A Type System for Functional Traversal-Based Aspects

Bryan Chadwick
College of Computer & Information Science
Northeastern University, 360 Huntington Avenue
Boston, Massachusetts 02115 USA
chadwick@ccs.neu.edu

Karl Lieberherr
College of Computer & Information Science
Northeastern University, 360 Huntington Avenue
Boston, Massachusetts 02115 USA
lieber@ccs.neu.edu

ABSTRACT

We present a programming language model of the ideas behind Functional Adaptive Programming (AP-F) and our Java implementation, DemeterF. Computation in AP-F is encapsulated in sets of functions that implement a fold over a data structure with the help of a generic traversal. In this paper we define the syntax, semantics, and typing rules of a simple AP-F model, together with a proof of soundness that guarantees that traversal expressions result in a value of the expected type. Applying a function set to a different structure can then be statically checked to eliminate some runtime tests and sources of program errors.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory

General Terms

Languages,Theory

Keywords

Adaptive Programming, Functional Aspects, Traversals, Type Soundness

1. INTRODUCTION

Aspect Oriented Languages provide an enormous amount of flexibility to programmers, which comes from the specification of aspects over a join point model using pointcuts and advice. In [9] the authors discuss different models that fall under this view, one of which is data structure traversal specifications in DemeterJ [12], called Adaptive Programming (AP) [8]. In AP, join points (traversal *entry* and *exit* points) are selected using a *strategy*, which directs traversal, while advice is encapsulated in *visitors* with `before` and `after` methods. Computation remains adaptable to data structure changes, but computing via side effects (`void` methods) means that adaptability goes unchecked, since structural assumptions are implicit in order dependencies between advice.

We have recently developed a functional formulation of AP that maintains the separate traversal, control (strategies), and computation of DemeterJ, but with traversals that return values. Computation in Functional Adaptive Programming [3] (AP-F) and our implementation, DemeterF [2], is encapsulated in sets of functions (or *function objects*) that, together with a generic traversal function, implement a fold over a data structure. This limits a program's adaptability, but provides more structural information about the flow of data, through argument and return types. Function objects can then be checked statically to ensure safety. In this paper we present a limited model of AP-F, describing its syntax, semantics, and typing rules. We prove the type system sound, meaning that function objects always contain applicable functions (advice is *complete*) and that the resulting program returns a value of the expected type. Applying a function object to a different structure can then be statically checked to eliminate some runtime tests and sources of program errors.

2. SYNTAX

Figure 1 shows our model syntax including definitions, functions, and simple expressions: variable references, instance construction, and traversals. A program begins with a number of definitions. The types in concrete definitions represent constructor arguments, while each T_i in the definition of an abstract type A becomes a subtype of A.

$$
\begin{array}{rcl}
x & ::= & \text{variable names} \\
C & ::= & \text{concrete type names} \\
A & ::= & \text{abstract type names} \\
 & & \\
P & ::= & D_1 \ldots D_n \;\; e \\
D & ::= & \texttt{concrete}\ C\ \texttt{(}\, T_1, \ldots, T_n\, \texttt{)} \\
 & | & \texttt{abstract}\ A\ \texttt{(}\, T_1, \ldots, T_n\, \texttt{)} \\
T & ::= & C \mid A \\
 & & \\
e & ::= & x \mid \texttt{new}\ C\,\texttt{(}\, e_1, \ldots, e_n\, \texttt{)} \mid \texttt{traverse(}\, e_0,\ F\, \texttt{)} \\
F & ::= & \texttt{funcset(}f_1 \ldots f_n\texttt{)} \\
f & ::= & \texttt{(}T_0\ x_0, \ldots, T_n\ x_n\texttt{)\{ return}\ e\texttt{; \}}
\end{array}
$$

Figure 1: AP-F Model Language Syntax

For simplicity there are no local variables, fields in ab-

```
// Double a given number representation
abstract  Int (Succ, Zero)
concrete Succ (Int)
concrete Zero ()

traverse(new Succ(new Succ(new Succ(new Zero()))),
         funcset((Succ s, Int i)
                    { return new Succ(new Succ(i)); }
                  (Zero z)
                    { return z; }))
```

Figure 2: Example Program : `Double`

stract types, non-traversal functions, or traversal control, since these do not add anything new to the type system or proofs. Function sets correspond to DemeterF function objects (sets of methods) and are used to compute over traversals of constructed values. Each function provides its arguments with their types and a single body expression, which becomes the function's result. A function set is similar to a list of `lambda` expressions (anonymous functions), though we require that all functions have no free variables.

A simple example program is given in Figure 2 with a traversal that doubles a given number representation; in this case calculating $3 * 2$. For each successor object that is traversed we return a nested double successor with the same inner integer, bottom up. The function for `Zero` is applied first, and the result is subsequently passed to the `Succ` function three times, along with each original nested `Succ` instance. This looks very similar to `fold` in functional languages, but the `traversal` function expression implicitly adapts to different data structure shapes.

2.1 Well Formed Rules

We introduce rules similar to [5] and [4] to ensure that a given program is well formed, before type checking and/or evaluation. The rules are shown below with informal descriptions; the formal definitions are elided for space reasons. With a well formed program, we can now define evaluation of expressions (the program body) in the context of a program's definitions.

TypesOnce(P): Each type is only defined once

SingleSuper(P): Each type is used in at most one abstract definition

InductiveTypes(P): Objects of concrete types are *constructible* without mutation, *i.e.*, abstract types include at least one non-recursive subtype, and data structure cycles contain at least one abstract type.

CompleteTypes(P): Each type in an abstract definition is itself defined

ClosedFunctions(P): All functions in P contain no free variables

3. SEMANTICS

Our semantics describes object creation and a depth-first (bottom up) traversal scheme that applies a function from the given set when applicable. Figure 3 shows the syntax of runtime expressions, values, and evaluation contexts. For recursive traversals and function dispatch we add two expression forms not in the surface syntax. The first gives a

$$
\begin{array}{rcl}
e & ::= & \cdots \\
 & | & \texttt{recur(}\,F,\ v_0,\ e_1,\ \ldots,\ e_n\,\texttt{)} \\
 & | & \texttt{apply(}\,f,\ v_0,\ e_1,\ \ldots,\ e_n\,\texttt{)} \\
 & & \\
v & ::= & \texttt{new}\ C\,\texttt{(}\,v_1,\ \ldots,\ v_n\,\texttt{)} \\
 & & \\
E & ::= & [\,] \\
 & | & \texttt{new}\ C\,\texttt{(}\,v\ \ldots,\ E,\ e\ \ldots\,\texttt{)} \\
 & | & \texttt{traverse(}\,E,\ F\,\texttt{)} \\
 & | & \texttt{recur(}\,F,\ v_0,\ v\ \ldots,\ E\,,e\ \ldots\,\texttt{)}
\end{array}
$$

Figure 3: Runtime Expressions, Values, and Evaluation Contexts

reduction context for the recursive step when traversing a value and the second separates the recursive traversal from function application. Values, v, are defined as a subset of the expression forms, including only constructed objects. Rather than congruence rules, we present *evaluation contexts*, E, under which reduction is permitted. A context is not used for `apply(...)` since reduction proceeds directly from `apply` to argument substitution.

We define the operational semantics as a single step reduction relation between contexts, $\longrightarrow$, which is shown in Figure 4. Traversal of a constructed value proceeds by recurring on each field. Once all recursive results are completed (*i.e.*, reduce to values), a function is chosen based on the original object's type. For a simplified presentation and proof, a function is selected based only on the type of the originally traversed value (single dispatch)[1].

The meta-functions *type*, *types*, and *choose* are defined along with substitution in Figure 5. The *type* function simply returns the type name used in value construction (*i.e.*, reflection), while *types* returns the declared argument types of the given function. The implementation of *choose* selects the function in a set with a first argument that matches the given type. Substitution is the typical replacement of e' for all free occurrences of x in e. Note that variables are bound in functions, but functions are not the same as typical functional closures, since substitution does not occur inside function sets. This simplifies the traversal typing rules and proof by eliminating the need for a type environment in the traversal judgment, providing symmetry between runtime traversals and static traversal typings.

4. TYPE SYSTEM

For ease of presentation, our type system is divided into three separate judgments: expressions ($\vdash_e$), functions ($\vdash_F$), and traversals ($\vdash_{\mathcal{T}}$). All judgments are made in the context of a program's definitions, which provide a basis for the subtype relation ($\leq$).

4.1 Expressions : $\vdash_e$

Our type system is typical for non-traversal expressions, shown in Figure 6. Variables are looked up in a type environment, Γ, which is a list of variable/type pairs. The construction of *objects* requires each field to be a subtype of

[1]In later formulations (future work) *choose* will use the types of all recursive results, in addition to the original value's type (multiple dispatch).

the declared type. For traversal expressions we delegate to our traversal judgment, which determines the return type of a traversal of an instance of type T with function set F, notated $\langle T_0, F\rangle$. It begins with no *recursive types* ($\emptyset$) and the typing derivation must discharge all recursive *traversal constraints.*

4.2 Functions: $\vdash_F$

Functions are typed in the normal manner, typing the body expression with the argument names bound to their assumed types. The result type of a function is inferred from the argument types and body expression, though substitution could cause a subtype to be returned at runtime (considered in Section 5).

4.3 Traversals : $\vdash_{\mathcal{T}}$

The traversal typing judgment uses a set, $\mathcal{X}$, of recursed types (*i.e.*, a *stack*) to identify recursive type *uses* in concrete type definitions. A set of pairs, Φ, represents *traversal constraints* collected from recursive type uses. A constraint of (T, T') means the traversal of a value of type T must result in a subtype of T'.

We read the judgment $\mathcal{X} \vdash_{\mathcal{T}} \langle T, F\rangle : T'; \Phi$ as:

> *With recurred types $\mathcal{X}$, the traversal of a value of type T with function set F, returns a value of type T' with traversal constraints Φ.*

It is split into two rules shown in Figure 8; one each for concrete and abstract types. Essentially we connect the traversal of values to the static structural definitions in the program.

For traversal of a concrete type, C, we select the parameter types of the matching function in the given set, F. The meta-function *choose*(F, C) selects a function in the set F which has C as the type of its first argument, and *types* returns the sequence of argument types. The type of the function eventually becomes the type of the traversal, but there are several conditions to be checked.

The recursive traversal of each non-recursive field with type T_i is typed by including C in the recursive type set. The result types, T_i', are required to be subtypes of the function's argument types, T_i''. If the recursive traversals generate constraints on C then we require the function's result type to be a subtype of the constrained type(s). New constraints are created for field types that exist in the set of recursed types, $\mathcal{X}$. Since the types of traversals of recursed types are unknown, we assume that the function argument types are the correct sub-traversal result types. The final constraint set is constructed from the union of field constraints by *discharging* those that involve C; the underscore (_) stands for *any* type.

The typing of the traversal of an abstract type, A, is slightly simpler since the traversal depends only on the results of the subtypes of A. Subtype traversals are typed by including A in the set of recursive types. The final return type is a common supertype, T, of the subtype traversal results. Constraints on A are checked the same as in the concrete case and new constraints are generated from recursive subtypes in $\mathcal{X}$, requiring a subtype of T as a traversal result. Constraints on A are likewise discharged.

5. TYPE SOUNDNESS

To prove type soundness we use the standard technique from [15] of proving preservation and progress. Most rules are similar to those in [5]. We begin with required lemmas.

LEMMA 1 (SUBSTITUTION PRESERVES TYPE). *If $(\Gamma, x{:}T_x) \vdash_e e : T$, $\vdash_e e' : T_x'$, and $T_x' \leq T_x$ then $\Gamma \vdash_e e[e'/x] : T'$ and $T' \leq T$.*

PROOF: By straight-forward induction on the structure of e, using the definition of substitution (Figure 5) and LEMMA 3. For a traversal expression we require that the traversal of a subtype return a subtype of the originally assigned type.

When we apply a function during traversal, substitution into the body always results in a subtype of the expected return type. The proof extends to the runtime expressions (`recur` and `apply`).

LEMMA 2 (COMPLETE FUNCTIONS). *For any well typed traversal expression $e =$ `traverse(` e_0, F `)`, the call choose(F, C) will not fail.*

PROOF: By straight-forward induction on the typing derivation of e, using the traversal judgment ($\vdash_{\mathcal{T}}$) rules, T-CTRAV and T-ATRAV (Figure 8).

In our implementation, DemeterF, the main concern is actually LEMMA 2, since at runtime we must be able to select an applicable function (*i.e.*, advice) from a given function object during traversal. Other lemmas/theorems give the stronger result that the type of the value returned from traversal is predictable.

LEMMA 3 (SUBTYPE TRAVERSALS RETURN SUBTYPES). *For any well typed traversal expression $e =$ `traverse(` e_0, F `)` with $\Gamma \vdash_e e_0 : T_0$ and result type T, the traversal of e_0', where $\Gamma \vdash_e e_0' : T_0'$ and $T_0' \leq T_0$, has result type T' with $T' \leq T$.*

PROOF: By induction on the typing derivation of e, using the traversal judgment rules, T-CTRAV and T-ATRAV.

Note that by our syntax and well-formed rules, there are no subtypes of a concrete type C, so the type of a construction expression will not change during evaluation. Our function selection (*choose*(F, C)) is deterministic and complete by LEMMA 2, and as such will return the same type for a given concrete traversal. The T-ATRAV rule also requires that the result of all subtype traversals be a subtype of the result type.

LEMMA 4 (WELL TYPED CONTEXTS). *For any closed expressions e, e', and context E, if $\vdash_e e : T$, $\vdash_e e' : T'$ with $T' \leq T$, and $\Gamma \vdash_e E[e] : T_0$, then $\Gamma \vdash_e E[e'] : T_0'$ and $T_0' \leq T_0$.*

PROOF: By induction on the structure of the context E and the typing derivation of $E[e]$, using LEMMA 3.

This aids the preservation proof below, since each reduction occurs on limited expression forms and contexts.

THEOREM 1 (PRESERVATION). *If $\vdash_e E[e] : T$ and $E[e] \rightarrow E[e']$ then $\vdash_e E[e'] : T'$ with $T' \leq T$.*

PROOF: By straight-forward induction on the structure and typing derivation of $E[e]$, using LEMMAS 1, 3, and 4.

This gives the first half of soundness: reduction preserves type, also referred to as *subject reduction.*

THEOREM 2 (PROGRESS). *Suppose that e is a closed expression. If $\vdash_e e : T$ then either e is a value, or $e = E[e_0]$ and $E[e_0] \rightarrow E[e_0']$.*

PROOF: By straight-forward induction on the structure and typing derivation of e.

We now have all requirements for the full theorem: well typed terms reduce to values of the expected type.

THEOREM 3 (TYPE SOUNDNESS). *Suppose that e is a closed expression and $\vdash_e e : T$, then either e is a value of type T, or $e \rightarrow e'$ and $\vdash_e e : T'$, with $T' \leq T$.*

PROOF: By PROGRESS and PRESERVATION theorems: e is either a value or can be reduced. If e reduces to e', then the type of e' (T') must be a subtype of T.

By PROGRESS and PRESERVATION we conclude that a well formed, well typed program will reduce to a value of predicted type, which allows us to precalculate the selection of certain functions from a set, or eliminate error checking from our dispatch implementation.

6. RELATED WORK

Most related work on semantics of Aspect Oriented Programming (AOP) Languages differs from our approach in that we do not describe a weaving semantics in order to provide a cleaner soundness proof. What we do share is a notion of dynamically selected advice (*i.e.*, *choose*), which is sometimes referred to as *advice lookup* [1, 6], implemented as *match-pcd* in [14].

In [7] the authors discuss the formulation of type safety for an AOP language in the theorem prover Coq, developing a more typical model. Our approach is specific to adaptive programming using traversals, which simplifies our soundness proof, but reduces the overall power of our model.

Similarly, in [13] the authors discuss an aspect extension to ML [10]. Labels are used to provide explicit join points, with first-class advice and side effects, providing most, if not all, of the flexibility of mainstream AOP languages. Around advice is similar to our function dispatch, though our syntax has been simplified as a first step in modeling our AP-F implementation.

Object Oriented type inference [11] has been used to provide a sound type system for a pure OO language using constraints. Constraints are generated and solved to establish that a *message-not-understood* error cannot occur. We have adapted typical typing rules based on their work and our experience with traversals. Ultimately their approach might lead to a simpler type checker, but full investigation is an item of future work.

7. CONCLUSION

We have presented the syntax, semantics, and type system of a restricted model of Functional Adaptive Programming (AP-F) and proven it type sound. AP-F provides a limited form of safe adaptive programming by way of functional, traversal-based aspects. The complication in the approach comes from the generalization of function set dispatch (*choose*), which delays function selection until recursive values are computed. This is done in order to later support a simple extension to unrestricted dispatch, as exists in our implementation.

7.1 Future Work

We are currently working on an unrestricted proof of type soundness with a full version of *choose* that selects a function based on all function arguments. With these results we hope to develop an approach for static dispatch of function objects during traversal, eliminating some of the overhead of reflection.

8. REFERENCES

[1] G. Bruns, R. Jagadeesan, A. Jeffrey, and J. Riely. μabc: A minimal aspect calculus. In *Proceedings of the 2004 International Conference on Concurrency Theory*, pages 209–224. Springer-Verlag, 2004.

[2] B. Chadwick. DemeterF: The functional adaptive programming library. Website, 2008. `http://www.ccs.neu.edu/home/chadwick/demeterf/`.

[3] B. Chadwick and K. Lieberherr. Functional Adaptive Programming. Technical Report NU-CCIS-08-75, CCIS/PRL, Northeastern University, Boston, October 2008.

[4] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *In POPL*, pages 171–183. ACM Press, 1998.

[5] A. Igarashi, B. Pierce, and P. Wadler. Featherweight java: A minimal core calculus for java and gj. In *TOPLAS*, pages 132–146, 1999.

[6] R. Jagadeesan, A. Jeffrey, and J. Riely. A calculus of untyped aspect-oriented programs. In *ECOOP*, pages 54–73, 2003.

[7] F. Kammüller and M. Voesgen. Towards type safety of aspect-oriented languages. In *AOSD 2006, FOAL Workshop*, 2009.

[8] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. 616 pages, ISBN 0-534-94602-X.

[9] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *ECOOP*, pages 2–28, 2003.

[10] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[11] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *OOPSLA*, pages 146–161, New York, NY, USA, 1991. ACM.

[12] The Demeter Group. The DemeterJ website. `http://www.ccs.neu.edu/research/demeter`, 2007.

[13] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *ICFP*, pages 127–139, New York, NY, USA, 2003. ACM.

[14] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *TOPLAS*, 26(5):890–910, 2004.

[15] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.

$$
\begin{array}{l}
E[\,\texttt{traverse(}\ \texttt{new}\ C\ \texttt{(}\ v_1,\ \ldots,\ v_n\ \texttt{)}\ \texttt{,}\ F\ \texttt{)}\,] \\
\qquad \rightarrow\ E[\,\texttt{recur(}\ F,\ \texttt{new}\ C\ \texttt{(}\ v_1,\ \ldots,\ v_n\ \texttt{)},\ \texttt{traverse(}\ v_1,\ F\ \texttt{)},\ \ldots,\ \texttt{traverse(}\ v_n,\ F\ \texttt{)}\ \texttt{)}\,] \\
E[\,\texttt{recur(}\ F,\ v_0,\ v_1,\ \ldots,\ v_n\ \texttt{)}\,] \rightarrow E[\,\texttt{apply(}\ \mathit{choose}(F,\ \mathit{type}(\ v_0)),\ v_0,\ v_1,\ \ldots,\ v_n\ \texttt{)}\,] \\
E[\,\texttt{apply(}\ \texttt{(}T_0\ x_0,\ \ldots,\ T_n\ x_n\texttt{)\{}\ \texttt{return}\ \ e\texttt{;}\ \texttt{\}}\ ,\ v_0,\ v_1,\ \ldots,\ v_n\ \texttt{)}\,] \rightarrow E[\ e[\overline{v_i/x_i}]\ ]
\end{array}
$$

Figure 4: Reduction Rules

$$\mathit{type}\,(\texttt{new}\ C\ \texttt{(}\ v_1,\ \ldots,\ v_n\ \texttt{)})\ =\ C$$

$$\mathit{types}\,(\texttt{(}T_0\ x_0,\ \ldots,\ T_n\ x_n\texttt{)\{}\ \texttt{return}\ \ e\texttt{;}\ \texttt{\}})\ =\ (T_0,\ \ldots,\ T_n)$$

$$\mathit{choose}\,(\texttt{funcset(}f\ldots,\texttt{(}C\ x_0,\ \ldots\texttt{)\{}\ \texttt{return}\ \ e\texttt{;}\ \texttt{\}},\ f\ldots\texttt{)},\ C)\ =\ \texttt{(}C\ x_0,\ \ldots\texttt{)\{}\ \texttt{return}\ \ e\texttt{;}\ \texttt{\}}$$

$$
\begin{array}{lcl}
x[e'/x] & = & e' \\
x'[e'/x] & = & x' \quad \text{if } x' \neq x \\
\texttt{new}\ C\ \texttt{(}\ e_1,\ \ldots,\ e_n\ \texttt{)}[e'/x] & = & \texttt{new}\ C\ \texttt{(}\ e_1[e'/x],\ \ldots,\ e_n[e'/x]\ \texttt{)} \\
\texttt{recur(}\ F,\ v_0,\ e_1,\ \ldots,\ e_n\ \texttt{)}[e'/x] & = & \texttt{recur(}\ F,\ v_0,\ e_1[e'/x],\ \ldots,\ e_n[e'/x]\ \texttt{)} \\
\texttt{apply(}\ f,\ v_0,\ e_1,\ \ldots,\ e_n\ \texttt{)}[e'/x] & = & \texttt{apply(}\ f,\ v_0,\ e_1[e'/x],\ \ldots,\ e_n[e'/x]\ \texttt{)} \\
\texttt{traverse(}\ e_0,\ F\ \texttt{)}[e'/x] & = & \texttt{traverse(}\ e_0[e'/x],\ F\ \texttt{)}
\end{array}
$$

Figure 5: Reflection, Function Selection and Substitution Definitions

[T-Var]

$$\frac{x : T \in \Gamma}{\Gamma \vdash_e x : T}$$

[T-New]

$$\frac{\begin{array}{c}\texttt{concrete}\ C\ \texttt{(}\ T_1,\ldots,\ T_n\ \texttt{)} \in P \\ \Gamma \vdash_e e_i : T_i' \quad T_i' \leq T_i \ \ \text{for all } i \in 1..n\end{array}}{\Gamma \vdash_e \texttt{new}\ C\ \texttt{(}\ e_1,\ \ldots,\ \ e_n\ \texttt{)} : C}$$

[T-Trav]

$$\frac{\Gamma \vdash_e e_0 : T_0 \quad \emptyset \vdash_{\mathcal{T}} \langle T_0,\ F\rangle : T;\ \emptyset}{\texttt{traverse(}\ e_0\ ,\ F\ \texttt{)} : T}$$

[T-Recur]

$$\frac{\vdash_e v_0 : C \quad \Gamma \vdash_e \texttt{apply(}\ \mathit{choose}(F,\ C),\ v_0,\ e_1,\ \ldots,\ e_n\ \texttt{)} : T}{\Gamma \vdash_e \texttt{recur(}\ F,\ v_0,\ e_1,\ \ldots,\ e_n\ \texttt{)} : T}$$

[T-Apply]

$$\frac{\begin{array}{c}\vdash_e v_0 : C \quad \mathit{types}(f) = (C,\ T_1'',\ \ldots\ ,T_n'') \quad \vdash_F f : T \\ \text{for all } i \in 1..n\ \ \Gamma \vdash_e e_i : T_i' \ \wedge\ T_i' \leq T_i''\end{array}}{\Gamma \vdash_e \texttt{apply(}\ f,\ v_0,\ e_1,\ \ldots,\ e_n\ \texttt{)} : T}$$

Figure 6: Expression Typing Rules

[T-Func]

$$\frac{\overline{x_i : T_i} \vdash_e e_0 : T}{\vdash_F\ \texttt{(}T_0\ x_0,\ \ldots,\ T_n\ x_n\texttt{)\{}\ \texttt{return}\ \ e_0\texttt{;}\ \texttt{\}} : T}$$

Figure 7: Function Typing Rule

[T-CTrav]

$$\frac{\begin{array}{c}\texttt{concrete}\ C\ \texttt{(}\ T_1,\ \ldots,\ T_n\ \texttt{)} \in P \quad \mathit{types}(\mathit{choose}(F,C)) = (C,\ T_1'',\ \ldots\ ,T_n'') \quad \vdash_F \mathit{choose}(F,C) : T \\ \text{for all } i \in 1..n\ \ T_i \notin \mathcal{X} \Rightarrow\ \mathcal{X} \cup \{C\} \vdash_{\mathcal{T}} \langle T_i, F\rangle : T_i';\Phi_i \ \wedge\ T_i' \leq T_i'' \\ (C,\ T') \in (\Phi_1 \cup \cdots \cup \Phi_n)\ \Rightarrow\ T \leq T' \qquad \Phi = \{\ (T_j,\ T_j'')\ \mid\ j \in 1..n\ \wedge\ T_j \in \mathcal{X}\ \} \qquad \Phi' = \Phi \cup (\Phi_1 \cup \cdots \cup \Phi_n)\backslash(C,\ \_)\end{array}}{\mathcal{X} \vdash_{\mathcal{T}} \langle C, F\rangle : T;\Phi'}$$

[T-ATrav]

$$\frac{\begin{array}{c}\texttt{abstract}\ A\ \texttt{(}\ T_1,\ \ldots,\ T_n\ \texttt{)} \in P \\ \text{for all } i \in 1..n\ \ T_i \notin \mathcal{X} \Rightarrow\ \ \mathcal{X} \cup \{A\} \vdash_{\mathcal{T}} \langle T_i, F\rangle : T_i';\Phi_i \ \wedge\ T_i' \leq T \\ (A,\ T') \in (\Phi_1 \cup \cdots \cup \Phi_n)\ \Rightarrow\ T \leq T' \qquad \Phi = \{\ (T_j,\ T)\ \mid\ j \in 1..n\ \wedge\ T_j \in \mathcal{X}\ \} \qquad \Phi' = \Phi \cup (\Phi_1 \cup \cdots \cup \Phi_n)\backslash(A,\ \_)\end{array}}{\mathcal{X} \vdash_{\mathcal{T}} \langle A, F\rangle : T;\Phi'}$$

Figure 8: Traversal Typing Rules