Going Against the Flow for Typeless Programming

Sean McDirmid
Microsoft Research Asia
Beijing China
smcdirm@microsoft.com

Abstract

We present YinYang, a language designed for enhanced code completion. So that type declarations do not obscure completions, YinYang supports **typeless programming** as an aggressive form of type inference that works well with object-oriented features like subtyping. YinYang forgoes traditional set-based types, instead tracking usage requirements in modular term assignment graphs where necessary usage requirements **flow backward** toward assignees to provide semantic feedback and a basis to form objects. Programs then lack type declarations but can still be separately compiled to support local reasoning on semantic feedback like type errors. This paper describes our design, trade-offs made for feasibility, and an implementation.

1. Introduction

Although types enable code completion, type declarations obscure completions by restricting how objects can be used: if the programmer does not know what type to select or selects the wrong type, they will miss completions useful in forming their program. We discussed in [15] how inference can avoid this problem by automatically defining object types based on how the objects will be used. That approach is refined in this paper into what we call *typeless programming* as an aggressive form of type inference where programs completely lack type declarations. Such a goal has yet to be achieved even in universal type inferred languages like OCaml [12] and Haskell [8] where type inference does not work very well across data abstractions.

Typeless programming forgoes computing traditional set-based types to instead compute term **usage requirements**. Consider how the assignment "`X = Y`" typically restricts usage of the assigned term `X` by generalizing `X`'s type to contain `Y`'s type [1, 22–25]. Typing instead **flows backwards** in our approach: the assignee term `Y`'s requirements, such as classes needed for use, are expanded by `X`'s requirements. Traditional types describe what terms provide for use, while we **prescribe** term needs based on usage to eliminate type declarations. Object definitions are then inferred from usage, leading to the more flexible code completion as in [15].

Beyond better code completion, we also found that by avoiding types, subtype, parametric, and data polymorphism are no longer explicit concerns of this type system or the programmer using it. The "type" of a compilation unit is a graph of its assignments between terms that can be encapsulated into *signatures* to enable **separate compilation** [4] for efficiency and local reasoning about semantic feedback. A compilation unit's signature can then be applied **generically** by clients without re-processing the unit or global analysis.

Type information in this approach cannot be used alone to resolve overloaded names. However, rather than have types drive symbol selection, symbol selection drives object type definitions so inference can be used more widely. Name ambiguity can be resolved as needed using context and type-informed search-like interactive code completion [15].

This paper describes our type system as realized in the YinYang language [15]. Although our theory is undeveloped, we show through example how YinYang is a new and interesting option in the type system design space. Term usage requirements in YinYang are extensions to mixin-like [3, 19] traits [6, 21] that are more flexible than classes; trait extensions are then propagated toward assignee terms in unit assignment graphs. YinYang's design involves many challenges that we address in successive sections:

- Assigning encapsulated objects to visible terms *freezes* them so that they cannot be expanded outside of their units (Section 2);
- Aliasing relationships between object *slots*, like fields, are accurately encoded in the assignment graph to capture covariant restrictions; also, our support for recursively nested terms is safe but not well developed (Section 3);
- Tension between polymorphism, encapsulation, and overriding limits YinYang to one level of genericity: traits and top-level methods are generic, while trait methods share their type variables with containing objects (Section 4);

- YinYang's lack of type-based name resolution, computed types, and explicit signatures presents many usability challenges that are handled by the editor (Section 5); and
- YinYang is a live programming language [16] and so requires an incremental implementation (Section 6).

Section 7 describes related work and Section 8 concludes.

2. Basics

We begin by describing traits and basic inference of trait extensions; the examples used in this section are simplistic since we avoid discussing fields and slots until Section 3. Traits in YinYang are similar to those in Scala [21] in that, unlike classes, they support a form of mixin-style [3, 19] linearized multiple inheritance. Inference along assignments propagates trait extensions required for each object, which are seeded by term method invocations; consider:

```
trait Duck:
| def Waddle(): ...
...
var A = new() # allocate ▷ Duck
A.Waddle()     # seed A ▷ Duck
```

YinYang is based on Python-like syntax [30] where indentation determines block structure; we precede indented lines with lightly shaded bars that indicate nesting depth. The # character precedes Python-like comments in YinYang that we use to document type system results, where ▷ indicates extension. A "new" expression creates a new object whose trait extensions will be inferred; e.g. the new object of this example assigned to the A variable extends the Duck trait because Waddle is called on A. What traits an object extends then depends on what it could do, reversing the more traditional role of extension in restricting usage. Because they are mixins, traits support fine-grained functionality; consider:

```
trait Bordered:
| var Thickness
trait Button:
| def Pressed(): ...
...
var A = new()         # allocate ▷ Button ▷ Bordered
if A.Pressed():       # seed A ▷ Button
| A.Thickness = 10    # seed A ▷ Bordered
```

Propagation in YinYang is flow, path, and context insensitive: only the assignment topology is relevant and condition or assignment order are ignored. The object assigned to the A variable extends the Button trait because the Pressed method is called on it, while it extends the Bordered trait because its Thickness field is assigned. With inference, programmers can ignore the traits and focus on calling methods that they discover through completion menus [15]. Library implementers can then factor functionality into many small traits without overwhelming programmers with lots of little traits.

Because traits support multiple inheritance and are extended on demand, it might appear that YinYang lacks type errors. However, many combinations of trait extensions are obviously not sensical; e.g. an object that extends both Int and String. A type error occurs in YinYang when two traits extended by a term are not "compatible" with each other; e.g. if they implement the same abstract method. Consider:

```
trait Animal:
| abstract def AnimalSound()
trait Duck:
| def Waddle(): ...
| implement def AnimalSound():
| return ''Quack''      # seed this ▷ Animal
trait Cat:
| def Purr(): ...
| implement def AnimalSound():
| return ''Meow''      # seed this ▷ Animal
...
var A = new()
A.Waddle()              # infer A ▷ Duck
A.Purr()                # infer A ▷ Cat
conflict: Duck cannot be Cat
```

The Duck and Cat traits are incompatible because they each implement the abstract AnimalSound method defined in the Animal trait. The last assignment of the example then leads to the type error `conflict: Duck cannot be Cat`, meaning incompatible extensions cannot be satisfied by any object assigned to the A variable. Incompatibility is also specified for primitives like the Int and String traits whose representations conflict.

Code completion presents **all** methods to programmers that can be invoked without error on an object; e.g. after Waddle is called on A, Purr will no longer be visible in A's completion menu. This can lead to many options in code completion menus, which is beyond the scope of this paper and we handle using probability in [15].

Term Assignment Graph

A *term assignment graph* is built during compilation to propagate trait extensions as inference results. The vertices $\overline{A}$ (overbar indicates set) of compilation unit U's *implementation* (term assignment) graph $G(U) = (\overline{A \triangleright \overline{T}}, \overline{B = C})$ are the terms used in U's construction, such as the parameters and return value of methods, local and temporary variables, new objects, and a trait's this variable; Section 3 will adds to this nested terms that refer to slots. The edges of G(U) are assignments $\overline{B = C}$ between term vertices in $\overline{A}$. G(U) also defines extensions $\triangleright \overline{T}$ for each term A, which extends all traits in $\overline{T}$.

All typeful operations in YinYang are translated into assignments and extensions according to their semantics to form G(U); e.g. an assignment A = B evaluates to one edge; a C-style condition expression "A ? B : C" evaluates to a temporary variable term D where D = B and D = C where A is also seeded to extend the Bool trait (A ▷ Bool); and the call C.Waddle() seeds C to extend Duck (C ▷ Duck).

Unit type checking propagates trait extensions from assigned (LHS) terms to assignee (RHS) terms; i.e.

$$\frac{D \triangleright \overline{S} \in \overline{A \triangleright \overline{T}} \qquad E \triangleright \overline{R} \in \overline{A \triangleright \overline{T}} \qquad D = E \in \overline{B = C}}{G(U) = (\overline{A \triangleright \overline{T}}, \overline{B = C}) \vdash \overline{S} \subseteq \overline{R}}$$

This is basically a form of iterative data-flow analysis. Consider an example of propagation:

```
trait Wild:
| def DoWildThing(): ...
...
var A = new()      # allocate ▷ Wild ▷ Cat
var B = new()      # allocate ▷ Wild ▷ Duck
var D = A          # infer A ▷ Cat ▷ Wild
var C = D          # infer D ▷ Wild
C = B              # infer B ▷ Wild
B.Waddle()         # seed B ▷ Duck
C.DoWildThing()    # seed C ▷ Wild
D.Purr()           # seed D ▷ Cat
```

The Wild trait, which defines the DoWildThing method, is seeded on the C variable and propagated to the B, D, and A variables. The Cat trait is seeded at D and propagates to A; so the code allocates a wild cat, assigned to A, and a wild duck, assigned to B.

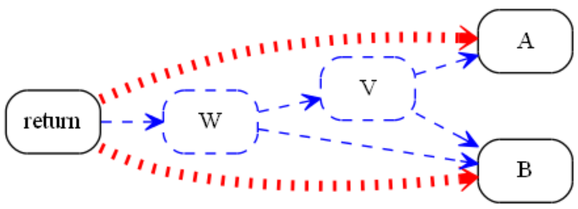
Encapsulation

Compilation units in YinYang are either top-level methods or traits. Each compilation unit U has a *signature* H(U) that encapsulates its implementation graph G(U) by: (a) pruning all hidden term vertices of U and the extensions and edges that refer to them; (b) adding assignments to preserve indirect assignment relationships between visible terms that occur through hidden terms; and (c) flagging visible terms that are assigned from hidden object definitions as *frozen*. Units are then instantiated by other client units via these signatures as a form of linking in separate compilation [4].

The "root" terms that remain visible in the signature of a unit are arguments and return value for top-level methods, or the this parameter of a trait. Other terms represent field or other slot accesses, and have more complex visibilities that we discuss in Section 3. All terms encoding temporary and local variables, constants, defined objects, and so on, are elided from a unit's signature graph. All edges that refer to hidden terms are also elided from a unit's signature graph, while new edges are added to the signature as necessary so indirect assignments are preserved between visible terms. For now we will only consider encapsulation of top-level methods; trait encapsulation is more complex and will be described more fully in Section 4; consider:

```
def Foo(A,B,C):
| var V = A
| var W = B
| if true: V = B
| if C: W = V
| return W
```

Foo's signature hides the V and W local variables, and assigns A and B to return to preserve their implementation assignment relationships; i.e.

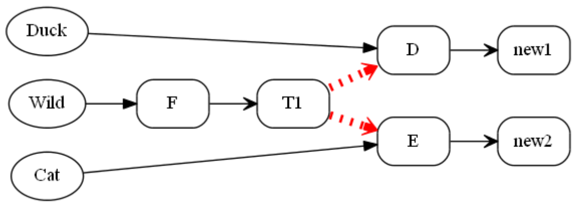


Terms here are rounded rectangles; and assignments are edges from assigned to assignee terms in direction of propagation. Dashed (and blue) nodes and edges are hidden by Foo's unit signature, and dotted (and red) assignment edges are added to Foo's unit signature to preserve indirect assignment relationship between visible terms.

Units are *instantiated* by splicing their signature graphs into client unit implementation graphs, substituting their visible root terms for binding terms of the client unit; consider:

```
var D = new()           # allocate new1 ▷ Wild ▷ Duck
var E = new()           # allocate new2 ▷ Wild ▷ Cat
var F = Foo(D, E, false) # T1 used for return
D.Waddle()              # seed D ▷ Duck
E.Purr()                # seed E ▷ Cat
F.DoWildThing()         # seed F ▷ Wild
```

The Foo method of the previous example is instantiated here with arguments D, E, and false bound to the original A, B, and C arguments. Additionally, a temporary term (T1) is bound to Foo's return value. Instantiation copies assignment edges and required trait extensions splicing in bound terms of the client unit for original root terms of the signature; i.e.:



Dotted (and red) edges are copied in on application of the Foo method; traits are ellipses; and trait extensions are edges from extended traits to extending terms. In this graph, D and E are assigned to T1 because A and B are assigned to RET in the signature of Foo; the false constant is also required to extend Bool because of the instantiation, even if redundant in this case. Because edges are copied, the Foo call is polymorphic so the Wild trait extension propagates from F to D and E through T1.

Local objects cannot support further inferred trait extensions outside of their defining units; consider:

```
def Bar():
| var V = new()    # allocate ▷ Duck
| V.Waddle()
| return V
var W = Bar()      # infer T2 ▷ Wild, fails
      frozen: cannot be Wild
W.DoWildThing()  # seed W ▷ Wild (causes error on T2)
W.Waddle()       # seed W ▷ Duck (without error)
```

Bar returns an object that is absolutely just a Duck; clients of Bar cannot cause it to extend additional traits as its definition cannot be updated with additional configuration details such as constructor parameters. A visible term that a local object is assigned to, directly or indirectly, becomes *frozen* in its unit signature, and their extensions will be restricted by what the local object is already inferred to extend. In our example, Bar's return value is frozen to extend only what is inferred by the DoDuckThing call (Duck and Animal). Frozen signature term extensions are then passed to binding terms from the client unit, so in this example, the temporary term (T2) that is bound to Bar's return value is frozen to be just Duck and Animal, and inferring Wild causes a type error.

Frozen term trait restrictions are intersected if multiple objects are assigned to the same visible term. Also, terms frozen on unit instantiation are considered like objects if they are hidden on further encapsulation. Finally, all constants except null are pre-frozen with fixed trait extensions.

3. Slots

Without object fields, we can hardly use YinYang to encode interesting programs. YinYang supports fields as *slots* that fully participate in type inference; for reasons presented in Section 4, trait method arguments and return values are also encoded as slots. The type inference of slots is also essential to expressing generic collections like lists and maps as YinYang does not have explicit type variables. Consider the definition of a simple generic Cell trait:

```
trait Cell:
| var ValC
...
var C = new(), X = new(), Y = new()

C.ValC = X             # infer X ▷ Wild, fail infer X ▷ Cat
          conflict: Duck cannot be Cat
C.ValC = Y             # infer Y ▷ Wild ▷ Cat
C.ValC.DoWildThing()   # seed C.ValC ▷ Wild
X.Waddle()             # seed X ▷ Duck
C.ValC.Purr()          # seed C.ValC ▷ Cat
```

When a trait is instantiated, all of its visible nested terms are spliced into the extending object; e.g. when C extends Cell in this example, C.ValC is a unique nested term in the assignment graph. Inference affects nested terms that access slots as it does root terms, so the nested term C.ValC, which accesses field ValC on root term C, will propagate the Wild trait extension to both X and Y. Likewise, the Cat trait extension is propagated to both X and Y given the call of Purr on C.ValC, but a type error occurs on X, which also extends Duck. This type error is reported on the assignment of X to C.ValC, which is where the incompatibility is detected.

Programs consist of multiple objects whose collective structure can determine the capabilities of each; e.g. a Windows UI [20] FrameworkElement object can be manually position when it is in a Canvas. However, no type relationship in C# exists between an element object and the Panel object that contains it: setting an object position that is not contained in a Canvas object is silently ignored! YinYang improves on this with reasoning about graphs of objects; consider:

```
var P = new(), W = new()
P.AddP(W)           # L1: seed P ▷ Panel,      infer W ▷ Widget
W.SetPos((10,10))   # L2: seed W ▷ WidgetInC, infer P ▷ Canvas
W.PutLeft()     # L3: seed W ▷ WidgetInD, infer P ▷ Dock (fails)
conflict: Canvas cannot be Dock in ParentW
```

```
trait Panel:
| def AddP(W): W.ParentW = this    # seed W ▷ Widget
| abstract def LayoutP()
trait Canvas:
| implement def LayoutP(): ...      # seed this ▷ Panel
| def MovedC(W): ...
trait Widget:
| var ParentW
trait WidgetInC:
| def SetPos(P):
| | this.ParentW.              # seed this ▷ Widget
| | | MovedC(this)             # seed this.ParentW ▷ Canvas
trait Dock:
| implement def LayoutP(): ...  # seed this ▷ Panel
| def InvalidateD(W): ...
trait WidgetInD:
| def PutLeft():
| | this.ParentW.              # seed this ▷ Widget
| | | InvalidateD(this)        # seed this.ParentW ▷ Dock
```

Figure 1: Traits that encode widget/panel relationships.

This example uses UI traits defined in Figure 1. Variables W and P respectively extend the Widget and Panel traits given the AddP call (L1), which also assigns P to W.ParentW. As a result, the SetPos call (L2) causes W to extend WidgetInC, and also causes P to extend Canvas because of the aforementioned assignment edge. The new object assigned to P will then be allocated as a Canvas without further programmer intervention. As with root terms, the trait incompatibilities of slotted terms are detected as soon as connections in the assignment graph cause the error to manifest. The SetLeft call (L3) infers that P extends the Dock trait, which is incompatible with the previous Canvas extension as both traits implement Panel's LayoutP method. An error is then flagged at L3 on the ParentW field of W, which is the node in the implementation graph where the incompatibility is detected. The ParentW slot must be mentioned in the error message since it is not specified in the term where the error is detected.

Covariance

We next address how aliasing between nested terms is handled, which is not very straightforward. Consider code that uses the Cell trait defined at the start of this section:

```
var A = new(), B = new()
var D
D = A
D = B
...
A.ValC.Waddle()        # seed A ▷ Cell, A.ValC ▷ Duck
B.ValC.Purr()          # seed B ▷ Cell, B.ValC ▷ Cat
D.ValC.DoWildThing()   # seed D ▷ Cell, C.ValC ▷ Wild
D.ValC = new()
          conflict: Duck cannot be Cat
```

Because A and B are assigned to D, D.ValC has an alias relationship with A.ValC and B.ValC. Accordingly, trait extensions of D.ValC are propagated to both; i.e. A.ValC is a Wild Duck and B.ValC is a Wild Cat. However, aliasing is not just a unidirectional relationship: a new object cannot be safely assigned to D.ValC because such an object would extend Duck when assigned to A.ValC and would extend Cat when assigned to B.ValC, which are incompatible. In fact, ValC has become **covariant** in D: it can be used but not assigned. The term assignment graph as described in Section 2 appears incapable of encoding such covariance.

Our solution to this problem relies on the observation that even though D.ValC is an alias of A.ValC and B.ValC, A.ValC and B.ValC themselves lack an aliasing relationship because D cannot be assigned from A and B at the same time. We then leverage this observation by encoding each term in the assignment graph as two nodes: a hard node that tracks real trait extensions for terms as before, and a soft node that tracks latent trait extensions that only turn into hard extensions on an explicit assignment to the term. On an explicit assignment X = Y and for any slot S that is directly accessible in both X and Y, the following 5 kinds of assignment edges are added between hard and soft nodes in the term assignment graph; we use → and ← to indicate propagation direction of :

```
1: X_hard   → Y_hard    # as before
2: X_soft   → Y_hard    # propagate soft reqs as hard reqs
3: X.S_hard → Y.S_hard  # propagate hard reqs, ignore soft reqs
4: X.S_soft ← Y.S_hard  # propagate in reverse both hard and
5: X.S_soft ← Y.S_soft  #     soft reqs as soft reqs
```

The first two edges deal with explicitly assigned terms. Edge 1 is expected as assignment should always propagate hard trait extensions to the assignee. Edge 2 goes further: any soft trait extensions in the assigned term become hard trait extensions in the assignee term on the intuition that aliasing causes propagation on assignment, and does not otherwise interfere with usage. Edges 3–5 are duplicated for each slot S accessible in both X and Y; a slot S defined in trait T is accessible from term C if C ▷ T (where ▷ is a hard extension). Edge 3 deals with standard propagation of hard constraints; soft extensions are not propagated from X.S to Y.S as this would capture soft extensions from peer aliases that we observe are unrelated. Finally, edges 4 and 5 propagate the hard and soft extensions of Y.S in reverse to X.S as soft extensions, setting up a propagation that is triggered on explicit assignment to X.S via edge 2.

Consider the assignment graph of our example in Figure 2 where the Duck and Cat traits are propagated (dotted red left-to-right edges) via D.ValC's soft nodes to become hard extensions of the new object, leading to the detected incompatibility. On the other hand, D.ValC can still propagate (dashed blue right-to-left edges) the Wild trait to A.ValC and B.ValC without error.

Edges are added for an explicit assignment between sub-slots to mirror those of their parent terms; e.g. given X.S_soft ← Y.S_hard and a R slot accessible in X.S and Y.S, X.S.R_soft ← Y.S.R_hard is added to the graph. However, such implicit edges are problematic with respect to **recursive** terms.

Recursion

YinYang does not have a problem per se with recursion: term assignment graphs can have cycles while methods can call each other without restriction. However, recursion that results in terms of infinite lengths are a problem; consider:

```
def Explode(C): Explode(C.ValC)
```

The recursive non-nonsensical top-level Explode method requires that C is a cell of cell of cell, and so on. The problem is not the recursive instantiation of Explode, but the fact that an infinite term C.ValC.ValC... can be derived at all. Even sensible basic looping code can be problematic; consider:

```
trait LinkedNode::
│ var ValLN
│ var NextLN
...
def DoAllDucks(H):
│ A = H
│ while A != null:
│ │ A.ValLN.Waddle()
│ │ A = A.NextLN
```

The DoAllDucks method walks a linked list starting from its H argument, calling Waddle on each element. YinYang would derive infinite H.NextLN term if terms could be nested indefinitely, which it does not for this reason. Instead, operations that would derive a nested term that mentions the same slot twice will instead derive the term up until the first mention of the slot, effectively merging their type information. For example, if an operation would derive the H.NextLN.NextLN term, YinYang will instead derive the term H.NextLN. Eliminating slot recursion in this way allows type checking to adequately deal with the Explode and DoAllDucks methods correctly. However, this approach is very conservative and cannot express reasonable constructions; consider:

```
var C = new(), M = new(), D = new()
M.ValC = D
C.ValC = M
C.ValC.ValC.Waddle()
  conflict: Cell cannot be Duck
```

In this example, the second ValC field access in the above code will collapse to just one ValC access, causing the prob-

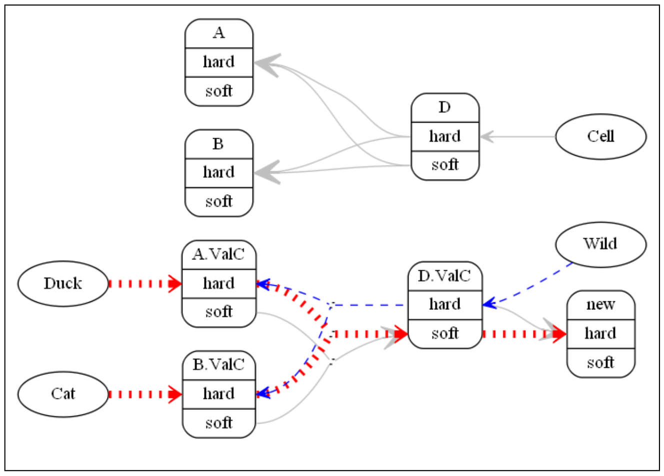


Figure 2: Soft/hard assignment graph for our example; different ports are indicated for each term's hard and soft node.

lem where the type system thinks that C must be a cell of cell and cell of duck. At best, an incompatibility is falsely raised, and at worst, a weird Duck Cell object is created!

Our current solution for recursion is inadequate but still allows us to express interesting programs. As future work, we hope to relax this restriction and detect non-convergent nested term recursion with type errors.

4. Trait Methods

With our support for slots in Section 3, we can now discuss encapsulated fields and trait methods. For one thing, the visibility of a nested term, which has a parent term and refers to a slot, depends recursively on the visibility of its parent and the visibility of the slot in the unit being compiled; i.e. given the term A.S, the visibility of this term is the conjunction of A's visibility and, if S is private, if the unit that contains A.S is the trait that defines S.

As mentioned in Section 2, top-level methods, which are not nested in traits, are instantiated polymorphically. Traits are also compilation units that are instantiated polymorphically, replacing the single this root term in their signature with the term extending the trait. Unlike top-level methods, traits have two encapsulating signatures: a *private* signature for use inside the trait that elides local and temporary terms of its method definitions, but still has terms that refer to trait private members; and a *public* signature for use outside of the trait that elides private members. Private signatures is only used for non-this terms that extend a trait and are used inside that trait's definition.

YinYang does not treat trait methods as compilation units; instead their arguments and return values are treated as slots of the object being defined, which is either a root this or a term substituted for it. We do not treat trait methods as units because methods of the same trait share its private fields, making encapsulation extremely complicated; consider:

```
trait Cell:
  private var ValC
  def GetC(): return this.ValC
  def SetC(B): this.ValC = B
  def Foo(A): A.ValC = this.ValC
```

If Foo, GetC, and SetC were instantiated polymorphically, we would have to add additional edges to the instantiating graph to express that traits would propagate from whatever was bound to the B term of each call to SetC to whatever was bound to the return value of each to GetC. This scheme is conceivable but would ruin the nice properties of our graph abstraction as additional book-keeping would be needed to ensure that these edges were generated automatically. The Foo method also creates an inter-object relationship through an encapsulated field so that the book-keeping would be required to synthesize these edges across multiple objects! Without encapsulation, there would be no problem since the intermediate fields needed to relate method arguments and return values would never be elided.

```
trait Map:
  abstract def GetM(K)
  abstract def PutM(K,E)

trait HashMap:
  private var Tbl = new(LenA = 100)
  implement def GetM(K): # public: GetM@RET = PutM@E
                         # public: GetM@K ▷ Hashable
    return this.Tbl.GetA(K.Hash % this.Tbl.CountA)
  implement def PutM(K,E):
    this.Tbl.SetA(K.Hash % this.Tbl.CountA, E)
  def PutH(K,E):      # public: PutM@E = PutH@E
    this.PutM(K,E)    # public: PutM@K = PutH@K
```

Figure 3: An abstract Map trait and a simple HashMap trait.

Overriding and Cat-calls

Another reason to forgo separate trait method instantiations involves overriding (and implementing) virtual and abstract methods. YinYang places no restriction on what assignment relationships can be added by a trait method definition, regardless of what it is overriding or implementing. Generic trait methods would interfere with that simplicity, perhaps leading to stricter rules in the assignments that overridden methods could introduce. Besides, top-level methods can be used to express common generic functionality such as mapping using public trait methods. The rest of this subsection explains how overriding works.

Overriding in YinYang is type checked through nested term assignment with no special rules imposed on overriding method implementations. Consider an abstract Map trait and implementing HashMap trait in Figure 4. The Map trait is completely devoid of code, so the relationship between the arguments and return values of its methods are unknown. In contrast, the HashMap trait implements the Map trait, establishing with its code the standard mutable map relationship where items put into the map via PutM calls are possibly returned via GetM calls. This is all missing for terms that just extend Map; consider:

```
def Bar(M, A, B, C):
  B.Purr()                 # seed B ▷ Cat
  M.PutM(A,B)              # seed M ▷ Map
                           # seed M.PutM@K = A
                           # seed M.PutM@E = B
  M.GetM(C).Waddle()       # seed M.GetM@K = B
                           # seed M.GetM@RET ▷ Duck
```

In Bar's implementation, B extends Cat and is put into M, which extends Map. Since Map specifies no relationship between GetM and PutM slots, B is not required to also be a Duck by the last call. Consider an instantiation of Bar:

```
var H = new(), J = new(), K = new()
H.PutH(0, K)      # seed H ▷ HashMap
Bar(H, 1, J, 1)
           conflict: Cat cannot be Duck
```

A PutH method is added to HashMap in Figure 4 to infer that H is a HashMap when the method is called on it. Although contrived, the programmer must indicate a usage signal so the right trait is chosen; in this case, the signal indicates that

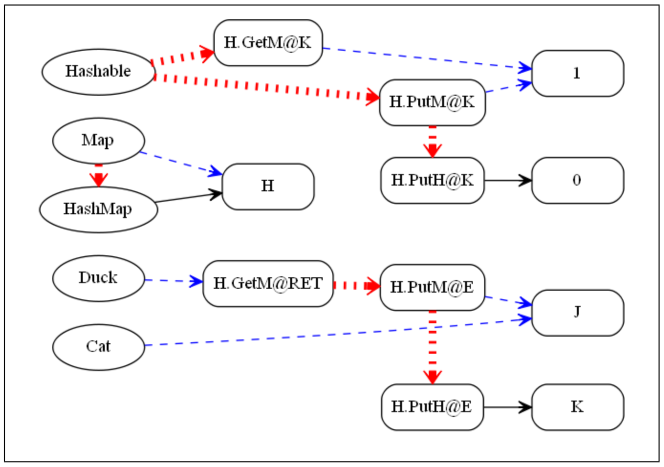


Figure 4: A graph of code that instantiates Bar and HashMap.

an element is put in the map via "hashed keys." We will come back to this design issue later.

In our example, the editor has correctly detected that Bar's third argument (B) extends both a Duck and Cat. No global analysis is necessary to detect this type error as it can be detected by instantiating the public signatures of HashMap and Bar into the implementation graph of this code. Even though Bar is unaware that it will be passed a HashMap, it does "assign" the K arguments of M.PutM and it seeds the return value of M.GetM with an extension to Duck, both of which are visible in its signature. Combined with HashMap's assignment relationships present in its public signature, this is enough to detect the type error as shown in this graph for the example in Figure 4. All term nodes shown are hard as soft extensions play no interesting role in the example; dashed (blue) edges are due to the instantiation of the Bar method; and dotted (red) edges are due to the instantiation of the HashMap trait on H. Given just dashed (blue) edges copied from Bar's signature, only the Cat trait propagates to J; it is only in combination with the bold (red) edges copied from HashMap's public signature that the graph can detect that the Duck trait also propagates to J and there is an incompatibility.

As defined this way, overriding is fairly easy from both a semantic and implementation standpoint: overridden methods are free to introduce new assignment relationships, and these will be taken into account at some point during modular type checking of compilation units. Combined with how covariance in nested terms are dealt with in Section 3, YinYang traits also safely avoids Meyer's "polymorphic catcall" [17] problem without requiring programmers to plan ahead in super classes (traits); e.g. if cows eat grass and pandas eat bamboo, both Cow and Panda traits can implement Animal's eat-food method each with their required food type.

Even if type errors will be detected eventually, programmers are deprived of timely semantic feedback when calling abstract methods that lack code. This problem is solved with *abstract code* that resembles normal YinYang code but whose only role is to establish relationships in the implementation graph of the trait being compiled. Consider a richer definition of the Map trait:

```
trait Map:
| type E
| abstract def GetM(K):
| | return this.E              # public: GetM@RET = E
| abstract def PutM(K,E):
| | this.E = E                  # public: E = PutM@E
```

The type keyword defines slots that are used to build the unit implementation graph but do not exist during execution. Given this definition of the Map trait, the incompatibility of the method Bar's B argument would be detected inside Bar as opposed to when Bar was instantiated. Again, overriding definitions are under no restriction with respect to abstract code, whose assignments need not be reproduced. However, they must live with the conservative consequences of these assignments, meaning superfluous assignment will never make the program less safe but could result in unnecessary type errors and trait extensions.

Other Design Issues

We have assumed so far that trait (type) selection can be completely inferred based on the methods selected by the programmer, which is dubious in practice; consider:

```
var H = new(), K = new()
          abstract: Map unimplemented
H.PutM(0, K)      # seed H ▷ Map
```

As the Map trait has abstract methods, inferring just it alone on an object raises an "unimplemented" type error. We avoided this issue previously by using HashMap's freshly defined PutH method, which was admittedly contrived. In this case, HashMap's functionality is indistinct compared to Map's, so method selection alone is not good enough. Rather than rely on contrived methods, other options include:

- The programmer could select HashMap for H by adding an explicit type assertion to the code; or
- The system could infer HashMap as the implementation of Map, as this what the programmer probably wants.

The first option goes against our "typeless programming" goals but is pragmatic. Although the second option involves dangerous speculation, it could be coupled with a warning so that programmers are aware of the under specification but can still run their programs anyways.

As mentioned in Section 2, objects defined with new() operations must be completely defined in their containing compilation unit and are frozen to further extension by clients. YinYang also allows for objects to be declared as public trait members that are open to extension up until their containing become hidden; consider:

```
trait Farm:
| object FarmAnimal
trait DuckFarm:
| def Init(): this.FarmAnimal.Waddle()
```

The Farm trait has one FarmAnimal object member that becomes a Duck in a DuckFarm trait that extends Farm.

We have yet to support function parameters in YinYang, but they would be supported in a way that is similar to a trait method; i.e. an instance of a FunctionN trait whose Apply method argument and return value slots encode both binding from the caller and use by the callee. Because functions are not instantiated independently, they are best used in top-level methods that are called polymorphically.

Although YinYang typing is flexible, there is still a need for dynamic downcasting; consider iterating over a list of objects where Cat objects are singled out for special attention:

```
var A = new()                       # allocate ▷ Duck
var B = new()                       # allocate ▷ Cat ▷ Wild
Es.AddL(A)                          # infer A ▷ Cat ⇒ Wild
Es.AddL(B)                          # infer B ▷ Cat ⇒ Wild
A.Waddle()                          # seed ▷ Duck
B.Purr()                            # seed ▷ Cat, infer ▷ Wild
for E in Es:               # infer Es.GetL@RET ▷ Cat ⇒ Wild
| switch E:                         # infer E ▷ Cat ⇒ Wild
| | case C ▷ Cat: C.DoWildThing()   # seed ▷ Cat ▷ Wild
| | default: continue
```

This code iterates over a list of two objects, a Cat and a Duck, choosing elements that extend Cat for an operation that causes them to extend the Wild trait. Although we have yet to implement this in YinYang, extended traits inferred for a downcast result must be propagated as *latent* extensions of the original term; e.g. the Wild extension by downcast result C propagates to original term E as ▷ Cat ⇒ Wild, meaning Wild is only extended by objects in the list that extend Cat.

5. The Programming Experience

YinYang's lack of type-based name resolution, computed types, and explicit signatures presents many usability challenges that can be compensated for by a language-aware editor. As suggested in [15], tooling should be a part of the language so trade-offs can be freely made between a language's semantics, syntax, and supporting tools, like the editor. Accordingly, we take a non-traditional approach or design that considers editing as a first-class part of the language.

Although this paper avoids name overloading for reasons of clarity, our technique does not preclude it. Instead, other signals such as probability and explicit programmer selection can be used to resolve ambiguous names. Our previous work [15] even shows how code completion benefits from being informed, but not driven, by types, where otherwise useful options are hidden from programmers because they have yet to specify the right types. There is the converse problem that programmers, without the use of types as filters, will be overwhelmed with too many options in code completion menus. However, these options can still be filtered by which ones will cause type errors, and the rest can be categorized and prioritized using probability [15]. Programmer decisions about each symbol must be recorded in code; e.g. code refers to GUIDs while ambiguous short names are input by and presented to programmers [7].

Type Debugging

The lack of type declarations or inspectable syntactic types leads to significant usability challenges in how programmers understand type errors or how objects are formed. The editor can help by making trait extensions visible as meta-text through *probing*; consider a modification of our second example from Section 2:

```
trait Widget:
| abstract def Render()
trait Bordered:
| override def Render(): ...
| var Thickness
trait Button:
| implement def Render(): ...
| def Pressed(): ...
...
var A = ^new()
         ▷ Button ▷ Bordered
if A.Pressed(): A.Thickness = 10
```

The caret symbol can be prepended to any term to view trait extensions of that term in the editor, just like a type error. In this example, we see that the object being created extends Button and Bordered; it also extends Widget but this is elided because Button and Thickness themselves already extend it. Extensions for nested terms are not shown, but can be viewed by just writing the expression along with the probe; consider this example from Section 3:

```
var W = new()
W.SetPos((10,10))
W.^ParentW
    Canvas
```

Here the programmer can see that W.ParentW extends Canvas. Probing syntax allows programmers to focus on the type information they care about at the expensive of having to make direct edits; i.e. they can interactively "debug" [28] type inferences as needed.

Probes, however, do not help programmers reason about the assignment graph that propagated traits to a term in the first place Although this reasoning is "local," the assignment graph is formed from a compilation unit's entire implementation, not to mention the obtuse public signature included for the units it instantiates. The computer should help out since it knows the assignment graph. Each trait listed in a type probe or error message can be clicked on to navigate to the statement where the extended trait propagates or is seeded into the unit. However, even this is not always good enough: the programmer might need to examine the graph visually to determine what assignment or instantiation led to a type incompatibility. McAdam [14] describes how graphs can be recorded, which we already do in YinYang, and presented to programmers. Future work on YinYang can explore how term assignment graphs can be used to help programmers debug their type errors.

Unit signatures are essentially undocumented given that they are inferred like everything else. As with reasoning about an implementation, the editor can reveal the graph to the programmer, but, especially given that trait method arguments and return values are nodes in this graph, it would often be very messy. Unit signatures can also change very easily when their implementation change, possibly breaking client units in unintended ways. The best idea we can offer on this point is to add the ability to "lock" units to ensure their signatures do not change. An implementation change that changes the signature can then be detected and flagged as an error. At any rate, our implementation tracks dependencies between units and is able to incrementally re-check client code whenever unit signatures change.

6. Technology

Although we have yet to explore our type system from a rigorous theoretical perspective, we have implemented much of YinYang, sans downcasting and function parameters, in a prototype that we describe here. The core type system and inference engine consists of about 1100 lines of C# code that plugs into 2000 lines of language-specific type checking and parsing code. The type system also relies on a framework called **Glitch** (1300 lines) that supports incremental and iterative computations using optimistic transaction-like processing and dependency tracing along with the ability to roll-back effects that are no longer performed. Glitch enables incremental parsing and type checking, which is essential to YinYang's overall goal of **live programming** [16] that provides continuous semantic and execution feedback to programmers as they type. This feedback is then presented to the programmer inline with edited code as in Section 5.

Glitch greatly simplifies the implementation of our type system: type checking logic is expressed along with parsing logic as if it could all be done in one pass, which is not true in general given possible cyclic dependencies between terms, not to mention compilation units! When an AST node is processed, dependencies of what it depends on are traced and its processing is *replayed* whenever they change. Effects performed during processing are also recorded and are undone if replay does not re-perform them. Re-processing between cyclic dependent nodes continues until they no longer change; termination is ensured because we avoid repetition of slots in nested terms (Section 3). This approach also transparently handles editing by the programmer, which induces re-processing of the edited AST nodes, undoing effects that are no longer performed, and propagating new effects through the program.

Language-specific code hooks into the type system using just two operations: **assignment** that express the dataflow of encoded assignments, built-in operations, and trait method calls; and **seeding** that express that a trait is known to be immediately required on a term because of a built-in operation, a method or field to be called or accessed, and so on. The type system will handle top-level method and trait instantiation directly. Language-specific code can also allocate temporary terms, which are bound to AST nodes so that the number of root terms in a unit are finite. As an example, consider code for a C-like built-in condition operation (C ? T : F); we write a simplified version of this code in YinYang though the real more-detailed code is written in C#:

```
trait Condition:
  implement def Parse(Cursor):
    var C = Cursor.ParseExpr()
    Cursor.Consume(Lexis.Question)
    var T = Cursor.ParseExpr()
    Cursor.Consume(Lexis.Colon)
    var F = Cursor.ParseExpr()
    C.Term.Seed(Primitives.Bool)
    var Temp = this.AllocateTemp()
    Temp.AssignT(T.Term)
    Temp.AssignT(F.Term)
    this.Term = Temp
```

This code performs parsing of the condition's syntactic structure and seeds the test condition (C) with a Bool extension. The code then allocates a temporary term (Temp) and assigns T and F to it, and uses this temporary as the Term that is the expression's result to the type system. Seeding and assignment for Term inside the type system resembles the following YinYang code:

```
trait Term:
  def Seed(Trait):
    this.Hard.ExtendHS(Trait)
  def AssignT(RHS):
    this.Hard.EdgeHS(RHS.Hard)  # Edge 1
    this.Soft.EdgeHS(RHS.Hard)  # Edge 2
    for C in this.Children:
      var D = RHS.BySlot(C.Slot)
      if D == null: continue       # not shared slots
      C.Hard.EdgeHS(D.Hard)        # Edge 3
      D.Soft.EdgeHS(C.Soft)        # Edge 4
      D.Hard.EdgeHS(C.Soft)        # Edge 5
```

The Seed method adds Trait to the hard constraint set of the node while the AssignT method inserts edges as specified in Section 3 (Aliasing). Incompatibility and freeze errors are checked whenever traits are added to the hard constraint node of a term. Traits are not actually added if an error is detected and reported, which causes non-determinism in type reporting as error messages depend on order of trait extension, but prevents type errors from cascading via trait propagation. The EdgeHS method of a hard or soft constraint node simply records the assignment and propagates traits in the assignee direction; i.e.:

```
trait HardSoftNode:
  private var Extended = new()
  var Edges = new()
  def ExtendHS(Trait):
    if this.Extended.CouldAddS(Trait):
      ... # check for errors if Hard
  def EdgeHS(RHS):
    if this.Edges.AddS(RHS):
      for T in this.Extended: RHS.ExtendHS(T)
    ... # propagate assignments implicitly to child slots if necessary
```

If a nested term, EdgeHS also propagates assignment relationships down to shared child slots. Finally, a unit's implementation graph formed from HardSoftNode objects will be encapsulated into a signature by finding a new edge set (EEdges) for each visible node that refers only to other visible nodes:

```
trait HardSoftNode:
  ...
  def Encapsulate(Sig):
    if !Sig.Visible(this.Term): return
    for E in this.Edges: Encapsulate0(Sig, E)
    for C in this.Children: C.Encapsulate(Sig)
  private def Encapsulate0(Sig, E):
    if Sig.Visible(E.Term): Sig.EEdges(this).AddS(E)
    else:
      for F in E.Edges: this.Encapsulate0(Sig, F)
```

Additional logic breaks assignment loops during processing. Though not shown here, the Encapsulate method is also where whether the node is frozen or not is discovered, and what the freezing constraints are. On unit instantiation, these encapsulated edges are applied to each binding term's hard and soft constraint nodes:

```
trait HardSoftNode:
  ...
  def Instantiate(Sig, Bindings):
    var X = this.HS(Bindings.GetM(this.Term))
    for E in Sig.EEdges(this):
      X.EdgeHS(E.HS(Bindings.GetM(E.Term)))
    for T in this.Extended: X.ExtendHS(T)
    for C in this.Children:
      if Sig.Visible(C): C.Instantiate(Sig, Bindings)
  abstract def HS(Term)  # returns Term.Hard if this is Hard
                         # returns Term.Soft if this is Soft
```

If the original term has been frozen in the unit's signature, this relationship is also propagated to the binding term.

Performance Concerns

One of the major problems with our last prototype in [15] was performance: graph-processing lacked modularity and involved propagating edge bindings (not just traits), causing the graph to explode in complexity. Learning from our mistakes, we focused more on separate compilation as well as discovering a more sane way of dealing with assignment and aliasing (Section 3) so that our graphs could be "normal."

Our prototype's performance on small programs is adequate, but we have yet to reach a point in our implementation where we can test on larger more realistic programs. Given our use of Glitch, non-incremental batch performance should suffer from dependency and effect tracing overhead; i.e. we should see long load times for large compilation units followed by a responsive editing experience.

Because of separate compilation, the complexity of an implementation graph is determined by the number of root terms in the compilation unit, which loosely correspond to expressions, followed by the number of slots that each term inherits from extended traits. This is problematic because, as mentioned in Section 4, all arguments and return values of the trait's methods are also represented as slots, leading to many nested terms that is only bounded by YinYang's inability to deal with recursive nested terms. For this reason, when a trait is applied to a term, we **defer** manifesting the trait's slots as nested terms along with their sub-graphs until they are needed as part of an assignment. It remains to be seen if this is good enough for scaling, especially when slot recursion is relaxed to be more realistic.

7. Related Work

Most type systems define *type* [5] as a set of values that constraints how these values can be used. YinYang lacks such types as well as subtyping based on partial order and/or subset relationships and explicit type variables that abstract over types. Yet YinYang still provides features that are typically provided by types: type errors are detected while various kinds of inclusion, parametric, data polymorphism happen. We are unaware of any other type system that focuses on supporting code completion.

Unlike YinYang, most type inference work has been based on some form Hindley-Milner style unification [10, 18]. Relating YinYang to all previous type inference work is not practical, but a few stand out. Leroy [11] investigates how to support type inference in the presence of assignment by treating type variables involved in assignments as *dangerous* and disallowing generalization, meaning they will only be inferred to have one type. YinYang is not based on generalization and, in fact, assignment is involved in most of the YinYang's typing aspects.

Agesen explores type inference for objects in the presence of parametric polymorphism with his Cartesian Product Algorithm [1] (CPA) for type inference in Self. CPA resembles YinYang by also maintaining a graph built from assignments (an innovation of Palsberg [23]); however, in CPA's case, "clone families" propagate from assignees to assignments, generalizing their type. YinYang signatures are analogous to CPA templates; however, templates are polyvariant and must be recomputed for every combination of input types they are used for while a YinYang signature applies to all uses. CPA also does not support the data polymorphism necessary to model collection types; YinYang supports data polymorphism by instantiating trait signature graphs on extending terms, and providing the terms with their own slots to represent members of the trait. Other work [26, 27, 31] has dealt with data polymorphism using various global analyses, but we are the first to do it modularly while, from a local typing perspective, dealing precisely with covariance using the soft term nodes of Section 3.

The "type" of a YinYang compilation unit is derived directly from its implementation, which is similar to a type recovery analysis [29]. Palsberg [24] demonstrates equivalence between type systems and type recovery analysis in properties that can be checked. Anderson et al. describe type recovery for JavaScript [2] while Zhao [32] improves on these results with support for polymorphic types; however, both systems are unable to handle full JavaScript and focus on detection of structural problems rather than full semantic feedback. Recency Types [9] can recovery type information from JavaScript programs more completely, but does so with expensive flow-sensitive analysis that models the heap. YinYang, in contrast, avoids the modularity and scalability issues of type recovery by not recovering types at all, instead focusing on just usage requirements.

Typeless code appears similar to code in dynamic languages like Python [30] or Ruby [13]. However, while typing in YinYang is definitely flexible, it is still not as flexible as full dynamic typing. Additionally, "duck typing" changes fundamentally in YinYang as an object becomes a duck if it quacks, with the duck trait providing the implementation of quack; rather than, as in dynamic languages, being usable as a duck if it has its own implementation of quack. This can be seen as less safe: at least Python programmers would get a "message not understood" at run-time if the incorrect object was used as a duck, whereas a YinYang programmer might just unintentionally turn some object into a duck!

The design of mixin-like traits in YinYang are based on Scala's [21], but are enhanced with compatibility relationships to temper multiple inheritance so that traits can fully replace the role of classes. Scala supports only local type inference, but then supports advanced typing features beyond what could be expressed in YinYang such as existential types and GADTs. On the other hand, having inference define objects makes fined-grained traits much more viable as they no longer need to be extended explicitly.

8. Conclusion

Two roads diverged in a wood, and I—I took the one less traveled by, and that has made all the difference – Robert Frost.

We introduced a new option in the type system design space that diverges from much of the conventional wisdom developed about type systems over the last forty years. YinYang is not based on set theory, generalization, or unification. Instead, term assignment graphs derived directly from implementation code traces usage requirements backwards from how assignments are typically dealt with. Surprisingly, this approach works: the type system is quite flexible, encoding many kinds of polymorphism just through assignments, while separate compilation is achieved by encapsulating assignment graphs into signatures that can be instantiated without re-analyzing units for each use. There are also no special typing rules for the programmer to learn—all feedback is derived only from assignment and trait propagation.

Our results are not all roses, however. Type errors can only be understood in the context of a unit implementation as no easy-to-read syntactic types are provided, leading to a need for debugging. Name overloading is thrown away, even for members in unrelated traits, while letting the type system form object definitions without explicit programmer buy-in appears dangerous. However, programmers can reasonably cope with these problems through good tooling; and actually, the precision of the type system even enhances how tooling is capable of helping the programmer.

As future work, YinYang must more flexibly support nested term recursion while ensuring adequate scaling before being feasible for general use. We should also formally reason about the soundness of how the type system handles encapsulation and slot aliasing/covariance, and, more importantly, determine how our type system's properties compare with those of well known type systems.

References

[1] O. Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proc. of ECOOP*, pages 2–26, 1995.

[2] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for javascript. In *Proc. of ECOOP*, pages 428–452, 2005.

[3] G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. of OOPSLA/ECOOP*, pages 303–311, 1990.

[4] L. Cardelli. Program fragments, linking, and modularization. In *Proc. of POPL*, pages 266–277, 1997.

[5] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM CSUR*, 17(4):471–523, Dec. 1985.

[6] G. Curry, L. Baer, D. Lipkie, and B. Lee. Traits: An approach to multiple-inheritance subclassing. In *Proc. of SIGOA*, pages 1–9, June 1982.

[7] J. Edwards. Subtext: uncovering the simplicity of programming. In *Proc. of OOPSLA*, pages 505–518, 2005.

[8] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in haskell. *ACM TOPLAS*, 18(2):109–138, Mar. 1996.

[9] P. Heidegger and P. Thiemann. Recency types for analyzing scripting languages. In *Proc. of ECOOP*, pages 200–224, 2010.

[10] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of AMS*, 146:29–60, 1969.

[11] X. Leroy and P. Weis. Polymorphic type inference and assignment. In *Proc. of POPL*, pages 291–302, 1991.

[12] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system. caml.inria.fr, 1995–2013.

[13] Y. Matsumoto. The Ruby programming language. www.ruby-lang.org, 1996–2013.

[14] B. J. McAdam. Graphs for recording type information. Technical report, UNIVERSITY OF EDINBURGH, 1999.

[15] S. McDirmid. Escaping the maze of twisty classes. In *Proc. of SPLASH Onward!*, pages 127–138, Oct. 2012.

[16] S. McDirmid. Usable live programming. In *Proc. of SPLASH Onward!*, 2013. To Appear.

[17] B. Meyer. Beware of polymorphic catcalls. www.eiffel.com/-doc/manuals/technology/typing/cat.html, 1995.

[18] R. Milner. A theory of type polymorphism in programming. *JCSS*, 17:348–375, 1978.

[19] D. A. Moon. Object-oriented programming with Flavors. In *Proc. of OOPSLA*, pages 1–8, 1986.

[20] A. Nathan. *Windows Presentation Foundation Unleashed (WPF) (Unleashed)*. Sams, 2006.

[21] M. Odersky and M. Zenger. Scalable component abstractions. In *Proc. of OOPSLA*, pages 41–57, 2005.

[22] N. Oxhøj, J. Palsberg, and M. I. Schwartzbach. Making type inference practical. In *Proc. of ECOOP*, pages 329–349, 1992.

[23] J. Palsberg. Efficient inference of object types. In *Proc. of LICS*, pages 186–195, 1994.

[24] J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. *ACM TOPLAS*, 17(4):576–599, July 1995.

[25] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *Proc. of OOPSLA*, pages 146–161, 1991.

[26] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. In *Proc. of OOPSLA*, pages 324–340, 1994.

[27] S. A. Spoon and O. Shivers. Dynamic data polyvariance using source-tagged classes. In *Proc. of DLS*, pages 35–48, 2005.

[28] P. J. Stuckey, M. Sulzmann, and J. Wazny. Interactive type debugging in Haskell. In *Proc. of Haskell*, pages 72–83, 2003.

[29] A. M. Tenenbaum. *Type Determination for Very High Level Languages*. PhD thesis, Courant Institute of Mathematical Sciences, 1974.

[30] G. van Rossum. The Python programming language manual. www.python.org, 1990–2013.

[31] T. Wang and S. F. Smith. Polymorphic constraint-based type inference for objects, 2006.

[32] T. Zhao. Polymorphic type inference for scripting languages with object extensions. In *Proc. of DLS*, pages 37–50, 2011.