp*: *distinct* (*params c*) **and** *len*: *length* (*mem-nms c*) = *length* (*mem-tys c*)
and *rftv*: $\bigcup\ (map\ (\lambda p.\ \bigcup\ (map\ ftvg\ (snd\ p)))\ (rfn\ c)) \subseteq set\ (params\ c)$
and *mftv*: $\bigcup\ (map\ ftvg\ (mem\text{-}tys\ c)) \subseteq set\ (params\ c)$
and *ccp*: $C \subseteq C'$ **and** *cpok*: $C'\ ok$
from *ccp cpok rs* **have** *rsp*: $C' \models_d rfn\ c \rightsquigarrow \tau s$ **using** *grow-env-pres-trans* **by** *blast*
from *ccp cpok ms* **have** *msp*: $C' \models mem\text{-}tys\ c \rightsquigarrow \sigma s$ **using** *grow-env-pres-trans* **by** *blast*
from *rsp msp dp len rftv mftv* **show** $C' \vdash c\ ok$ **using** *wf-c* **by** *blast*
qed

corollary *grow-env-pres-c-ok*: $[\![\ C \vdash cd\ ok;\ C'\ ok;\ C \subseteq C'\ ]\!] \Longrightarrow C' \vdash cd\ ok$
using *grow-env-pres-wf-concepts* **apply** *blast* **done**

The third property is that if a concept is in a sane concept environment, then the concept is sane.

lemma *c-mem-implies-c-ok-mutual*:
$(C \vdash ci\ ok \longrightarrow True) \wedge (C\ ok \longrightarrow (\forall\ c\ ci.\ C\ ok \wedge (c,ci) \in C \longrightarrow C \vdash ci\ ok))$
apply (*induct rule*: *wf-concept-wf-concept-env.induct*)
apply *simp*+ **apply** *clarify* **apply** (*case-tac* (*ca*,*ci*) = (*n*,*c*))
using *grow-env-pres-c-ok* **apply** *blast* **using** *grow-env-pres-c-ok* **by** *blast*

corollary *c-mem-implies-c-ok*: $[\![\ C\ ok;\ (c,ci) \in C\ ]\!] \Longrightarrow C \vdash ci\ ok$
using *c-mem-implies-c-ok-mutual* **by** *blast*

8.4 Properties of the Type Translation

This section establishes several properties of the translation from types in F^G to types in System F.

The inversion lemma for the translation of a concept instantiation to a dictionary type is heavily used. The following lemma is an easier to use variant of that inversion lemma. Instead of a conclusion that gives the existence of a concept definition for concept *c*, the lemma instead includes a premise for the concept definition *cd* which the conclusion gives its results in terms of.

lemma *inv-r-d2*:
assumes *D*: $C \vdash_d c\ \varrho s \rightsquigarrow \tau$ **and** *Cok*: $C\ ok$ **and** *cC*: $(c,cd) \in C$
shows $\exists\ \delta s\ \sigma s\ \tau s'.\ C \models \varrho s \rightsquigarrow \tau s' \wedge C \models_d rfn\ cd \rightsquigarrow \delta s$
$\wedge\ C \models mem\text{-}tys\ cd \rightsquigarrow \sigma s \wedge length\ \varrho s = length\ (params\ cd)$
$\wedge\ \tau = \langle\{params\ cd \mapsto \tau s'\}(\delta s\ @\ \sigma s)\rangle$
proof −
from *D* **obtain** $\delta s\ \sigma s\ \varrho s'\ cd'$ **where** *cpC*: $(c,cd') \in C$ **and** *rs-rsp*: $C \models \varrho s \rightsquigarrow \varrho s'$
and *Ds*: $C \models_d rfn\ cd' \rightsquigarrow \delta s$ **and** *ms-ss*: $C \models mem\text{-}tys\ cd' \rightsquigarrow \sigma s$
and *lrsp*: $length\ \varrho s = length\ (params\ cd')$
and *T*: $\tau = \langle\{params\ cd' \mapsto \varrho s'\}(\delta s @ \sigma s)\rangle$ **by** (*rule inv-r-d*, *auto*)
from *Cok cC cpC* **have** *cd-cdp*: $cd = cd'$ **by** (*rule unique-concept*)
from *cd-cdp* **have** *Ds2*: $C \models_d rfn\ cd \rightsquigarrow \delta s$ **by** *simp*
from *cd-cdp* **have** *ms-ss2*: $C \models mem\text{-}tys\ cd \rightsquigarrow \sigma s$ **by** *simp*
from *cd-cdp lrsp* **have** *lrsp2*: $length\ \varrho s = length\ (params\ cd)$ **by** *simp*
from *cd-cdp T* **have** *T2*: $\tau = \langle\{params\ cd \mapsto \varrho s'\}(\delta s @ \sigma s)\rangle$ **by** *simp*
from *rs-rsp Ds2 ms-ss2 lrsp2 T2* **show** *?thesis* **by** *auto*
qed

The next lemma states that the type translation is a function. The proof is a mutual induction on the four type translation judegements.

lemma *fun-dict-trans-ty*:
$(C \vdash \tau \rightsquigarrow \tau' \longrightarrow C\ ok \longrightarrow (\forall\ \tau''.\ C \vdash \tau \rightsquigarrow \tau'' \longrightarrow \tau' = \tau''))$
$\wedge\ (C \models \tau s \rightsquigarrow \tau s' \longrightarrow C\ ok \longrightarrow (\forall\ \tau s''.\ C \models \tau s \rightsquigarrow \tau s'' \longrightarrow \tau s' = \tau s''))$
$\wedge\ (C \vdash_d c\ \varrho s \rightsquigarrow dt \longrightarrow C\ ok \longrightarrow (\forall\ dt'.\ C \vdash_d c\ \varrho s \rightsquigarrow dt' \longrightarrow dt' = dt))$
$\wedge\ (C \models_d ws \rightsquigarrow dts \longrightarrow C\ ok \longrightarrow (\forall\ dts'.\ C \models_d ws \rightsquigarrow dts' \longrightarrow dts' = dts))$
(**is** $(C \vdash \tau \rightsquigarrow \tau' \longrightarrow ?P1\ C\ \tau\ \tau') \wedge (C \models \tau s \rightsquigarrow \tau s' \longrightarrow ?P2\ C\ \tau s\ \tau s')$
$\wedge\ (C \vdash_d c\ \varrho s \rightsquigarrow dt \longrightarrow ?P3\ C\ c\ \varrho s\ dt) \wedge (C \models_d ws \rightsquigarrow dts \longrightarrow ?P4\ C\ ws\ dts))$

apply (*induct rule*: *trans-ty-trans-tys-req-dict-reqs-dicts.induct*)
apply *clarify* **apply** (*rule inv-trans-var*) **apply** *simp* **apply** *simp*
prefer *3* **apply** *clarify* **apply** (*rule inv-trans-bool*) **apply** *simp* **apply** *simp*
prefer *3* **apply** *clarify* **apply** (*rule inv-trans-int*) **apply** *simp* **apply** *simp*
prefer *3* **apply** *clarify* **apply** (*rule inv-trans-nil*) **apply** *simp* **apply** *simp*
prefer *5* **apply** *clarify* **apply** (*rule inv-rs-ds-nil*) **apply** *simp* **apply** *simp*
proof −
fix $C\ \tau\ \tau'\ \tau s\ \tau s'$ **assume** *?P2* $C\ \tau s\ \tau s'$ **and** *?P1* $C\ \tau\ \tau'$
thus *?P1* $C\ (fn\ \tau s \to \tau)\ (fn\ \tau s' \to \tau')$ **apply** *clarify* **by** (*rule inv-trans-fun*, *auto*)
next
fix $C\ \delta s\ \tau\ \tau'\ ts\ ws$ **assume** *?P4* $C\ ws\ \delta s$ **and** *?P1* $C\ \tau\ \tau'$
thus *?P1* $C\ (\forall\ ts\ where\ ws.\ \tau)\ (\forall\ ts.\ fn\ \delta s \to \tau')$
apply *clarify* **by** (*rule inv-trans-all2*, *auto*)
next
fix $C\ \tau\ \tau'\ \tau s\ \tau s'$ **assume** *?P1* $C\ \tau\ \tau'$ **and** *?P2* $C\ \tau s\ \tau s'$
thus *?P2* $C\ (\tau \# \tau s)\ (\tau' \# \tau s')$ **apply** *clarify* **by** (*rule inv-trans-cons*, *auto*)
next
fix $C\ \delta s\ \sigma s\ \tau s\ \tau s'\ c$ **and** *ci*::*concept-info* **assume** *cC*: $(c,ci) \in C$
and *IH1*: *?P2* $C\ \tau s\ \tau s'$ **and** *IH2*: *?P4* $C\ (rfn\ ci)\ \delta s$ **and** *IH3*: *?P2* $C\ (mem\text{-}tys\ ci)\ \sigma s$
show *?P3* $C\ c\ \tau s\ ([params\ ci \mapsto \tau s'](\langle \delta s\ @\ \sigma s\rangle))$
proof *clarify*
fix dt' **assume** *Cok*: *C ok* **and** *D*: $C \vdash_d c\ \tau s \rightsquigarrow dt'$
from *D Cok cC* **obtain** $\delta s'\ \sigma s'\ \tau s''$
where *ts-tspp*: $C \models \tau s \rightsquigarrow \tau s''$ **and** *r-dsp*: $C \models_d rfn\ ci \rightsquigarrow \delta s'$
and *ms-sp*: $C \models mem\text{-}tys\ ci \rightsquigarrow \sigma s'$
and *dtp*: $dt' = \langle\{params\ ci \mapsto \tau s''\}(\delta s' @ \sigma s')\rangle$ **using** *inv-r-d2* **by** *blast*
from *IH1 Cok ts-tspp* **have** *tseq*: $\tau s' = \tau s''$ **by** *simp*
from *IH2 Cok r-dsp* **have** *dseq*: $\delta s = \delta s'$ **by** *simp*
from *IH3 Cok ms-sp* **have** *mseq*: $\sigma s = \sigma s'$ **by** *simp*
from *dtp tseq dseq mseq* **show** $dt' = [params\ ci \mapsto \tau s'](\langle \delta s\ @\ \sigma s\rangle)$ **by** *simp*
qed
next
fix $C\ \delta\ \delta s\ \tau s\ c\ rs$ **assume** *?P3* $C\ c\ \tau s\ \delta$ **and** *?P4* $C\ rs\ \delta s$
thus *?P4* $C\ ((c,\tau s)\# rs)\ (\delta \# \delta s)$ **apply** *clarify* **by** (*rule inv-rs-ds-cons*, *auto*)
qed

The length of type list is invariant under translation. The number of requirements in where clause is equal the length of the list of dictionary types.

lemma *trans-length*:
$(C \vdash \tau \rightsquigarrow \tau' \longrightarrow True) \wedge (C \models \sigma s \rightsquigarrow \sigma s' \longrightarrow length\ \sigma s = length\ \sigma s')$
$\wedge\ (C \vdash_d c\ \varrho s \rightsquigarrow dt \longrightarrow True) \wedge (C \models_d rs \rightsquigarrow dts \longrightarrow length\ rs = length\ dts)$
by (*induct rule*: *trans-ty-trans-tys-req-dict-reqs-dicts.induct*, *auto*)

corollary *trans-length-tys*: $C \models \sigma s \rightsquigarrow \sigma s' \Longrightarrow length\ \sigma s = length\ \sigma s'$
using *trans-length* **apply** *blast* **done**

corollary *trans-length-r-d*: $C \models_d rs \rightsquigarrow dts \Longrightarrow length\ rs = length\ dts$
using *trans-length* **apply** *blast* **done**

If the list of types σs translates to $\sigma s'$, then the ith element of σs translates to the ith element of $\sigma s'$.

lemma *trans-nth-helper*:
 $(C \vdash \tau \rightsquigarrow \tau' \longrightarrow True) \wedge (C \models \sigma s \rightsquigarrow \sigma s' \longrightarrow (\forall\ i < length\ \sigma s.\ C \vdash \sigma s!i \rightsquigarrow \sigma s'!i))$
 $\wedge (C \vdash_d c\ \varrho s \rightsquigarrow dt \longrightarrow True) \wedge (C \models_d rs \rightsquigarrow dts \longrightarrow True)$
 apply (*induct rule*: *trans-ty-trans-tys-req-dict-reqs-dicts.induct*)
 apply *auto* **apply** (*case-tac i*) **apply** *auto* **done**

corollary *trans-nth*: $[\![\ C \models \sigma s \rightsquigarrow \sigma s';\ i < length\ \sigma s\ ]\!] \Longrightarrow C \vdash \sigma s!i \rightsquigarrow \sigma s'!i$
 using *trans-nth-helper* **by** *blast*

The next few lemmas and definitions build up to the proof that type translation respects substitution. The following fact characterizes the affect of substitution on free variables.

lemma *ftv-subst-ty*: $length\ ts = length\ \sigma s \Longrightarrow ftv\ [ts{\mapsto}\sigma s]\tau \subseteq (ftv\ \tau - set\ ts) \cup \bigcup (map\ ftv\ \sigma s)$

The proof will be a induction on the structure of types, and thus a mutual induction proving the following two statements.

constdefs *ftv-subst-ty* :: $ty \Rightarrow bool$
 $ftv\text{-}subst\text{-}ty\ \tau \equiv$
 $(\forall\ ts\ (\sigma s{::}ty\ list).\ length\ ts = length\ \sigma s$
 $\longrightarrow ftv\ [ts{\mapsto}\sigma s]\tau \subseteq (ftv\ \tau - set\ ts) \cup \bigcup (map\ ftv\ \sigma s))$
constdefs *ftv-subst-tys* :: $ty\ list \Rightarrow bool$
 $ftv\text{-}subst\text{-}tys\ \tau s \equiv (\forall\ ts\ (\sigma s{::}ty\ list).$
 $length\ ts = length\ \sigma s$
 $\longrightarrow \bigcup (map\ ftv\ (sub\text{-}tys\ ts\ \sigma s\ \tau s)) \subseteq (\bigcup (map\ ftv\ \tau s) - set\ ts) \cup \bigcup (map\ ftv\ \sigma s))$

The case for variables is the only interesting case. There are two subcases to consider, when t is substituted, and when it is not.

lemma *ftv-subst-var*: *ftv-subst-ty* (*‘t*)
proof (*simp only*: *ftv-subst-ty-def*, *clarify*)
 fix $ts\ \sigma s\ x$ **assume** *xfv*: $x \in ftv\ [ts{\mapsto}\sigma s]\,‘t$ **and** *xfss*: $x \notin \bigcup (map\ ftv\ \sigma s)$
 and *len*: $length\ ts = length\ \sigma s$
 show $x \in ftv\ (‘t) - set\ ts$
 proof (*cases* $t \in set\ ts$)
 assume *tts*: $t \in set\ ts$
 from *tts len* **obtain** *i* **where** *I*: $i < length\ ts$ **and** *L*: $lookup\ t\ ts\ \sigma s\ 0 = Some\ (\sigma s!i,i)$
 using *lookup-succeeds*[*of t ts σs 0*] **by** *auto*
 hence *st*: $[ts{\mapsto}\sigma s]\,‘t = \sigma s!i$ **by** *simp*
 from *I len* **have** *iss*: $i < length\ (map\ ftv\ \sigma s)$ **using** *length-map* **by** *simp*
 from *iss* **have** $(map\ ftv\ \sigma s)!i \subseteq \bigcup\ (map\ ftv\ \sigma s)$ **using** *union-list-elem-subset* **by** *blast*
 with *st iss* **have** $ftv\ [ts{\mapsto}\sigma s]\,‘t \subseteq \bigcup\ (map\ ftv\ \sigma s)$ **using** *nth-map* **by** *simp*
 with *xfv xfss* **have** *False* **by** *auto* **thus** *?thesis* **by** *simp*
 next
 assume *tts*: $t \notin set\ ts$
 from *tts* **have** $lookup\ t\ ts\ \sigma s\ 0 = None$ **by** (*rule lookup-fails*)

with *xfv tts* **show** *?thesis* **by** *simp*
qed
qed

lemma *ftv-subst-mutual*: *ftv-subst-ty* $\tau \wedge$ *ftv-subst-tys* $\tau s \wedge$ *ftv-subst-tys* τs
apply (*induct rule*: *ty.induct*) **apply** (*rule ftv-subst-var*)
apply (*simp*, *blast*)+ **apply** *simp*+ **apply** *blast* **apply** *simp* **by** (*simp*, *blast*)

corollary *ftv-subst-ty*: $\mathit{length}\ ts = \mathit{length}\ \sigma s$
$\Longrightarrow \mathit{ftv}\ [ts{\mapsto}\sigma s]\tau \subseteq (\mathit{ftv}\ \tau - \mathit{set}\ ts) \cup \bigcup (\mathit{map}\ \mathit{ftv}\ \sigma s)$
using *ftv-subst-mutual* **by** *simp*

corollary *ftv-subst-tys*: $\mathit{length}\ ts = \mathit{length}\ \sigma s$
$\Longrightarrow \bigcup (\mathit{map}\ \mathit{ftv}\ \{ts{\mapsto}\sigma s\}\tau s) \subseteq (\bigcup (\mathit{map}\ \mathit{ftv}\ \tau s) - \mathit{set}\ ts) \cup \bigcup (\mathit{map}\ \mathit{ftv}\ \sigma s)$
using *ftv-subst-mutual* **by** *simp*

corollary *ftv-subst-ty2*:
assumes *ftts*: $\mathit{ftv}\ \tau \subseteq \mathit{set}\ ts$ **and** *len*: $\mathit{length}\ ts = \mathit{length}\ \sigma s$
shows $\mathit{ftv}\ [ts{\mapsto}\sigma s]\tau \subseteq \bigcup (\mathit{map}\ \mathit{ftv}\ \sigma s)$
proof −
from *len* **have** $\mathit{ftv}\ [ts{\mapsto}\sigma s]\tau \subseteq (\mathit{ftv}\ \tau - \mathit{set}\ ts) \cup \bigcup (\mathit{map}\ \mathit{ftv}\ \sigma s)$
by (*rule ftv-subst-ty*)
with *ftts* **show** *?thesis* **by** *auto*
qed

corollary *ftv-subst-tys2*:
assumes *ftts*: $\bigcup (\mathit{map}\ \mathit{ftv}\ \tau s) \subseteq \mathit{set}\ ts$ **and** *len*: $\mathit{length}\ ts = \mathit{length}\ \sigma s$
shows $\bigcup (\mathit{map}\ \mathit{ftv}\ \{ts{\mapsto}\sigma s\}\tau s) \subseteq \bigcup (\mathit{map}\ \mathit{ftv}\ \sigma s)$
proof −
from *len* **have** $\bigcup (\mathit{map}\ \mathit{ftv}\ \{ts{\mapsto}\sigma s\}\tau s) \subseteq (\bigcup (\mathit{map}\ \mathit{ftv}\ \tau s) - \mathit{set}\ ts) \cup \bigcup (\mathit{map}\ \mathit{ftv}\ \sigma s)$
by (*rule ftv-subst-tys*)
with *ftts* **show** *?thesis* **by** *auto*
qed

The translation never adds free variables to a type. This is proved by induction on the translation judgments, with the only interesting case being the case for a requirement in a where clause.

lemma *trans-reduces-ftv*:
$(C \vdash \tau \rightsquigarrow \tau' \longrightarrow C\ \mathit{ok} \longrightarrow \mathit{ftv}\ \tau' \subseteq \mathit{ftvg}\ \tau)$
$\wedge (C \models \tau s \rightsquigarrow \tau s' \longrightarrow C\ \mathit{ok} \longrightarrow \bigcup (\mathit{map}\ \mathit{ftv}\ \tau s') \subseteq \bigcup (\mathit{map}\ \mathit{ftvg}\ \tau s))$
$\wedge (C \vdash_d c\ \varrho s \rightsquigarrow dt \longrightarrow C\ \mathit{ok} \longrightarrow \mathit{ftv}\ dt \subseteq \bigcup (\mathit{map}\ \mathit{ftvg}\ \varrho s))$
$\wedge (C \models_d rs \rightsquigarrow dts \longrightarrow C\ \mathit{ok} \longrightarrow \bigcup (\mathit{map}\ \mathit{ftv}\ dts) \subseteq \bigcup (\mathit{map}\ (\lambda\, p.\ \bigcup (\mathit{map}\ \mathit{ftvg}\ (\mathit{snd}\ p)))\ rs))$
apply (*induct rule*: *trans-ty-trans-tys-req-dict-reqs-dicts.induct*)
apply *simp* **apply** (*simp*, *blast*) **apply** (*simp*,*blast*) **apply** *simp* **apply** *simp* **apply** *simp*
apply (*simp*, *blast*) **prefer** *2* **apply** *simp* **prefer** *2* **apply** (*simp*, *blast*)
proof *clarify*
fix *C* **and** δs::*ty list* **and** $\sigma s\ \tau s\ \tau s'\ c\ ci\ x$
assume *cC*: $(c, ci) \in C$ **and** *ts-tsp*: $C \models \tau s \rightsquigarrow \tau s'$
and *xfds*: $x \in \mathit{ftv}\ [\mathit{params}\ ci{\mapsto}\tau s'](\langle \delta s @ \sigma s \rangle)$

and *IH1*: $\bigcup\ (map\ ftv\ \tau s') \subseteq \bigcup\ (map\ ftvg\ \tau s)$
and *IH2*: $\bigcup (map\ ftv\ \delta s) \subseteq \bigcup (map\ (\lambda p.\ \bigcup\ (map\ ftvg\ (snd\ p)))\ (rfn\ ci))$
and *IH3*: $\bigcup (map\ ftv\ \sigma s) \subseteq \bigcup (map\ ftvg\ (mem\text{-}tys\ ci))$
and *lts*: $length\ \tau s = length\ (params\ ci)$ **and** *Cok*: $C\ ok$
from *Cok cC* **have** *ciok*: $C \vdash ci\ ok$ **by** (*rule c-mem-implies-c-ok*)
from *ciok* **have** *rsps*: $\bigcup\ (map\ (\lambda p.\ \bigcup\ (map\ ftvg\ (snd\ p)))\ (rfn\ ci)) \subseteq set\ (params\ ci)$
by (*rule inv-wf-c*, *simp*)
from *ciok* **have** *msps*: $\bigcup\ (map\ ftvg\ (mem\text{-}tys\ ci)) \subseteq set\ (params\ ci)$ **by** (*rule inv-wf-c*, *simp*)
from *ts-tsp lts* **have** *ltsp*: $length\ (params\ ci) = length\ \tau s'$ **by** (*simp add*: *trans-length*)
from *IH2 rsps* **have** *fdsps*: $\bigcup (map\ ftv\ \delta s) \subseteq set\ (params\ ci)$ **by** *simp*
from *fdsps ltsp* **have**
A: $\bigcup (map\ ftv\ (\{params\ ci \mapsto \tau s'\}\delta s)) \subseteq \bigcup (map\ ftv\ \tau s')$ **by** (*rule ftv-subst-tys2*)
from *IH3 msps* **have** *fssps*: $\bigcup (map\ ftv\ \sigma s) \subseteq set\ (params\ ci)$ **by** *simp*
from *fssps ltsp* **have**
B: $\bigcup (map\ ftv\ (\{params\ ci \mapsto \tau s'\}\sigma s)) \subseteq \bigcup (map\ ftv\ \tau s')$ **by** (*rule ftv-subst-tys2*)
from *A B* **have** $ftv\ [params\ ci \mapsto \tau s'](\langle \delta s @ \sigma s \rangle) \subseteq \bigcup (map\ ftv\ \tau s')$
by (*induct* δs *rule*: *list.induct*, *auto*)
with *IH1 xfds* **show** $x \in \bigcup (map\ ftvg\ \tau s)$ **by** *auto*
qed

Substitution respects type translation That is, if τ translates to τ', then $[ts \mapsto \tau s]\tau$ translates to $[ts \mapsto \tau s']\tau'$, provided that τs translates to $\tau s'$. The proof is by induction on the derivation of the translation. There are two interesting cases, for translating type variables, and the case for translating a concept instantiation in a where clause. This first lemma handles the translation of type variables.

lemma *subst-respects-trans-var*: $(C \vdash (VarTG\ t) \rightsquigarrow (VarT\ t)$
$\longrightarrow (\forall\ ts\ \tau s\ \tau s'.\ distinct\ ts \wedge length\ ts = length\ \tau s \wedge C \models \tau s \rightsquigarrow \tau s'$
$\longrightarrow C \vdash [ts \mapsto \tau s](VarTG\ t) \rightsquigarrow [ts \mapsto \tau s'](VarT\ t)))$
proof (*clarify*)
fix *ts*::*var list* **and** $\tau s\ \tau s'$
assume *D*: *distinct ts* **and** *L*: $length\ ts = length\ \tau s$ **and** *ts-tsp*: $C \models \tau s \rightsquigarrow \tau s'$
show $C \vdash [ts \mapsto \tau s](VarTG\ t) \rightsquigarrow [ts \mapsto \tau s'](VarT\ t)$
proof (*cases* $t \in set\ ts$)
assume *tm*: $t \in set\ ts$
from *tm L* **obtain** *i* **where** *il*: $i < length\ ts$ **and** *tsi*: $ts!i = t$
and *lts*: $lookup\ t\ ts\ \tau s\ 0 = Some\ (\tau s!i,i)$
using *lookup-succeeds*[*of t ts* τs *0*] **by** *auto*
from *ts-tsp* **have** $length\ \tau s = length\ \tau s'$ **by** (*rule trans-length-tys*)
with *L* **have** *L2*: $length\ ts = length\ \tau s'$ **by** *simp*
from *tm L2* **obtain** $i'\ \tau'$ **where**
ipl: $i' < length\ ts$ **and** *tsip*: $ts!i' = t$ **and** *tausip*: $\tau s'!i' = \tau'$
and *ltsp*: $lookup\ t\ ts\ \tau s'\ 0 = Some\ (\tau s'!i',i')$
using *lookup-succeeds*[*of t ts* $\tau s'$ *0*] **by** *auto*
from *D il ipl tsi tsip* **have** *i-ip*: $i = i'$ **using** *distinct-conv-nth* **by** *auto*
note *ts-tsp*
moreover from *L il* **have** $i < length\ \tau s$ **by** *simp*
ultimately have $C \vdash \tau s!i \rightsquigarrow \tau s'!i$ **by** (*rule trans-nth*)
with *lts ltsp tausip i-ip* **show** *?thesis* **by** *auto*
next

assume *tm*: $t \notin$ *set ts*
from *tm* **have** *lookup t ts* τs *0* = *None* **by** (*rule lookup-fails*)
moreover from *tm* **have** *lookup t ts* $\tau s'$ *0* = *None* **by** (*rule lookup-fails*)
ultimately show *?thesis* **by** (*simp add*: *trans-var*)
qed
qed

The following abbreviations are used for the conclusions of the statements that will be proved.

constdefs *srt-ty* :: [*Cenv*,*tyg*,*ty*] $\Rightarrow$ *bool*
srt-ty C τ τ' $\equiv$ ($\forall$ *ts* τs $\tau s'$. $C \models \tau s \rightsquigarrow \tau s' \wedge$ *C ok* $\wedge$ *distinct ts* $\wedge$ *length ts* = *length* τs
$\longrightarrow C \vdash$ *sub-tyg*(*ts*,τs,τ) $\rightsquigarrow$ *sub-ty*(*ts*,$\tau s'$,τ'))
constdefs *srt-tys* :: [*Cenv*,*tyg list*,*ty list*] $\Rightarrow$ *bool*
srt-tys C τs $\tau s'$ $\equiv$ ($\forall$ *ts* σs $\sigma s'$. $C \models \sigma s \rightsquigarrow \sigma s' \wedge$ *C ok* $\wedge$ *distinct ts* $\wedge$ *length ts* = *length* σs
$\longrightarrow C \models$ *sub-tygs ts* σs τs $\rightsquigarrow$ *sub-tys ts* $\sigma s'$ $\tau s'$)
constdefs *srt-dict* :: [*Cenv*, *var*, *tyg list*, *ty*] $\Rightarrow$ *bool*
srt-dict C c ϱs *dt* $\equiv$ ($\forall$ *ts* τs $\tau s'$. ($C \models \tau s \rightsquigarrow \tau s' \wedge$ *C ok* $\wedge$ *distinct ts* $\wedge$ *length ts* = *length* τs
$\longrightarrow C \vdash_d c$ (*sub-tygs ts* τs ϱs) $\rightsquigarrow$ *sub-ty*(*ts*,$\tau s'$,*dt*)))
constdefs *srt-ds* :: [*Cenv*, *where-clause*, *ty list*] $\Rightarrow$ *bool*
srt-ds C rs dts $\equiv$ ($\forall$ *ts* τs $\tau s'$. $C \models \tau s \rightsquigarrow \tau s' \wedge$ *C ok* $\wedge$ *distinct ts* $\wedge$ *length ts* = *length* τs
$\longrightarrow C \models_d$ {|*ts*$\mapsto\tau s$|}*rs* $\rightsquigarrow$ {*ts*$\mapsto\tau s'$}*dts*)

The case for translating a requirement in a where clause is handled by the following lemma.

lemma *subst-respects-trans-dict*:
assumes *cC*: (*c*, *ci*) $\in$ *C* **and** *ts-tsp*: $C \models \tau s \rightsquigarrow \tau s'$ **and** *IH1*: *srt-tys C* τs $\tau s'$
and *Rs*: $C \models_d$ *rfn ci* $\rightsquigarrow \delta s$ **and** *IH2*: *srt-ds C* (*rfn ci*) δs
and *Ms*: $C \models$ *mem-tys ci* $\rightsquigarrow \sigma s$ **and** *IH3*: *srt-tys C* (*mem-tys ci*) σs
and *lts*: *length* τs = *length* (*params ci*)
shows *srt-dict C c* τs [*params ci*$\mapsto\tau s'$]($\langle\delta s$ @ $\sigma s\rangle$)
proof (*simp only*: *srt-dict-def*, *clarify*)
fix *ts*::*var list* **and** τsa::*tyg list* **and** $\tau sa'$::*ty list*
assume *tsa-tsap*: $C \models \tau sa \rightsquigarrow \tau sa'$
and *Cok*: *C ok* **and** *dist*: *distinct ts* **and** *len*: *length ts* = *length* τsa
let *?dt* = [*params ci*$\mapsto\tau s'$]($\langle\delta s$ @ $\sigma s\rangle$)
let *?ts* = {*ts*$\mapsto\tau sa$}τs **and** *?tsp* = {*ts*$\mapsto\tau sa'$}$\tau s'$
let *?ms* = {*ts*$\mapsto\tau sa$}*mem-tys ci* **and** *?ss* = {*ts*$\mapsto\tau sa'$}σs
let *?rs* = {|*ts*$\mapsto\tau sa$|}(*rfn ci*) **and** *?ds* = {*ts*$\mapsto\tau sa'$}δs
note *cC* **moreover from** *tsa-tsap Cok dist len IH1* **have**
ts-tsp: $C \models$ *?ts* $\rightsquigarrow$ *?tsp* **by** *simp*
moreover note *Rs* **and** *Ms*
moreover from *lts* **have** *length* {*ts*$\mapsto\tau sa$}τs = *length* (*params ci*)
using *substg-length* **by** *simp*
ultimately have $C \vdash_d c$ *?ts* $\rightsquigarrow$ [*params ci*$\mapsto$*?tsp*]($\langle\delta s$@$\sigma s\rangle$) **by** (*rule r-d*)
moreover have [*params ci*$\mapsto$*?tsp*]($\langle\delta s$@$\sigma s\rangle$) = [*ts*$\mapsto\tau sa'$]*?dt*
proof –
— We can alpha-convert to change the concept parameters so that they are distinct from *ts* and from the variables in $\tau sa'$.
have *A*: *set* (*params ci*) $\cap$ *set ts* = {} **sorry**

have *B*: *set* (*params ci*) $\cap \bigcup$ (*map otv* $\tau sa'$) = {} **sorry**
have *C*: *set ts* $\cap$ *otv* ($\langle\delta s$ @ $\sigma s\rangle$) = {}
proof −
have *ofb*: *otv* ($\langle\delta s$ @ $\sigma s\rangle$) = *ftv* ($\langle\delta s$ @ $\sigma s\rangle$) $\cup$ *btv* ($\langle\delta s$ @ $\sigma s\rangle$)
using *otv-ftv-btv* **by** *simp*
from *Cok cC* **have** *ciok*: $C \vdash$ *ci ok* **by** (*rule c-mem-implies-c-ok*)
from *ciok* **have** *frsps*: $\bigcup$ (*map* (λp. $\bigcup$ (*map ftvg* (*snd p*))) (*rfn ci*)) $\subseteq$ *set* (*params ci*)
by (*rule inv-wf-c*, *simp*)
from *ciok* **have** *fmsps*: $\bigcup$ (*map ftvg* (*mem-tys ci*)) $\subseteq$ *set* (*params ci*)
by (*rule inv-wf-c*, *simp*)
from *Rs Cok* **have** $\bigcup$ (*map ftv* δs) $\subseteq \bigcup$ (*map* (λp. $\bigcup$ (*map ftvg* (*snd p*))) (*rfn ci*))
using *trans-reduces-ftv* **by** *simp*
with *frsps* **have** *fdsps*: $\bigcup$ (*map ftv* δs) $\subseteq$ *set* (*params ci*) **by** *simp*
from *Ms Cok* **have** $\bigcup$ (*map ftv* σs) $\subseteq \bigcup$ (*map ftvg* (*mem-tys ci*))
using *trans-reduces-ftv* **by** *simp*
with *fmsps* **have** *fssps*: $\bigcup$ (*map ftv* σs) $\subseteq$ *set* (*params ci*) **by** *simp*
have *ftv* ($\langle\delta s$ @ $\sigma s\rangle$) = $\bigcup$ (*map ftv* δs) $\cup \bigcup$ (*map ftv* σs)
by (*induct* δs *rule*: *list.induct*, *auto*)
with *fdsps fssps* **have** *ftv* ($\langle\delta s$ @ $\sigma s\rangle$) $\subseteq$ *set* (*params ci*) **by** *auto*
with *A* **have** *tsfds*: *set ts* $\cap$ *ftv* ($\langle\delta s$ @ $\sigma s\rangle$) = {} **by** *auto*
— We can alpha-convert the bound variables to be distinct from *ts*.
have *tsbds*: *set ts* $\cap$ *btv* ($\langle\delta s$ @ $\sigma s\rangle$) = {} **sorry**
from *tsfds tsbds ofb* **show** *?thesis* **by** *auto*
qed
from *ts-tsp* **have** *length ?ts* = *length ?tsp* **using** *trans-length* **by** *blast*
with *lts* **have** *D*: *length* (*params ci*) = *length* $\tau s'$
by (*simp add*: *subst-length substg-length*)
from *tsa-tsap* **have** *length* τsa = *length* $\tau sa'$ **using** *trans-length* **by** *blast*
with *len* **have** *E*: *length ts* = *length* $\tau sa'$ **by** *simp*
from *Cok cC* **have** $C \vdash$ *ci ok* **by** (*rule c-mem-implies-c-ok*)
hence *F*: *distinct* (*params ci*) **by** (*rule inv-wf-c*, *auto*)
from *A B C D E F* **have** [*ts*$\mapsto\tau sa'$]*?dt* = [*params ci*$\mapsto$*?tsp*]($\langle\delta s$@$\sigma s\rangle$)
using *substitution-lemma2* **apply** *blast* **done**
thus *?thesis* **by** *simp*
qed
ultimately show $C \vdash_d c$ {*ts*$\mapsto\tau sa$}$\tau s \rightsquigarrow$ *sub-ty*(*ts*,$\tau sa'$,*?dt*) **by** *simp*
qed

The rest of the cases are trivial and proved automatically by Isabelle.

lemma *subst-respects-trans*:
($C \vdash \tau \rightsquigarrow \tau' \longrightarrow$ *srt-ty C* τ τ') $\wedge$ ($C \models \tau s \rightsquigarrow \tau s' \longrightarrow$ *srt-tys C* τs $\tau s'$)
$\wedge$ ($C \vdash_d c$ $\varrho s \rightsquigarrow dt \longrightarrow$ *srt-dict C c* ϱs *dt*) $\wedge$ ($C \models_d rs \rightsquigarrow dts \longrightarrow$ *srt-ds C rs dts*)
apply (*induct rule*: *trans-ty-trans-tys-req-dict-reqs-dicts.induct*)
using *subst-respects-trans-var* **apply** *simp* **apply** *simp* **apply** *simp*
apply *simp* **apply** *simp* **apply** *simp* **apply** *simp*
using *subst-respects-trans-dict* **by** *simp*+

corollary *subst-r-d*:
assumes *D*: $C \vdash_d c$ $\varrho s \rightsquigarrow dt$ **and** *Cok*: *C ok* **and** *dist*: *distinct ts*

and L: *length ts* $=$ *length* τs **and** *ts-tsp*: $C \models \tau s \rightsquigarrow \tau s'$
shows $C \vdash_d c\ \{ts {\mapsto} \tau s\}\varrho s \rightsquigarrow [ts \mapsto \tau s']dt$
proof $-$
have $C \vdash_d c\ \varrho s \rightsquigarrow dt \longrightarrow$ *srt-dict C c* ϱs *dt* **using** *subst-respects-trans* **by** *simp*
with *Cok D dist L ts-tsp* **show** *?thesis* **by** *auto*
qed

corollary *subst-ds*:
assumes *Ds*: $C \models_d rs \rightsquigarrow dts$ **and** *Cok*: *C ok* **and** *dist*: *distinct ts*
and L: *length ts* $=$ *length* τs **and** *ts-tsp*: $C \models \tau s \rightsquigarrow \tau s'$
shows $C \models_d \{\!|ts \mapsto \tau s|\!\}rs \rightsquigarrow \{ts \mapsto \tau s'\}dts$
proof $-$
have $C \models_d rs \rightsquigarrow dts \longrightarrow$ *srt-ds C rs dts* **using** *subst-respects-trans* **by** *simp*
with *Cok Ds dist L ts-tsp* **show** *?thesis* **by** *auto*
qed

corollary *subst-trans-ty*:
assumes *Ds*: $C \vdash \tau \rightsquigarrow \tau'$ **and** *Cok*: *C ok* **and** *dist*: *distinct ts*
and L: *length ts* $=$ *length* τs **and** *ts-tsp*: $C \models \tau s \rightsquigarrow \tau s'$
shows $C \vdash [ts \mapsto \tau s]\tau \rightsquigarrow [ts \mapsto \tau s']\tau'$
proof $-$
have $C \vdash \tau \rightsquigarrow \tau' \longrightarrow$ *srt-ty C* $\tau\ \tau'$ **using** *subst-respects-trans* **by** *simp*
with *Cok Ds dist L ts-tsp* **show** *?thesis* **by** *auto*
qed

corollary *subst-trans-tys*:
assumes *Ds*: $C \models \sigma s \rightsquigarrow \sigma s'$ **and** *Cok*: *C ok* **and** *dist*: *distinct ts*
and L: *length ts* $=$ *length* τs **and** *ts-tsp*: $C \models \tau s \rightsquigarrow \tau s'$
shows $C \models \{ts \mapsto \tau s\}\sigma s \rightsquigarrow \{ts \mapsto \tau s'\}\sigma s'$
proof $-$
have $C \models \sigma s \rightsquigarrow \sigma s' \longrightarrow$ *srt-tys C* $\sigma s\ \sigma s'$ **using** *subst-respects-trans* **by** *simp*
with *Cok Ds dist L ts-tsp* **show** *?thesis* **by** *auto*
qed

If a concept is never referred to in a type, removing the concept from the environment does not affect the translation of that type. We skip the proof of this straightforward lemma due to time constraints.

lemma *remove-concept-pres-trans*:
$(insert\ (c,ci)\ C \vdash \tau \rightsquigarrow \tau' \longrightarrow (c,\tau) \notin c\text{-}occurs\text{-}ty \longrightarrow C \vdash \tau \rightsquigarrow \tau')$
$\wedge\ (insert\ (c,ci)\ C \models \sigma s \rightsquigarrow \sigma s' \longrightarrow (c,\tau) \notin c\text{-}occurs\text{-}ty \longrightarrow C \models \sigma s \rightsquigarrow \sigma s')$
$\wedge\ (insert\ (c,ci)\ C \vdash_d c\ \varrho s \rightsquigarrow dt \longrightarrow (c,\tau) \notin c\text{-}occurs\text{-}ty \longrightarrow C \vdash_d c\ \varrho s \rightsquigarrow dt)$
$\wedge\ (insert\ (c,ci)\ C \models_d rs \rightsquigarrow dts \longrightarrow (c,\tau) \notin c\text{-}occurs\text{-}ty \longrightarrow C \models_d rs \rightsquigarrow dts)$
sorry

corollary *remove-concept-pres-trans-ty*:
$[\![\ insert\ (c,ci)\ C \vdash \tau \rightsquigarrow \tau';\ (c,\tau) \notin c\text{-}occurs\text{-}ty\ ]\!] \Longrightarrow C \vdash \tau \rightsquigarrow \tau'$
using *remove-concept-pres-trans* **by** *blast*

Adding concepts to the environment (weakening) does not affect the translation of types.

lemma *add-concept-pres-trans*:
$(C \vdash \tau \rightsquigarrow \tau' \longrightarrow (\forall\ c\ ci.\ insert\ (c,ci)\ C \vdash \tau \rightsquigarrow \tau'))$
$\wedge\ (C \models \sigma s \rightsquigarrow \sigma s' \longrightarrow (\forall\ c\ ci.\ insert\ (c,ci)\ C \models \sigma s \rightsquigarrow \sigma s'))$
$\wedge\ (C \vdash_d c\ \varrho s \rightsquigarrow dt \longrightarrow (\forall\ c'\ ci'.\ insert\ (c',ci')\ C \vdash_d c\ \varrho s \rightsquigarrow dt))$
$\wedge\ (C \models_d rs \rightsquigarrow dts \longrightarrow (\forall\ c\ ci.\ insert\ (c,ci)\ C \models_d rs \rightsquigarrow dts))$
apply (*induct rule*: *trans-ty-trans-tys-req-dict-reqs-dicts.induct*)
using *r-d* **by** *auto*

The type translation is a function. The premise *C ok* is need to ensure that the concept environment contains no more than one definition for each concept name. Again, we skip the proof due to time constraints.

lemma *unique-trans-tys*: $[\![\ C \models \tau s \rightsquigarrow \sigma s;\ C\ ok;\ C \models \tau s \rightsquigarrow \sigma s'\ ]\!] \Longrightarrow \sigma s = \sigma s'$
sorry

Next we prove a lemma concerning substitution and the translation of refinments to dictionary types. The proof will use this basic fact about list append.

lemma *append-eq-len*: $\bigwedge ls1'\ ls2\ ls2'.\ [\![\ length\ ls1 = length\ ls1';\ ls1\ @\ ls2 = ls1'\ @\ ls2'\ ]\!]$
$\Longrightarrow ls1 = ls1' \wedge ls2 = ls2'$ **by** (*induct ls1*, *simp*, *case-tac ls1'*, *simp*, *simp*)

lemma *refine-dict-types*:
assumes *D*: $C \vdash_d c\ \tau s \rightsquigarrow \langle dts @ \sigma s\rangle$ **and** *Cok*: *C ok* **and** *cC*: $(c,\ ci) \in C$
and *L*: $length\ dts = length\ (rfn\ ci)$
shows $C \models_d \{\!|params\ ci {\mapsto} \tau s|\!\} rfn\ ci \rightsquigarrow dts$
proof −
from *D Cok cC* **obtain** $dts'\ \sigma s'\ \tau s'$ **where** *ts-tsp*: $C \models \tau s \rightsquigarrow \tau s'$
and *Ds*: $C \models_d rfn\ ci \rightsquigarrow dts'$
and *lpts*: $length\ \tau s = length\ (params\ ci)$
and *tp*: $\langle dts @ \sigma s\rangle = \langle\{params\ ci {\mapsto} \tau s'\}(dts' @ \sigma s')\rangle$ **using** *inv-r-d2* **by** *blast*
from *tp* **have** $\langle dts @ \sigma s\rangle = \langle\{params\ ci {\mapsto} \tau s'\}dts' @ \{params\ ci {\mapsto} \tau s'\}\sigma s'\rangle$
by (*simp only*: *subst-append*)
hence *T*: $dts @ \sigma s = \{params\ ci {\mapsto} \tau s'\}dts' @ \{params\ ci {\mapsto} \tau s'\}\sigma s'$ **by** *simp*
from *L* **have** $length\ dts = length\ (rfn\ ci)$ **.**
also from *Ds* **have** $\ldots = length\ dts'$ **by** (*rule trans-length-r-d*)
also have $\ldots = length\ \{params\ ci {\mapsto} \tau s'\}dts'$ **using** *subst-length* **by** *simp*
finally have *L1*: $length\ dts = length\ \{params\ ci {\mapsto} \tau s'\}dts'$ **by** *simp*
from *T L1 append-eq-len* **have** *dts*: $dts = \{(params\ ci) \mapsto \tau s'\}dts'$ **by** *simp*
from *T L1 append-eq-len* **have** *ss*: $\sigma s = \{params\ ci {\mapsto} \tau s'\}\sigma s'$ **by** *simp*
— So we finally have the dictionary types for the refinements.
have $C \models_d \{\!|params\ ci {\mapsto} \tau s|\!\} rfn\ ci \rightsquigarrow \{params\ ci {\mapsto} \tau s'\}dts'$
proof −
from *Cok cC* **have** *ciok*: $C \vdash ci\ ok$ **by** (*rule c-mem-implies-c-ok*)
from *ciok* **have** *dist*: *distinct* (*params ci*) **by** (*rule inv-wf-c*, *simp*)
from *Cok Ds dist lpts ts-tsp* **show** *?thesis* **by** (*simp only*: *subst-ds*)
qed
with *dts* **show** *?thesis* **by** *simp*
qed

Given that a list of F^{G} types translates to a list of F types, the ith F^{G} type translates to the ith F type.

lemma *trans-tys-nth*: $\bigwedge C\ \sigma s'\ i\ \tau.\ [\![\ C \models \sigma s \rightsquigarrow \sigma s';\ i < \mathit{length}\ \sigma s;\ \sigma s!\ i = \tau\ ]\!]$
$\Longrightarrow C \vdash \tau \rightsquigarrow \sigma s'!i$
proof $(\mathit{induct}\ \sigma s\ \mathit{rule}{:}\ \mathit{list.induct},\ \mathit{simp})$
fix $a\ \mathit{list}\ C\ \sigma s'\ i\ \tau$
assume $\mathit{IH}{:}\ \bigwedge C\ \sigma s'\ i\ \tau.\ [\![C \models \mathit{list} \rightsquigarrow \sigma s';\ i < \mathit{length}\ \mathit{list};\ \mathit{list}\ !\ i = \tau]\!] \Longrightarrow C \vdash \tau \rightsquigarrow \sigma s'\ !\ i$
and $\mathit{alss}{:}\ C \models a\ \#\ \mathit{list} \rightsquigarrow \sigma s'$ **and** $\mathit{il}{:}\ i < \mathit{length}\ (a\ \#\ \mathit{list})$ **and** $\mathit{alit}{:}\ (a\ \#\ \mathit{list})\ !\ i = \tau$
from *alss* **obtain** $\tau'\ \tau s'$ **where** $\mathit{t\text{-}tp}{:}\ C \vdash a \rightsquigarrow \tau'$ **and** $\mathit{ssp}{:}\ \sigma s' = \tau'\#\tau s'$
and $\mathit{ltsp}{:}\ C \models \mathit{list} \rightsquigarrow \tau s'$ **by** $(\mathit{rule}\ \mathit{inv\text{-}trans\text{-}cons},\ \mathit{auto})$
show $C \vdash \tau \rightsquigarrow \sigma s'\ !\ i$
proof $(\mathit{cases}\ i)$
assume $\mathit{iz}{:}\ i = 0$ **from** *iz alit* **have** $\mathit{at}{:}\ a = \tau$ **by** *simp*
from *at t-tp ssp iz* **show** *?thesis* **by** *simp*
next fix *j* **assume** $I{:}\ i = \mathit{Suc}\ j$
from *alit I* **have** $\mathit{ljt}{:}\ \mathit{list}!j = \tau$ **by** *simp*
from *il I* **have** $\mathit{jl}{:}\ j < \mathit{length}\ \mathit{list}$ **by** *simp*
from *ltsp jl ljt IH* **have** $C \vdash \tau \rightsquigarrow \tau s'!j$ **by** *blast*
with *I ssp* **show** *?thesis* **by** *simp*
qed
qed

8.5 Paths Through Dictionaries

There are several places in Figure 15 where the environment is extended with concepts, models, or variables. In Section 8.6 we show that the environment correspondence is maintained in each case. However, first we need several lemmas regarding paths through dictionaries.

The following two lemmas extend a path through a dictionary. The first extends the path to the sub-dictionary for a refinement. The second extends the path to a member of the dictionary. Both lemmas are straightforward inductions on the path *ns*.

lemma *dict-path-to-super*:
$\bigwedge \mathit{dts}\ \mathit{dt}\ \sigma s\ i\ \tau.\ [\![\ i < \mathit{length}\ \mathit{dts};\ \mathit{dt} = \langle \mathit{dts}@\sigma s\rangle;\ \tau{-}\mathit{ns}{\rightarrow}\mathit{dt}\ ]\!] \Longrightarrow \tau{-}\mathit{ns}@[i]{\rightarrow}\mathit{dts}!i$
proof $(\mathit{induct}\ \mathit{ns})$
fix $\mathit{dts}\ \mathit{dt}\ \sigma s\ i\ \tau$
assume $I{:}\ i < \mathit{length}\ \mathit{dts}$ **and** $\mathit{dt}{:}\ \mathit{dt} = \langle \mathit{dts}@\sigma s\rangle$ **and** $\mathit{t\text{-}dt}{:}\ \tau{-}[\,]{\rightarrow}\mathit{dt}$
from *t-dt* **have** $\mathit{eq}{:}\ \tau = \mathit{dt}$ **apply** $(\mathit{rule}\ \mathit{inv\text{-}path\text{-}nil})$ **apply** *simp* **done**
from *I* **have** $(\mathit{dts}@\sigma s)!i = \mathit{dts}!i$ **apply** $(\mathit{simp}\ \mathit{add}{:}\ \mathit{nth\text{-}append})$ **done**
hence $(\mathit{dts}@\sigma s)!i{-}[\,]{\rightarrow}\mathit{dts}!i$ **by** $(\mathit{simp}\ \mathit{add}{:}\ \mathit{p\text{-}nil})$
hence $\langle \mathit{dts}\ @\ \sigma s\rangle{-}i\#[\,]{\rightarrow}\mathit{dts}!i$ **by** $(\mathit{rule}\ \mathit{p\text{-}cons})$
with *eq dt* **show** $\tau{-}[\,]@[i]{\rightarrow}\ \mathit{dts}!i$ **by** *simp*
next fix $a\ \mathit{list}\ \mathit{dts}\ \mathit{dt}\ \sigma s\ i\ \tau$
assume $\mathit{IH}{:}\ \bigwedge \mathit{dts}\ \mathit{dt}\ \sigma s\ i\ \tau.\ [\![i < \mathit{length}\ \mathit{dts};\ \mathit{dt} = \langle \mathit{dts}@\sigma s\rangle;\ \tau{-}\mathit{list}{\rightarrow}\mathit{dt}]\!] \Longrightarrow \tau{-}\mathit{list}@[i]{\rightarrow}\mathit{dts}!i$
and $I{:}\ i < \mathit{length}\ \mathit{dts}$ **and** $\mathit{dt}{:}\ \mathit{dt} = \langle \mathit{dts}\ @\ \sigma s\rangle$
and $P{:}\ (\tau,\ a\ \#\ \mathit{list},\ \mathit{dt}) \in \mathit{path\text{-}ty}$
from *P* **obtain** τs **where** $\mathit{P2}{:}\ (\tau s!a,\ \mathit{list},\ \mathit{dt}) \in \mathit{path\text{-}ty}$

and T: $\tau = \langle \tau s \rangle$ **apply** (*rule inv-path-cons*) **apply** *simp* **done**
from *I dt P2 IH* **have** *P3*: $\tau s!a{-}list@[i]{\rightarrow}dts!i$ **by** *simp*
hence $\langle \tau s \rangle {-} a\#(list\ @\ [i]){\rightarrow}dts!i$ **by** (*rule p-cons*)
with T **have** $\tau{-}a\#(list@[i]){\rightarrow}dts!i$ **by** *simp*
thus $\tau{-}(a\#list)@[i]{\rightarrow}dts!i$ **by** *auto*
qed

lemma *dict-path-to-member*:
$\bigwedge dts\ dt\ \sigma s\ i\ \tau.\ [\![\ i < length\ \sigma s;\ dt{=}\langle dts@\sigma s\rangle;\ \tau{-}ns{\rightarrow}dt\ ]\!] \Longrightarrow \tau{-}ns@[length\ dts{+}i]{\rightarrow}\sigma s!i$
proof (*induct ns*)
fix $dts\ dt\ \sigma s\ i\ \tau$
assume I: $i < length\ \sigma s$ **and** dt: $dt = \langle dts@\sigma s\rangle$ **and** *t-dt*: $\tau{-}[]{\rightarrow}dt$
from *t-dt* **have** *eq*: $\tau = dt$ **apply** (*rule inv-path-nil*) **apply** *simp* **done**
from I **have** $(dts@\sigma s)!(length\ dts + i) = \sigma s!i$
apply (*simp add*: *nth-append-length-plus*) **done**
hence $(dts@\sigma s)!(length\ dts + i){-}[]{\rightarrow}\sigma s!i$ **by** (*simp add*: *p-nil*)
hence $\langle dts@\sigma s\rangle{-}(length\ dts{+}i)\#[]{\rightarrow}\sigma s!i$ **by** (*rule p-cons*)
with *eq dt* **show** $\tau{-}[]@[length\ dts + i]{\rightarrow}\sigma s!i$ **by** *simp*
next fix $a\ list\ dts\ dt\ \sigma s\ i\ \tau$
assume *IH*: $\bigwedge dts\ dt\ \sigma s\ i\ \tau.\ [\![i < length\ \sigma s;\ dt = \langle dts\ @\ \sigma s\rangle;\ \tau{-}list{\rightarrow}dt\ ]\!]$
$\Longrightarrow \tau{-}list@[length\ dts + i]{\rightarrow}\sigma s!i$
and I: $i < length\ \sigma s$ **and** dt: $dt = \langle dts@\sigma s\rangle$ **and** P: $\tau{-}a\#list{\rightarrow}dt$
from P **obtain** τs **where** *P2*: $\tau s!a{-}list{\rightarrow}dt$
and T: $\tau = \langle \tau s\rangle$ **apply** (*rule inv-path-cons*) **apply** *simp* **done**
from *I dt P2 IH* **have** *P3*: $\tau s!a{-}list@[length\ dts + i]{\rightarrow}\sigma s!i$ **by** *simp*
hence $\langle \tau s\rangle{-}a\#(list\ @\ [length\ dts + i]){\rightarrow}\sigma s!i$ **by** (*rule p-cons*)
with T **have** $\tau{-}a\#(list@[length\ dts + i]){\rightarrow}\sigma s!i$ **by** *simp*
thus $\tau{-}(a\#list)@[length\ dts + i]{\rightarrow}\sigma s!i$ **by** *auto*
qed

The next lemma states that the ith entry in the dictionary type for concept c is the dictionary type for the "super" concept c'. This lemma is proved by induction on the refinement list *rs*.

lemma *dict-at-i*: $\bigwedge C\ dts\ i\ c'\ \tau s'.\ [\![\ C \models_d rs \leadsto dts;\ rs!i = (c',\tau s');\ Suc\ i \leq length\ dts\ ]\!]$
$\Longrightarrow (\exists\ dts'\ \sigma s'\ ci'.\ C \vdash_d c'\ \tau s' \leadsto dts!i \wedge dts!i = \langle dts'@\sigma s'\rangle$
$\wedge\ (c',ci') \in C \wedge length\ (rfn\ ci') = length\ dts')$
apply (*induct rs rule*: *list.induct*) **prefer** *2* **apply** *clarify* **prefer** *2*
proof −
fix $C\ dts\ i$ **and** c'::*var* **and** $\tau s'$::*tyg list*
assume *Ds*: $C \models_d [] \leadsto dts$ **and** L: $Suc\ i \leq length\ dts$
from *Ds* **have** $dts = []$ **by** (*rule inv-rs-ds-nil*, *simp*)
with L **have** *False* **by** *simp*
thus $\exists\ dts'\ \sigma s'\ ci'.\ C \vdash_d c'\ \tau s' \leadsto dts!i \wedge dts!i = \langle dts'\ @\ \sigma s'\rangle$
$\wedge\ (c',ci') \in C \wedge length\ (rfn\ ci') = length\ dts'$ **by** *simp*
next fix $a\ b\ list\ C\ dts\ i\ c'\ \tau s'$
assume *IH*: $\bigwedge C\ dts\ i\ c'\ \tau s'.\ [\![\ C \models_d list \leadsto dts;\ list!i = (c',\ \tau s');\ Suc\ i \leq length\ dts\ ]\!]$
$\Longrightarrow (\exists dts'\ \sigma s'\ ci'.\ C \vdash_d c'\ \tau s' \leadsto dts!i \wedge dts!i = \langle dts'\ @\ \sigma s'\rangle$
$\wedge\ (c',ci') \in C \wedge length\ (rfn\ ci') = length\ dts')$
and *Ds*: $C \models_d (a,\ b)\ \#\ list \leadsto dts$

and *at*: $((a, b) \# list) \,!\, i = (c', \tau s')$
and *I*: $Suc\ i \le length\ dts$
from *Ds* **obtain** $\tau\ \tau s$ **where** *D*: $C \vdash_d a\ b \rightsquigarrow \tau$ **and** *Ds2*: $C \models_d list \rightsquigarrow \tau s$
and *dts*: $dts = \tau \# \tau s$ **by** (*rule inv-rs-ds-cons*, *simp*)
show $\exists\, dts'\ \sigma s'\ ci'.\ C \vdash_d c'\ \tau s' \rightsquigarrow dts!i \wedge dts!i = \langle dts' @ \sigma s'\rangle \wedge (c', ci') \in C$
$\wedge\ length\ (rfn\ ci') = length\ dts'$
proof (*cases i*)
assume *iz*: $i = 0$
from *iz at* **have** *eq*: $(a,b) = (c',\tau s')$ **by** *simp*
from *D eq* **have** *D2*: $C \vdash_d c'\ \tau s' \rightsquigarrow \tau$ **by** *simp*
from *D2* **obtain** $\delta s\ \sigma s\ \tau s''\ ci$ **where** *cC*: $(c',ci) \in C$ **and** *ts-tsp*: $C \models \tau s' \rightsquigarrow \tau s''$
and *Ds*: $C \models_d rfn\ ci \rightsquigarrow \delta s$ **and** *Ms*: $C \models mem\text{-}tys\ ci \rightsquigarrow \sigma s$
and *tp*: $\tau = \langle\{params\ ci \mapsto \tau s''\}(\delta s @ \sigma s)\rangle$ **by** (*rule inv-r-d*, *auto*)
from *tp* **have** *T*: $\tau = \langle(\{params\ ci \mapsto \tau s''\}\delta s @ \{params\ ci \mapsto \tau s''\}\sigma s)\rangle$
by (*simp only*: *subst-append*)
from *T D2* **have**
D3: $C \vdash_d c'\ \tau s' \rightsquigarrow \langle(\{params\ ci \mapsto \tau s''\}\delta s\ @\ \{params\ ci \mapsto \tau s''\}\sigma s)\rangle$ **by** *simp*
from *T iz dts* **have**
dtsi: $dts \,!\, i = \langle(\{params\ ci \mapsto \tau s''\}\delta s\ @\ \{params\ ci \mapsto \tau s''\}\sigma s)\rangle$ **by** *simp*
from *Ds trans-length* **have** $length\ (rfn\ ci) = length\ \delta s$ **by** *blast*
hence *L*: $length\ (rfn\ ci) = length\ \{(params\ ci) \mapsto \tau s''\}\delta s$ **using** *subst-length* **by** *simp*
from *D3 dtsi* **have** *D4*: $C \vdash_d c'\ \tau s' \rightsquigarrow dts!i$ **by** *simp*
from *D4 dtsi cC L* **show** *?thesis* **by** *blast*
next fix *j* **assume** *ij*: $i = Suc\ j$
from *I ij dts* **have** *J*: $Suc\ j \le length\ \tau s$ **by** *simp*
from *ij at* **have** *at2*: $list \,!\, j = (c',\tau s')$ **by** *simp*
from *Ds2 at2 J IH* **obtain** $dts'\ \sigma s'\ ci'$ **where** *D2*: $C \vdash_d c'\ \tau s' \rightsquigarrow \tau s!j$
and *at3*: $\tau s!j = \langle dts' @ \sigma s'\rangle$ **and** *cC*: $(c',ci') \in C$
and *L*: $length\ (rfn\ ci') = length\ dts'$ **by** *blast*
from *D2 dts ij* **have** *D3*: $C \vdash_d c'\ \tau s' \rightsquigarrow dts!i$ **by** *simp*
from *dts ij at3* **have** *at4*: $dts!i = \langle dts' @ \sigma s'\rangle$ **by** *simp*
from *D3 at4 cC L* **show** *?thesis* **by** *auto*
qed
qed

8.6 Preserving the Environment Correspondence

The environment correspondence defined in Figure 17 must be preserved in the face of changes made to the environment. For example, in *fg-abs*, the variables *xs* are added to the variable environment, bound to the types τs. To maintain the correspondence, we also add the variables *xs* to the System F environment, bound to the types $\tau s'$, where *concepts* $\Gamma \models \tau s \rightsquigarrow \tau s'$. The following lemma is proved by induction on the judgment $C \models \tau s \rightsquigarrow \tau s'$ (and the other judgments that it was mutually defined with).

lemma *add-vars-preserves-var-env*:
$(C \vdash \tau \rightsquigarrow \tau' \longrightarrow True)$
$\wedge\ (C \models \tau s \rightsquigarrow \tau s' \longrightarrow (\forall\ xs.\ C \vdash_v V \rightsquigarrow S \wedge length\ xs = length\ \tau s$
$\longrightarrow C \vdash_v V{,}xs{:}\tau s \rightsquigarrow S{,}xs{:}\tau s'))$
$\wedge\ (C \vdash_d c\ \varrho s \rightsquigarrow dt \longrightarrow True) \wedge (C \models_d rs \rightsquigarrow dts \longrightarrow True)$

apply (*induct rule*: *trans-ty-trans-tys-req-dict-reqs-dicts.induct*)
apply *auto* **apply** (*case-tac xs*) **using** *cv-cons* **by** *auto*

The following lemma provides a convenient way to use the invariants captured in $C \vdash_v V \rightsquigarrow S$. This lemma is used in the *fg-var* case of the main theorem.

lemma *var-mem-trans-implies*:
$[\![C \vdash_v V \rightsquigarrow S; (x,\tau) \in V]\!] \Longrightarrow (\exists\ \tau'.\ C \vdash \tau \rightsquigarrow \tau' \wedge (x,\tau') \in S)$
by (*induct rule*: *trans-var-env.induct*, *auto*)

The next two "weakening" lemmas show that adding a concept to the environment does not affect variable and model environment correspondences.

lemma *add-concept-preserves-var-env*: $C \vdash_v V \rightsquigarrow S \Longrightarrow insert\ (c,ci)\ C \vdash_v V \rightsquigarrow S$
apply (*induct rule*: *trans-var-env.induct*)
apply (*simp add*: *cv-nil*) **using** *add-concept-pres-trans cv-cons* **by** *auto*

lemma *add-concept-preserves-model-env*: $C \vdash_m M \rightsquigarrow S \Longrightarrow insert\ (c,ci)\ C \vdash_m M \rightsquigarrow S$
apply (*induct rule*: *trans-model-env.induct*)
apply (*simp add*: *cm-nil*) **using** *add-concept-pres-trans cm-cons* **apply** *simp*
proof −
fix $C\ M\ S\ \tau\ \tau'\ \tau s\ ca\ d\ ns$
assume *m-s*: $insert\ (c,\ ci)\ C \vdash_m M \rightsquigarrow S$ **and** *N*: $ns \neq []$
and *dt*: $(d,\ \tau) \in S$ **and** *D*: $C \vdash_d ca\ \tau s \rightsquigarrow \tau'$ **and** *P*: $path\text{-}ty\ \tau\ ns\ \tau'$
from *D* **have** *D2*: $insert\ (c,ci)\ C \vdash_d ca\ \tau s \rightsquigarrow \tau'$ **using** *add-concept-pres-trans* **by** *simp*
from *m-s N dt D2 P* **show** $insert\ (c,\ ci)\ C \vdash_m insert\ (ca,\ \tau s,\ d,\ ns)\ M \rightsquigarrow S$
by (*rule cm-drop*)
qed

Next we prove several lemmas that show how the correspondence with a System F typing environment is preserved as models are added to the environment. First we show that adding models for the where clause of a type abstraction preserves the correspondence . In particular, if we start with some model environment *M* in correspondence with some System F environment *S*, and if *ds* are the dictionary variables for the added models, and *dts* are the types of the dictionaries for the models, then the new model environment M' will correspond to *S*,*ds*:*dts*.

lemma *add-models-where-preserves*:
$[\![C \vdash ws\ ds\ M \Rightarrow M'; C\ ok; C \models_d ws \rightsquigarrow dts; C \vdash_m M \rightsquigarrow S]\!] \Longrightarrow C \vdash_m M' \rightsquigarrow S,ds{:}dts$

The judgment $C \vdash ws\ ds\ M \Rightarrow M'$ processes each requirement in the where clause using $\vdash_\flat$. The judgment $\vdash_\flat$ adds a model to the environment and then uses $\models_\flat$ to add models for all of its concept refinements. We prove two lemmas with regards to how $\vdash_\flat$ and $\models_\flat$ preserve the environment correspondence while adding models to the environment. The first lemma, in Figure 18, handles the case when $\vdash_\flat$ is used on a refinement, and thus the dictionary for the model will be a sub-dictionary of some other model. The dictionary path will be non-empty in this case. The second lemma, in Figure 19, handles when $\vdash_\flat$ is applied to a requirement in a where clause, when the dictionary path for the model is empty. Figure 20 uses this lemma to show preservation of the correspondence for all the requirements in the where clause.

Figure 18: Adding models to the environment for concept refinements preserves the environment correspondence.

lemma *add-models-rfns-pres*:
 $(C \vdash_\flat c\ \varrho s\ d\ ns\ M \Rightarrow M' \longrightarrow (\forall\ S\ \tau\ dts\ \sigma s\ ci.\ C\ ok \wedge ns \neq []$
  $\wedge\ C \vdash_d c\ \varrho s \rightsquigarrow \langle dts @ \sigma s\rangle \wedge (d,\tau) \in S \wedge (c,ci) \in C$
  $\wedge\ length\ (rfn\ ci) = length\ dts \wedge \tau{-}ns{\rightarrow}\langle dts @ \sigma s\rangle \wedge C \vdash_m M \rightsquigarrow S$
  $\longrightarrow C \vdash_m M' \rightsquigarrow S))$
 $\wedge\ (C \models_\flat i\ rs\ d\ ns\ M \Rightarrow M' \longrightarrow (\forall\ S\ dts\ \tau\ \sigma s.\ C\ ok \wedge C \models_d rs \rightsquigarrow dts$
  $\wedge\ (d,\tau) \in S \wedge \tau{-}ns{\rightarrow}\langle dts @ \sigma s\rangle \wedge i \leq length\ dts \wedge C \vdash_m M \rightsquigarrow S$
  $\longrightarrow C \vdash_m M' \rightsquigarrow S))$
 (**is** $(C \vdash_\flat c\ \varrho s\ d\ ns\ M \Rightarrow M' \longrightarrow\ ?P\ C\ c\ \varrho s\ d\ ns\ M\ M')$
  $\wedge\ (C \models_\flat i\ rs\ d\ ns\ M \Rightarrow M' \longrightarrow\ ?PS\ C\ i\ rs\ d\ ns\ M\ M'))$
proof (*induct rule*: *flat-m-flat-ms.induct*)
 fix $C{::}Cenv$ **and** $M\ M'\ M''\ \tau s\ c\ ci\ d\ i\ ns$
 assume *cC*: $(c, ci) \in C$ **and** *Mp*: $M' = insert\ (c, \tau s, d, ns)\ M$
  and *IH*: $?PS\ C\ (length\ (rfn\ ci))\ (\{\!|params\ ci {\mapsto} \tau s|\!\}rfn\ ci)\ d\ ns\ M'\ M''$
 show $?P\ C\ c\ \tau s\ d\ ns\ M\ M''$
 proof *clarify* **fix** $S\ \tau\ dts\ \sigma s\ ci'$ **assume** *Cok*: $C\ ok$ **and** *N*: $ns \neq []$
  and *D*: $C \vdash_d c\ \tau s \rightsquigarrow \langle dts @ \sigma s\rangle$ **and** *DT*: $(d,\tau) \in S$
  and *cpC*: $(c,ci') \in C$ **and** *L*: $length(rfn\ ci') = length\ dts$
  and *P*: $\tau{-}ns{\rightarrow}\langle dts @ \sigma s\rangle$ **and** *m-s*: $C \vdash_m M \rightsquigarrow S$
  from *Cok cC cpC* **have** *ci-cip*: $ci = ci'$ **by** (*rule unique-concept*)
  from *L ci-cip* **have** *L2*: $length\ dts = length\ (rfn\ ci)$ **by** *simp*
  from *D Cok cC L2* **have** *Ds2*: $C \models_d \{\!|params\ ci {\mapsto} \tau s|\!\}rfn\ ci \rightsquigarrow dts$
   by (*rule refine-dict-types*)
  from *L2* **have** *L3*: $length\ (rfn\ ci) \leq length\ dts$ **by** *simp*
  from *m-s N DT D P* **have** $C \vdash_m insert\ (c,\tau s,d,ns)\ M \rightsquigarrow S$ **by** (*rule cm-drop*)
  with *Mp* **have** *mp-s*: $C \vdash_m M' \rightsquigarrow S$ **by** *simp*
  from *Cok Ds2 DT P L3 mp-s IH* **show** $C \vdash_m M'' \rightsquigarrow S$ **by** *auto*
 qed
next fix $C\ M\ d\ ns\ rs$ **show** $?PS\ C\ 0\ rs\ d\ ns\ M\ M$ **by** *simp*
next fix $C\ M\ M'\ M''\ \tau s'\ c'\ d\ i\ ns\ rs$ **assume** *rsi*: $rs\ !\ i = (c', \tau s')$
  and *IH1*: $?P\ C\ c'\ \tau s'\ d\ (ns@[i])\ M\ M'$ **and** *IH2*: $?PS\ C\ i\ rs\ d\ ns\ M'\ M''$
 show $?PS\ C\ (Suc\ i)\ rs\ d\ ns\ M\ M''$
 proof *clarify*
  fix $S\ dts\ \tau\ \sigma s$ **assume** *Cok*: $C\ ok$ **and** *Rs*: $C \models_d rs \rightsquigarrow dts$
   and *DT*: $(d, \tau) \in S$ **and** *P*: $\tau{-}ns{\rightarrow}\langle dts\ @\ \sigma s\rangle$
   and *I*: $Suc\ i \leq length\ dts$ **and** *m-s*: $C \vdash_m M \rightsquigarrow S$
  from *Rs rsi I Cok* **obtain** $dts'\ \sigma s'\ ci'$ **where**
   D: $C \vdash_d c'\ \tau s' \rightsquigarrow dts!i$ **and** *dtsp*: $dts!i = \langle dts' @ \sigma s'\rangle$
   and *cC*: $(c',ci') \in C$ **and** *LR*: $length\ (rfn\ ci') = length\ dts'$
   using *dict-at-i* **by** *blast*
  from *D dtsp* **have** *D2*: $C \vdash_d c'\ \tau s' \rightsquigarrow \langle dts' @ \sigma s'\rangle$ **by** *simp*
  from *I P* **have** $\tau{-}ns@[i]{\rightarrow}dts!i$ **by** (*simp add*: *dict-path-to-super*)
  with *I dtsp* **have** *P2*: $\tau{-}ns@[i]{\rightarrow}\langle dts'\ @\ \sigma s'\rangle$ **by** *simp*
  from *Cok D2 DT cC LR P2 m-s IH1* **have** *mp-s*: $C \vdash_m M' \rightsquigarrow S$ **by** *blast*
  from *I* **have** *I2*: $i \leq length\ dts$ **by** *simp*
  from *Cok Rs DT P I2 mp-s IH2* **show** $C \vdash_m M'' \rightsquigarrow S$ **by** *auto*
 qed
qed

The following corollary captures first half of Lemma *add-models-rfns-pres*, which we use in Lemma *add-models-req-preserves*.

corollary *add-models-rfns-preserves*: $[\![\, C \vdash_\flat c\ \tau s\ d\ ns\ M \Rightarrow M'; C\ ok; ns \neq []$;
$C \vdash_d c\ \tau s \rightsquigarrow \langle dts@\sigma s\rangle$; $(d,\tau) \in S$; $(c,ci) \in C$; $length\ (rfn\ ci) = length\ dts$;
$\tau{-}ns{\to}\langle dts@\sigma s\rangle$; $C \vdash_m M \rightsquigarrow S\,]\!] \Longrightarrow C \vdash_m M' \rightsquigarrow S$
using *add-models-rfns-pres* **by** *blast*

The other place the model environment is extended is, of course, at model definitions. The lemma in Figure 21 proves that we can add model $(c,\varrho s,d,[])$ to the environment, and the corresponding System F environment will be $S,d{:}\langle [params\ ci{\mapsto}\varrho s'] dts\ @\ \sigma s'\rangle$, where d is bound to the dictionary type for the model. The main work of the proof is to show *Dt* which states that the dictionary type is correct.

8.7 Model Member Lookup

In preparation for proving the case in the main theorem for model member access, we need to show that the member access judgment $\vdash^\flat$ returns a type τ and dictionary path ns' such that the path leads to a type τ' that is the translation of τ.

lemma *dict-member*: $[\![\, C \vdash^\flat x\ c\ \tau s\ ns \Rightarrow \tau\ ns'; C\ ok; C \vdash_d c\ \tau s \rightsquigarrow dt'; dt{-}ns{\to}dt'\,]\!]$
$\Longrightarrow (\exists\ \tau'.\ dt{-}ns'{\to}\tau' \wedge C \vdash \tau \rightsquigarrow \tau')$

The member access judgment $\vdash^\flat$ is mutually recursive with the judgment $\models^\flat$ which looks for a member among the refinements. Thus, our proof is an induction on the derivation of both judgments. There are four cases to consider. The proof is fairly long and tedious, so we summarize the proof here before presenting the proof itself. The first case of the proof is when the member x appears in the current concept c. We rely on the Lemma *lookup-succeeds* to get the type and position of the member. We then use Lemma *dict-path-to-member* to show that we can extend the current path to this member. The second case is for when $\vdash^\flat$ uses $\models^\flat$ to find the member in a refinement. We simply use the assumptions with the induction hypothesis. The third case is when the ith refinement, concept c' with type arguments $\tau s'$ has the member. This case is complicated by the substitutions that occur for the type parameters of the concept . The fourth case is for continuing on to the next refinement in concept c. This case is trivial, since we just use the assumptions with the induction hypothesis. The following is the proof in its entirety.

lemma *lookup-found*: $\bigwedge x\ \tau s\ i\ j\ \tau.\ lookup\ x\ ts\ \tau s\ i = Some\ (\tau, j) \Longrightarrow x \in set\ ts$
apply (*induct ts*) **apply** *simp* **apply** (*case-tac* τs) **apply** *simp* **apply** *simp*
apply (*case-tac a* = *x*) **by** *simp*+

lemma *dict-member-helper*:
$(C \vdash^\flat x\ c\ \tau s\ ns \Rightarrow \tau\ ns' \longrightarrow (\forall\ dt\ dt'.\ C\ ok \wedge C \vdash_d c\ \tau s \rightsquigarrow dt' \wedge dt{-}ns{\to}dt'$
$\longrightarrow (\exists\ \tau'.\ dt{-}ns'{\to}\tau' \wedge C \vdash \tau \rightsquigarrow \tau')))$
$\wedge\ (C \models^\flat x\ i\ c\ \tau s\ ns \Rightarrow \tau\ ns' \longrightarrow (\forall\ dt\ dt'\ ci.\ C\ ok \wedge C \vdash_d c\ \tau s \rightsquigarrow dt' \wedge dt{-}ns{\to}dt'$

Figure 19: Adding models for a requirement in a where clause preserves the environment correspondence.

lemma *add-models-req-preserves*:
  $(C \vdash_\flat c\ \varrho s\ d\ ns\ M \Rightarrow M' \longrightarrow (\forall\ S\ \tau.\ C\ ok \wedge C \vdash_d c\ \varrho s \rightsquigarrow \tau \wedge ns = []$
    $\wedge\ C \vdash_m M \rightsquigarrow S \longrightarrow C \vdash_m M' \rightsquigarrow (S,d{:}\tau)))$
  $\wedge\ (C \models_\flat i\ rs\ d\ ns\ M \Rightarrow M' \longrightarrow (\forall\ S\ dts\ \tau\ \sigma s.\ C\ ok \wedge C \models_d rs \rightsquigarrow dts \wedge (d,\tau) \in S$
    $\wedge\ \tau{-}ns{\rightarrow}\langle dts@\sigma s\rangle \wedge i \leq length\ dts \wedge C \vdash_m M \rightsquigarrow S \longrightarrow C \vdash_m M' \rightsquigarrow S))$
  (**is** $(C \vdash_\flat c\ \varrho s\ d\ ns\ M \Rightarrow M' \longrightarrow ?P\ C\ c\ \varrho s\ d\ ns\ M\ M')$
    $\wedge\ (C \models_\flat i\ rs\ d\ ns\ M \Rightarrow M' \longrightarrow ?PS\ C\ i\ rs\ d\ ns\ M\ M')$)
proof (*induct rule*: *flat-m-flat-ms.induct*)
  fix $C\ M\ M'\ M''\ \tau s\ \tau s'\ c\ ci\ d\ ns$
  assume *C*: $(c,ci) \in C$ **and** *Mp*: $M' = insert\ (c,\tau s,d,ns)\ M$
    and *IH*: $?PS\ C\ (length\ (rfn\ ci))\ (\{\!|params\ ci{\mapsto}\tau s|\!\}(rfn\ ci))\ d\ ns\ M'\ M''$
  { **fix** $S\ \tau$ **assume** *Cok*: $C\ ok$ **and** *D*: $C \vdash_d c\ \tau s \rightsquigarrow \tau$ **and** *N*: $ns = []$
    and *m-s*: $C \vdash_m M \rightsquigarrow S$
    from *m-s D* **have** *mp-s*: $C \vdash_m insert\ (c,\tau s,d,[])\ M \rightsquigarrow S,d{:}\tau$ **by** (*rule cm-cons*)
    from *D* **obtain** $dts\ \sigma s\ \tau s'\ ci'$ **where** *cip*: $(c,ci') \in C$ **and** *ts-tsp*: $C \models \tau s \rightsquigarrow \tau s'$
      and *Dsp*: $C \models_d rfn\ ci' \rightsquigarrow dts$ **and** *lts*: $length\ \tau s = length\ (params\ ci')$
      and *tp*: $\tau = \langle\{params\ ci'{\mapsto}\tau s'\}(dts@\sigma s)\rangle$ **by** (*rule inv-r-d*, *auto*)
    from *Cok C cip* **have** *ci-cip*: $ci = ci'$ **by** (*rule unique-concept*)
    let $?Tup = \langle\{params\ ci{\mapsto}\tau s'\}dts\ @\ \{params\ ci{\mapsto}\tau s'\}\sigma s\rangle$
    from *ci-cip tp* **have** *T*: $\tau = ?Tup$ **by** (*simp only*: *subst-append*)
    from *T N* **have** *P*: $\tau{-}ns{\rightarrow}?Tup$ **using** *p-nil* **by** *simp*
    from *Cok cip ci-cip* **have** *distinct* (*params ci*)
      using *c-mem-implies-c-ok inv-wf-c* **by** *blast*
    with *Cok Dsp ci-cip lts ts-tsp* **have**
      Ds2: $C \models_d \{\!|params\ ci{\mapsto}\tau s|\!\}(rfn\ ci) \rightsquigarrow \{params\ ci{\mapsto}\tau s'\}dts$ **by** (*simp only*: *subst-ds*)
    have *DT*: $(d,\tau) \in S,d{:}\tau$ **by** *simp*
    from *Dsp ci-cip* **have** *L*: $length\ (rfn\ ci) \leq length\ \{params\ ci{\mapsto}\tau s'\}dts$
      using *trans-length-r-d subst-length* **by** *simp*
    from *Cok Ds2 DT P L mp-s Mp N IH* **have** $C \vdash_m M'' \rightsquigarrow S,d{:}\tau$ **by** *blast*
  } **thus** $?P\ C\ c\ \tau s\ d\ ns\ M\ M''$ **by** *simp*
next fix $C\ M\ d\ ns\ rs$ **show** $?PS\ C\ 0\ rs\ d\ ns\ M\ M$ **by** *simp*
next fix $C\ M\ M'\ M''\ \tau s'\ c'\ d\ i\ ns\ rs$ **assume** *rsi*: $rs\ !\ i = (c', \tau s')$
    and *F*: $C \vdash_\flat c'\ \tau s'\ d\ ns\ @\ [i]\ M \Rightarrow M'$ **and** *IH2*: $?PS\ C\ i\ rs\ d\ ns\ M'\ M''$
  show $?PS\ C\ (Suc\ i)\ rs\ d\ ns\ M\ M''$
  proof *clarify* **fix** $S\ dts\ \tau\ \sigma s$ **assume** *Cok*: $C\ ok$ **and** *Rs*: $C \models_d rs \rightsquigarrow dts$
      and *DT*: $(d, \tau) \in S$ **and** *P*: $\tau{-}ns{\rightarrow}\langle dts@\sigma s\rangle$
      and *I*: $Suc\ i \leq length\ dts$ **and** *m-s*: $C \vdash_m M \rightsquigarrow S$
    from *Rs rsi I Cok* **obtain** $dts'\ \sigma s'\ ci'$ **where** *D*: $C \vdash_d c'\ \tau s' \rightsquigarrow dts!i$
      and *dtsp*: $dts!i = \langle dts'@\sigma s'\rangle$ **and** *cpC*: $(c',ci') \in C$
      and *LR*: $length\ (rfn\ ci') = length\ dts'$ **using** *dict-at-i* **by** *blast*
    from *I P* **have** $\tau{-}ns@[i]{\rightarrow}dts!i$ **by** (*simp add*: *dict-path-to-super*)
    with *dtsp* **have** *P2*: $\tau{-}ns@[i]{\rightarrow}\langle dts'@\sigma s'\rangle$ **by** *simp*
    from *F Cok D dtsp DT cpC LR P2 m-s* **have**
      mp-s: $C \vdash_m M' \rightsquigarrow S$ **by** (*simp add*: *add-models-rfns-preserves*)
    from *I* **have** *I3*: $i \leq length\ dts$ **by** *simp*
    from *Cok Rs DT P I3 mp-s IH2* **show** $C \vdash_m M'' \rightsquigarrow S$ **by** *auto*
  qed
qed

Figure 20: Adding models for the where clause of a type abstraction preserves the environment correspondence.

lemma *add-models-where-preserves*:
$C \vdash ws\ ds\ M \Rightarrow M' \Longrightarrow (\bigwedge dts\ S.\ [\![\ C\ ok; C \models_d ws \rightsquigarrow dts; C \vdash_m M \rightsquigarrow S\ ]\!]$
$\Longrightarrow C \vdash_m M' \rightsquigarrow S,ds{:}dts \wedge length\ ds = length\ dts)$
proof (*induct rule*: *add-models.induct*)
fix $C\ M\ dts\ S$ **assume** D: $C \models_d [] \rightsquigarrow dts$ **and** *m-s*: $C \vdash_m M \rightsquigarrow S$
from D **have** *dn*: $dts = []$ **by** (*rule inv-rs-ds-nil*, *simp*)
hence $S = S,[]{:}dts$ **by** *simp*
with *m-s dn* **show** $C \vdash_m M \rightsquigarrow S,[]{:}dts \wedge length\ [] = length\ dts$ **by** *auto*
next fix $C\ M\ M'\ M''\ \varrho s\ c\ d\ ds\ ws\ dts\ S$
assume F: $C \vdash_\flat c\ \varrho s\ d\ []\ M \Rightarrow M'$
and *IH*: $\bigwedge dts\ S.\ [\![\ C\ ok; C \models_d ws \rightsquigarrow dts; C \vdash_m M' \rightsquigarrow S]\!]$
$\Longrightarrow C \vdash_m M'' \rightsquigarrow S,ds{:}dts \wedge length\ ds = length\ dts$
and *Cok*: $C\ ok$ **and** *Ds*: $C \models_d (c,\varrho s)\#ws \rightsquigarrow dts$ **and** *m-s*: $C \vdash_m M \rightsquigarrow S$
from *Ds* **obtain** $dt\ dts'$ **where** D: $C \vdash_d c\ \varrho s \rightsquigarrow dt$ **and** *Dsp*: $C \models_d ws \rightsquigarrow dts'$
and *DTS*: $dts = dt\#dts'$ **by** (*rule inv-rs-ds-cons*, *auto*)
from *F Cok D m-s add-models-req-preserves* **have**
mp-sd: $C \vdash_m M' \rightsquigarrow S,d{:}dt$ **by** *blast*
from *Cok Dsp mp-sd IH* **have**
mpp-sp: $C \vdash_m M'' \rightsquigarrow (S,d{:}dt),ds{:}dts' \wedge length\ ds = length\ dts'$ **by** *simp*
from *DTS* **have** $(S,d{:}dt),ds{:}dts' = S,(d\#ds){:}dts$ **by** (*simp only*: *pushs-env-assoc*)
with *mpp-sp DTS* **show** $C \vdash_m M'' \rightsquigarrow S,(d\#ds){:}dts \wedge length\ (d\#ds) = length\ dts$ **by** *simp*
qed

Figure 21: Adding a model to the model environment for a model definition preserves the environment correspondence.

lemma *add-model-preserves*:
  assumes *g-s*: $\Gamma \rightsquigarrow S$ **and** *Cok*: *concepts* Γ *ok* **and** *C*: $(c, ci) \in$ *concepts* Γ
  and *rs-rsp*: *concepts* $\Gamma \models \varrho s \rightsquigarrow \varrho s'$ **and** *Ds*: *concepts* $\Gamma \models_d$ *rfn ci* $\rightsquigarrow$ *dts*
  and *ss-ssp*: *concepts* $\Gamma \models \sigma s \rightsquigarrow \sigma s'$ **and** *memtys*: $\sigma s = \{params\ ci \mapsto \varrho s\}(mem\text{-}tys\ ci)$
  and *lps*: *length* (*params ci*) = *length* ϱs
  shows Γ,*model* $(c,\varrho s,d,[]) \rightsquigarrow S(\!|\ tys := (tys\ S), d{:}(\langle\{params\ ci \mapsto \varrho s'\}dts @ \sigma s'\rangle)\ |\!)$
proof $-$
  let $?Gp = \Gamma$,*model* $(c, \varrho s, d, [])$ **and** $?sdts = \{params\ ci \mapsto \varrho s'\}dts$
  from *g-s* **obtain** *Sv Sm* **where** *v-s*: *concepts* $\Gamma \vdash_v$ *vars* $\Gamma \rightsquigarrow Sv$
    and *m-s*: *concepts* $\Gamma \vdash_m$ *models* $\Gamma \rightsquigarrow Sm$ **and** *tvsg*: *tvars S* = *tyvars* Γ
    and *s*: *tys S* = $Sm \cup Sv$ **by** *auto*
  from *v-s* **have** *v-s2*: *concepts ?Gp* $\vdash_v$ *vars ?Gp* $\rightsquigarrow Sv$ **by** *simp*
  from *m-s* **have** *m-s2*: *concepts ?Gp* $\vdash_m$ *models* $\Gamma \rightsquigarrow Sm$ **by** *simp*
  have *Dt*: *concepts ?Gp* $\vdash_d c\ \varrho s \rightsquigarrow \langle ?sdts\ @\ \sigma s'\rangle$
  proof $-$
    from *C* **have** *C2*: $(c,ci) \in$ *concepts ?Gp* **by** *simp*
    from *rs-rsp* **have** *rs-rsp2*: *concepts ?Gp* $\models \varrho s \rightsquigarrow \varrho s'$
      by (*simp add*: *add-concept-pres-trans*)
    from *Ds* **have** *Ds2*: *concepts ?Gp* $\models_d$ (*rfn ci*) $\rightsquigarrow$ *dts*
      by (*simp add*: *add-concept-pres-trans*)
    from *Cok C* **have** *ciok*: *concepts* $\Gamma \vdash$ *ci ok* **by** (*rule c-mem-implies-c-ok*)
    from *ciok* **obtain** $\sigma s''$ **where** *ms-ssp*: *concepts* $\Gamma \models$ *mem-tys ci* $\rightsquigarrow \sigma s''$
      by (*rule inv-wf-c*, *auto*)
    from *ms-ssp* **have** *ms-ssp2*: *concepts ?Gp* $\models$ *mem-tys ci* $\rightsquigarrow \sigma s''$
      by (*simp add*: *add-concept-pres-trans*)
    from *lps* **have** *lrs*: *length* ϱs = *length* (*params ci*) **by** *simp*
    from *C2 rs-rsp2 Ds2 ms-ssp2 lrs*
    have *concepts ?Gp* $\vdash_d c\ \varrho s \rightsquigarrow [params\ ci \mapsto \varrho s'](\langle dts @ \sigma s''\rangle)$ **by** (*rule r-d*)
    hence *D*: *concepts ?Gp* $\vdash_d c\ \varrho s \rightsquigarrow (\langle ?sdts\ @\ \{params\ ci \mapsto \varrho s'\}\sigma s''\rangle)$
      using *subst-append* **by** *simp*
    from *Cok C* **have** *dist*: *distinct* (*params ci*) **using** *c-mem-implies-c-ok inv-wf-c* **by** *blast*
    from *Cok ms-ssp2 dist lps rs-rsp2* **have**
      concepts ?Gp $\models \{params\ ci \mapsto \varrho s\}(mem\text{-}tys\ ci) \rightsquigarrow \{params\ ci \mapsto \varrho s'\}\sigma s''$
      using *subst-trans-tys* **by** *simp*
    with *memtys* **have** *concepts ?Gp* $\models \sigma s \rightsquigarrow \{params\ ci \mapsto \varrho s'\}\sigma s''$ **by** *simp*
    with *Cok ss-ssp* **have** $\sigma s' = \{params\ ci \mapsto \varrho s'\}\sigma s''$ **using** *fun-dict-trans-ty* **by** *simp*
    with *D* **show** *?thesis* **by** *simp*
  qed
  from *m-s2 Dt* **have** *m-s3*: *concepts ?Gp* $\vdash_m$ *models ?Gp* $\rightsquigarrow Sm,d{:}\langle ?sdts\ @\ \sigma s'\rangle$
    using *cm-cons* **by** *simp*
  from *s* **have** *s2*: *tys* $S,d{:}\langle ?sdts\ @\ \sigma s'\rangle = Sm,d{:}\langle ?sdts @ \sigma s'\rangle \cup Sv$ **by** *simp*
  from *v-s2 m-s3 s2 tvsg* **show** *?thesis* **by** *auto*
qed

$\wedge$ (*c*,*ci*) $\in$ *C* $\wedge$ *i* $\leq$ *length* (*rfn ci*) $\longrightarrow$ ($\exists\ \tau'$. *dt*$-$*ns*$'$$\rightarrow$$\tau'$ $\wedge$ *C* $\vdash$ $\tau \rightsquigarrow \tau'$)))
(**is** (*C* $\vdash^\flat$ *x c* τ*s ns* $\Rightarrow$ τ *ns*$'$ $\longrightarrow$ *?P C x c* τ*s ns* τ *ns*$'$)
$\wedge$ (*C* $\models^\flat$ *x i c* τ*s ns* $\Rightarrow$ τ *ns*$'$ $\longrightarrow$ *?PS C x i c* τ*s ns* τ *ns*$'$))
proof (*induct rule*: *lookup-mem-lookup-mem-rs.induct*)
fix *C*::*Cenv* **and** τ τ*s c ci i ns x*
assume *cC*: (*c*, *ci*) $\in$ *C* **and** *F*: *lookup x* (*mem-nms ci*) (*mem-tys ci*) *0* = *Some* (τ, *i*)
show *?P C x c* τ*s ns* [*params ci*$\mapsto$$\tau$*s*]$\tau$ (*ns* @ [*length* (*rfn ci*) + *i*])
proof *clarify* **fix** *dt dt*$'$
assume *Cok*: *C ok* **and** *D*: *C* $\vdash_d$ *c* τ*s* $\rightsquigarrow$ *dt*$'$ **and** *P*: *dt*$-$*ns*$\rightarrow$*dt*$'$
from *D Cok cC* **obtain** δ*s* σ*s* τ*s*$'$ **where** *ts-tsp*: *C* $\models$ τ*s* $\rightsquigarrow$ τ*s*$'$
and *Ds*: *C* $\models_d$ *rfn ci* $\rightsquigarrow$ δ*s* **and** *ms-ss*: *C* $\models$ *mem-tys ci* $\rightsquigarrow$ σ*s*
and *ltsp*: *length* τ*s* = *length* (*params ci*)
and *T*: *dt*$'$ = $\langle$\{*params ci*$\mapsto$$\tau$*s*$'$\}($\delta$*s*@$\sigma$*s*)$\rangle$ **using** *inv-r-d2* **by** *blast*
let *?DS* = \{*params ci*$\mapsto$$\tau$*s*$'$\}$\delta$*s* **and** *?SS* = \{*params ci*$\mapsto$$\tau$*s*$'$\}$\sigma$*s*
from *T* **have** *T2*: *dt*$'$ = $\langle$*?DS*@*?SS*$\rangle$ **using** *subst-append* **by** *auto*
from *Cok cC* **have** *C* $\vdash$ *ci ok* **by** (*rule c-mem-implies-c-ok*)
hence *ltn*: *length* (*mem-tys ci*) = *length* (*mem-nms ci*) **by** (*rule inv-wf-c*, *simp*)
from *F* **have** *xms*: *x* $\in$ *set* (*mem-nms ci*) **by** (*rule lookup-found*)
from *xms ltn* **obtain** *i*$'$ **where** *Ip*: *i*$'$ < *length* (*mem-nms ci*)
and *mipt*: (*mem-nms ci*)!*i*$'$ = *x*
and *F2*: *lookup x* (*mem-nms ci*) (*mem-tys ci*) *0* = *Some*((*mem-tys ci*)!*i*$'$,*i*$'$)
using *lookup-succeeds*[*of x mem-nms ci mem-tys ci 0*] **by** *auto*
from *F F2 mipt* **have** *mit*: (*mem-tys ci*)!*i* = τ **by** *auto*
from *F F2 Ip* **have** *I1*: *i* < *length* (*mem-nms ci*) **by** *simp*
from *ms-ss* **have** *length* (*mem-tys ci*) = *length ?SS*
using *trans-length-tys subst-length* **by** *simp*
with *I1 ltn* **have** *I2*: *i* < *length ?SS* **by** *arith*
from *I2 T2 P* **have** *dt*$-$(*ns* @ [*length ?DS* + *i*])$\rightarrow$*?SS*!*i* **by** (*rule dict-path-to-member*)
moreover from *Ds* **have** *length ?DS* = *length* (*rfn ci*)
using *trans-length-r-d subst-length* **by** *auto*
ultimately have *A*: *dt*$-$(*ns* @ [*length* (*rfn ci*) + *i*])$\rightarrow$*?SS*!*i* **by** *simp*
have *B*: *C* $\vdash$ [*params ci*$\mapsto$$\tau$*s*]$\tau$ $\rightsquigarrow$ *?SS*!*i*
proof $-$
from *Cok cC* **have** *dist*: *distinct* (*params ci*)
using *c-mem-implies-c-ok inv-wf-c* **by** *blast*
from *Cok ms-ss dist ltsp ts-tsp* **have** *mss*: *C* $\models$ \{*params ci*$\mapsto$$\tau$*s*\}(*mem-tys ci*) $\rightsquigarrow$ *?SS*
by (*simp only*: *subst-trans-tys*)
have *length* (*mem-tys ci*) = *length* \{*params ci*$\mapsto$$\tau$*s*\}(*mem-tys ci*)
using *substg-length* **by** *simp*
with *I1 ltn* **have** *ilsm*: *i* < *length* \{*params ci*$\mapsto$$\tau$*s*\}(*mem-tys ci*) **by** *arith*
from *mit I1 ltn* **have** *mit2*: (\{*params ci*$\mapsto$$\tau$*s*\}*mem-tys ci*)!*i* = [*params ci*$\mapsto$$\tau$*s*]$\tau$
using *substg-nth* **by** *simp*
from *mss ilsm mit2* **show** *?thesis* **by** (*rule trans-tys-nth*)
qed
from *A B* **show** $\exists\,\tau'$. *dt*$-$(*ns* @ [*length* (*rfn ci*) + *i*])$\rightarrow$$\tau'$ $\wedge$ *C* $\vdash$ [*params ci*$\mapsto$$\tau$*s*]$\tau$ $\rightsquigarrow$ τ'
by *auto*
qed
next
fix *C* τ τ*s c ci ns ns*$'$ *x*
assume *cC*: (*c*, *ci*) $\in$ *C* **and** *F*: *lookup x* (*mem-nms ci*) (*mem-tys ci*) *0* = *None*

and *L*: $C \models^{\flat} x\ \mathit{length}\ (\mathit{rfn}\ \mathit{ci})\ c\ \tau s\ \mathit{ns} \Rightarrow \tau\ \mathit{ns}'$
and *IH*: $\mathit{?PS}\ C\ x\ (\mathit{length}(\mathit{rfn}\ \mathit{ci}))\ c\ \tau s\ \mathit{ns}\ \tau\ \mathit{ns}'$
show $\mathit{?P}\ C\ x\ c\ \tau s\ \mathit{ns}\ \tau\ \mathit{ns}'$
proof *clarify*
fix $\mathit{dt}\ \mathit{dt}'$ **assume** *Cok*: $C\ \mathit{ok}$ **and** *D*: $C \vdash_d c\ \tau s \rightsquigarrow \mathit{dt}'$ **and** *P*: $\mathit{dt}{-}\mathit{ns}{\rightarrow}\mathit{dt}'$
from *D Cok cC* **obtain** $\delta s\ \sigma s\ \tau s'$ **where** *ts-tsp*: $C \models \tau s \rightsquigarrow \tau s'$
and *Ds*: $C \models_d \mathit{rfn}\ \mathit{ci} \rightsquigarrow \delta s$ **and** *ms-ss*: $C \models \mathit{mem\text{-}tys}\ \mathit{ci} \rightsquigarrow \sigma s$
and *ltsp*: $\mathit{length}\ \tau s = \mathit{length}\ (\mathit{params}\ \mathit{ci})$
and *T*: $\mathit{dt}' = \langle\{\mathit{params}\ \mathit{ci} \mapsto \tau s'\}(\delta s @ \sigma s)\rangle$ **using** *inv-r-d2* **by** *blast*
from *Cok D P cC IH* **show** $\exists \tau'.\ \mathit{dt}{-}\mathit{ns}'{\rightarrow}\tau' \wedge C \vdash \tau \rightsquigarrow \tau'$ **by** *blast*
qed
next
fix $C\ \tau\ \tau s\ \tau s'\ c\ c'\ \mathit{ci}\ i\ \mathit{ns}\ \mathit{ns}'\ x$
assume *cC*: $(c, \mathit{ci}) \in C$ **and** *ri*: $\mathit{rfn}\ \mathit{ci}\ !\ i = (c', \tau s')$
and *L*: $C \vdash^{\flat} x\ c'\ \{\mathit{params}\ \mathit{ci}{\mapsto}\tau s\}\tau s'\ \mathit{ns}\ @\ [i] \Rightarrow \tau\ \mathit{ns}'$
and *IH*: $\mathit{?P}\ C\ x\ c'\ \{\mathit{params}\ \mathit{ci}{\mapsto}\tau s\}\tau s'\ (\mathit{ns}@[i])\ \tau\ \mathit{ns}'$
show $\mathit{?PS}\ C\ x\ (\mathit{Suc}\ i)\ c\ \tau s\ \mathit{ns}\ \tau\ \mathit{ns}'$
proof *clarify*
fix $\mathit{dt}\ \mathit{dt}'\ \mathit{cia}$
assume *Cok*: $C\ \mathit{ok}$ **and** *D*: $C \vdash_d c\ \tau s \rightsquigarrow \mathit{dt}'$ **and** *P*: $\mathit{dt}{-}\mathit{ns}{\rightarrow}\mathit{dt}'$
and *ciaC*: $(c, \mathit{cia}) \in C$ **and** *I*: $\mathit{Suc}\ i \leq \mathit{length}\ (\mathit{rfn}\ \mathit{cia})$
from *Cok cC ciaC* **have** *ci-cia*: $\mathit{ci} = \mathit{cia}$ **by** (*rule unique-concept*)
from *D Cok cC* **obtain** $\delta s\ \sigma s\ \tau s''$ **where** *ts-tsp*: $C \models \tau s \rightsquigarrow \tau s''$
and *Ds*: $C \models_d \mathit{rfn}\ \mathit{ci} \rightsquigarrow \delta s$ **and** *ms-ss*: $C \models \mathit{mem\text{-}tys}\ \mathit{ci} \rightsquigarrow \sigma s$
and *lts*: $\mathit{length}\ \tau s = \mathit{length}\ (\mathit{params}\ \mathit{ci})$
and *T*: $\mathit{dt}' = \langle\{\mathit{params}\ \mathit{ci} \mapsto \tau s''\}(\delta s @ \sigma s)\rangle$ **using** *inv-r-d2* **by** *blast*
let $\mathit{?DS} = \{\mathit{params}\ \mathit{ci} \mapsto \tau s''\}\delta s$ **and** $\mathit{?SS} = \{\mathit{params}\ \mathit{ci} \mapsto \tau s''\}\sigma s$
from *T subst-append* **have** *T2*: $\mathit{dt}' = \langle \mathit{?DS}@\mathit{?SS}\rangle$ **by** *auto*
have *D2*: $C \vdash_d c'\ \{\mathit{params}\ \mathit{ci}{\mapsto}\tau s\}\tau s' \rightsquigarrow \mathit{?DS}!i$
proof −
have *sil*: $\mathit{Suc}\ i \leq \mathit{length}\ \delta s$
proof −
from *Ds* **have** $\mathit{length}\ (\mathit{rfn}\ \mathit{ci}) = \mathit{length}\ \delta s$ **by** (*rule trans-length-r-d*)
moreover with *I ci-cia* **have** $\mathit{Suc}\ i \leq \mathit{length}\ (\mathit{rfn}\ \mathit{ci})$ **by** *simp*
ultimately show *?thesis* **by** *simp*
qed
from *Ds ri sil* **obtain** $\mathit{dts}'\ \sigma s'\ \mathit{ci}'$ **where** *cpD*: $C \vdash_d c'\ \tau s' \rightsquigarrow \delta s!i$
and *cpC*: $(c',\mathit{ci}') \in C$ **using** *dict-at-i* **by** *blast*
from *Cok cC* **have** *dist*: $\mathit{distinct}\ (\mathit{params}\ \mathit{ci})$
using *c-mem-implies-c-ok inv-wf-c* **by** *blast*
from *Cok cpD dist lts ts-tsp*
have $C \vdash_d c'\ \{\mathit{params}\ \mathit{ci}{\mapsto}\tau s\}\tau s' \rightsquigarrow [\mathit{params}\ \mathit{ci}{\mapsto}\tau s''](\delta s!i)$ **by** (*simp only*: *subst-r-d*)
moreover from *sil* **have** $\mathit{?DS}!i = [\mathit{params}\ \mathit{ci} \mapsto \tau s''](\delta s!i)$ **by** (*simp only*: *subst-nth*)
ultimately show *?thesis* **by** *simp*
qed
from *Ds ci-cia* **have** $\mathit{length}\ \delta s = \mathit{length}\ (\mathit{rfn}\ \mathit{cia})$ **using** *trans-length-r-d* **by** *simp*
hence $\mathit{length}\ \mathit{?DS} = \mathit{length}\ (\mathit{rfn}\ \mathit{cia})$ **using** *subst-length* **by** *simp*
with *I* **have** *I2*: $i < \mathit{length}\ \mathit{?DS}$ **by** *simp*
from *I2 T2 P* **have** *P2*: $\mathit{dt}{-}\mathit{ns}@[i]{\rightarrow}\mathit{?DS}!i$ **by** (*rule dict-path-to-super*)
from *Cok D2 P2 IH* **show** $\exists \tau'.\ \mathit{dt}{-}\mathit{ns}'{\rightarrow}\tau' \wedge C \vdash \tau \rightsquigarrow \tau'$ **by** *auto*

qed
next
fix $C\ \tau\ \tau s\ c\ i\ ns\ ns'\ x$
assume $C \models^{\flat} x\ i\ c\ \tau s\ ns \Rightarrow \tau\ ns'$
and IH: $\forall dt\ dt'\ ci.\ C\ ok \wedge C \vdash_d c\ \tau s \leadsto dt' \wedge dt{-}ns{\rightarrow}dt' \wedge (c, ci) \in C$
$\wedge\ i \leq length\ (rfn\ ci) \longrightarrow (\exists \tau'.\ dt{-}ns'{\rightarrow}\tau' \wedge C \vdash \tau \leadsto \tau')$
show $\forall dt\ dt'\ ci.\ C\ ok \wedge C \vdash_d c\ \tau s \leadsto dt' \wedge dt{-}ns{\rightarrow}dt' \wedge (c, ci) \in C$
$\wedge\ Suc\ i \leq length\ (rfn\ ci) \longrightarrow (\exists \tau'.\ dt{-}ns'{\rightarrow}\tau' \wedge C \vdash \tau \leadsto \tau')$
proof *clarify*
fix $dt\ dt'\ ci$
assume Cok: $C\ ok$ **and** D: $C \vdash_d c\ \tau s \leadsto dt'$
and P: $dt{-}ns{\rightarrow}dt'$ **and** cC: $(c, ci) \in C$
and I: $Suc\ i \leq length\ (rfn\ ci)$
from I **have** $I2$: $i \leq length\ (rfn\ ci)$ **by** *simp*
from *Cok D P cC I2 IH*
show $\exists \tau'.\ dt{-}ns'{\rightarrow}\tau' \wedge C \vdash \tau \leadsto \tau'$ **by** *auto*
qed
qed

corollary *dict-member*:
$[\![\ C \vdash^{\flat} x\ c\ \tau s\ ns \Rightarrow \tau\ ns';\ C\ ok;\ C \vdash_d c\ \tau s \leadsto dt';\ dt{-}ns{\rightarrow}dt'\ ]\!]$
$\Longrightarrow (\exists\ \tau'.\ dt{-}ns'{\rightarrow}\tau' \wedge C \vdash \tau \leadsto \tau')$
using *dict-member-helper* **apply** *blast* **done**

8.8 Properties of Dictionary Access

There are three places in the translation where the translation must produce System F terms that evaluates to a dictionary. In *fg-tapp*, a list of dictionaries is needed to satisfy the requirements of the where clause of the type abstraction. In the *fg-mdl*, dictionaries corresponding to the refinements in the concept are needed. In *fg-mem*, the dictionary for the specified model must be accessed, and then the appropriate member extracted. The function *mk-nth* is used to construct a System F term to access a dictionary, and the *mk-nths* function constructs a list of terms that access a list of dictionaries. In this section we prove that *mk-nth* and *mk-nths* produce well typed System F terms.

The first lemma states that *mk-nth* produces well typed terms and is a proof by induction on the derivation of the path $\tau{-}ns{\rightarrow}dt$.

lemma *mk-nth-wt*: $\tau{-}ns{\rightarrow}dt \Longrightarrow (\bigwedge S\ de.\ S \vdash_F de : \tau \Longrightarrow S \vdash_F mk\text{-}nth\ de\ ns : dt)$
proof (*induct rule*: *path-ty.induct*)
fix $\tau\ S\ de$ **assume** $S \vdash_F de : \tau$
thus $S \vdash_F mk\text{-}nth\ de\ [] : \tau$ **by** *simp*
next fix $\tau'\ \tau s\ n\ ns\ S\ de$
assume IH: $\bigwedge S\ de.\ S \vdash_F de : \tau s!n \Longrightarrow S \vdash_F mk\text{-}nth\ de\ ns : \tau'$ **and** $d\text{-}wt$: $S \vdash_F de : \langle \tau s \rangle$
from $d\text{-}wt$ **have** $S \vdash_F Nth\ de\ n : \tau s!n$ **by** (*simp add*: *wt-f-nth*)
with IH **show** $S \vdash_F mk\text{-}nth\ de\ (n \# ns) : \tau'$ **by** *simp*
qed

The following lemma is needed to prove that *mk-nths* produces well typed terms. This lemma provides a more convenient way to access the invariants expressed by $C \vdash_m M \rightsquigarrow S$. The proof is by induction on the derivation of $C \vdash_m M \rightsquigarrow S$.

```
lemma model-trans: ⟦ C ⊢m M ⇝ S; (c,τs,d,ns) ∈ M ⟧
   ⟹ (∃ τ τ'. C ⊢d c τs ⇝ τ' ∧ (d, τ) ∈ S ∧ τ−ns→τ')
proof (induct rule: trans-model-env.induct, simp)
 fix C M S τ τsa ca da
 assume IH: (c, τs, d, ns) ∈ M ⟹ ∃ τ τ'. C ⊢d c τs ⇝ τ' ∧ (d, τ) ∈ S ∧ path-ty τ ns τ'
  and D: C ⊢d ca τsa ⇝ τ and M: (c, τs, d, ns) ∈ insert (ca, τsa, da, []) M
 show ∃ τa τ'. C ⊢d c τs ⇝ τ' ∧ (d, τa) ∈ S,da:τ ∧ path-ty τa ns τ'
 proof (cases (c, τs, d, ns) = (ca, τsa, da, []))
  assume eq: (c, τs, d, ns) = (ca, τsa, da, [])
  from eq D have D2: C ⊢d c τs ⇝ τ by simp
  from eq have dt: (d, τ) ∈ S,da:τ by simp
  from eq have P: τ−ns→τ using p-nil by simp
  from D2 dt P show ?thesis by auto
 next assume neq: (c, τs, d, ns) ≠ (ca, τsa, da, [])
  from neq M have M2: (c, τs, d, ns) ∈ M by auto
  from M2 IH show ?thesis by auto
 qed
next fix C M S τ τ' τsa ca da nsa
 assume C ⊢m M ⇝ S and IH: (c, τs, d, ns) ∈ M ⟹
    ∃ τ τ'. C ⊢d c τs ⇝ τ' ∧ (d, τ) ∈ S ∧ τ−ns→τ'
  and nsa ≠ [] and dt: (da, τ) ∈ S and D: C ⊢d ca τsa ⇝ τ'
  and P: τ−nsa→τ' and M: (c, τs, d, ns) ∈ insert (ca, τsa, da, nsa) M
 show ∃ τ τ'. C ⊢d c τs ⇝ τ' ∧ (d, τ) ∈ S ∧ path-ty τ ns τ'
 proof (cases (c, τs, d, ns) = (ca, τsa, da, nsa))
  assume eq: (c, τs, d, ns) = (ca, τsa, da, nsa)
  from eq D have D2: C ⊢d c τs ⇝ τ' by simp
  from eq dt have dt2: (d,τ) ∈ S by simp
  from eq P have P2: τ−ns→τ' by simp
  from D2 dt2 P2 show ?thesis by auto
 next assume neq: (c, τs, d, ns) ≠ (ca, τsa, da, nsa)
  from neq M have M2: (c, τs, d, ns) ∈ M by auto
  from M2 IH show ?thesis by auto
 qed
qed
```

The proof of Lemma *mk-nths-wt*, that *mk-nths* produces well typed terms, is by induction on the derivation of the translation $M \models ws \rightsquigarrow ds, nns$.

```
lemma mk-nths-wt: M ⊨ ws ⇝ ds, nns ⟹ (⋀ T C V S dts. ⟦ C ok;
   (|tyvars = T, vars = V, concepts = C, models = M|) ⇝ S; C ⊨d ws ⇝ dts ⟧
   ⟹ S ⊨F (mk-nths ds nns) : dts)
proof (induct rule: fg-where.induct)
 fix Γ T C V S dts
 assume Ds: C ⊨d [] ⇝ dts
 from Ds have dts = [] by (rule inv-rs-ds-nil, simp)
 also have S ⊨F mk-nths [] [] : [] by (simp add: wt-f-nil)
 ultimately show S ⊨F mk-nths [] [] : dts by simp
next fix M τs c d ds nns ns ws T C V S dts
```

assume M: $(c, \tau s, d, ns) \in M$ **and** W: $M \models ws \leadsto ds, nns$
and IH: $\bigwedge T\ C\ V\ S\ dts.\ [\![\ C\ ok;\ (\!|tyvars = T, vars = V, concepts = C, models = M|\!) \leadsto S;$
$C \models_d ws \leadsto dts\ ]\!] \Longrightarrow S \models_F mk\text{-}nths\ ds\ nns : dts$
and Cok: $C\ ok$ **and** $g\text{-}s$: $(\!|tyvars = T, vars = V, concepts = C, models = M|\!) \leadsto S$
and D: $C \models_d (c,\tau s)\#ws \leadsto dts$
from $g\text{-}s$ **obtain** $Sv\ Sm$ **where** T: $C \vdash_m M \leadsto Sm$ **and** TV: $tvars\ S = T$
and S: $tys\ S = Sm \cup Sv$ **by** *auto*
from $M\ T\ model\text{-}trans$ **obtain** $\tau\ \tau'$ **where** $D2$: $C \vdash_d c\ \tau s \leadsto \tau'$
and $dt\text{-}sm$: $(d,\tau) \in Sm$ **and** P: $\tau{-}ns{\rightarrow}\tau'$ **by** *blast*
from $dt\text{-}sm\ S$ **have** $dt\text{-}s$: $(d,\tau) \in tys\ S$ **by** *simp*
from $dt\text{-}s$ **have** $wt\text{-}d$: $S \vdash_F$ ‘$d : \tau$ **by** $(rule\ wt\text{-}f\text{-}var)$
from $P\ wt\text{-}d$ **have** A: $S \vdash_F mk\text{-}nth\ (‘d)\ ns : \tau'$ **by** $(rule\ mk\text{-}nth\text{-}wt)$
from D **obtain** $dt\ dts'$ **where** Dt: $C \vdash_d c\ \tau s \leadsto dt$ **and** Ds: $C \models_d ws \leadsto dts'$
and dts: $dts = dt\#dts'$ **by** $(rule\ inv\text{-}rs\text{-}ds\text{-}cons, auto)$
from $D2\ Cok\ Dt$ **have** $\tau' = dt$ **using** *fun-dict-trans-ty* **apply** *blast* **done**
with dts **have** $dts2$: $dts = \tau'\#dts'$ **by** *simp*
from $Cok\ g\text{-}s\ Ds\ IH$ **have** B: $S \models_F mk\text{-}nths\ ds\ nns : dts'$ **by** *simp*
from $A\ B$ **have** $S \models_F (mk\text{-}nth\ (‘d)\ ns)\#(mk\text{-}nths\ ds\ nns) : \tau'\#dts'$ **by** $(rule\ wt\text{-}f\text{-}cons)$
with $dts2$ **have** $S \models_F (mk\text{-}nth\ (‘d)\ ns)\#(mk\text{-}nths\ ds\ nns) : dts$ **by** *simp*
thus $S \models_F mk\text{-}nths\ (d\ \#\ ds)\ (ns\ \#\ nns) : dts$ **by** *simp*
qed

8.9 The Main Theorem

The main theorem, that the translation produces well-typed terms of System F, is proved by mutual induction on derivations of $\Gamma \vdash e : \tau \leadsto f$ and of $\Gamma \models es : \tau s \leadsto fs$. Comments are embedded in the proof that summarize the main points of each sub-case.

theorem *fg-pres-ty*:
$(\Gamma \vdash e : \tau \leadsto f \longrightarrow$
$(\forall\ S.\ concepts\ \Gamma\ ok \wedge \Gamma \leadsto S \longrightarrow (\exists\ \tau'.\ S \vdash_F f : \tau' \wedge concepts\ \Gamma \vdash \tau \leadsto \tau')))$
$\wedge\ (\Gamma \models es : \tau s \leadsto fs \longrightarrow$
$(\forall\ S.\ concepts\ \Gamma\ ok \wedge \Gamma \leadsto S \longrightarrow (\exists\ \tau s'.\ S \models_F fs : \tau s' \wedge concepts\ \Gamma \models \tau s \leadsto \tau s')))$
(**is** $(\Gamma \vdash e : \tau \leadsto f \longrightarrow ?P\ \Gamma\ \tau\ f) \wedge (\Gamma \models es : \tau s \leadsto fs \longrightarrow ?PS\ \Gamma\ \tau s\ fs)$)
proof $(induct\ rule: fg\text{-}fg\text{-}list.induct)$
— Case *fg-tabs*: The sub-term e is translated in an environment extended with models for each requirement in the where clause. We use the lemma from Figure 20 to show that the environment correspondence holds for the extended environment. We then invoke the induction hypothesis for $\Gamma(\!|models := M|\!) \vdash e : \sigma \leadsto f$ and assemble the typing derivation for the output term $\Lambda\ ts.\ (\lambda\ ds{:}\tau s.\ f)$.
fix $M\ \Gamma\ \sigma\ \tau s\ ds\ e\ f$ **and** $ts{::}var\ list$ **and** ws
assume Ds:$concepts\ \Gamma \models_d ws \leadsto \tau s$ **and** M: $concepts\ \Gamma \vdash ws\ ds\ (models\ \Gamma) \Rightarrow M$
and $dist$: $distinct\ ts$ **and** $e\text{-}f$: $\Gamma(\!|models := M|\!)(\!|tyvars := tyvars\ \Gamma \cup set\ ts|\!) \vdash e : \sigma \leadsto f$
and IH: $?P\ (\Gamma(\!|models := M|\!)(\!|tyvars := tyvars\ \Gamma \cup set\ ts|\!))\ \sigma\ f$
show $?P\ \Gamma\ (\forall\ ts\ where\ ws.\ \sigma)\ (\Lambda\ ts.\ (\lambda\ ds{:}\tau s.\ f))$
proof *clarify*
fix S **assume** Cok: $concepts\ \Gamma\ ok$ **and** $g\text{-}s$: $\Gamma \leadsto S$
from $g\text{-}s$ **obtain** $Sv\ Sm$ **where** $v\text{-}s$: $concepts\ \Gamma \vdash_v vars\ \Gamma \leadsto Sv$
and $m\text{-}s$: $concepts\ \Gamma \vdash_m models\ \Gamma \leadsto Sm$ **and** sv: $tvars\ S = tyvars\ \Gamma$

and *s-svm*: *tys S* $=$ *Sm* $\cup$ *Sv* **by** *auto*
from *M Cok Ds m-s* **have** *mp-sd*: *concepts* $\Gamma \vdash_m M \rightsquigarrow$ *Sm*,*ds*:τs $\wedge$ *length ds* $=$ *length* τs
by (*rule add-models-where-preserves*)
let *?Gp* $= \Gamma(\!|$ *models* $:=$ *M* $|\!)(\!|$ *tyvars* $:=$ *tyvars* $\Gamma \cup$ *set ts*$|\!)$
and *?Sp* $= (\!|$*tys* $=$ (*Sm* $\cup$ *Sv*),*ds*:τs, *tvars* $=$ *tvars S* $\cup$ *set ts*$|\!)$
have *eq*: (*Sm*,*ds*:τs) $\cup$ *Sv* $=$ (*Sm* $\cup$ *Sv*),*ds*:τs **by** (*simp only*: *push-union-commute*)
from *sv v-s mp-sd* **have** *?Gp* $\rightsquigarrow (\!|$*tys* $=$ (*Sm*,*ds*:τs) $\cup$ *Sv*, *tvars* $=$ *tvars S* $\cup$ *set ts*$|\!)$ **by** *auto*
with *eq* **have** *gp-sp*: *?Gp* $\rightsquigarrow$ *?Sp* **by** *simp*
from *Cok* **have** *Gpok*: *concepts ?Gp ok* **by** *simp*
from *Gpok gp-sp IH* **obtain** τ' **where** *wt-f*: *?Sp* $\vdash_F f : \tau'$ **and** *s-tp*: *concepts ?Gp* $\vdash \sigma \rightsquigarrow \tau'$
by *blast*
from *wt-f* **have** *ft*: *?Sp* $\vdash_F f : \tau'$ **by** *simp*
let *?Sp2* $= (\!|$*tys* $=$ *Sm*$\cup$*Sv*, *tvars* $=$ *tvars S* $\cup$ *set ts*$|\!)$
from *ft* **have** *wtf*: *?Sp2*$(\!|$*tys* $:=$ (*tys ?Sp2*),*ds*:$\tau s|\!) \vdash_F f : \tau'$ **by** *simp*
have *dsty*: *set ds* $\cap$ *dom* (*tys ?Sp2*) $= \{\}$ **sorry** — Can alpha-convert to get this
from *wtf mp-sd dsty* **have** *wtlf*: *?Sp2* $\vdash_F \lambda$ *ds*:τs. f : *fn* $\tau s \rightarrow \tau'$ **using** *wt-f-abs* **by** *auto*
let *?Sp3* $= (\!|$*tys* $=$ *Sm*$\cup$*Sv*, *tvars* $=$ *tvars S*$|\!)$
from *wtlf* **have** *wtlf2*: *?Sp3*$(\!|$ *tvars* $:=$ *tvars ?Sp3* $\cup$ *set ts*$|\!) \vdash_F \lambda$ *ds*:τs. f : *fn* $\tau s \rightarrow \tau'$ **by** *simp*
have *tstsp*: *set ts* $\cap$ *tvars ?Sp3* $= \{\}$ **sorry** — alpha-convert to get this
have *tsfs*: *set ts* $\cap$ *FTV* (*tys ?Sp3*) $= \{\}$ **sorry** — alpha-convert to get this
from *wtlf2 tstsp tsfs dist* **have** *sp3*: *?Sp3* $\vdash_F$ (Λ *ts*. (λ *ds*:τs. f)) : ($\forall$ *ts*. *fn* $\tau s \rightarrow \tau'$)
by (*rule wt-f-tabs*)
from *s-svm* **have** $S =$ *?Sp3* **by** *simp*
with *sp3* **have** *A*: $S \vdash_F$ (Λ *ts*. (λ *ds*:τs. f)) : ($\forall$ *ts*. *fn* $\tau s \rightarrow \tau'$) **by** *auto*
from *s-tp* **have** *s-tp2*: *concepts* $\Gamma \vdash \sigma \rightsquigarrow \tau'$ **by** *simp*
from *Ds s-tp2 dist* **have** *B*: *concepts* $\Gamma \vdash \forall$ *ts where ws*. $\sigma \rightsquigarrow$ ($\forall$ *ts*. *fn* $\tau s \rightarrow \tau'$)
by (*rule trans-all*)
from *A B* **show** ($\exists\ \tau'$. $S \vdash_F \Lambda$ *ts*. (λ *ds*:τs. f) : $\tau' \wedge$ *concepts* $\Gamma \vdash \forall$ *ts where ws*. $\sigma \rightsquigarrow \tau'$)
by *auto*
qed

next — Case *fg-tapp*: We must show that the output term, which is the application $f[\tau s'] \cdot$ *mk-nths ds nns* is well typed. We use the induction hypothesis to show that f is well typed and Lemma *mk-nths-wt* from Section 8.8 to show that the result of *mk-nths* is well typed.

fix $\Gamma\ \sigma\ \tau s\ \tau s'$ *ds e f nns ts ws*
assume *e-f*: $\Gamma \vdash e : \forall$ *ts where ws*. $\sigma \rightsquigarrow f$ **and** *IH*: *?P* Γ ($\forall$ *ts where ws*. σ) f
and *lts*: *length ts* $=$ *length* τs **and** *Ws*: *models* $\Gamma \models \{\!|$*ts*$\mapsto\tau s|\!\}$*ws* $\rightsquigarrow$ *ds*, *nns*
and *ts-tsp*: *concepts* $\Gamma \models \tau s \rightsquigarrow \tau s'$
show *?P* Γ ([*ts*$\mapsto\tau s$]σ) ($f[\tau s'] \cdot$ *mk-nths ds nns*)
proof *clarify*
fix *S* **assume** *Cok*: *concepts* Γ *ok* **and** *g-s*: $\Gamma \rightsquigarrow S$
from *Cok g-s IH* **obtain** τ' **where** *wt-f*: $S \vdash_F f : \tau'$
and *alls-tp*: *concepts* $\Gamma \vdash \forall$ *ts where ws*. $\sigma \rightsquigarrow \tau'$ **by** *blast*
from *alls-tp* **obtain** $\tau''\ \sigma s$ **where** *Rs*: *concepts* $\Gamma \models_d$ *ws* $\rightsquigarrow \sigma s$
and *s-tpp*: *concepts* $\Gamma \vdash \sigma \rightsquigarrow \tau''$ **and** *dist*: *distinct ts*
and *tp*: $\tau' = \forall$ *ts*. *fn* $\sigma s \rightarrow \tau''$ **by** (*rule inv-trans-all2*, *simp*)
from *wt-f tp* **have** *wt-f2*: $S \vdash_F f : \forall$ *ts*. *fn* $\sigma s \rightarrow \tau''$ **by** *simp*
from *ts-tsp* **have** *length* $\tau s =$ *length* $\tau s'$ **by** (*simp add*: *trans-length*)
with *lts* **have** *ltsp*: *length ts* $=$ *length* $\tau s'$ **by** *simp*
from *wt-f2 ltsp* **have** $S \vdash_F f[\tau s']$: [*ts*$\mapsto\tau s'$](*fn* $\sigma s \rightarrow \tau''$) **by** (*rule wt-f-tapp*)
hence *A*: $S \vdash_F f[\tau s']$: (*fn* (*sub-tys ts* $\tau s'\ \sigma s$) $\rightarrow$ ([*ts*$\mapsto\tau s'$]τ'')) **by** *simp*

from *Rs Cok dist lts ts-tsp* **have** *Rs2*: *concepts* $\Gamma \models_d \{\!|ts \mapsto \tau s|\!\}ws \rightsquigarrow \{ts \mapsto \tau s'\}\sigma s$
by (*rule subst-ds*)
from *Ws Cok g-s Rs2* **have** *B*: $S \models_F$ *mk-nths ds nns* : $\{ts \mapsto \tau s'\}\sigma s$ **by** (*simp add*: *mk-nths-wt*)
have *eq*: *id* $\models_F \{ts \mapsto \tau s'\}\sigma s = \{ts \mapsto \tau s'\}\sigma s$ **by** (*rule f-eqs-refl*)
from *A B eq* **have** *C*: $S \vdash_F (f[\tau s'] \cdot$ *mk-nths ds nns*) : $[ts \mapsto \tau s']\tau''$ **by** (*rule wt-f-app*)
from *s-tpp Cok dist lts ts-tsp* **have** *D*: *concepts* $\Gamma \vdash [ts \mapsto \tau s]\sigma \rightsquigarrow [ts \mapsto \tau s']\tau''$
by (*rule subst-trans-ty*)
from *C D* **show** $\exists \tau'.\ S \vdash_F f[\tau s'] \cdot$ *mk-nths ds nns* : $\tau' \wedge$
concepts $\Gamma \vdash [ts \mapsto \tau s]\sigma \rightsquigarrow \tau'$ **by** *blast*
qed

next — Case *fg-cpt*: The sub-term *e* is translated in an environment extended with the new concept. To invoke the induction hypothesis we must show that the new environment corresponds to a System F environment, which is handled by the lemmas from Section 8.6. From the induction hypothesis we get $\{(c, ci)\} \cup$ *concepts* $\Gamma \vdash \tau \rightsquigarrow \tau'$, from which we have *concepts* $\Gamma \vdash \tau \rightsquigarrow \tau'$ because *c* is not permitted to appear in τ.

fix *C* Γ **and** σs::*tyg list* **and** $\sigma s'\ \tau\ \tau s\ c$ **and** *ci*::*concept-info*
and *e f* **and** *rs*::*where-clause* **and** *ts xs*
assume *CD*: $c \notin$ *dom* (*concepts* Γ) **and** *R*: *concepts* $\Gamma \models_d rs \rightsquigarrow \tau s$
and *ss-ssp*: *concepts* $\Gamma \models \sigma s \rightsquigarrow \sigma s'$
and *CI*: *ci* = (|*params* = *ts*, *rfn* = *rs*, *mem-nms* = *xs*, *mem-tys* = σs|)
and *e-f*: (Γ,*concept c ci*) $\vdash e : \tau \rightsquigarrow f$ **and** *IH*: *?P* (Γ,*concept c ci*) τ *f*
and *lxs*: *length xs* = *length* σs **and** *dist*: *distinct ts*
and *frs*: $\bigcup$ (*map* ($\lambda p.\ \bigcup$ (*map ftvg* (*snd p*))) *rs*) $\subseteq$ *set ts*
and *fms*: $\bigcup$ (*map ftvg* σs) $\subseteq$ *set ts*
and *O*: $(c,\tau) \notin$ *c-occurs-ty*
show *?P* $\Gamma\ \tau$ *f*
proof *clarify*
fix *S* **assume** *Cok*: *concepts* Γ *ok* **and** *g-s*: $\Gamma \rightsquigarrow S$
have *Cok2*: *concepts* (Γ,*concept c ci*) *ok*
proof *simp*
from *R ss-ssp dist lxs CI frs fms* **have** *CIok*: *concepts* $\Gamma \vdash$ *ci ok* **by** (*simp add*: *wf-c*)
from *CD CIok Cok* **show** *insert* (*c*,*ci*) (*concepts* Γ) *ok* **by** (*simp add*: *wf-cs-cons*)
qed
from *g-s* **obtain** *Sv Sm* **where** *v-s*: *concepts* $\Gamma \vdash_v$ *vars* $\Gamma \rightsquigarrow Sv$
and *m-s*: *concepts* $\Gamma \vdash_m$ *models* $\Gamma \rightsquigarrow Sm$ **and** *sv*: *tvars S* = *tyvars* Γ
and *s-svm*: *tys S* = *Sv* $\cup$ *Sm* **by** *auto*
from *v-s* **have** *v-s2*: *concepts* (Γ,*concept c ci*) $\vdash_v$ *vars* $\Gamma \rightsquigarrow Sv$
using *add-concept-preserves-var-env* **by** *simp*
from *m-s* **have** *m-s2*: *concepts* (Γ,*concept c ci*) $\vdash_m$ *models* $\Gamma \rightsquigarrow Sm$
using *add-concept-preserves-model-env* **by** *simp*
from *sv v-s2 m-s2 s-svm* **have** *g-s2*: Γ,*concept c ci* $\rightsquigarrow S$ **by** *auto*
from *Cok2 g-s2 IH* **obtain** τ' **where** *wt-f*: $(S, f, \tau') \in$ *wt-f*
and *t-tp*: *concepts* (Γ,*concept c ci*) $\vdash \tau \rightsquigarrow \tau'$ **by** *blast*
from *t-tp* **have** *t-tpb*: *insert* (*c*, *ci*) (*concepts* Γ) $\vdash \tau \rightsquigarrow \tau'$ **by** *simp*
from *t-tpb O* **have** *t-tp2*: *concepts* $\Gamma \vdash \tau \rightsquigarrow \tau'$
by (*rule remove-concept-pres-trans-ty*)
from *wt-f t-tp2* **show** $\exists \tau'.\ (S, f, \tau') \in$ *wt-f* $\wedge$ *concepts* $\Gamma \vdash \tau \rightsquigarrow \tau'$ **by** *blast*
qed

next — Case *fg-mdl*: The output term will be (*let d* := *de in f*), where *de* is the term for the dictionary for the model. We use Lemma *mk-nths-wt* to show that the part of the dictionary for refinements is well typed. We will use the induction hypothesis to get a well-typed *f*. However, we first show that adding the model to the environment preserves the environment correspondence. We invoke Lemma *add-model-preserves* to prove this.

fix Γ *ϱs ϱs′ σs τ c ci d de ds dts e es f fs ns xs*
assume *C*: $(c, ci) \in$ *concepts* Γ **and** *rs-rsp*: *concepts* $\Gamma \models$ *ϱs* $\rightsquigarrow$ *ϱs′*
and *memns*: *xs* = *mem-nms ci* **and** *es-fs*: $\Gamma \models$ *es* : *σs* $\rightsquigarrow$ *fs*
assume *IH1*: *?PS* Γ *σs fs* **and** *memtys*: *σs* = {*params ci*↦*ϱs*}(*mem-tys ci*)
and *Ds*: *concepts* $\Gamma \models_d$ *rfn ci* $\rightsquigarrow$ *dts*
assume *W*: *models* $\Gamma \models$ ⦃*params ci*↦*ϱs*⦄*rfn ci* $\rightsquigarrow$ *ds*, *ns*
and *D*: *de* = ⟨*mk-nths ds ns* @ *fs*⟩ **and** *lps*: *length* (*params ci*) = *length ϱs*
and *IH2*: *?P* (Γ,*model* (*c*,*ϱs*,*d*,[])) *τ f*
let *?Gp* = Γ,*model* (*c*, *ϱs*, *d*, [])
show *?P* Γ *τ* (*let d* := *de in f*)
proof *clarify*
fix *S* **assume** *Cok*: *concepts* Γ *ok* **and** *g-s*: $\Gamma \rightsquigarrow S$
from *Cok g-s IH1* **obtain** *σs′* **where**
wt-fs: $S \models_F$ *fs* : *σs′* **and** *ss-ssp*: *concepts* $\Gamma \models$ *σs* $\rightsquigarrow$ *σs′* **by** *blast*
from *Cok C* **have** *dist*: *distinct* (*params ci*)
using *c-mem-implies-c-ok inv-wf-c* **by** *blast*
let *?sdts* = {*params ci*↦*ϱs′*}*dts*
from *Ds Cok dist lps rs-rsp* **have**
Ds2: *concepts* $\Gamma \models_d$ ⦃*params ci*↦*ϱs*⦄(*rfn ci*) $\rightsquigarrow$ *?sdts* **by** (*rule subst-ds*)
from *W Cok g-s Ds2* **have**
wt-mk: $S \models_F$ *mk-nths ds ns* : *?sdts* **by** (*simp add*: *mk-nths-wt*)
from *wt-mk wt-fs* **have** $S \models_F$ (*mk-nths ds ns*) @ *fs* : *?sdts* @ *σs′*
by (*simp add*: *wt-f-append*)
hence $S \vdash_F$ ⟨*mk-nths ds ns* @ *fs*⟩ : ⟨*?sdts* @ *σs′*⟩ **by** (*rule wt-f-tuple*)
with *D* **have** *wt-de*: $S \vdash_F$ *de* : ⟨*?sdts* @ *σs′*⟩ **by** *simp*
from *Cok* **have** *Cok2*: *concepts ?Gp ok* **by** *simp*
let *?Sp* = *S*⦇*tys* := (*tys S*),*d*:⟨*?sdts* @ *σs′*⟩⦈
from *g-s Cok C rs-rsp Ds ss-ssp memtys lps*
have *g2-s*: *?Gp* $\rightsquigarrow$ *?Sp* **by** (*rule add-model-preserves*)
from *Cok2 g2-s IH2* **obtain** *τ′* **where** *wt-f*: *?Sp* $\vdash_F$ *f* : *τ′*
and *t-tp*: *concepts* (Γ,*model* (*c*,*ϱs*,*d*,[])) $\vdash$ *τ* $\rightsquigarrow$ *τ′* **by** *blast*
have *dS*: $d \notin$ *dom* (*tys S*) **sorry** — d is fresh
from *wt-de wt-f dS* **have** *A*: $S \vdash_F$ *let d* := *de in f* : *τ′* **by** (*rule wt-f-let*)
from *t-tp* **have** *B*: *concepts* $\Gamma \vdash$ *τ* $\rightsquigarrow$ *τ′* **by** *simp*
from *A B* **show** $\exists$ *τ′*. (*S*, *let d* := *de in f*, *τ′*) $\in$ *wt-f* $\wedge$ *concepts* $\Gamma \vdash$ *τ* $\rightsquigarrow$ *τ′* **by** *auto*
qed

next — Case *fg-mem*: We take advantage of the environment correspondence $\Gamma \rightsquigarrow S$ to obtain the path *σ*−*ns*→*dt* from the dictionary *d* to the appropriate sub-dictionary for this model. We then use Lemma *dict-member* from Section 8.7 to extend the path to the appropriate member. Lemma *mk-nth-wt* shows that *mk-nth* (‘*d*) *ns′* is well typed.

fix Γ::*FGenv* **and** *τ τs c d ns ns′ x*
assume *M*: (*c*, *τs*, *d*, *ns*) $\in$ *models* Γ **and** *F*: *concepts* $\Gamma \vdash^\flat$ *x c τs ns* $\Rightarrow$ *τ ns′*
show *?P* Γ *τ* (*mk-nth* (‘*d*) *ns′*)
proof *clarify*
fix *S* **assume** *Cok*: *concepts* Γ *ok* **and** *g-s*: $\Gamma \rightsquigarrow S$
from *g-s* **obtain** *Sv Sm* **where** *v-s*: *concepts* $\Gamma \vdash_v$ *vars* $\Gamma \rightsquigarrow$ *Sv*
and *m-s*: *concepts* $\Gamma \vdash_m$ *models* $\Gamma \rightsquigarrow$ *Sm* **and** *sv*: *tvars S* = *tyvars* Γ

and *s-svm*: *tys* $S = Sv \cup Sm$ **by** *auto*
from *M m-s model-trans* **obtain** σ *dt* **where** *D*: *concepts* $\Gamma \vdash_d c\ \tau s \rightsquigarrow dt$
and *DS*: $(d,\sigma) \in Sm$ **and** *P*: $\sigma{-}ns{\rightarrow}dt$ **by** *blast*
from *DS s-svm* **have** *DS2*: $(d,\sigma) \in tys\ S$ **by** *auto*
from *F Cok D P dict-member* **obtain** τ' **where** *P2*: $\sigma{-}ns'{\rightarrow}\tau'$
and *t-tp*: *concepts* $\Gamma \vdash \tau \rightsquigarrow \tau'$ **by** *blast*
from *DS2* **have** *wt-d*: $S \vdash_F$ ‘$d : \sigma$ **by** (*rule wt-f-var*)
from *P2 wt-d* **have** *wt-nth*: $S \vdash_F$ *mk-nth* (‘d) $ns' : \tau'$ **by** (*rule mk-nth-wt*)
from *wt-nth t-tp* **show**
$\exists\,\tau'.\ (S,$ *mk-nth* (‘d) $ns', \tau') \in$ *wt-f* $\wedge$ *concepts* $\Gamma \vdash \tau \rightsquigarrow \tau'$ **by** *auto*
qed

next — Case *fg-var*: Again we rely on the environment correspondence $\Gamma \rightsquigarrow S$. This time we use it to obtain the translation of type τ for variable x.
fix Γ::*FGenv* **and** τ x **assume** *XT*: $(x,\tau) \in vars\ \Gamma$
show *?P* Γ τ (‘x)
proof *clarify*
fix S **assume** *Cok*: *concepts* Γ *ok* **and** *g-s*: $\Gamma \rightsquigarrow S$
from *g-s* **obtain** *Sv Sm* **where** *v-s*: *concepts* $\Gamma \vdash_v vars\ \Gamma \rightsquigarrow Sv$
and *m-s*: *concepts* $\Gamma \vdash_m models\ \Gamma \rightsquigarrow Sm$ **and** *sv*: *tvars* S = *tyvars* Γ
and *s-svm*: *tys* $S = Sv \cup Sm$ **by** *auto*
from *v-s XT var-mem-trans-implies* **obtain** τ' **where**
t-tp: *concepts* $\Gamma \vdash \tau \rightsquigarrow \tau'$ **and** *XTP*: $(x,\tau') \in Sv$ **by** *blast*
from *XTP s-svm* **have** *XTP2*: $(x,\tau') \in tys\ S$ **by** *simp*
from *XTP2* **have** *wt-x*: $S \vdash_F$ ‘$x : \tau'$ **by** (*rule wt-f-var*)
from *wt-x t-tp* **show** $\exists\,\tau'.\ S \vdash_F$ ‘$x : \tau' \wedge$ *concepts* $\Gamma \vdash \tau \rightsquigarrow \tau'$ **by** *auto*
qed

next — Case *fg-app*: This case is straightforward.
fix Γ σs $\sigma s'$ τ e *es* f *fs* **assume** *IH1*: *?P* Γ (*fn* $\sigma s \rightarrow \tau$) f **and** *IH2*: *?PS* Γ $\sigma s'$ *fs*
and *ss-sp*: $id \models \sigma s = \sigma s'$
show *?P* Γ τ ($f \cdot fs$)
proof *clarify*
fix S **assume** *Cok*: *concepts* Γ *ok* **and** *g-s*: $\Gamma \rightsquigarrow S$
from *Cok g-s IH1* **obtain** τ' **where** *wt-f*: $S \vdash_F f : \tau'$
and *t-tp*: *concepts* $\Gamma \vdash fn\ \sigma s \rightarrow \tau \rightsquigarrow \tau'$ **by** *blast*
from *Cok g-s IH2* **obtain** $\tau s'$ **where** *wt-fs*: $S \models_F fs : \tau s'$
and *ss-tp*: *concepts* $\Gamma \models \sigma s' \rightsquigarrow \tau s'$ **by** *blast*
from *t-tp* **obtain** τ'' $\tau s''$ **where** *ss-tpp*: *concepts* $\Gamma \models \sigma s \rightsquigarrow \tau s''$
and *s-tpp*: *concepts* $\Gamma \vdash \tau \rightsquigarrow \tau''$ **and** *tp*: $\tau' = fn\ \tau s'' \rightarrow \tau''$
by (*rule inv-trans-fun*, *blast*)
from *tp wt-f* **have** *wt-f2*: $S \vdash_F f : fn\ \tau s'' \rightarrow \tau''$ **by** *simp*
— Need to change lemma *fun-dict-trans-ty* to take into accound alpha-equal types
from *Cok ss-tp ss-tpp ss-sp* **have** *eq*: $id \models_F \tau s' = \tau s''$ **using** *fun-dict-trans-ty* **sorry**
from *eq* **have** *eq2*: $id \models_F \tau s'' = \tau s'$ **by** (*rule f-eqs-symm*)
from *wt-fs eq* **have** *wt-fs2*: $S \models_F fs : \tau s''$ **by** (*rule equal-preserves-wts*)
from *wt-f2 wt-fs eq2* **have** *wt-ap*: $S \vdash_F f \cdot fs : \tau''$ **by** (*rule wt-f-app*)
from *s-tpp wt-ap* **show** $\exists\,\tau'.\ S \vdash_F f \cdot fs : \tau' \wedge$ *concepts* $\Gamma \vdash \tau \rightsquigarrow \tau'$ **by** *auto*
qed

next — Case *fg-abs*: In this case the sub-term is translated in an environment extended with variable bindings for the parameters. We use a lemma from Section 8.6 to show that the environment correspondence is maintained.

fix Γ σs $\sigma s'$ τ *e f xs* **assume** *IH*: *?P* $(\Gamma, xs{:}\sigma s)$ τ *f* **and** *ss-ssp*: *concepts* $\Gamma \models \sigma s \rightsquigarrow \sigma s'$
and *lxs*: *length xs* = *length* σs
from *ss-ssp* **have** *length* σs = *length* $\sigma s'$ **by** (*simp add*: *trans-length*)
with *lxs* **have** *lxs2*: *length xs* = *length* $\sigma s'$ **by** *simp*
show *?P* Γ (*fn* $\sigma s \rightarrow \tau$) ($\lambda$ *xs*:$\sigma s'$. *f*)
proof *clarify*
fix *S* **assume** *Cok*: *concepts* Γ *ok* **and** *g-s*: $\Gamma \rightsquigarrow S$
have *eq*: *concepts* $(\Gamma, xs{:}\sigma s)$ = *concepts* Γ **by** (*simp add*: *push-vars-def*)
have *meq*: *models* $(\Gamma, xs{:}\sigma s)$ = *models* Γ **by** (*simp add*: *push-vars-def*)
from *g-s* **obtain** *Sv Sm* **where** *v-s*: *concepts* $\Gamma \vdash_v$ *vars* $\Gamma \rightsquigarrow$ *Sv*
and *m-s*: *concepts* $\Gamma \vdash_m$ *models* $\Gamma \rightsquigarrow$ *Sm* **and** *sv*: *tvars S* = *tyvars* Γ
and *s-svm*: *tys S* = *Sv* $\cup$ *Sm* **by** *auto*
from *ss-ssp v-s lxs* **have** *concepts* $\Gamma \vdash_v$ (*vars* Γ),*xs*:$\sigma s \rightsquigarrow$ *Sv*,*xs*:$\sigma s'$
using *add-vars-preserves-var-env* **by** *simp*
with *eq* **have** *v-s2*: *concepts* $(\Gamma, xs{:}\sigma s) \vdash_v$ (*vars* Γ),*xs*:$\sigma s \rightsquigarrow$ *Sv*,*xs*:$\sigma s'$ **by** *simp*
from *m-s eq meq* **have** *m-s2*: *concepts* $(\Gamma, xs{:}\sigma s) \vdash_m$ *models* $(\Gamma, xs{:}\sigma s) \rightsquigarrow$ *Sm* **by** *simp*
have (*Sv*,*xs*:$\sigma s'$) $\cup$ *Sm* = (*Sv* $\cup$ *Sm*),*xs*:$\sigma s'$ **using** *push-union-commute* **by** *simp*
hence *s-svm2*: (*Sv* $\cup$ *Sm*),*xs*:$\sigma s'$ = *Sm* $\cup$ (*Sv*,*xs*:$\sigma s'$) **by** *auto*
obtain *S'* **where** *sp*: *S'* = (*Sv* $\cup$ *Sm*),*xs*:$\sigma s'$ **by** *simp*
from *s-svm2 sp* **have** *sp-svm*: *S'* = *Sm* $\cup$ (*Sv*,*xs*:$\sigma s'$) **by** *simp*
let *?Sp* = *S*(|*tys* := (*tys S*),*xs*:$\sigma s'$|)
from *sv v-s2 m-s2 sp-svm* **have** Γ,*xs*:$\sigma s \rightsquigarrow$ *S*(|*tys* := *S'*|)
using *trans-env-def push-vars-def* **by** *auto*
with *s-svm sp* **have** *g-s2*: Γ,*xs*:$\sigma s \rightsquigarrow$ *?Sp* **by** *simp*
from *eq Cok* **have** *Cok2*: *concepts* $(\Gamma, xs{:}\sigma s)$ *ok* **by** *simp*
from *Cok2 g-s2 IH* **obtain** τ' **where** *wt-f*: *?Sp* $\vdash_F$ *f* : τ'
and *t-tp*: *concepts* $(\Gamma, xs{:}\sigma s) \vdash \tau \rightsquigarrow \tau'$ **by** *blast*
from *t-tp eq* **have** *t-tp2*: *concepts* $\Gamma \vdash \tau \rightsquigarrow \tau'$ **by** *simp*
have *xsds*: *set xs* $\cap$ *dom* (*tys S*) = {} **sorry** — can alpha-convert xs to get this
from *wt-f xsds lxs2* **have** *wt-l*: *S* $\vdash_F$ λ *xs*:$\sigma s'$. *f* : *fn* $\sigma s' \rightarrow \tau'$ **by** (*rule wt-f-abs*)
from *ss-ssp t-tp2*
have *T*: *concepts* $\Gamma \vdash$ *fn* $\sigma s \rightarrow \tau \rightsquigarrow$ *fn* $\sigma s' \rightarrow \tau'$ **by** (*rule trans-fun*)
from *wt-l T*
show $\exists \tau'$. *S* $\vdash_F$ λ *xs*:$\sigma s'$. *f* : $\tau' \wedge$ *concepts* $\Gamma \vdash$ *fn* $\sigma s \rightarrow \tau \rightsquigarrow \tau'$
by *auto*
qed
next — Case *fg-bool*: This case is trivial.
fix Γ::*FGenv* **and** *b*
{ **fix** *S*
have *S* $\vdash_F$ *Boolean b* : *BoolT* **by** (*rule wt-f-bool*)
moreover have *concepts* $\Gamma \vdash$ *BoolG* $\rightsquigarrow$ *BoolT* **by** (*rule trans-bool*)
ultimately have $\exists \tau'$. *S* $\vdash_F$ *Boolean b* : $\tau' \wedge$ *concepts* $\Gamma \vdash$ *BoolG* $\rightsquigarrow \tau'$
by *blast*
} **thus** $\forall$ *S*. *concepts* Γ *ok* $\wedge$ $\Gamma \rightsquigarrow S \longrightarrow$
($\exists \tau'$. *S* $\vdash_F$ *Boolean b* : $\tau' \wedge$ *concepts* $\Gamma \vdash$ *BoolG* $\rightsquigarrow \tau'$) **by** *simp*
next — Case *fg-int*: This case is trivial.
fix Γ::*FGenv* **and** *i*
{ **fix** *S* **have** *S* $\vdash_F$ *Integer i* : *IntT* **by** (*rule wt-f-int*)
moreover have *concepts* $\Gamma \vdash$ *IntG* $\rightsquigarrow$ *IntT* **by** (*rule trans-int*)
ultimately have $\exists \tau'$. *S* $\vdash_F$ *Integer i* : $\tau' \wedge$ *concepts* $\Gamma \vdash$ *IntG* $\rightsquigarrow \tau'$ **by** *blast*

} **thus** $\forall S.\ \mathit{concepts}\ \Gamma\ \mathit{ok} \wedge \Gamma \rightsquigarrow S \longrightarrow (\exists \tau'.\ S \vdash_F \mathit{Integer}\ i : \tau' \wedge \mathit{concepts}\ \Gamma \vdash \mathit{IntG} \rightsquigarrow \tau')$
by *simp*
next — Case *fg-nil*: This case is trivial.
fix Γ **show** $\forall S.\ \mathit{concepts}\ \Gamma\ \mathit{ok} \wedge \Gamma \rightsquigarrow S \longrightarrow (\exists \tau s'.\ S \models_F [] : \tau s' \wedge \mathit{concepts}\ \Gamma \models [] \rightsquigarrow \tau s')$
proof *clarify*
fix S **have** A: $S \models_F [] : []$ **by** (*rule wt-f-nil*)
have B: $\mathit{concepts}\ \Gamma \models [] \rightsquigarrow []$ **by** (*rule trans-nil*)
from $A\ B$ **show** $\exists \tau s'.\ S \models_F [] : \tau s' \wedge \mathit{concepts}\ \Gamma \models [] \rightsquigarrow \tau s'$ **by** *auto*
qed
next — Case *fg-cons*: This case is straightforward.
fix $\Gamma\ \tau\ \tau s\ e\ es\ f\ fs$
assume *IH1*: $\forall S.\ \mathit{concepts}\ \Gamma\ \mathit{ok} \wedge \Gamma \rightsquigarrow S \longrightarrow (\exists \tau'.\ S \vdash_F f : \tau' \wedge \mathit{concepts}\ \Gamma \vdash \tau \rightsquigarrow \tau')$
and *IH2*: $\forall S.\ \mathit{concepts}\ \Gamma\ \mathit{ok} \wedge \Gamma \rightsquigarrow S \longrightarrow (\exists \tau s'.\ S \models_F fs : \tau s' \wedge \mathit{concepts}\ \Gamma \models \tau s \rightsquigarrow \tau s')$
show $\forall S.\ \mathit{concepts}\ \Gamma\ \mathit{ok} \wedge \Gamma \rightsquigarrow S \longrightarrow (\exists \tau s'.\ S \models_F f \# fs : \tau s' \wedge \mathit{concepts}\ \Gamma \models \tau \# \tau s \rightsquigarrow \tau s')$
proof *clarify*
fix S **assume** *Cok*: $\mathit{concepts}\ \Gamma\ \mathit{ok}$ **and** *g-s*: $\Gamma \rightsquigarrow S$
from *Cok g-s IH1* **obtain** τ' **where** *wt-f*: $S \vdash_F f : \tau'$
and *t-tp*: $\mathit{concepts}\ \Gamma \vdash \tau \rightsquigarrow \tau'$ **by** *blast*
from *Cok g-s IH2* **obtain** $\tau s'$ **where** *wt-fs*: $S \models_F fs : \tau s'$
and *ts-tsp*: $\mathit{concepts}\ \Gamma \models \tau s \rightsquigarrow \tau s'$ **by** *blast*
from *wt-f wt-fs* **have** A: $S \models_F f\#fs : \tau'\#\tau s'$ **by** (*rule wt-f-cons*)
from *t-tp ts-tsp* **have** B: $\mathit{concepts}\ \Gamma \models \tau\#\tau s \rightsquigarrow \tau'\#\tau s'$ **by** (*rule trans-cons*)
from $A\ B$ **show** $\exists \tau s'.\ S \models_F f \# fs : \tau s' \wedge \mathit{concepts}\ \Gamma \models \tau \# \tau s \rightsquigarrow \tau s'$ **by** *auto*
qed
qed

9 Conclusion

The main contribution of this report is the development of a language, named F^G, that captures the essence of concepts and thus language support for generic programming. We present a formal type system for the language and provide semantics via a translation to System F. We prove the translation preserves typing, and thus type soundness for F^G.

The language definition was formalized using the Isabelle proof assistant, and the proof of soundness for the translation was written in the Isar language and verified using Isabelle. This was a fairly difficult proof engineering task, but the definition of F^G was sharpened considerably as a result. One aspect of the proof we did not formalize in Isabelle was the use of the variable convention: we assumed that bound variable could be renamed. The standard solution to this issue is to change to De Bruijn indices. We chose not to use De Bruijn indices for this report because they are more difficult to reason about. However, rewriting the proof to use De Bruijn indices should now be a straightforward, but tedious, task.

There are several language features that are important for generic programming that we do not cover in this report. Those features include:

Associated Types. Part 2 of this report will extend F^G with associated types.

Implicit instantiation of type abstractions. Ideally we would introduce a subsumption rule based on Mitchell's containment relation [31]. However, that relation is undecidable [47]. There are two interesting restrictions that are decidable: no coercion under a function arrow [25] and restriction of type arguments to monomorphic types [36]. We plan further investigation in this area.

Statically resolved function overloading, as is found in C++ and Java. This is needed to remove the clutter of model member access such as <Monoid(t)>.binary_op.

Named models, as in [20]. This provides a mechanism for managing overlapping models, and is a straightforward addition to F^G.

Parameterized models (equivalent to parameterized instances in Haskell) are important for models that use parameterized type such as list<T>.

Defaults for concept members (as in Haskell) provide a mechanism for implementing a rich interface in terms of a few functions.

Algorithm specialization is used in C++ to provide automatic dispatching to different versions of an algorithm based on properties of a type, such as an iterator providing random access. The natural way to add this to F^G would be to have function overloading based on the where clauses of generic functions [17].

Acknowledgments

We would like to thank Ronald Garcia, Jeremiah Willcock, Doug Gregor, Jaakko Järvi, Dave Abrahams, Dave Musser, and Alexander Stepanov for many discussions and collaborations that informed this work. We thank Simon Peyton Jones for an interesting email discussion with regards to associated types. This work was supported by NSF grant EIA-0131354 and by a grant from the Lilly Endowment.

References

[1] *Ada 95 Reference Manual*, 1997.

[2] L. Augustsson. Implementing Haskell overloading. In *Functional Programming Languages and Computer Architecture*, pages 65–73, 1993.

[3] H. Barendregt. *The Lambda Calculus*, volume 103 of *Studies in Logic*. Elsevier, 1984.

[4] J.-D. Boissonnat, F. Cazals, F. Da, O. Devillers, S. Pion, F. Rebufat, M. Teillaud, and M. Yvinec. Programming with CGAL: the example of triangulations. In *Proceedings of the fifteenth annual symposium on Computational geometry*, pages 421–422. ACM Press, 1999.

[5] Boost. *Boost C++ Libraries*. http://www.boost.org/.

[6] G. Bracha, N. Cohen, C. Kemper, S. Marx, et al. *JSR 14: Add Generic Types to the Java Programming Language*, April 2001. http://www.jcp.org/en/jsr/detail?id=014.

[7] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the fourth international conference on functional programming languages and computer architecture*, 1989.

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

[9] M. Chakravarty, G. Keller, S. P. Jones, and S. Marlow. Associated types with class. In *POPL*, 2005. submitted.

[10] K. Chen, P. Hudak, and M. Odersky. Parametric type classes. In *LISP and Functional Programming*, pages 170–181, 1992.

[11] G. J. Ditchfield. Overview of Cforall. University of Waterloo, August 1996.

[12] R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock. A comparative study of language support for generic programming. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 115–134. ACM Press, Oct. 2003.

[13] J.-Y. Girard. *Interprtation Fonctionnelle et Élimination des Coupures de l'Arithmtique d'Ordre Suprieur*. Thse de doctorat d'tat, Universit Paris VII, Paris, France, 1972.

[14] J. A. Goguen, T. Winker, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In *Applications of Algebraic Specification using OBJ*. Cambridge University Press, 1992.

[15] C. V. Hall, K. Hammond, S. L. P. Jones, and P. L. Wadler. Type classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996.

[16] International Standardization Organization (ISO). *ANSI/ISO Standard 14882, Programming Language C++*. 1 rue de Varembé, Case postale 56, CH-1211 Genève 20, Switzerland, 1998.

[17] J. Järvi, J. Willcock, and A. Lumsdaine. Algorithm specialization and concept constrained genericity. In *Concepts: a Linguistic Foundation of Generic Programming*. Adobe Systems, Apr. 2004.

[18] M. P. Jones. Dictionary-free overloading by partial evaluation. In *Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne)*, pages 107–117, 1994.

[19] M. P. Jones. Type classes with functional dependencies. In *European Symposium on Programming*, number 1782 in LNCS, pages 230–244. Springer-Verlag, March 2000.

[20] W. Kahl and J. Scheffczyk. Named instances for Haskell type classes. In R. Hinze, editor, *Proc. Haskell Workshop 2001*, volume 59 of *ENTCS*, 2001. See also: http://ist.unibw-muenchen.de/Haskell/NamedInstances/.

[21] D. Kapur and D. Musser. Tecton: a framework for specifying and verifying generic system components. Technical Report RPI–92–20, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, New York 12180, July 1992.

[22] D. Kapur, D. R. Musser, and X. Nie. An overview of the tecton proof system. *Theoretical Computer Science*, 133:307–339, Oct. 1994.

[23] A. Kennedy and D. Syme. Design and implementation of generics for the .NET Common Language Runtime. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, Snowbird, Utah, June 2001.

[24] U. Köthe. *Handbook on Computer Vision and Applications*, volume 3, chapter Reusable Software in Computer Vision. Acadamic Press, 1999.

[25] D. Le Botlan and D. Rémy. MLF: Raising ML to the power of System-F. In *Proceedings of the International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden*, pages 27–38. ACM Press, aug 2003.

[26] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, 1977.

[27] D. MacQueen. An implementation of Standard ML modules. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming, Snowbird, UT*, pages 212–223, New York, NY, 1988. ACM.

[28] B. Meyer. *Eiffel: the Language*. Prentice Hall, New York, NY, first edition, 1992.

[29] Microsoft Corporation. Generics in C#, September 2002. Part of the Gyro distribution of generics for .NET available at http://research.microsoft.com/projects/clrgen/.

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

[31] J. C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2-3):211–249, 1988.

[32] D. R. Musser and A. A. Stepanov. A library of generic algorithms in Ada. In *Using Ada (1987 International Ada Conference)*, pages 216–225, New York, NY, Dec. 1987. ACM SIGAda.

[33] D. R. Musser and A. A. Stepanov. Generic programming. In P. P. Gianni, editor, *Symbolic and algebraic computation: ISSAC '88, Rome, Italy, July 4–8, 1988: Proceedings*, volume 358 of *Lecture Notes in Computer Science*, pages 13–25, Berlin, 1989. Springer Verlag.

[34] T. Nipkow. Structured Proofs in Isar/HOL. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2646, pages 259–278, 2003.

[35] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[36] M. Odersky and K. Läufer. Putting type annotations to work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 54–67. ACM Press, 1996.

[37] B. C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 11, 1996.

[38] W. R. Pitt, M. A. Williams, M. Steven, B. Sweeney, A. J. Bleasby, and D. S. Moss. The bioinformatics template library: generic components for biocomputing. *Bioinformatics*, 17(8):729–737, 2001.

[39] E. Poll and S. Thompson. The Type System of Aldor. Technical Report 11-99, Computing Laboratory, University of Kent at Canterbury, Kent CT2 7NF, UK, July 1999.

[40] J. C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, Berlin, 1974. Springer-Verlag.

[41] J. Siek, L.-Q. Lee, and A. Lumsdaine. The generic graph component library. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 399–414. ACM Press, 1999.

[42] J. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[43] J. G. Siek and A. Lumsdaine. *Advances in Software Tools for Scientific Computing*, chapter A Modern Framework for Portable High Performance Numerical Linear Algebra. Springer, 2000.

[44] Silicon Graphics, Inc. *SGI Implementation of the Standard Template Library*, 2004. http://www.sgi.com/tech/stl/.

[45] A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, May 1994.

[46] B. Stroustrup. Parameterized types for C++. In *USENIX C++ Conference*, October 1988.

[47] J. Tiuryn and P. Urzyczyn. The subtyping problem for second-order types is undecidable. *Information and Computation*, 179(1):1–18, 2002.

[48] M. Troyer, S. Todo, S. Trebst, and A. F. and. *ALPS: Algorithms and Libraries for Physics Simulations*. http://alps.comp-phys.org/.

[49] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, Jan. 1989.

[50] J. Walter and M. Koch. *uBLAS*. Boost. http://www.boost.org/libs/numeric/ublas/doc/index.htm.

[51] J. Willcock, J. Järvi, A. Lumsdaine, and D. Musser. A formalization of concepts for generic programming. In *Concepts: a Linguistic Foundation of Generic Programming at Adobe Tech Summit*. Adobe Systems, Apr. 2004.