Essential Language Support for Generic Programming

Jeremy Siek
Open Systems Lab, Indiana University
jsiek@osl.iu.edu

Andrew Lumsdaine
Open Systems Lab, Indiana University
lums@osl.iu.edu

Abstract

Concepts are an essential language feature for generic programming in the large. Concepts allow for succinct expression of constraints on type parameters of generic algorithms, enable systematic organization of problem domain abstractions, and make generic algorithms easier to use. In this paper we present 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. Concepts include a novel combination of associated types and same-type constraints that do not appear in type classes, but that are similar to nested types and type sharing in ML.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—abstract data types, constraints, polymorphism; D.2.13 [*Software Engineering*]: Reusable Software—reusable libraries; D.3.2 [*Programming Languages*]: Language Classifications—multiparadigm languages

General Terms Languages, Design

Keywords generic programming, polymorphism, C++, Standard ML, Haskell

1. Introduction

In the 1980's Musser and Stepanov developed a methodology for creating highly reusable algorithm libraries [25, 39], using the term "generic programming" for their work. They applied this methodology to the construction of sequence and graph algorithms in Ada, C, and Scheme [28, 40, 58]. In the early 1990's they applied their work to C++ and took advantage of templates [60] to construct the Standard Template Library [59] (STL). The STL became part of the C++ Standard [18], which brought generic programming into the mainstream. Since then, generic programming has been successfully applied to the creation of generic libraries for numerous problem domains [2, 29, 49, 53, 56, 62, 64].

A distinguishing characteristic of generic programming is that generic algorithms are expressed in terms of properties of types, rather than in terms of a particular type. A generic algorithm can be used (more importantly, reused) with any type that has the necessary properties. Although a statically typed language must therefore provide type parameterization ("generics") to support generic programming, generic programming as a development methodology is much richer than simply 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 four approaches to addressing these issues: subtype bounds, type classes, structural matching, and by-name operation lookup. We describe each of these approaches below and show examples in Figure 1.

Subtype Bounds (Figure 1 (a)) In object-oriented languages, constraints on type parameters are typically expressed via subtyping [6, 7, 47]. 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 in a virtual table. This approach is used in Eiffel [35] and in the generics extensions to Java [4] and C# [27, 36].

Type Classes (Figure 1 (b)) In Haskell, type classes are used to describe the set of admissible types to a generic function [63]. 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 at a call site. 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).

Structural Matching (Figure 1 (c)) Many languages take a structural approach to expressing constraints: the name of an interface does not matter (as it does for a type class), only the content of the interface matters (which operations must be provided). This is the case for CLU type sets [32, 33], ML signatures [37], and O'Caml object types [31]. In CLU, polymorphic functions are explicitly instantiated on particular types, and the corresponding cluster definitions for those types must supply the operations required in the where clause. In ML, a functor is explicitly instantiated with a structure, and the structure must match the required signature. In O'Caml, the type of the object passed into a polymorphic function must structurally match the parameter's object type, and if successful the polymorphic function is implicitly instantiated.

By-Name Operation Lookup (Figure 1 (d)) In Cforall [10, 11] and C++, the operations used in a generic function are not necessarily class methods, but can be free-standing functions. In

PLDI'05, June 12–15, 2005, Chicago, Illinois, USA.

```
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: BigInt is a subtype of the Number interface (Java).

```
class Number u where
    mult :: u →u →u

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

instance Number Int where
   mult = (∗)

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

(b) Type classes: Int is an instance of the Number type class (Haskell).

```
number = { u | u has mul: proctype (u,u) returns (u)
                        signals (underflow,overflow) }

square = proc[t:type](a: t) returns (t)
  where t in number
  return (t$mul(a, a))
  end square

start_up = proc()
    out:stream := stream$primary_output()
    stream$putl(out,int$unparse(square[int](4)))
  end start_up
```

(c) Structural matching: type int has static methods for all operations in the number type set (CLU).

```
spec number(type U) {
  U mult(U, U);
};
forall(type T | number(T))
T square(T x) {
  return mult(x,x);
}
int mult(int x, int y) { return x ∗ y; }

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

(d) By-name operation lookup: a function named "mult" is defined for type int (Cforall).

Figure 1. Some approaches to realizing generic programming.

Cforall, constraints are specified in terms of function signatures and in C++ they are specified in the accompanying documentation in terms of valid expressions. In either case, when a call is made to a generic function, the compiler tries to locate function declarations with the appropriate name and signature.

In [14] we implemented a generic graph library (based on the Boost Graph Library [54]) using programming languages in each of the above four 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 a list of requirements which denotes a set of conforming types. A function specified in terms of concepts can be used with any types satisfying the requirements given by those concepts. Concepts as specifications were formalized in the generic programming literature [23, 24, 65], but are more widely known through their use in the documentation of C++ template libraries [3, 57].

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 [15, 52] 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 instance 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.

Road map. Concepts have a number of similarities to the type classes of Haskell [17, 63] and F^G has a number of similarities (and differences) with existing work, which we discuss in Section 2. We split our presentation of F^G into two parts to simplify the presentation and the technical development. The first part adds concepts, models, and where clauses to System F. We informally introduce the syntax and semantics in Section 3 and present some examples that demonstrate its characteristics. We provide a formal semantics in Section 4 with a translation from F^G to System F that preserves typing (similar to the translation of type classes to System F in [17]). The second part of our presentation adds support for associated types, which turns out to be a non-trivial addition to the language. In Section 5 we discuss the motivation for associated types and then extend the syntax of F^G and the translation to System F to handle associated types. The language F^G omits a number of important but less essential features for generic programming due to the scope (and page limit) of this paper. We briefly describe those features in Section 6 where we also discuss directions for future work.

2. Related Work

Of existing language features, 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 target 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. 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. Difficulties arising from this difference are described in Section 3.2.

Another lesson learned was the importance of language support for associated types. In our study [14] 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 [14], Chakravarty *et al* proposed an extension to Haskell for associating algebraic data types with type classes [8]. Our work differs from theirs in three ways. First, our associated types are not algebraic data types but simply requirements for a type definition, which is 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, as we explain in Section 5. Third, we include concept inheritance (refinement) in our formalism. Earlier extensions to Haskell [9,21] address some of the same issues solved by associated types, but they did not address the problems of clutter and encapsulation.

A rough analogy can be made between ML signatures [37] and F^G concepts, and between ML structures and F^G models. However, there are significant differences. First, 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, which perform automatic lookup of the required model or instance. The associated types and same-type constraints of F^G are roughly equivalent to types nested in ML signatures and to type sharing respectively. We reuse some implementation techniques from ML such as a union/find-based algorithm for deciding type equality [34]. There are numerous other languages with parameterized modules [1, 16, 50] that require explicit instantiation with a structure.

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 depth in this paper.

As discussed in the introduction, many object-oriented languages choose to express bounds on type parameters via subtyping [4, 7, 26, 27, 35, 36]. For a detailed account of the problems we encountered with the subtype-based approach we refer the reader to our study [14]. One of the problems was the inability to group constraints on several types.

Scala [44, 45] and gbeta [12, 13] have some support for associated types in the form of object-dependent types. This differs from F^G, where types are associated with a model which is a static entity. A model could be represented with an object in Scala or gbeta, however F^G provides the convenience that models are implicitly passed to generic functions. Further, we found it difficult to express the accumulate function in Section 5, especially the return type, using Scala's nested abstract types. Scala's view construct is similar to model in F^G: it allows for retroactive conformance of a type to an interface. However in Scala, member operations must have the modeling type as a parameter, so operations such as identity_elt of the Monoid concept (see Section 3) can not be expressed.

O'Caml's object types [31,51] and polymorphism over row variables provide fairly good support for generic programming. However, O'Caml lacks support for associated types so it too suffers from clutter due to extra type parameters. PolyTOIL [5], with its match-bound polymorphism, provides similar support for generic programming as O'Caml but also lacks associated types.

Type sets in CLU [32, 33] are analogous to concepts in F^G, and of course the where clause of F^G was inspired by by CLU's where clause. Type sets differ from concepts in that they rely on structural matching whereas concepts use nominal conformance established by a model definition. Also, F^G provides a means for composing concepts via refinement whereas CLU does not provide a means for composing type sets. Finally, CLU does not provide support for associated types.

Figure 2. Types and Terms of System F

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

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)
```

3. F^G = System F + Concepts

System F, the polymorphic lambda calculus, is the prototypical tool for studying type parameterization [15, 52]. The syntax of System F is shown in Figure 2. We omit the type rules for System F as they are standard. The variable f ranges over System F expressions; we reserve e for System F^G expressions. We use an over-bar, such as $\overline{\tau}$, to denote repetition: $\tau_1, \ldots, \tau_n$. We use mult-parameter functions and type abstractions in System F to ease the translation from F^G to F. We also include a let expression with the following type rule.

$$(\textsc{Let})\ \frac{\Sigma \vdash f_1 : \sigma \qquad \Sigma, x : \sigma \vdash f_2 : \tau}{\Sigma \vdash \mathsf{let}\ x = f_1\ \mathsf{in}\ f_2 : \tau}$$

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

Figure 4. Types and Terms of F^G

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

3.1 Adding Concepts

F^G adds support for generic programming through the addition of concepts, models, and where clauses to System F. The concept feature is a mechanism for grouping and organizing requirements. The model feature establishes that a type meets the requirements of a concept. The where clause, which is written in terms of concepts, constrains how a polymorphic function may be instantiated and dually introduces models that may be used inside a polymorphic function. Figure 4 shows the abstract syntax of the basic formulation of F^G. Associated types and same-type constraints are added to F^G in Section 5.

To illustrate the features of F^G, we evolve the sum function. 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 slightly more generalized concept: a Semigroup is some type together with an associative binary operation. In F^G, the Semigroup concept is defined as follows.

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

The generic sum function requires more than just addition; it also requires a zero object 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;
}
```

Note that 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 one as the identity element also form a monoid. 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 presently 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. The function has been generalized to work with any type that has an associative binary operation with an identity element (no longer necessarily addition), so a more appropriate name for this function is accumulate. As in System F, type parameterization in F^G is provided by the Λ expression. F^G adds a where clause to the Λ expression for listing requirements.

```
let accumulate = (Λ t where Monoid<t>. /*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. In F^G, instantiation also involves the following steps:

1. int is substituted for t in the where clause.
2. For each requirement 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.

Consider the body of the accumulate function listed below. 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 dot operator is used inside the body to access the binary operator and identity element of the Monoid.

```
let accumulate =
  (Λ t where Monoid<t>.
    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 could be ambiguous without the incorporation of overloading. For example, suppose that a generic function has two type parameters, s and t, and requires each to be a Monoid. Then a call to binary_op might refer to either Monoid<s>.binary_op or Monoid<t>.binary_op. While the convenience of function overloading is important, we did not wish to complicate F^G with this additional feature. We discuss future work on function overloading in Section 6.

The complete program for this example is in Figure 5. As with System F, F^G is an expression-oriented programming language. The concept and model declarations are like let: they add to the lexical environment for the enclosed expression (after the in).

Figure 5. 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 Monoid<t>.
    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)
```

Figure 6. 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))
```

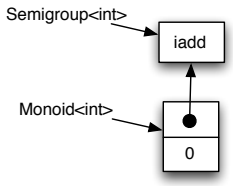

Figure 7. Dictionaries for Semigroup<int> and Monoid<int>.

3.2 Lexically Scoped Models and Overlapping

The lexical scoping of models declarations is an important feature of F^G, and one that distinguishes it from Haskell. We illustrate this distinction with an example. There are multiple ways for the set of integers to model Monoid besides addition with the zero identity element. For example, in F^G, the Monoid consisting of integers with multiplication for the binary operation and 1 for the identity element would be declared as follows.

```
model Semigroup<int> {
  binary_op = imult;
}
model Monoid<int> {
  identity_elt = 1;
}
```

Borrowing from Haskell terminology, this creates overlapping model declarations, since there are now two models declarations for the Semigroup<int> and Monoid<int> concepts. 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 6 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.

4. Translation of F^G to System F

We describe a translation from F^G to System F similar to the type-directed translation of Haskell type classes presented in [17]. The translation described here is intentionally simple; its purpose is to communicate the semantics of F^G and to aid in the proof of type safety. We show that the translation from F^G to System F preserves typing, which together with the fact that System F is type safe [48], ensures the type safety of F^G. 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, we use tuples to represent dictionaries. Thus, the model declarations for Semigroup<int> and Monoid<int> translate to a pair of let expressions that bind freshly generated dictionary names to the dictionaries (tuples) for the models. We show a diagram of the dictionary representation of these models in Figure 7 and we show the translation to System F below.

```
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 Monoid<t>. /*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
```

The formal translation rules are in Figure 9. We write $[t \mapsto \sigma]\tau$ for the capture avoiding substitution of σ for t in τ. We write $[\overline{t} \mapsto \overline{\sigma}]\tau$ for simultaneous substitution. The function FTV returns the set of free type variables and CV returns the concept names occurring in the where clauses within a type. We write $\mathrm{distinct}\ \overline{t}$ to mean that each item in the list appears at most once. We subscript a nested tuple type with a non-empty sequence of natural numbers to mean the following:

$$(\tau_1 \times \ldots \times \tau_k)_i = \tau_i$$
$$(\tau_1 \times \ldots \times \tau_k)_{i,\overline{n}} = (\tau_i)_{\overline{n}}$$

The environment Γ consists of four parts: 1) the usual type assignment for variables, 2) the set of type variables currently in scope, 3) information about concepts and their corresponding dictionary types, and 4) information about models, including the identifier and path to the corresponding dictionary in the translation.

The (MEM) rule uses the auxiliary function $\flat(c, \overline{\rho}, \overline{n}, \Gamma)$ to obtain a set of concept members together with their their types and the paths (sequences of natural numbers) to the members through the dictionary. A path instead of a single index is necessary because dictionaries may be nested due to concept refinement.

$\flat(c, \overline{\rho}, \overline{n}, \Gamma)$ =
  $M := \emptyset$
  for $i = 0, \ldots, |\overline{c'}| - 1$
    $M := M \cup \flat(c'_i, [\overline{t} \mapsto \overline{\rho}]\overline{\rho'}_i, (\overline{n}, i), \Gamma)$
  for $i = 0, \ldots, |\overline{x}| - 1$
    $M := M \cup \{x_i : ([\overline{t} \mapsto \overline{\rho}]\sigma_i, (\overline{n}, |\overline{c'}| + i))\}$
  return M
where concept $c\langle\overline{t}\rangle\{\text{refines } \overline{c'\langle\overline{\rho'}\rangle};\ \overline{x : \sigma};\} \mapsto \delta \in \Gamma$

The (TABS) rule uses the auxiliary function $\flat^w$ to collect proxy model definitions from the where clause of a type abstraction and also computes the dictionary type for each requirement. The function $\flat^m$, defined below, is applied to each concept requirement.

$\flat^w([], \Gamma) = (\Gamma, [])$
$\flat^w((c\langle\overline{\rho}\rangle, c'\langle\overline{\rho'}\rangle), \Gamma)$ =
  generate fresh d
  $(\Gamma, \delta) := \flat^m(c, \overline{\rho}, d, [], \Gamma)$
  $(\Gamma, \overline{\delta'}) := \flat^w(\overline{c'\langle[\overline{t} \mapsto \overline{\rho}]\overline{\rho'}\rangle}, \Gamma)$
  return $(\Gamma, (\delta, \overline{\delta'}))$
where concept $c\langle\overline{t}\rangle\{\text{refines } \overline{c'\langle\overline{\rho'}\rangle};\ \overline{x : \sigma};\} \mapsto \delta \in \Gamma$

The function $\flat^m(c, \overline{\rho}, d, \overline{n}, \Gamma)$ collects the model definitions and dictionary type for the model $c\langle\overline{\rho}\rangle$. The model information inserted into the environment includes a dictionary name d and a path $\overline{n}$ that gives the location inside d for the dictionary of $c(\overline{\tau})$.

$\flat^m(c, \overline{\rho}, d, \overline{n}, \Gamma)$ =
    check $\Gamma \vdash \overline{\rho} \rightsquigarrow -$
    $\overline{\tau} := []$
    for $i = 0, \ldots, |\overline{c'}| - 1$
      $(\Gamma, \delta') := \flat^m(c'_i, [\overline{t} \mapsto \overline{\rho}]\overline{\rho'}_i, d, (\overline{n}, i), \Gamma)$
      $\overline{\tau} := \overline{\tau}, \delta'$
    $\overline{\tau} := \overline{\tau}@[\overline{t} \mapsto \overline{\rho}]\overline{\sigma}$
    $\Gamma := \Gamma, (\text{model } c\langle\overline{\rho}\rangle \mapsto (d, \overline{n}))$
    return $(\Gamma, \overline{\tau})$
where concept $c\langle\overline{t}\rangle\{\text{refines } \overline{c'\langle\overline{\rho'}\rangle};\ \overline{x : \sigma};\} \mapsto \delta \in \Gamma$

Figure 8. Well-formedness of F^{G} types and translation to System F types. Formation of dictionary types.

$\boxed{\Gamma \vdash \tau \rightsquigarrow \tau'}$

$$(\textsc{TyVar})\ \frac{t \in \Gamma}{\Gamma \vdash t \rightsquigarrow t}$$

$$(\textsc{TyAbs})\ \frac{\Gamma \vdash \overline{\sigma} \rightsquigarrow \overline{\sigma'} \quad \Gamma \vdash \tau \rightsquigarrow \tau'}{\Gamma \vdash\ \text{fn } \overline{\sigma} \to \tau \rightsquigarrow\ \text{fn } \overline{\sigma'} \to \tau'}$$

$$(\textsc{TyTAbs})\ \frac{(\Gamma', \overline{\delta}) = \flat^w(\overline{c\langle\overline{\rho}\rangle}, (\Gamma, \overline{t})) \quad \Gamma' \vdash \tau \rightsquigarrow \tau'}{\Gamma \vdash \forall \overline{t} \text{ where } \overline{c\langle\overline{\rho}\rangle}.\ \tau \rightsquigarrow \forall \overline{t}.\ \text{fn } \overline{\delta} \to \tau'}$$

Figure 10. Well-formed F^{G} environment that is in correspondence with a System F environment.

$\boxed{\Gamma \rightsquigarrow \Sigma}$

$$\frac{}{\emptyset \rightsquigarrow \emptyset} \qquad \frac{\Gamma \rightsquigarrow \Sigma \quad \Gamma \vdash \tau \rightsquigarrow \tau'}{\Gamma, x : \tau \rightsquigarrow \Sigma, x : \tau'} \qquad \frac{\Gamma \rightsquigarrow \Sigma}{\Gamma, t \rightsquigarrow \Sigma, t}$$

$$\frac{\Gamma \rightsquigarrow \Sigma \quad (-, \delta) = \flat^m(c, \overline{\tau}, -, -, \Gamma)}{\Gamma, (\text{model } c\langle\overline{\tau}\rangle \mapsto (d, [])) \rightsquigarrow \Sigma, d : \delta}$$

$$\frac{\Gamma \rightsquigarrow \Sigma \quad 0 < |\overline{n}| \quad d : \delta \in \Sigma \quad (-, \delta_{\overline{n}}) = \flat^m(c, \overline{\tau}, -, -, \Gamma)}{\Gamma, (\text{model } c\langle\overline{\tau}\rangle \mapsto (d, \overline{n})) \rightsquigarrow \Sigma}$$

$$\frac{\Gamma \rightsquigarrow \Sigma \quad (\Gamma', \overline{\delta'}) = \flat^w(\overline{c'\langle\overline{\tau}\rangle}, (\Gamma, \overline{t})) \quad \Gamma' \vdash \overline{\sigma} \rightsquigarrow \overline{\sigma'}}{\Gamma, (\text{concept } c\langle\overline{t}\rangle\{\text{refines } \overline{c'\langle\overline{\tau}\rangle};\ \overline{x : \sigma};\} \mapsto \overline{\delta'}@\overline{\sigma'}) \rightsquigarrow \Sigma}$$

Figure 8 defines the translation from F^{G} types to System F types.

We now come to our main result for this section: translation produces well typed terms of System F, or more precisely, if $\Gamma \vdash e : \tau \rightsquigarrow f$ and Σ is a System F environment corresponding to Γ, then there exists some type τ' such that $\Sigma \vdash f : \tau'$. Figure 10 defines what we mean by correspondence between an F^{G} environment and System F environment.

Several lemmas are used in the theorem. The proofs of these lemmas are omitted here but appear in the accompanying technical report [55]. The technical report formalizes the lemmas and theorem in the Isar proof language [42] and the Isabelle proof assistant [43] was used to verify the proofs.

The first lemma relates the type of a model member returned by the $\flat$ function to the member type in the dictionary for the model given by the $\flat^m$.

Figure 9. Type Rules for $\mathrm{F^G}$ and Translation to System F

$$\boxed{\Gamma \vdash e : \tau \rightsquigarrow f} \qquad (\textsc{Cpt})\ \frac{\begin{array}{c}\text{distinct } \overline{t} \qquad (\Gamma', -) = \flat^w(\overline{c'\!<\!\overline{\rho}\!>}, (\Gamma, \overline{t})) \qquad \Gamma' \vdash \overline{\tau} \rightsquigarrow \overline{\tau'} \\ \delta = (\overline{[\overline{t'} \mapsto \overline{\rho'}]\delta'})@\overline{\tau'} \qquad \Gamma, (\mathsf{concept}\ c\!<\!\overline{t}\!>\{\mathsf{refines}\ \overline{c'\!<\!\overline{\rho}\!>};\ \overline{x : \tau};\} \mapsto \delta) \vdash e : \tau \rightsquigarrow f \qquad c \notin \mathrm{CV}(\tau)\end{array}}{\Gamma \vdash \mathsf{concept}\ c\!<\!\overline{t}\!>\{\mathsf{refines}\ \overline{c'\!<\!\overline{\rho}\!>};\ \overline{x : \tau};\}\ \mathsf{in}\ e : \tau \rightsquigarrow f}$$

$$(\textsc{Mdl})\ \frac{\begin{array}{c}\mathsf{concept}\ c\!<\!\overline{t}\!>\{\mathsf{refines}\ \overline{c'\!<\!\overline{\rho'}\!>};\ \overline{x : \tau};\} \mapsto \delta \in \Gamma \qquad \Gamma \vdash \overline{\rho} \rightsquigarrow \overline{\tau'} \qquad \Gamma \vdash \overline{e : \sigma} \rightsquigarrow \overline{f} \\ \overline{\mathsf{model}\ c'\!<\![\overline{t} \mapsto \overline{\rho}]\overline{\rho'}\!> \mapsto (d', \overline{n})} \in \Gamma \qquad \overline{x : [\overline{t} \mapsto \overline{\rho}]\tau \subseteq y : \sigma} \qquad d\ \text{fresh} \\ \Gamma, (\mathsf{model}\ c\!<\!\overline{\rho}\!> \mapsto (d, [])) \vdash e : \tau \rightsquigarrow f \qquad \overline{d''} = \overline{(\ \mathsf{nth}\ \ldots (\ \mathsf{nth}\ d'\ n_1) \ldots n_k)}\end{array}}{\Gamma \vdash \mathsf{model}\ c\!<\!\overline{\rho}\!>\ \{\overline{y = e};\}\ \mathsf{in}\ e : \tau \rightsquigarrow \mathsf{let}\ d = (\overline{d''}@[\overline{y} \mapsto \overline{f}]\overline{x})\ \mathsf{in}\ f}$$

$$(\textsc{TAbs})\ \frac{\text{distinct } \overline{t} \qquad \overline{t} \cap \mathrm{FTV}(\Gamma) = \emptyset \qquad (\Gamma', \overline{\delta}) = \flat^w(\overline{c\!<\!\overline{\rho}\!>}, (\Gamma, \overline{t})) \qquad \Gamma' \vdash e : \tau \rightsquigarrow f}{\Gamma \vdash \Lambda\overline{t}\ \mathsf{where}\ \overline{c\!<\!\overline{\rho}\!>}.\ e : \forall\overline{t}\ \mathsf{where}\ \overline{c\!<\!\overline{\rho}\!>}.\ \tau \rightsquigarrow \Lambda\overline{t}.\ \lambda\overline{d : \delta}.\ f}$$

$$(\textsc{TApp})\ \frac{\Gamma \vdash \overline{\sigma} \rightsquigarrow \overline{\sigma'} \qquad \Gamma \vdash e : \forall\overline{t}\ \mathsf{where}\ \overline{c\!<\!\overline{\rho}\!>}.\ \tau \rightsquigarrow f \qquad \overline{\mathsf{model}\ c\!<\![\overline{t} \mapsto \overline{\sigma}]\overline{\rho}\!> \mapsto (d, \overline{n})} \in \Gamma}{\Gamma \vdash e[\overline{\sigma}] : [\overline{t} \mapsto \overline{\sigma}]\tau \rightsquigarrow f[\overline{\sigma'}](\overline{\ \mathsf{nth}\ \ldots (\ \mathsf{nth}\ d\ n_1) \ldots n_k})}$$

$$(\textsc{Mem})\ \frac{\Gamma \vdash \overline{\rho} \rightsquigarrow \overline{\rho'} \qquad (\mathsf{model}\ c\!<\!\overline{\rho}\!> \mapsto (d, \overline{n})) \in \Gamma \qquad (x : (\tau, \overline{n'})) \in \flat(c, \overline{\rho}, \overline{n}, \Gamma)}{\Gamma \vdash c\!<\!\overline{\rho}\!>.x : \tau \rightsquigarrow (\ \mathsf{nth}\ \ldots (\ \mathsf{nth}\ d\ n'_1) \ldots n'_k)} \qquad (\textsc{Var})\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau \rightsquigarrow x}$$

$$(\textsc{Abs})\ \frac{\Gamma, \overline{x : \sigma} \vdash e : \tau \rightsquigarrow f \qquad \Gamma \vdash \overline{\sigma} \rightsquigarrow \overline{\sigma'}}{\Gamma \vdash \lambda\overline{x : \sigma}.\ e :\ \mathsf{fn}\ \overline{\sigma} \to \tau \rightsquigarrow \lambda\overline{x : \sigma'}.\ f} \qquad (\textsc{App})\ \frac{\Gamma \vdash e_1 :\ \mathsf{fn}\ \overline{\sigma} \to \tau \rightsquigarrow f_1 \qquad \Gamma \vdash \overline{e_2 : \sigma} \rightsquigarrow \overline{f_2}}{\Gamma \vdash e_1(\overline{e_2}) : \tau \rightsquigarrow f_1(\overline{f_2})}$$

LEMMA 1.

If $(x : (\tau, \overline{n'})) \in \flat(c, \overline{\rho}, \overline{n}, \Gamma)$ *and* $(-, \delta_{\overline{n}}) = \flat^m(c, \overline{\rho}, -, -, \Gamma)$
then $\Gamma \vdash \tau \rightsquigarrow \delta_{\overline{n'}}$

The next lemma states that the type of the dictionaries in the environment match the concept's dictionary type δ. The purpose of the sequence $\overline{n}$ is to map from the dictionary d for a "derived" concept to the nested tuple for the "super" concept c.

LEMMA 2.

If $(\mathsf{model}\ c\!<\!\overline{\tau}\!> \mapsto (d, \overline{n})) \in \Gamma$ *and* $\Gamma \rightsquigarrow \Sigma$
and $(-, \delta) = \flat^m(c, \overline{\tau}, -, -, \Gamma)$
then $\Sigma \vdash (\ \mathsf{nth}\ \ldots (\ \mathsf{nth}\ d\ n_1) \ldots n_k) : \delta$

The following lemma states that extending the $\mathrm{F^G}$ environment with proxy models from a where clause, and extending the System F environment with $\overline{d} : \overline{\delta}$, preserves the environment correspondence.

LEMMA 3.

If $\Gamma \rightsquigarrow \Sigma$ *and* $(\Gamma', \overline{\delta}) = \flat^w(\overline{c\!<\!\overline{\rho}\!>}, \Gamma)$
then $\Gamma' \rightsquigarrow \Sigma, \overline{d} : \overline{\delta}$

We now state the main result. The proof is in the Appendix.

THEOREM 1 (Translation preserves well typing).

If $\Gamma \vdash e : \tau \rightsquigarrow f$ *and* $\Gamma \rightsquigarrow \Sigma$
then there exists τ' *such that* $\Sigma \vdash f : \tau'$ *and* $\Gamma \vdash \tau \rightsquigarrow \tau'$

5. Associated Types and Same-Type Constraints

Associated types are types that play a role in the operations of a concept, that may vary from model to model, and that are determined by the type parameters of the concept. An example of an associated type is the "pointed-to" type of an iterator. In Standard ML, associated types are typically represented with nested types within a signature. Similarly, in $\mathrm{F^G}$ nested types within a concept are used to represent associated types. The following Iterator concept declares a requirement for a nested type named elt.

```
concept Iterator<Iter> {
  types elt;
  next : fn(Iter)→Iter,
  curr : fn(Iter)→elt,
  at_end : fn(Iter)→bool;
}
```

Any model of the Iterator concept must provide a type assignment for elt.

```
model Iterator<list int> {
  types elt = int;
  next = (λ ls : list int. cdr[int](ls)),
  curr = (λ ls : list int. car[int](ls)),
  at_end = (λ ls : list int. null[int](ls));
}
```

In the following code the accumulate function is modified to take an iterator as input instead of a list. The function is now parameterized on the iterator type, not on the element type of the list. However, accumulate still needs a way to refer to the element type, for example, to write the return type and to state the requirement that the element type must model Monoid. We extend the model member access syntax to apply to associated

Figure 11. F^G with Associated Types and Same Type Constraints

$$
\begin{array}{lll}
c & \in & \text{Concept Names} \\
s, t & \in & \text{Type Variables} \\
x, y & \in & \text{Term Variables} \\
\rho, \sigma, \tau & ::= & t \mid \mathsf{fn}\ \overline{\tau} \to \tau \mid \forall \overline{t}\ \mathsf{where}\ \overline{c\langle\overline{\sigma}\rangle};\ \overline{\sigma = \tau}\ .\ \tau \\
 & & \mid\ c\langle\overline{\tau}\rangle.t \\
e & ::= & x \mid e(\overline{e}) \mid \lambda \overline{y : \tau}.\ e \\
 & & \mid\ \Lambda \overline{t}\ \mathsf{where}\ \overline{c\langle\overline{\sigma}\rangle};\ \overline{\sigma = \tau}\ .\ e \mid e[\overline{\tau}] \\
 & & \mid\ \mathsf{concept}\ c\langle\overline{t}\rangle\ \{ \\
 & & \quad \mathsf{types}\ \overline{s};\ \ \mathsf{refines}\ \overline{c\langle\overline{\sigma}\rangle}; \\
 & & \quad \overline{x : \tau};\ \ \overline{\sigma = \tau}; \\
 & & \}\ \mathsf{in}\ e \\
 & & \mid\ \mathsf{model}\ c\langle\overline{\tau}\rangle\ \{ \\
 & & \quad \mathsf{types}\ \overline{t = \sigma}; \\
 & & \quad \overline{x = e}; \\
 & & \}\ \mathsf{in}\ e \\
 & & \mid\ c\langle\overline{\tau}\rangle.x \\
 & & \mid\ \mathsf{type}\ t = \tau\ \mathsf{in}\ e
\end{array}
$$

types: Iterator<Iter>.elt. The accumulate function can now be expressed as follows.

```
let accumulate =
  (Λ Iter where Iterator<Iter>, Monoid<Iterator<Iter>.elt>.
    fix (λ accum : fn(Iter) →Iterator<Iter>.elt.
          λiter : Iter. /* body */))
```

From inside a generic function, associated types are opaque, that is, no information is known about them unless otherwise specified. For example, associated types from different models are assumed to be different types. However, an algorithm may need two associated types to be the same type, such as in the merge algorithm shown below, where the element type of the two iterators must be the same type. To accommodate this, we introduce *same-type constraints*.

```
let merge =
  (Λ Iter1, Iter2, OutIter where
    Iterator<Iter1>, Iterator<Iter2>,
    OutputIterator<OutIter, Iterator<Iter1>.elt>,
    LessThanComparable<Iterator<Iter1>.elt>;
    Iterator<Iter1>.elt = Iterator<Iter2>.elt. /* body */)
```

5.1 Syntax and Typing Rules

The syntax of F^G with associated types and same-type constraints is given in Figure 11 with the additions highlighted in gray. The syntax for concepts is extended to include requirements for associated types and for type equalities. We add type assignments to models declarations. In addition, where clauses are extended with type equalities.

We have also added an expression for creating type aliases. Type aliases were singled out in [14] as an important feature and the semantics of type aliases is naturally expressed using the type equality infrastructure for same-type constraints.

Type checking is complicated by the addition of same-type constraints because type equality is no longer syntactic equality: it must take into account the same-type declarations. We extend environments to include type equalities, and introduce a new type equality relation $\Gamma \vdash \sigma = \tau$. This relation is the congruence that includes all the type equalities in Γ. Deciding type equality is equivalent to the quantifier free theory of equality with uninterpreted function symbols, for which there is an efficient $O(n \log n)$ time algorithm [41]. We prefix operations on sets of types and type assignments with $\Gamma \vdash$ because type equality now depends on the environment Γ.

Figure 13 gives the typing rules for F^G with associated types and same-type constraints and the translation to System F. The (MDL) rule must check that all required associated types are given type assignments and that the same-type requirements of the concept are satisfied. Also, when comparing the model's operations to the operations in the concept, in addition to substituting $\overline{\rho}$ for the concept parameters $\overline{t}$, occurrences of associated types must be replaced with their type assignments from the body of the model and from models of the concepts c refines. The (TABS) and (TAPP) rules are changed to introduce same-type constraints into the environment and to check same-type constraints respectively. The (APP) rule has been changed from requiring syntactic equality between the parameter and argument types to requiring type equality based on the congruence of the type equalities in the environment. The new rule (ALS) for type aliasing checks the body in an environment extended with a type equality that expresses the aliasing.

5.2 Translation

The main idea of the translation is to turn associated types into extra type parameters on type abstractions, an approach we first outlined in [19] and which is also used in [8]. The following code shows an example of this translation. The copy function requires a model of Iterator, which has an associated type elt.

```
let copy = (Λ Iter, OutIter where Iterator<Iter>,
    OutputIterator<OutIter, Iterator<Iter>.elt>. /* body */)
```

An extra type parameter for the associated type is added to the translated version of copy.

```
let copy =
  (Λ Iter, OutIter, elt.
  (λ Iterator_21:(fn(Iter)→Iter)*(fn(Iter)→elt)*(fn(Iter)→bool),
    OutputIterator_23:(fn(OutIter,elt)→OutIter).
    /* body */)
```

However, there are two complications here that are not present in [8]: same-type constraints and concept refinement. Due to the same-type constraints, all type expressions in the same equivalence class must be translated to the same System F type. Fortunately, the congruence closure algorithm for type equality [41] is based on a union-find data structure which maintains a representative for each type class. Therefore the translation outputs the representative for each type expression. The translation of the merge function shows an example of this. There are two type parameters elt1 and elt2 for each of the two Iterator constraints. Note that in the types for the three dictionaries, only elt1 is used, since it was chosen as the representative.

```
let merge =
  (Λ In1, In2, Out, elt1, elt2.
  (λ Iterator_78:(fn(In1)→In1)*(fn(In1)→elt1)*(fn(In1)→bool),
     Iterator_80:(fn(In2)→In2)*(fn(In2)→elt1)*(fn(In2)→bool),
     OutputIterator_84:(fn(Out,elt1)→Out),
     LessThanComparable_88:(fn(elt1,elt1)→bool). /* body */))
```

The second complication is the presence of concept refinement. As mentioned in [8], this causes there to be extra type parameters for not just the associated types of a concept c mentioned in the where clause, but also an extra type parameter for every associated type in concepts that c refines. Furthermore, there may be diamonds in the refinement diagram. To preclude duplicate associated types we keep track of which concepts (with particular type arguments) have already been processed.

Figure 13 presents the translation from F^G with associated types and same-type constraints to System F. We omit the (Mem), (Var),

and (Abs) rules since they do not change. The functions $\flat$ and $\flat^m$ need to be changed to take into account associated types that may appear in the type of a concept member or refinement. For example, in the body of function below, the expression <B(r)>.bar(x) has type <B(r)>.z, not just z. Also, the refinement for A(z) in B translates to <B(r)>.z modeling A.

```
concept A<u> { foo : fn(u)→u; } in
concept B<t> {
  types z;
  refines A<z>;
  bar : fn(t)→z;
} in
(Λ r where B<r>.
 λx:r. A<B<r>.z>.foo(B<r>.bar(x)))
```

We define a function $\flat^a$ to collect all the associated types from a concept c and from the concepts refined by c and map them to their concept-qualified names.

$\flat^a(c, \overline{\tau})$ =
  $S := \overline{s : c\langle\overline{\tau}\rangle.s}$
  for $i = 0, \ldots, |\overline{c'}| - 1$
    $S := S, \flat^a(c'_i, S(\overline{\tau'}_i))$
  return S
where
  concept $c\langle\overline{t}\rangle\{\text{types } \overline{s}\ ;\ \text{refines } \overline{c'\langle\overline{\tau'}\rangle};\ \overline{x : \sigma};\ \overline{\rho = \rho'}\} \in \Gamma$

Here is the new definition of $\flat$.

$\flat(c, \overline{\tau}, \overline{n}, \Gamma)$ =
  $S := \flat^a(c, \overline{\tau}), \overline{t : \tau}$
  $M := \emptyset$
  for $i = 0, \ldots, |\overline{c'}| - 1$
    $M := M \cup \flat(c'_i, S(\overline{\tau'}_i), (\overline{n}, i), \Gamma)$
  for $i = 0, \ldots, |\overline{x}| - 1$
    $M := M \cup \{x_i : (S(\sigma_i), (\overline{n}, |\overline{c'}| + i))\}$
  return M
where
  concept $c\langle\overline{t}\rangle\{\text{types } \overline{s}\ ;\ \text{refines } \overline{c'\langle\overline{\tau'}\rangle};\ \overline{x : \sigma};\ \overline{\rho = \rho'}\} \in \Gamma$

We used $\flat^m$ in Section 4 to collect the the models from a concept c and the concepts that c refines. We change $\flat^m$ to also collect the same-type constraints from the concepts. In addition, for every associated type s in c we generate a fresh type variable s' and add the same-type constraint $s' = \langle c(\overline{\tau})\rangle.s$. The function $\flat^m$ also returns a substitution mapping the associated type names to these generated type variables.

$\flat^m(c, \overline{\rho}, d, \overline{n}, \Gamma)$ =
  check $\Gamma \vdash \overline{\rho} \rightsquigarrow -$ and generate fresh variables $\overline{s'}$
  $\Gamma := \Gamma, \overline{s' = c\langle\overline{\rho}\rangle.s}$
  $A := \flat^a(c, \overline{\rho}), \overline{t : \rho}$
  $\overline{\tau} := []$
  for $i = 0, \ldots, |\overline{c'}| - 1$
    $(\Gamma, \overline{s}, \delta') := \flat^m(c'_i, A(\overline{\rho'}_i), d, (\overline{n}, i), \Gamma)$
    $\overline{\tau} := \overline{\tau}, \delta'$
  $\overline{\tau} := \overline{\tau}@A(\overline{\sigma})$
  $\Gamma := \Gamma, \overline{A(\eta) = A(\eta')}$
  $\Gamma := \Gamma, \text{model } c\langle\overline{\rho}\rangle \mapsto (d, \overline{n}, A)$
  return $(\Gamma, (\overline{s}, \overline{s'}), \overline{\tau})$
where
  concept $c\langle\overline{t}\rangle\{\text{types } \overline{s}\ ;\ \text{refines } \overline{c'\langle\overline{\rho'}\rangle};\ \overline{x : \sigma};\ \overline{\eta = \eta'}\} \in \Gamma$

The where clause of a type abstraction is processed sequentially so that later requirements in the where clause can refer to requirements (e.g., their associated types) that appear earlier in the list.

$\flat^w([], \Gamma) = (\Gamma, [])$

Figure 12. Well-formed F^G types (now with associated types and same-type constraints) and translation to System F types.

$\boxed{\Gamma \vdash \tau \rightsquigarrow \tau'}$

$$(\text{TyVar})\ \frac{t \in \Gamma}{\Gamma \vdash t \rightsquigarrow [t]_\Gamma}$$

$$(\text{TyAbs})\ \frac{\Gamma \vdash \overline{\sigma} \rightsquigarrow \overline{\sigma'} \qquad \Gamma \vdash \tau \rightsquigarrow \tau'}{\Gamma \vdash\ \text{fn } \overline{\sigma} \to \tau \rightsquigarrow\ \text{fn } \overline{\sigma'} \to \tau'}$$

$$(\text{TyTAbs})\ \frac{(\Gamma', \overline{s}, \overline{\delta}) = \flat^w(\overline{c\langle\overline{\rho}\rangle}, (\Gamma, \overline{t})) \qquad \Gamma' \vdash \tau \rightsquigarrow \tau'}{\Gamma \vdash \forall \overline{t}\ \text{where } \overline{c\langle\overline{\rho}\rangle},\ \overline{\eta = \eta'}\ .\ \tau \rightsquigarrow \forall \overline{t},\ \overline{s}\ .\ \text{fn } \overline{\delta} \to \tau'}$$

$$(\text{TyAsc})\ \frac{\Gamma \vdash \overline{\rho} \rightsquigarrow \overline{\rho'} \qquad \Gamma \vdash \text{model } c\langle\overline{\rho}\rangle \ldots \in \Gamma}{\Gamma \vdash c\langle\overline{\rho}\rangle.x \rightsquigarrow [c\langle\overline{\rho}\rangle.x]_\Gamma}$$

$\flat^w((c\langle\overline{\rho}\rangle, \overline{c'\langle\overline{\rho'}\rangle}), \Gamma)$ =
  generate fresh d
  $(\Gamma, \overline{s}, \delta) := \flat^m(c, \overline{\rho}, d, [], \Gamma)$
  $(\Gamma, \overline{s'}, \overline{\delta'}) := \flat^w(\overline{c'\langle[\overline{t} \mapsto \overline{\rho}]\overline{\rho'}\rangle}, \Gamma)$
  return $(\Gamma, (\overline{s}, \overline{s'}), (\delta, \overline{\delta'}))$
where
  concept $c\langle\overline{t}\rangle\{\text{types } \overline{s}\ ;\ \text{refines } \overline{c'\langle\overline{\rho'}\rangle};\ \overline{x : \sigma};\ \overline{\eta = \eta'}\} \in \Gamma$

Figure 12 shows the changes to the translation of F^G types to System F types. Type variables and member access types are mapped to their representative, written as $[-]_\Gamma$.

The proof that the translation to System F preserves well typing can be modified to take into account the changes we have made for associated types and same-type constraints and will appear in the first author's Ph.D. thesis.

THEOREM 2 (Translation preserves well typing).

If $\Gamma \vdash e : \tau \rightsquigarrow f$ *and* $\Gamma \rightsquigarrow \Sigma$

then there exists τ' *such that* $\Sigma \vdash f : \tau'$ *and* $\Gamma \vdash \tau \rightsquigarrow \tau'$

6. Conclusion

The main contribution of this paper is the development of a language, named F^G, that provides first-class support for concepts, thereby capturing the essence of 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 that the translation preserves typing, and thus type soundness for F^G.

There are several language features that are important for generic programming that we do not cover in this paper due to space considerations as well as a desire to provide an uncluttered presentation of concepts. Those features include:

Nested Requirements. Concepts often include requirements on associated types. For example, a container's associated iterator a would be required to model the Iterator concept. This form of concept composition is slightly different from refinement but close enough that we did not wish to clutter our presentation of F^G.

Figure 13. Type Rules for F^G with Associated Types and Same-Type Constraints and Translation to System F

$\boxed{\Gamma \vdash e : \tau \rightsquigarrow f}$

$$\text{(Cpt)}\ \frac{\begin{array}{c}\text{distinct } \overline{t} \quad \text{distinct } \overline{s} \quad \overline{\text{concept } c'\langle\overline{t'}\rangle\{\ldots\} \mapsto \delta'} \in \Gamma \quad \Gamma, \overline{t}, \overline{s} \vdash \overline{\rho} \rightsquigarrow \overline{\rho'} \quad \Gamma, \overline{t}, \overline{s} \vdash \overline{\tau} \rightsquigarrow \overline{\tau'} \\ \Gamma, \overline{t}, \overline{s} \vdash \overline{\sigma} \rightsquigarrow \overline{\nu} \qquad \Gamma, \overline{t}, \overline{s} \vdash \overline{\sigma'} \rightsquigarrow \overline{\nu'} \qquad \delta = (\overline{[t' \mapsto \rho']\delta'})@\overline{\tau'} \\ \Gamma, (\text{concept } c\langle\overline{t}\rangle\{\text{types } \overline{s}\ ;\ \text{refines } \overline{c'\langle\overline{\rho}\rangle};\ \overline{x : \tau};\ \overline{\sigma = \sigma'}\} \mapsto \delta) \vdash e : \tau \rightsquigarrow f\end{array}}{\Gamma \vdash \text{concept } c\langle\overline{t}\rangle\{\text{types } \overline{s}\ ;\ \text{refines } \overline{c'\langle\overline{\rho}\rangle};\ \overline{x : \tau};\ \overline{\sigma = \sigma'}\} \text{ in } e : \tau \rightsquigarrow f}$$

$$\text{(Mdl)}\ \frac{\begin{array}{c}\text{concept } c\langle\overline{t}\rangle\{\text{types } \overline{s'}\ ;\ \text{refines } \overline{c'\langle\overline{\rho'}\rangle};\ \overline{x : \tau};\ \overline{\eta = \eta'}\} \mapsto \delta \in \Gamma \quad \Gamma \vdash \overline{\rho} \rightsquigarrow \overline{\tau'} \quad \Gamma \vdash \overline{\nu} \rightsquigarrow \overline{\nu'} \quad \Gamma \vdash \overline{e : \sigma} \rightsquigarrow \overline{f} \\ \overline{s'} \subseteq \overline{s} \qquad S = \overline{t : \rho}, \overline{s' : [\overline{s} \mapsto \overline{\nu}]s'} \qquad \Gamma \vdash \overline{\text{model } c'\langle S(\overline{\rho'})\rangle \mapsto (d', \overline{n}, \ A'\ )} \in \Gamma \qquad S' = S, \cup\overline{A'} \\ \Gamma \vdash \overline{x : S'(\tau)} \subseteq \overline{y : \sigma} \qquad \Gamma \vdash \overline{S'(\eta) = S'(\eta')} \qquad d \text{ fresh} \\ \Gamma, (\text{model } c\langle\overline{\rho}\rangle \mapsto (d, [], \ (\cup\overline{A'}, \overline{s' : [\overline{s} \mapsto \overline{\nu}]s'})\ )) \vdash e : \tau \rightsquigarrow f \qquad \overline{d''} = \overline{(\ \text{nth}\ \ldots (\ \text{nth}\ d'\ n_1) \ldots n_k)}\end{array}}{\Gamma \vdash \text{model } c\langle\overline{\rho}\rangle\ \{\ \text{types } \overline{s = \nu}; \overline{y = e}\} \text{ in } e : \tau \rightsquigarrow \text{let } d = (\overline{d''}@[\overline{y} \mapsto \overline{f}]\overline{x}) \text{ in } f}$$

$$\text{(TAbs)}\ \frac{\text{distinct } \overline{t} \quad \overline{t} \cap \text{FTV}(\Gamma) = \emptyset \quad (\Gamma', \ \overline{s}\ , \overline{\delta}) = \flat^w(\overline{c\langle\overline{\rho}\rangle}, (\Gamma, \overline{t})) \quad \Gamma', \ \overline{\tau = \tau'}\ \vdash e : \tau \rightsquigarrow f}{\Gamma \vdash \Lambda\overline{t} \text{ where } \overline{c\langle\overline{\rho}\rangle}, \ \overline{\tau = \tau'}\ . e : \forall\overline{t} \text{ where } \overline{c\langle\overline{\rho}\rangle}, \ \overline{\tau = \tau'}\ . \tau \rightsquigarrow \Lambda\overline{t}, \ \overline{s}\ . \lambda\overline{d : \delta}.\ f}$$

$$\text{(TApp)}\ \frac{\begin{array}{c}\Gamma \vdash \overline{\sigma} \rightsquigarrow \overline{\sigma'} \qquad \Gamma \vdash e : \forall\overline{t} \text{ where } \overline{c\langle\overline{\rho}\rangle}, \ \overline{\eta = \eta'}\ . \tau \rightsquigarrow f \\ \Gamma \vdash \overline{\text{model } c\langle[\overline{t} \mapsto \overline{\sigma}]\overline{\rho}\rangle \mapsto (d, \overline{n}, \ \overline{s : \nu}\ )} \in \Gamma \qquad \Gamma \vdash \overline{[\overline{t} \mapsto \overline{\sigma}]\eta = [\overline{t} \mapsto \overline{\sigma}]\eta'}\end{array}}{\Gamma \vdash e[\overline{\sigma}] : [\overline{t} \mapsto \overline{\sigma}]\tau \rightsquigarrow f[\overline{\sigma'}, \ \overline{\nu}\ ](\overline{\text{nth}\ \ldots (\ \text{nth}\ d\ n_1) \ldots n_k})}$$

$$\text{(Als)}\ \frac{t \notin \text{FTV}(\Gamma) \quad \Gamma, t = \tau \vdash e : \tau \rightsquigarrow f}{\Gamma \vdash \text{type } t = \tau \text{ in } e : \tau \rightsquigarrow f} \qquad \text{(App)}\ \frac{\Gamma \vdash e_1 : \text{fn } \overline{\sigma} \to \tau \rightsquigarrow f_1 \quad \Gamma \vdash \overline{e_2 : \sigma'} \rightsquigarrow \overline{f_2} \quad \Gamma \vdash \overline{\sigma = \sigma'}}{\Gamma \vdash e_1\ \overline{e_2} : \tau \rightsquigarrow f_1(\overline{f_2})}$$

Implicit instantiation of type abstractions. Ideally we would introduce a subsumption rule based on Mitchell's containment relation [38]. However, that relation is undecidable [61]. There are two interesting restrictions that are decidable: no coercion under a function arrow [30] and restriction of type arguments to monomorphic types [46]. We plan further investigation in this area.

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

Named models, as in [22]. This feature provides a mechanism for managing overlapping models, and would be a straightforward addition to F^G.

Parameterized models (equivalent to parameterized instances in Haskell) are important for the case when the modeling type is parameterized, 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 [20].

A. Appendix

PROOF. (of Theorem 1) The proof is by induction of the derivation of $\Gamma \vdash e : \tau \rightsquigarrow f$.

Cpt Let $\Gamma' = \Gamma, \text{concept } c\langle\overline{t}\rangle\{\text{refines } \overline{c'\langle\overline{\nu}\rangle};\ \overline{x : \tau};\}$. By inversion we have:

$$\overline{\text{concept } c'\langle\overline{t'}\rangle\{\ldots\} \mapsto \delta} \in \Gamma \tag{1}$$

$$\Gamma, \overline{t} \vdash \overline{\tau} \rightsquigarrow \overline{\tau'} \tag{2}$$

$$\Gamma' \vdash e : \tau \rightsquigarrow f \tag{3}$$

$$c \notin \text{CV}(\tau) \tag{4}$$

From the assumption $\Gamma \rightsquigarrow \Sigma$ and from (1) and (2) we have $\Gamma' \rightsquigarrow \Sigma$. Then by (3) and the induction hypothesis we have $\Sigma \vdash f : \tau'$ and $\Gamma' \vdash \tau \rightsquigarrow \tau'$. Then from (4) we have $\Gamma \vdash \tau \rightsquigarrow \tau'$.

Mdl Let $\Gamma' = \Gamma, (\text{model } c\langle\overline{\rho}\rangle) \mapsto (d, [])$. We have the following by inversion:

$$\Gamma \vdash \overline{e : \sigma} \rightsquigarrow \overline{f} \tag{5}$$

$$\overline{\text{model } c'\langle[\overline{t} \mapsto \overline{\rho}]\overline{\rho'}\rangle \mapsto (d', \overline{n'})} \subseteq \Gamma \tag{6}$$

$$\overline{x : [\overline{t} \mapsto \overline{\rho}]\tau} \subseteq \overline{y : \sigma} \tag{7}$$

$$\Gamma' \vdash e : \tau \rightsquigarrow f \tag{8}$$

$$\text{concept } c\langle\overline{t}\rangle\{\text{refines } \overline{c'\langle\overline{\rho'}\rangle};\ \overline{x : \tau};\} \mapsto \delta \in \Gamma \tag{9}$$

Let Σ such that $\Gamma \rightsquigarrow \Sigma$. With (5) and the induction hypothesis there exists σ' such that $\Sigma \vdash \overline{f} : \overline{\sigma'}$ and $\Gamma \vdash \overline{\sigma} \rightsquigarrow \overline{\sigma'}$. Next, let

$$\overline{r} = \overline{(\ \text{nth}\ \ldots (\ \text{nth}\ d'\ n'_1) \ldots n'_k)}$$

From $\Gamma \rightsquigarrow \Sigma$ and (9) we have $(-, \overline{\delta'}) = \flat^w(\overline{c'\!<\!\overline{\rho'}\!>}, \Gamma)$. and therefore $(-, [\overline{t} \mapsto \overline{\rho}]\overline{\delta'}) = \flat^w(\overline{c'\!<\![\overline{t} \mapsto \overline{\rho}]\overline{\rho'}\!>}, \Gamma)$. Together with (6) and Lemma 2 we have $\Sigma \vdash \overline{r} : [\overline{t} \mapsto \overline{\rho}]\overline{\delta'}$. With (7) we have a well typed dictionary:

$$\Sigma \vdash (\overline{r}@[\overline{y} \mapsto \overline{f}]\overline{x}) : \delta \tag{10}$$

Let Σ' be $\Sigma, d : \delta$ so $\Gamma' \rightsquigarrow \Sigma'$. Then with (8) and the induction hypothesis there exists τ' such that $\Sigma' \vdash f : \tau'$ and $\Gamma' \vdash \tau \rightsquigarrow \tau'$. From (10) we show $\Sigma \vdash \mathsf{let}\ d = (\overline{r}@[\overline{y} \mapsto \overline{f}]\overline{x})\ \mathsf{in}\ f : \tau'$.

TAbs By inversion we have:

$$(\Gamma', \overline{\delta}) = \flat^w(\overline{c\!<\!\overline{\rho}\!>}, (\Gamma, \overline{t})) \tag{11}$$

$$\Gamma', \overline{t}, \overline{M} \vdash e : \tau \rightsquigarrow f \tag{12}$$

From the assumption $\Gamma \rightsquigarrow \Sigma$ we have $\Gamma, \overline{t} \rightsquigarrow \Sigma, \overline{t}$. Then with (11) we apply Lemma 3 to get $\Gamma' \rightsquigarrow \Sigma, \overline{t}, \overline{d} : \overline{\delta}$. We then apply the induction hypothesis with (12), so there exists τ' such that $\Sigma, \overline{t}, \overline{d} : \overline{\delta} \vdash f : \tau'$ and $\Gamma' \vdash \tau \rightsquigarrow \tau'$. Hence we have $\Sigma, \overline{t} \vdash \lambda\overline{d} : \overline{\delta}.\ f : \mathsf{fn}\ \overline{\delta} \to \tau'$ and therefore $\Sigma \vdash \Lambda\overline{t}.\ \lambda\overline{d} : \overline{\delta}.\ f : \forall\overline{t}.\ \mathsf{fn}\ \overline{\delta} \to \tau'$. Also, from $\Gamma' \vdash \tau \rightsquigarrow \tau'$ we have $\Gamma, \overline{t} \vdash \tau \rightsquigarrow \tau'$. Then with (11) we have $\Gamma \vdash \forall\overline{t}\ \mathsf{where}\ \overline{c\!<\!\overline{\rho}\!>}.\ \tau \rightsquigarrow \forall\overline{t}.\ \mathsf{fn}\ \overline{\delta} \to \tau'$.

TApp By inversion of the (TAPP) rule we have:

$$\Gamma \vdash \overline{\sigma} \rightsquigarrow \overline{\sigma'} \tag{13}$$

$$\Gamma \vdash e : \forall\overline{t}.\ \mathsf{where}\ \overline{c\!<\!\overline{\rho}\!>}.\ \tau \rightsquigarrow f \tag{14}$$

$$\overline{\mathsf{model}\ c\!<\![\overline{t} \mapsto \overline{\sigma}]\overline{\rho}\!>} \mapsto (d, \overline{n}) \in \Gamma \tag{15}$$

From (14) and the induction hypothesis there exists τ' such that $\Sigma \vdash f : \tau'$ and $\Gamma \vdash \forall\overline{t}\ \mathsf{where}\ \overline{c\!<\!\overline{\rho}\!>}.\ \tau \rightsquigarrow \tau'$. By inversion there exists $\overline{\delta}$, τ'', and Γ' such that

$$\tau' = \forall\overline{t}.\ \ \mathsf{fn}\ \overline{\delta} \to \tau'' \tag{16}$$

$$(\Gamma', \overline{\delta}) = \flat^w(\overline{c\!<\!\overline{\rho}\!>}, \Gamma) \tag{17}$$

$$\Gamma' \vdash \tau \rightsquigarrow \tau'' \tag{18}$$

Using (16) we have

$$\Sigma \vdash f[\overline{\sigma'}] : [\overline{t} \mapsto \overline{\sigma'}](\ \mathsf{fn}\ \overline{\delta} \to \tau'') \tag{19}$$

From (17) and (13) we have

$$(\Gamma', [\overline{t} \mapsto \overline{\sigma'}]\overline{\delta}) = \flat^w(\overline{c\!<\![\overline{t} \mapsto \overline{\sigma}]\overline{\rho}\!>}, \Gamma)) \tag{20}$$

Let $\overline{d'} = \overline{(\ \mathsf{nth}\ \ldots (\ \mathsf{nth}\ d\ n_1) \ldots n_k)}$. From the assumption $\Gamma \rightsquigarrow \Sigma$, (15), and (20) we apply Lemma 2 to get $\Sigma \vdash \overline{d'} : [\overline{t} \mapsto \overline{\sigma'}]\overline{\delta}$. Then with (19) we have $\Sigma \vdash f[\overline{\sigma'}](\overline{d'}) : [\overline{t} \mapsto \overline{\sigma}]\tau''$ and from (13) and (18) we have $\Gamma \vdash [\overline{t} \mapsto \overline{\sigma}]\tau \rightsquigarrow [\overline{t} \mapsto \overline{\sigma'}]\tau''$.

Mem By inversion we have

$$(\mathsf{model}\ c\!<\!\overline{\tau}\!> \mapsto (d, \overline{n})) \in \Gamma \tag{21}$$

$$x : (\tau, \overline{n'}) \in \flat(c, \overline{\tau}, \overline{n}, \Gamma) \tag{22}$$

From the assumption $\Gamma \rightsquigarrow \Sigma$ and (21), we have the following by inversion.

$$(d : \delta) \in \Sigma \tag{23}$$

$$(-, \delta_{\overline{n}}) = \flat^m(c, \overline{\tau}, -, -, \Gamma) \tag{24}$$

From (23) we have $\Sigma \vdash d : \delta$ and with (22) we show

$$\Sigma \vdash (\ \mathsf{nth}\ \ldots (\ \mathsf{nth}\ d\ n'_1) \ldots n'_k) : \delta_{\overline{n'}}$$

From (22), (24), and Lemma 1 we have $\Gamma \vdash \tau \rightsquigarrow \delta_{\overline{n'}}$.

Var, Abs, App The proofs of these cases are straightforward and omitted for brevity.

□

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. 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] 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.

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

[4] 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`.

[5] K. B. Bruce, A. Schuett, and R. van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In W. Olthoff, editor, *Proceedings of* ECOOP '95, number 952 in Lecture Notes in Computer Science, pages 27–51. Springer-Verlag, 1995.

[6] 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.

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

[8] M. Chakravarty, G. Keller, S. P. Jones, and S. Marlow. Associated types with class. In *Proceedings of the 32nd ACM-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California*, pages 1–13. ACM, Jan. 2005.

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

[10] G. J. Ditchfield. Contextual polymorphism, 1994.

[11] G. J. Ditchfield. Cforall reference manual and rationale, 1997.

[12] E. Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[13] E. Ernst. Family polymorphism. In *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 303–326. Springer, June 2001.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock. An analysis of constrained polymorphism for generic programming. In K. Davis and J. Striegnitz, editors, *Multiparadigm Programming in Object-Oriented Languages Workshop (MPOOL) at OOPSLA*, Anaheim, CA, Oct. 2003.

[20] 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.

[21] M. P. Jones. Type classes with functional dependencies. In *European*

Symposium on Programming, number 1782 in LNCS, pages 230–244. Springer-Verlag, March 2000.

[22] 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/`.

[23] 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.

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

[25] D. Kapur, D. R. Musser, and A. Stepanov. Operators and algebraic structures. In *Proc. of the Conference on Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire*. ACM, 1981.

[26] D. Katiyar, D. Luckham, and J. Mitchell. A type system for prototyping languages. In *Conference Record of POPL '94: 21st ACM SIGPLAN–SIGACT Symposium of Principles of Programming Languages, Portland, Oregon*, pages 138–150, 1994.

[27] 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.

[28] A. Kershenbaum, D. Musser, and A. Stepanov. Higher order imperative programming. Technical Report 88-10, Rensselaer Polytechnic Institute, 1988.

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

[30] 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.

[31] X. Leroy, D. Doligez, J. Garrigue, D. Remy, and J. Vouillon. *The Object Caml Documentation and User's Manual*, September 2003.

[32] B. Liskov, R. Atkinson, T. Bloom, E. Moss, C. Schaffert, B. Scheifler, and A. Snyder. CLU reference manual. Technical Report LCS-TR-225, Cambridge, MA, USA, October 1979.

[33] B. Liskov and J. Guttag. *Abstraction and specification in program development*. MIT Press, Cambridge, MA, USA, 1986.

[34] 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.

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

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

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

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

[39] D. R. Musser and A. Stepanov. Generic programming. In *ISSAC: Proceedings of the ACM SIGSAM International Symposium on Symbolic and Algebraic Computation*, 1988.

[40] 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.

[41] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980.

[42] 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.

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

[44] M. Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[45] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *Proc. ECOOP'03*, Springer LNCS, 2003.

[46] 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.

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

[48] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[49] 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.

[50] 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.

[51] D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1):27–50, 1998. A preliminary version appeared in the proceedings of the 24th ACM Conference on Principles of Programming Languages, 1997.

[52] 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.

[53] 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.

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

[55] J. Siek and A. Lumsdaine. Essential language support for generic programming: Formalization part 1. Technical Report 605, Indiana University, December 2004.

[56] 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.

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

[58] A. Stepanov. gclib. `http://www.stepanovpapers.com`, 1987.

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

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

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

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

[63] 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.

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

[65] 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.