Integrating Typed and Untyped Code in a Scripting Language

Tobias Wrigstad Francesco Zappa Nardelli* Sylvain Lebresne Johan Östlund Jan Vitek
PURDUE UNIVERSITY * INRIA

Abstract

Many large software systems originate from untyped scripting language code. While good for initial development, the lack of static type annotations can impact code-quality and performance in the long run. We present an approach for integrating untyped code and typed code in the same system to allow an initial prototype to smoothly evolve into an efficient and robust program. We introduce *like types*, a novel intermediate point between dynamic and static typing. Occurrences of like types variables are checked statically within their scope but, as they may be bound to dynamic values, their usage is checked dynamically. Thus like types provide some of the benefits of static typing without decreasing the expressiveness of the language. We provide a formal account of like types in a core object calculus and evaluate their applicability in the context of a new scripting language.

Categories and Subject Descriptors D Software [*D.3 Programming Languages*]: D.3.1 Formal Definitions and Theory

General Terms Theory

Keywords Compilers, Object-orientation, Semantics, Types

1. Introduction

Scripting languages facilitate the rapid development of fully functional prototypes thanks to powerful features that are often inherently hard to type. Scripting languages pride themselves on "optimizing programmer time rather than machine time," which is especially desirable in the early stages of program development before requirements stabilize or are properly understood. A lax view of what constitutes a valid program allows execution of incomplete programs, a requirement of test-driven development. The absence of types also obviates the need for early commitment to particular data structures and supports rapid evolution of systems. However, as programs stabilize and mature—e.g. a temporary data migration script finds itself juggling with the pension benefits of a small country [31]—the once liberating lack of types becomes a problem. Untyped code, or more precisely dynamically typed code, is hard to navigate, especially for maintenance programmers not involved in the original implementation. The effects of refactoring, bug fixes and enhancements are hard to trace. Moreover performance is often not on par with more static languages. A common way of dealing with this situation is to rewrite the untyped program in a statically typed language such as C# or C++. Apart from being costly and far from guaranteed to succeed [35], a complete rewrite is likely to slow down future development as it is a snapshot of the dynamic system at one particular point in time. Not surprisingly, the idea of being able to gradually evolve a prototype into a full-fledged program within the same language has been a long standing challenge in the dynamic language community [3, 8, 22, 28, 32, 33].

An early attempt at bridging the gap between dynamic and static typing is the *soft typing* proposed by Cartwright and Fagan [12] and subsequently applied to a variety of languages [2, 9, 17, 23, 24]. Soft typing tries to transparently superimpose a type system on unannotated programs, inferring types for variables and functions. When an operation cannot be typed, a dynamic check is emitted and, possibly, a warning for the programmer. A compiler equipped with a soft type checker would never reject a program, thus preserving expressivity of the dynamically typed language. The main benefit of soft typing is the promise of more efficient program execution and warnings for potentially dangerous constructs. Its drawback is the lack of guarantees that a given piece of code is free of errors. It is thus not possible for programmers to take key pieces of their system and "make" them safe, or fast. Furthermore, no guidance is given on how to refactor code that is not typable.

Incremental typing schemes have been explored by Bracha and Griswold in Strongtalk [8] which inspired pluggable types [7], in various gradual type systems [3, 19, 26, 29, 30], and recently Typed Scheme [32, 33]. In these works, dynamically typed programs can be incrementally annotated with static type information and untyped values are allowed to cross the boundary between static and dynamic code. Run-time type checks are inserted at appropriate points to ensure that values conform to the annotations on the variables they are bound to. The strength of incremental approaches is that programmers can decide which parts of their program to annotate and will get understandable error messages when code does not type check. The drawback is that any operation, even one that is fully annotated, may fail due to a non-conforming value passed in from an untyped context. This has a direct consequence on performance as type information can not be used for optimization. Even worse, program performance may decrease substantially when type annotations are added to an untyped program.

While our goal is related to this previous work, namely to explore practical techniques for evolving scripts to programs, we come from a different perspective which impacts some of our design decisions. Unlike most of the previous work which had its root in dynamically typed languages (Smalltalk, Scheme, Ruby and JavaScript) and tried to provide static checking, we would like to provide the flexibility of dynamic languages to static languages. At the language level, we are willing to forgo some of the most dynamic features of languages, such are run-time modification of object interfaces, in languages like JavaScript or Ruby. At the implementation level, the addition of opcodes to support dynamic languages in Java virtual machines makes it possible to envision mixing typed and untyped code without sacrificing performance. The research question is thus how to integrate these different styles

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL'10, January 17–23, 2009, Madrid, Spain.
Copyright © 2009 ACM 978-1-60558-479-9/10/01...$10.00

of programming within the same language. In particular, it would not be acceptable for statically typed code to either experience run-time failures or be compiled in a less efficient to support dynamic values. Conversely, the expressiveness of dynamic parts of the system should not be restricted by the mere presence of static types in unrelated parts of the system.

We use, as a vehicle for our experiments, a new object-oriented scripting language called Thorn [6] which runs on a JVM and ought to support the integration of statically and dynamically typed code. The statically typed part of Thorn sports a conventional nominal type system with multiple subtyping akin to that of Java. Thorn has also a fully dynamic part, where every object is of type **dyn** and all operations performed on **dyn** objects are checked at run-time. We introduce a novel intermediate point, dubbed a "like type," between dynamic and compile-time checked static types. For each type C, there is a type **like** C. Uses of variables of type **like** C are checked statically and must respect C's interface. However, at run-time, any value can flow into a **like** C variable and their conformance to C is checked dynamically. Like types allow the same degree of incrementality as previous proposals for gradual typing, but we have chosen a design which favors efficiency. In contrast to inference-based systems, like types allow static checking of operations against an explicit, programmer-declared, protocol. Notably, this allows catching spelling errors and argument type errors which are simple and frequent mistakes. Furthermore they make it possible to provide IDE support such as code completion.

To summarize, this paper makes the following contributions:

- A type system that incorporates dynamic types, concrete types and like types to provide a way to integrate untyped and typed code. The separation of concrete and like types makes it possible to optimize concretely typed code, and retain flexibility in the rest of the program.
- A formalization of the type system in an imperative class-based object-oriented language; a proof of the standard theorems for typed subsets of the code; a formalization of a wrapper-less compilation scheme, and a proof of its adequacy.
- An implementation in the Thorn compiler that supports the type system and performs optimizations for concretely typed code.
- A report on an application of like types to evolve an untyped script into a partially typed program.

A technical report extended with proofs is available at `http://moscova.inria.fr/~zappa/projects/liketypes`.

2. Background and Motivating Example

This section introduces closely related work dealing with the integration of dynamically typed and statically typed code through a series of examples written in Thorn [6].

The Typing of a Point. In a language that supports rapid prototyping, it is sometimes convenient to start development without committing to a particular representation for data. Declaring a two-dimensional `Point` class with two mutable fields `x` and `y` and three methods (`getX`, `getY`, and `move`) can be done with every variable and method declaration having the (implicit) type **dyn**. Run-time checks are then emitted to ensure that methods are present before attempting to invoke them.

```
class Point(var x, var y) {
  def getX() = x;
  def getY() = y;
  def move(pt) { x:=pt.getX(); y:=pt.getY() }
}
```

As a first step toward assurance, the programmer may choose to annotate the coordinates with concrete types, say `Int` for integer, but leave the `move` method unchanged allowing it to accept any object that understands `getX()` and `getY()`. The benefit of such a refactoring is that a compiler could emit efficient code for operations on the integer fields. As the argument to `move` is untyped, casts may be needed to ensure that values returned by the getter methods are of the right type.

```
class Point(var x: Int, var y: Int) {
  def getX(): Int = x;
  def getY(): Int = y;
  def move(pt){ x:= (Int)pt.getX(); y:= (Int)pt.getY()}
}
```

Of course, this modification is disruptive to clients of the class: all places where `Point` is constructed must be changed to ensure that arguments have the proper static type. In the long run, the programmer may want more assurance for invocations of `move()`, e.g., by annotating the argument of the method as `pt:Point`. This has the benefit that the casts in the method's body become superfluous. This has the drawback that all client code must (again) be revisited to add static type annotations on arguments and decreases flexibility of the code, as clients may call `move` passing an `Origin` object.

```
class Origin {
   def getX(): Int = 0;
   def getY(): Int = 0;
}
```

While not a subclass of point, and thus failing to type check, `Origin` has the interface required by the method. This is not unusual in dynamically typed programs. Part of the last issue could be somewhat mitigated by the adoption of structural subtyping [10]. This would lift the requirement that argument of `move` be a declared subtype `Point` and would accept any type with the same signature. Unfortunately, this is not enough here, as `Origin` is not a structural subtype either. The solution to this particular example is to invent a more general type, such as `getXgetY` which has exactly the interface required by `move`.

```
class getXgetY {
  def getX(): Int;
  def getY(): Int;
}
```

This solution does not generalize as, if it was applied systematically, it would give rise to many special purpose types with little meaning to the programmer. A combination of structural and intersection types are often the reasonable choice when starting with an existing untyped language such as Ruby, JavaScript or Scheme (see for example [17, 33]) but they add programmer burden, as a programmer must explicitly provide type declarations, and are brittle in the presence of small changes to the code. For these reasons, Typed Scheme is moving from structural to nominal typing.[1]

Soft Typing. A *soft typing* system in the tradition of Cartwright and Fagan [12] would infer a type such as `getXgetY` above without programmer intervention. Thus obviating the need to litter the code with overly specific types, but soft typing is inherently brittle as something as trivial as a spelling mistake in a method name will generate a constraint that will never be satisfied and only caught when the method is actually used by client code. Also, inferred types can easily get unwieldy and hard to understand for a human programmer. Furthermore, the absence of type declarations means programmers will not have much help from their IDE. In terms of performance, run-time checks are eliminated when the compiler can show that an operation is safe. This makes the performance

[1] Matthias Felleisen, presentation at the STOP'09 (Script to Program Evolution) workshop.

model opaque as a small change in the code can have a large impact on performance simply because it prevents the compiler from optimizing an operation in a hotspot. The work on soft typing can be traced to early work by Cartwright [11] and directly influenced research on soft Scheme [36] and Lagorio et al.'s Just [2, 23] bringing soft typing to Java.

Gradual typing. The gradual typing approach of Siek and Taha allows for typed and untyped values to commingle freely [26]. When an untyped value is coerced, or cast, to a typed value, a *wrapper* is inserted to verify that all further interactions through that particular reference behave according to the target type's contract. At the simplest a wrapper is a cast $\langle T \Leftarrow R \rangle$ saying, intuitively, that the value was of type R and must behave as a value of type T. The number of wrappers is variable and can, in pathological cases, be substantial [19]. In practice, any program that has more than a single wrapper for any value is likely to be visibly slower. In the presence of aliasing and side-effects the wrappers typically can not be discharged on the spot and have to be kept as long as the value is live. The impact of this design choice is that any operation on a value may fail if that value is a dynamic type which does not abide by the contract imposed by its wrapper. Wrapper have to be manipulated at run-time and compiler optimizations are inhibited as the compiler has to emit code that assumes the presence of wrappers everywhere. Some of these problems may be avoided with program analysis, but there is currently no published work that demonstrates this. To provide improved debugging support researchers have investigated the notion of *blame control* in the context of gradual typing, [14, 29, 32, 34]. The underlying notion is that concretely typed parts of a program should not be blamed for run-time type errors. As an example, let `T` be a type with a method `m` and `x` be a variable of type `T`. Now, if some object `o`, that does not understand `m`, is stored in `T`, blame tracking will not blame the call `x.m()`—which is correct as `x` has type `T`—for throwing a "message not understood" exception at run-time. Rather, it will identify the place in the code where `o` was cast to `T`. Fine-grained blame control requires that a reference "remembers" each cast it flows through, perhaps modulo optimizations on redundant casts. Storing such information in references and not in objects is key to achieve traceability, but incurs additional run-time overhead on top of the run-time type checks. Evaluating the performance impact of blame tracking and its practical impact on the ability to debug gradually typed programs has not yet been investigated. We use the term gradual typing to refer to a family of approaches that includes hybrid typing [15] and that have their roots in a contract-based approach of [14, 18].

3. A Type System for Program Evolution

In this paper we propose a type system for a class-based object-oriented programming language with three kinds of types. *Dynamic types*, denoted by the type **dyn**, represent values that are manipulated with no static checks. Dynamic types offer programmers maximal flexibility as any operation is allowed, as long as the target object implements the requested method. However, **dyn** gives little aid to find bugs, to capture design intents, or to prove properties. At the other extreme, we depart from previous work on gradual typing, by offering *concrete types*. Concrete types behave exactly how programmers steeped in statically typed languages would expect. A variable of concrete type `C` is guaranteed to refer to an instance of `C` or one of its subtypes. Concrete types drastically restrict the values that can be bound to a variable as they do not support the notion of wrapped values found in other gradual type systems. Concrete types are intended to facilitate optimizations such as unboxing and inlining as the compiler can rely on the static type information to emit efficient code. Finally, as an intermediate step between the two, we propose *like types*. Like types combine static and dynamic checking in a novel way. For any concrete type `C`, there is a corresponding like type, written **like** `C`, with an identical interface. Whenever a programmer uses a variable typed **like** `C`, all manipulations of that *variable* are checked statically against `C`'s interface, while, at run-time, all uses of the value bound to the variable are checked dynamically. Figure 1 shows the relations between types (**dyn** will be implicit in the code snippets). Full arrows indicate traditional subtype relations (so, for instance if `B` is a subtype of `A`, then **like** `B` is a subtype of **like** `A`), dotted lines indicate implicit **dyn** casts, and finally, dashed lines show situations where **like** casts are needed.

In this paper, we have chosen a nominal type system, thus subtype relation between concrete types must be explicitly declared by **extends** clauses. While we believe that our approach applies equally well to structural types, our choice is motivated by pragmatic reasons. Using class declarations to generate eponymous types is a compact and familiar (to most programmers) way to construct a type hierarchy. Moreover, techniques for generating efficient field access and method dispatch code sequences for nominal languages are well known and supported by most virtual machines.

The first key property of like type annotations is that they are local. This is both a strength and a limitation. It is a strength because it enables purely local type checking. Returning to our example, like types allow us to type the parameter to `move` thus:

```
def move(p: like Point) {
  x := p.getX(); y := p.getY();
  p.hog();     # !Raises a compile time error!
}
```

Declaring the variable `p` to be **like** a `Point`, makes the compiler check all operations on that variable against the interface of `Point`. Thus, the call to `hog` would be statically rejected since there is no such method in `Point`. The annotation provides the static information necessary to enable IDE support commonly found in statically typed languages (but not in dynamic ones).

The second key property is that like types do not restrict flexibility of the code. Declaring a variable to be **like** `C` is a promise on how that *variable* is used and not to what *value* that variable can be bound to. For the client code, a like typed parameter is similar to a **dyn**. The question of when to test conformance between a variable's type and the value it refers to is subtle. One of our goals was to ensure that the addition of like type annotations would not break working code. In particular, adding type annotations to a library class should not cause all of its clients to break. So instead of checking at invocation time, each use of a like typed variable is preceded by a check that the target object has the requested method. If the check fails, a run-time exception is thrown. Consider

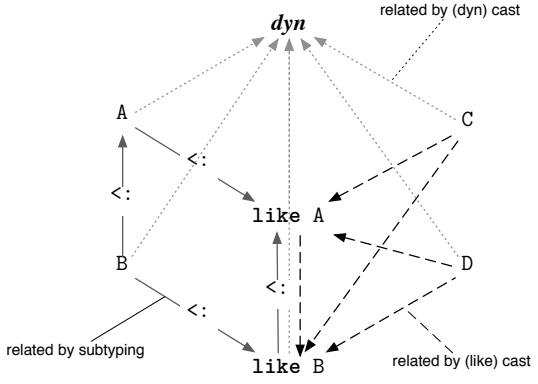

Figure 1. Type Relations. `C` and `D` are unrelated by inheritance.

the `Coordinate` class, which is similar to `Point`, but lacks a `move` method:

```
class Coordinate(var x: Int, var y: Int) {
  def getX(): Int = x;
  def getY(): Int = y;
}
```

In our running example, if `move` expects a **like** `Point`, then calling `move` with a `Coordinate` works exactly as in an untyped language. Even if `Coordinate` does not implement the entire `Point` protocol, it implements the *relevant* parts, the methods needed for `move` to run successfully. If it lacked a `getY` method, passing a `Coordinate` to `move` would compile fine, but result in an exception at run-time. More interestingly, `move` can also accept an untyped definition of `Coordinate`:

```
class Coord(x,y) {  def getX() = x;  def getY() = y;  }
```

Here, the run-time return value of `getX` and `getY` are tested against `Int`: invoking `move` with the argument `Coord(1,2)` would succeed, `Coord("a","b")` would raise an exception. Observe that if `Point` used **like** `Int`, checking the return type would not be necessary as assigning to a like type always succeeds.

Interfacing typed and untyped code. Consider a call `p1.move(p2)` with different declared types for variables `p1`, `p2` and `pt` (the type of the parameter in the `move` method). Depending on the static type information available on the receiver, different static checks are enabled, and different run-time checks are needed to preserve type-safety. We go through these in detail in Figure 2.

p1	p2	pt	Result
_	_	**dyn**	OK
_	_	**like** Point	OK
dyn	_	Point	OK
Point	**dyn**	Point	ERR
Point	**like** Point	Point	OK *
Point	Point	Point	OK
like Point	**dyn**	Point	ERR
like Point	**like** Point	Point	OK *
like Point	Point	Point	OK

Figure 2. Configurations of declared types. The column labeled Result indicate if there will be a compile-time error. Note that the calculus is slightly more strict and requires explicit casts in cases labeled *.

Assume that the parameter `pt` in `move` has type **dyn**, then all configurations of receiver and argument are allowed and will compile successfully. In case the parameter has the type **like** `Point`, again, all configurations are statically valid. The last case to consider is when `pt` has the concrete type `Point`. In that case, there are several subcases that need to be looked at. If the receiver `p1` is untyped, then, as expected, no static checks are possible. At run-time, we must consequently check that `p1` understands the `move` method and if so, that `p2`'s run-time type satisfies the type on the parameter in the `move` method. Since, `pt` is `Point`, a subtype test will be performed at run-time. If the receiver `p1` is a concrete type, the type of the argument `p2` will be statically checked: if it is **dyn**, a compile-time error will be reported; if it is **like** `Point`, the compiler will accept the call and emit a run-time subtype test; if `p2` is a `Point` a straightforward typed invocation sequence can be emitted. Finally, the case where the receiver is declared **like** `Point` is similar to the previous case, with the exception that a run-time test is emitted to check for the presence of a **move** method in `p1`.

If `move` had some concrete return type `C`, invoking it on a like typed receiver, would then check that the value returned from the method was indeed a (subtype of) `C`. If this cannot be determined statically, for instance if the actual method does not return a concrete type, then a type test is performed on the value returned. Calls with untyped receivers never need to type-check return values, as client code has no expectations that must be met. The concretely typed case follows from regular static checking.

Revisiting a previous example, consider a variant of `move` with a call to `getY` guarded by an `if` and assume that `p` is bound at run-time to an object that does not have a `getY`.

```
def move(p: like Point) {
  x := p.getX();
  if (unlikely) y := p.getY();
}
```

As the system only checks uses of `p`, the error triggers if the condition is true. Some situations, which are hard to type in systems that perform eager subtype tests, e.g., at the start of the method call, work smoothly thanks to this lazy checking. As a result like types are not structural, but "semi-structural" since they only require the methods called to be present.

Code evolution. Like types provide an intermediate step between dynamic and concrete types. In some cases the programmer might want to replace **like** `C` annotations with concrete `C` annotations, but this is not always straightforward. The reason is the shift in notion of subtype—from (a variant on) structural to nominal. Fortunately, studies of the use of dynamic features in practice in dynamically typed programs [4, 20] suggest that many dynamic programs are really not that polymorphic. When this is the case, the transition is as simple as removing the **like** keyword. Changing a piece of code that is largely like typed to use concrete types imposes an additional level of strictness on the code. Subsequently, stores from like typed (or **dyn**) variables into concretely typed variables must be guarded by type checks. The Thorn compiler inserts these checks automatically where needed and prints a warning to avoid suppressing actual compile-time errors. Notably, when accessing a concretely typed field or calling a method with concrete return type on a like typed receiver, the resulting value will be concretely typed. Subsequent operations on the returned value will enjoy the same strict type checking as all concrete values and can be compiled more efficiently than operations on like typed receivers.

In some cases, one can imagine going from typed code to untyped, for example to facilitate interaction with some larger untyped program, or to increase the flexibility in the code. Simply adding a **like** keyword in the relevant places, e.g., in front of types in the interface, or on key variables, immediately allows for a higher degree of flexibility without losing the local checking and still keeping the design intent in the code.

Compile-Time Optimizations In Thorn, all method calls go through a dispatching function. With like types, three different dispatching functions are used to perform the necessary run-time checks described above. Every user written method call is compiled down to one of those dispatching functions depending on the type information available at the call-site. The dispatching function used for untyped calls performs run-time type checks and unboxes boxed primitives. The like typed dispatching function checks that the intended method is actually present in the receiver and has compatible types. The concretely typed dispatching function performs a simple and fast lookup, knowing that the method is present. Additionally, if the static type of the argument is a like type when some concrete type is expected, the Thorn compiler will insert a run-time type test and issue a warning.

Like types allow interaction with an untyped object through a typed interface and guarantees that operations that succeed satisfy the typing constraints specified in the interface. Consider the following code snippet that declares two cells—one for untyped content and one for integers:

```
class Cell(var x) {
  def get() = x;
  def set(x') { x:= x' }
}
class IntCell(var i: Int) {
  def get(): Int = i;
  def set(j:Int) { i:= j }
}
box = Cell(32);
y = box.get();
ibox: like IntCell = box;
z: Int = ibox.get();
ibox.set(z+10);
```

If `ibox.get()` succeeds, we statically know its return type to be an `Int` since the cell is accessed through a like typed interface. Subsequent operations on `z` enjoy static type checking and can be optimized, contrarily to uses of `y`. For example, the + operation on the last line can be compiled into machine instructions or equivalent, rather than a high-level method call on an integer object. Alternatively, the programmer might explicitly cast `y` to `Int`. However, typing the cell `like IntBox` type checks all interactions with the cell statically and gives static type information about what is put into and taken from it: this requires a single annotation at a declaration rather than casts spread all over the code.

Relating Like Types to Previous Work Like types add local checking to code without restricting its use from untyped code. In contrast to gradual typing [3, 19, 26, 29, 30] and pluggable types [7], it introduces an intermediate step on the untyped–concretely typed spectrum and uses nominal rather than structural subtyping. Furthermore, it only requires operations to be present when actually used. As a result, operations on concrete types can be efficiently implemented and like types used where flexibility is desired. Typed Scheme [32, 33] uses contracts on a module level, rather that simple type annotations, and does not work with object structures. Soft typing [12] infers constraints from code, rather than lets programmers expressly encode design intent in the form of type annotations. Adding soft typing to Java [2, 23] faces similar although fewer problems. An important difference between like types and gradual typing systems like $\mathbf{Ob}^{?}_{<:}$ [26], is that code completely annotated with like types can go wrong due to a run-time type error. On the other hand, a code completely annotated with concrete types will not go wrong.

A perhaps unusual design decision is the lack of blame control. If a method fails, e.g., due to a missing method in an argument object, we cannot point to the place in the program that subsequently lead to this problem. In this respect, the blame tracking support offered by like types is not much better than what is offered by a run-time typecast error. This is a design decision. Nothing prevents adding blame control to like types in accordance with previous work (e.g., [1, 27]). The rationale for our design is to avoid performance penalties. Keeping like types blame-free allows for a wrapper-less implementation.

As part of the aborted ECMAScript 4 standard, Cormac Flanagan proposed a type system closely related to the one we present in this paper [16]. The Objective-C language has like types for objects and no concrete object types. Classes can be either $\mathbf{dyn}$ (called `id`) or like typed, and the compiler warns rather than rejects programs due to other language features that can make non-local changes to classes.

4. A Formalization of Like Types

To investigate the meta-theory of like types we define mini-Thorn, an imperative variant of FJ [21] extended with $\mathbf{dyn}$ and like types. Mini-Thorn is a language tailored to study the interaction between untyped and typed code. Compared to FJ, it lacks subexpressions but allows assignment, because aliasing, and understanding what happens when objects are accessed through different views, is essential to our study. Mini-Thorn also lacks some features of the Thorn type system (like multiple inheritance or method overloading on arity). Extending the formalization would not be difficult but would take us away from the purpose of this section. The Thorn compiler checks source code without these restrictions.

Types. We denote class names by C, D, the dynamic type by $\mathbf{dyn}$, and like types by **like** C where C is a class name.

t	::=	types
	C	class name
	like C	like class C
	$\mathbf{dyn}$	dynamic

The distinguished class name $Object$ is also the top of the subtype hierarchy. The function $\mathbf{concr}\,(t)$ tests if the type t is concrete, and returns true if t is a class name and false otherwise.

Programs. A program consists of a collection of class definitions plus a statement to be executed. A class definition

$$\mathbf{class}\ C\ \mathbf{extends}\ D\ \{\ fds\ ;\ mds\ \}$$

introduces a class named C with superclass D. The new class has fields fds and methods mds; a field is defined by a type annotation and a field name $t\,f$, while a method is defined by its name m, its signature, and its body:

$$t\ m\ (t_1\ x_1\ ..\ t_k\ x_k)\ \{\ s\ ;\ \mathbf{return}\ x\ \}\ .$$

Statements include object creation, field read, field update, method call, and cast. Fields are private to objects, and can be accessed only from an object's scope. With an abuse of notation, we will consider lists of statements, rather than trees. We omit null-pointers: the only run-time errors we are interested in this formalization are due to dynamic type-checks that fail. As a consequence, fields must be initialized at object creation.

s	::=	statements
	skip	skip
	$s_1\ ;\ s_2$	sequence
	$this\,.\,f\ =\ x$	field update
	$x\ =\ this\,.\,f$	field read
	$x\ =\ y\ .\ m\ (y_1\ ..\ y_n)$	method call
	$x\ =\ \mathbf{new}\ C\ (y_1\ ..\ y_n)$	object creation
	$x\ =\ y$	copy
	$x\ =\ (t)\ y$	cast

Static semantics. Figure 3 defines the static semantics of mini-Thorn. Method invocation on an object accessed through a variable which has a dynamic type, e.g. $x = y\ .\ m\ (y_1\ ..\ y_n)$ where $\Gamma\ \vdash\ y$: $\mathbf{dyn}$, is trivially well-typed: all type-checks are postponed to run-time. On the contrary, as in FJ, if the variable y has a concrete type, e.g. $\Gamma\ \vdash\ y\ :\ C$, then method invocation can be statically type checked; the run-time guarantees that the objects actually accessed through y are instances of the class C (or of subclasses of C) and no run-time checks are needed. Type-checking method invocation boils down to ensuring that the method exists, that the actual arguments matches the types expected by the method, and that the type of the result matches the type of the return variable. Observe that type-checking of values is performed only if the expected type is concrete, as in the hypothesis $\mathbf{concr}\,(t_i) \Rightarrow \Gamma \vdash y_i <:\ t_i$; since any value can be stored in a like or dynamic typed variable, no static type-checking is required. Like types behave as contracts between variables and contexts: if a variable has a like type, e.g. $\Gamma\ \vdash\ y$: **like** C, then a well-typed context uses it only as a variable pointing to an instance of the class C. Operations on such

The subtyping relation $<:$ is the reflexive and transitive relation closed under the rules below.

$$\frac{\mathbf{class}\ C\ \mathbf{extends}\ D\ \{\ fds\ ;\ mds\ \}}{C\ <:\ D} \qquad \frac{}{C\ <:\ Object} \qquad \frac{C_1\ <:\ C_2}{\mathbf{like}\ C_1\ <:\ \mathbf{like}\ C_2} \qquad \frac{}{C\ <:\ \mathbf{like}\ C}$$

Method lookup functions are inherited from FJ.

$$\frac{C = Object}{\mathbf{mtype}\,(m,\ C) = \bot} \qquad \frac{\mathbf{class}\ C\ \mathbf{extends}\ D\ \{\ fds\ ;\ mds\ \} \quad t\ m\ (t_1\ x_1\ ..\ t_k\ x_k)\ \{\ s\ ;\ \mathbf{return}\ x\ \} \in mds}{\mathbf{mtype}\,(m,\ C) = t_1\ ..\ t_k\ \rightarrow\ t} \qquad \frac{\mathbf{class}\ C\ \mathbf{extends}\ D\ \{\ fds\ ;\ mds\ \} \quad m \notin mds}{\mathbf{mtype}\,(m,\ C) = \mathbf{mtype}\,(m,\ D)}$$

$$\frac{\mathbf{class}\ C\ \mathbf{extends}\ D\ \{\ fds\ ;\ mds\ \} \quad t\ m\ (t_1\ x_1\ ..\ t_k\ x_k)\ \{\ s\ ;\ \mathbf{return}\ x\ \} \in mds}{\mathbf{mbody}\,(m,\ C) = x_1\ ..\ x_k\ .\ s\ ;\ \mathbf{return}\ x} \qquad \frac{\mathbf{class}\ C\ \mathbf{extends}\ D\ \{\ fds\ ;\ mds\ \} \quad m \notin mds}{\mathbf{mbody}\,(m,\ C) = \mathbf{mbody}\,(m,\ D)}$$

The typing judgment for statements, denoted $\Gamma\ \vdash\ s$, relies on the environment Γ to record the types of the local variables accessed by s. We write $\Gamma\ \vdash\ x\ <:\ t$ as a shorthand for $\Gamma\,(x) = t'$ and $t'\ <:\ t$.

$$\frac{}{\Gamma\ \vdash\ x : t}\ [\text{T\_VAR}]\ \ \Gamma\,(x) = t \qquad \frac{\Gamma\ \vdash\ s_1 \quad \Gamma\ \vdash\ s_2}{\Gamma\ \vdash\ s_1\ ;\ s_2}\ [\text{TYPE\_SEQUENCE}] \qquad \frac{\Gamma\,(x) = C \quad fields\,(C) = t_1\ f_1\ ..\ t_k\ f_k \quad \forall i\ .\ \mathbf{concr}\,(t_i)\ \Rightarrow\ \Gamma\ \vdash\ y_i\ <:\ t_i}{\Gamma\ \vdash\ x = \mathbf{new}\ C\,(y_1\ ..\ y_k)}\ [\text{TYPE\_NEW}]$$

$$\frac{\Gamma\ \vdash\ y : t}{\Gamma\ \vdash\ x = y}\ [\text{TYPE\_COPY}]\ \ \Gamma\ \vdash\ y\ <:\ t \qquad \frac{\Gamma\ \vdash\ y : t_2 \quad \Gamma\ \vdash\ x : t_1 \quad t\ <:\ t_1}{\Gamma\ \vdash\ x = (t)\ y}\ [\text{TYPE\_CAST}]$$

$$\frac{\Gamma\ \vdash\ this : C \quad fields\,(C) = t_1\ f_1\ ..\ t_k\ f_k \quad \Gamma\ \vdash\ x : t' \quad \mathbf{concr}\,(t')\ \Rightarrow\ t_i\ <:\ t'}{\Gamma\ \vdash\ x = this\ .\ f_i}\ [\text{TYPE\_FIELD}]$$

$$\frac{\Gamma\ \vdash\ this : C \quad fields\,(C) = t_1\ f_1\ ..\ t_k\ f_k \quad \Gamma\ \vdash\ x : t_i' \quad \mathbf{concr}\,(t_i)\ \Rightarrow\ t_i'\ <:\ t_i}{\Gamma\ \vdash\ this\ .\ f_i = x}\ [\text{TYPE\_ASSIGN}]$$

$$\frac{(\Gamma\ \vdash\ y : C\ \vee\ \Gamma\ \vdash\ y : \mathbf{like}\ C) \quad \mathbf{mtype}\,(m,\ C) = t_1\ ..\ t_k\ \rightarrow\ t' \quad \Gamma\ \vdash\ y_1 : t_1'\ \ ..\ \ \Gamma\ \vdash\ y_k : t_k' \quad \forall i\ .\ \mathbf{concr}\,(t_i)\ \Rightarrow\ t_i'\ <:\ t_i \quad \Gamma\ \vdash\ x : t \quad \mathbf{concr}\,(t)\ \Rightarrow\ t'\ <:\ t}{\Gamma\ \vdash\ x = y\ .\ m\,(y_1\ ..\ y_k)}\ [\text{TYPE\_CALL}]$$

$$\frac{\Gamma\ \vdash\ y : \mathbf{dyn} \quad \Gamma\ \vdash\ y_1 : t_1\ \ ..\ \ \Gamma\ \vdash\ y_k : t_k \quad \Gamma\ \vdash\ x : \mathbf{dyn}}{\Gamma\ \vdash\ x = y\ .\ m\,(y_1\ ..\ y_k)}\ [\text{TYPE\_CALL\_DYN}]$$

Typing of methods and classes is inherited from FJ.

$$\frac{x_1 : t_1,\ ..,\ x_k : t_k,\ this : C\ \vdash\ s \quad x_1 : t_1,\ ..,\ x_k : t_k,\ this : C\ \vdash\ x : t_0 \quad \mathbf{class}\ C\ \mathbf{extends}\ D\ \{\ fds\ ;\ mds\ \} \quad \mathbf{if}\ \mathbf{mtype}\,(m,\ D) = t_1'\ ..\ t_k'\ \rightarrow\ t_0'\ \mathbf{then}\ (t_1 = t_1'\ \ ..\ \ t_k = t_k')\ \wedge\ t_0 = t_0'}{C\ \vdash\ t_0\ m\,(t_1\ x_1\ ..\ t_k\ x_k)\ \{\ s\ ;\ \mathbf{return}\ x\ \}}$$

$$\frac{\mathbf{fieldnames}\,(D) = f_1'\ ..\ f_n' \quad \forall f \in f_1\ ..\ f_k\ .\ f \notin f_1'\ ..\ f_n' \quad C \vdash md_1\ ..\ C \vdash md_n}{\vdash\ \mathbf{class}\ C\ \mathbf{extends}\ D\ \{\ C_1\ f_1\ ..\ C_k\ f_k\ ;\ md_1\ ..\ md_n\ \}}$$

Figure 3. The type system

variables are then statically checked as if their type was concrete. However in this case the run-time does not guarantee that the object accessed are instances of the class C, and the conformance of the value actually accessed will be checked individually at each method invocation. These intuitions suggest that, even if it adds the overhead of redundant conformance checks, it is always safe to consider a variable of type C as a variable of type **like** C, as allowed by the subtyping rule $C\ <:\ \mathbf{like}\ C$. Similarly, it is easy to see that the **like** constructor is covariant. Since fields are private to each object, the operations to read or update them are always made in a context where the type of *this* is known with great precision, and the type constraints can be checked statically. The other rules are unsurprising.

Dynamic semantics. At run-time objects live in the heap and are referenced via pointers p. Different variables can have different views of the same object; for instance, the variables $x{:}C$, $y{:}\mathbf{like}\ D$ and $z{:}\mathbf{dyn}$ might be aliases and refer to the same object stored at location p. The dynamic semantics keeps track of a variable's view of an object using wrapped pointers (also called stack-values and denoted by sv). So the stack-value of z is $(\mathbf{dyn})\ p$ while that of y is $(\mathbf{like}\ D)\ p$. No wrapper is needed for x, whose stack-value is just the pointer p.

The dynamic semantics is then defined as a small-step operational semantics over configurations. A configuration consists of a heap H of locations p mapped to objects

$$C\,(f_1 = sv_1\ ;\ ..\ ;\ f_n = sv_n)$$

and of a stack S of activation records

$$\langle\ F_1\ \mid\ s_1\ \rangle\ ...\ \langle\ F_n\ \mid\ s_n\ \rangle$$

where each activation record consists of an environment F_i that maps variables to stack-values, and a statement s_i to be executed. Computation, defined in Figure 4, progresses by executing the statement in the stack-frame on the top of the stack. We write $H(p).f_i \mapsto sv$ to denote the object stored at $H(p)$ where the field f_i has been updated with the stack-value sv.

An invariant relates the type of variables and wrappers of stack-values:

Auxiliary functions to extract the run-time type of a pointer and of a stack-value, or to compute wrappers.

$$\frac{H(p) = C\,(...)}{\mathbf{ptype}\,(H,\ p) = C} \qquad \frac{H(p) = C\,(...)}{\mathbf{svtype}\,(H,\ p) = C} \qquad \frac{}{\mathbf{svtype}\,(H,\ (\mathbf{like}\ D)\,p) = \mathbf{like}\ D} \qquad \frac{}{\mathbf{svtype}\,(H,\ (\mathbf{dyn})\,p) = \mathbf{dyn}}$$

$$\frac{}{[\![\ C\ ]\!] =} \qquad \frac{}{[\![\ \mathbf{like}\ C\ ]\!] = (\mathbf{like}\ C)} \qquad \frac{}{[\![\ \mathbf{dyn}\ ]\!] = (\mathbf{dyn})} \qquad \frac{}{\mathbf{w2c}\,() =} \qquad \frac{}{\mathbf{w2c}\,((\mathbf{like}\ C)) = (\mathbf{like}\ C)} \qquad \frac{}{\mathbf{w2c}\,((\mathbf{dyn})) = (\mathbf{dyn})}$$

Dynamic semantics.

[RED_NEW]
$$\frac{p \text{ fresh for } H \quad \mathit{fields}\,(C) = t_1\,f_1\,..\,t_n\,f_n \quad F(y_1) = w_1\,p_1 \quad .. \quad F(y_n) = w_n\,p_n \quad sv_1 = [\![\ t_1\ ]\!]\,p_1 \quad .. \quad sv_n = [\![\ t_n\ ]\!]\,p_n}{H \mid \langle\ F \mid x = \mathbf{new}\ C\,(y_1\,..\,y_n)\,;\ s\ \rangle\ S \longrightarrow H\,[p \mapsto C\,(f_1 = sv_1\,;\,..\,;\,f_n = sv_n)] \mid \langle\ F\,[x \mapsto p] \mid s\ \rangle\ S}$$

[RED_COPY]
$$\frac{}{H \mid \langle\ F \mid x = y\,;\ s\ \rangle\ S \longrightarrow H \mid \langle\ F\,[x \mapsto F(y)] \mid s\ \rangle\ S}$$

[RED_RETURN]
$$\frac{}{H \mid \langle\ F_0 \mid \mathbf{return}\ x\ \rangle\ \langle\ F_1 \mid s_1\ \rangle\ S \longrightarrow H \mid \langle\ F_1\,[\mathit{ret} \mapsto F_0(x)] \mid s_1\ \rangle\ S}$$

[RED_CAST_CLASS]
$$\frac{F(y) = w\,p \quad \mathbf{ptype}\,(H,\ p) = D \quad D <:\ C}{H \mid \langle\ F \mid x = (C)\,y\,;\ s\ \rangle\ S \longrightarrow H \mid \langle\ F\,[x \mapsto p] \mid s\ \rangle\ S}$$

[RED_CAST_OTHER]
$$\frac{F(y) = w\,p \quad t = \mathbf{like}\ C\ \vee\ t = \mathbf{dyn}}{H \mid \langle\ F \mid x = (t)\,y\,;\ s\ \rangle\ S \longrightarrow H \mid \langle\ F\,[x \mapsto (t)\,p] \mid s\ \rangle\ S}$$

[RED_FIELD]
$$\frac{\begin{array}{l} F(\mathit{this}) = p \\ H(p) = C\,(f_1 = w_1\,p_1\,;\,..\,;\,f_n = w_n\,p_n) \\ F(x) = w'\,p' \end{array}}{H \mid \langle\ F \mid x = \mathit{this}\,.\,f_i\,;\ s\ \rangle\ S \longrightarrow H \mid \langle\ F\,[x \mapsto w'\,p_i] \mid s\ \rangle\ S}$$

[RED_ASSIGN]
$$\frac{\begin{array}{ll} F(\mathit{this}) = p & F(x) = w'\,p' \\ H(p) = C\,(f_1 = sv_1\,;\,..\,;\,f_n = sv_n) & \\ \mathit{fields}\,(C) = t_1\,f_1\,..\,t_n\,f_n & sv = [\![\ t_i\ ]\!]\,p' \end{array}}{H \mid \langle\ F \mid \mathit{this}\,.\,f_i = x\,;\ s\ \rangle\ S \longrightarrow H\,[p \mapsto (H\,(p)\,.\,f_i \mapsto sv)] \mid \langle\ F \mid s\ \rangle\ S}$$

[RED_CALL]
$$\frac{\begin{array}{lll} F(y) = p & \mathbf{ptype}\,(H,\ p) = C & \mathbf{mbody}\,(m,\ C) = x_1\,..\,x_n\,.\,s_0\,;\ \mathbf{return}\ x_0 \\ \mathbf{mtype}\,(m,\ C) = t_1\,..\,t_n \to t & F(y_1) = w_1\,p_1 \quad .. \quad F(y_n) = w_n\,p_n & sv_1 = [\![\ t_1\ ]\!]\,p_1 \quad .. \quad sv_n = [\![\ t_n\ ]\!]\,p_n \\ F(x) = w'\,p' & \mathit{cast} = \mathbf{w2c}\,(w') & \end{array}}{H \mid \langle\ F \mid x = y\,.\,m\,(y_1\,..\,y_n)\,;\ s\ \rangle\ S \longrightarrow H \mid \langle\ []\,[x_1 \mapsto sv_1\,..\,x_n \mapsto sv_n]\,[\mathit{this} \mapsto p] \mid s_0\,;\ \mathbf{return}\ x_0\ \rangle\ \langle\ F \mid x = \mathit{cast}\ \mathit{ret}\,;\ s\ \rangle\ S}$$

[RED_CALL_LIKE]
$$\frac{\begin{array}{lll} F(y) = (\mathbf{like}\ C)\,p & & \mathbf{ptype}\,(H,\ p) = D \\ \mathbf{mbody}\,(m,\ D) = x_1\,..\,x_n\,.\,s_0\,;\ \mathbf{return}\ x_0 & & \mathbf{mtype}\,(m,\ C) = t_1\,..\,t_n \to t \\ \mathbf{mtype}\,(m,\ D) = t'_1\,..\,t'_n \to t' & \forall i\,.\,t_i <:\ t'_i\ \vee\ t'_i = \mathbf{dyn} & \\ (\mathbf{concr}\,(t)\ \wedge\ \mathbf{concr}\,(t')) \Rightarrow t' <:\ t & & F(y_1) = w_1\,p_1 \quad .. \quad F(y_n) = w_n\,p_n \\ sv_1 = [\![\ t'_1\ ]\!]\,p_1 \quad .. \quad sv_n = [\![\ t'_n\ ]\!]\,p_n & F(x) = w'\,p' & \mathit{cast} = \mathbf{w2c}\,(w') \end{array}}{H \mid \langle\ F \mid x = y\,.\,m\,(y_1\,..\,y_n)\,;\ s\ \rangle\ S \longrightarrow H \mid \langle\ []\,[x_1 \mapsto sv_1\,..\,x_n \mapsto sv_n]\,[\mathit{this} \mapsto p] \mid s_0\,;\ \mathbf{return}\ x_0\ \rangle\ \langle\ F \mid x = \mathit{cast}\,(t)\ \mathit{ret}\,;\ s\ \rangle\ S}$$

[RED_CALL_DYN]
$$\frac{\begin{array}{lll} F(y) = (\mathbf{dyn})\,p & \mathbf{ptype}\,(H,\ p) = C & \mathbf{mbody}\,(m,\ C) = x_1\,..\,x_n\,.\,s_0\,;\ \mathbf{return}\ x_0 \\ \mathbf{mtype}\,(m,\ C) = t_1\,..\,t_n \to t & F(y_1) = w_1\,p_1 \quad .. \quad F(y_n) = w_n\,p_n & \\ \forall i\,.\,\mathbf{concr}\,(t_i) \Rightarrow \mathbf{svtype}\,(H,\ w_i\,p_i) <:\ t_i & sv_1 = [\![\ t_1\ ]\!]\,p_1 \quad .. \quad sv_n = [\![\ t_n\ ]\!]\,p_n & \end{array}}{H \mid \langle\ F \mid x = y\,.\,m\,(y_1\,..\,y_n)\,;\ s\ \rangle\ S \longrightarrow H \mid \langle\ []\,[x_1 \mapsto sv_1\,..\,x_n \mapsto sv_n]\,[\mathit{this} \mapsto p] \mid s_0\,;\ \mathbf{return}\ x_0\ \rangle\ \langle\ F \mid x = (\mathbf{dyn})\ \mathit{ret}\,;\ s\ \rangle\ S}$$

Figure 4. Dynamic semantics

1. if a variable x has a concrete type C, then its stack-value will always be an unwrapped pointer p and the pointer will always point in the heap to a valid object of type (or subtype of) C;
2. if a variable x has type **like** C, then its stack-value will always be a (**like** C) p wrapped pointer; no guarantee about the type of the object pointed to by p in the heap;
3. if a variable x has type $\mathbf{dyn}$, then its stack-value will always be a $(\mathbf{dyn})$ p wrapped pointer; no guarantee about the type of the object pointed to by p in the heap.

To preserve this invariant across reductions, operations on objects must perform different checks according to their view (that is, the wrapper stored in the stack-value) of the object.

Suppose that a stack-value $w\,p$ must be stored in a local variable x (or in an object field). Let t be the static type of x. If t is some concrete type C, then the static semantics and the run-time invariant guarantee that the type of the stack-value $w\,p$ is compatible with C: this implies that the wrapper w is empty and p points to an object of type D for $D <: C$. In this case, the link $x \mapsto p$ can be safely stored in the stack, and the invariant is preserved. If t is **like** C (resp. $\mathbf{dyn}$), then any pointer can be used to build a valid stack-value for x, provided that it is wrapped properly in a (**like** C) (resp. $(\mathbf{dyn})$) wrapper. The appropriate wrapper is built by the function $[\![t]\!]$ when the type t is known, or, in some cases, copied from the old stack-value of x.

For instance, when a new object is created, (rule ([RED_NEW]), its fields are initialized with stack-values that are built by wrapping (if needed) the actual arguments according to the field types. Field

update goes along similar lines. Field read illustrates a subtlety: when executing $x = this.f$ the static type of x is not easily accessible. However, since the variable x is in the scope, its current stack-value already reflects the view that the variable has of objects. In particular, if the static type of x was **like** C (resp. $\mathbf{dyn}$), then its current stack-value contains a (**like** C) (resp. $(\mathbf{dyn})$) wrapper. The semantics simply updates the pointer, bundling it with the older wrapper (a cast is built from a wrapper by the function **w2c**). Since fields are private to each object, the type of the enclosing object is known precisely (the variable $this$ always has a concrete type), and no extra care is required to check the type constraints.

Invoking a method (say $x = y.m(y_1 .. y_n)$), requires more care, as different checks and actions must be performed according to view that the variable y has of the object. The first condition of the three rules for method invocation tests exactly this: if the object is accessed via a like or dyn wrapper, or not.

If it is accessed directly (rule [RED_CALL]), that is if $F(y) = p$, then the run-time invariant guarantees that the object on which the method is called is an instance of the class statically checked. The static semantics guarantees that the method m exists. Let $t_1 .. t_n \rightarrow t$ be the type of the method; if some t_i is a concrete type, then the static semantics also guarantees that the actual argument y_i is an instance of t_i, and no run-rime type checks are needed. If t_i is like or dyn, then the actual arguments are wrapped with $[\![t_i]\!]$, and, again, no run-time checks are performed. The run-time allocates a new stack-frame to evaluate the body of the method in an environment where the actual arguments are bound to the method parameters and $this$ points to the object itself. The return value is passed to the previous stack-frame via the ret distinguished variable. If the return value must be stored in a variable that has like or dyn type, then a cast (computed from the previous stack-value of the return variable) around ret ensures that the new stack-value will be properly wrapped. Observe that this rule boils down to standard FJ method invocation if the type of the method m does not involve like or dyn types.

If the object is accessed via a $(\mathbf{dyn})$ wrapper, that is if $F(y) = (\mathbf{dyn})\, p$, (rule [RED_CALL_DYN]), then the run-time must verify that the method exists (contrarily to the previous case, the condition $\mathbf{mbody}(m, D) = ...$ might fail), and that the actual arguments are compatible with the types expected by the method, via the condition $\mathbf{svtype}(H,\ w_i\ p_i) <:\ t_i$ (this check is performed only if t_i is a concrete type, otherwise the arguments are simply wrapped according to t_i). Also, the returned pointer is bundled in a $(\mathbf{dyn})$ wrapper via a cast, since there are no static guarantees about the use that the context makes of the returned pointer.

If the object is accessed through a (**like** C) wrapper, that is if $F(y) = (\textbf{like}\ C)\, p$, (rule ([RED_CALL_LIKE]), then the arguments of the method call have been statically type-checked against the type of the method m in class C (). The run-time must then check that a method of name m exists in the object actually accessed (which can be an instance of some class D not related to C), and that its type is compatible with the type of m in C (via the condition $\forall i.t_i <: t'_i$). This strict type matching is not required when t'_i is of type $\mathbf{dyn}$, as the argument will be wrapped with $(\mathbf{dyn})$ anyway. The return value must be wrapped (via casts) not only to the type of the return variable, but also to the return type t of the method m in class C.

Mini-Thorn's dynamic semantics does not rely on chains of wrappers and every reference to an object goes through *at most* one wrapper. Upcast of class types is always allowed: when the cast is evaluated no new wrappers are added, and the previous one (if any) is discarded (rule [RED_CAST_CLASS]). A cast to a concrete type that is not a super-type of the type of the object fails. Casting to a like type as **like** C (resp. to $\mathbf{dyn}$) always succeeds (rule [RED_CAST_OTHER]); the run-time forgets the previous wrapper (if any) and insert a (**like** C) (resp. a $\mathbf{dyn}$) wrapper.

The rule for copy of a variable is straightforward, while the $\mathbf{return}\, x$ instruction simply deallocates the current stack-frame and stores the stack-value of x in the distinguished ret variable of the previous stack-frame.

4.1 Meta-theory

A configuration is *well-typed* if it satisfies the run-time invariant informally described above. The invariant relates the static type of each variable to the stack-value and heap object it can refer to, and we show that well-typed configurations are closed under reductions.

The environment Σ associates class names C to pointers p, and it records the concrete types of the objects in the heap. We then define a type relation for stack-values:

$$\frac{\Sigma(p) = D \quad D <:\ t}{\Sigma \vdash\ p : t} \qquad \frac{\Sigma(p) = E \quad E <:\ C}{\Sigma \vdash\ (\textbf{like}\ D)\, p : \textbf{like}\ C} \qquad \frac{\Sigma(p) = C}{\Sigma \vdash\ (\mathbf{dyn})\, p : \mathbf{dyn}}$$

The key property of this relation is that either it reflects the wrapper of the stack-value (imposing no conditions on the actual object accessed), or, if no wrappers are present, the concrete type of the object actually accessed. A heap H is then well-typed in Σ if:

$$\frac{\begin{array}{l}\Sigma \vdash\ H \qquad p \notin dom(H)\ \ \Sigma(p) = C \\ fields\,(C) = t_1\, f_1 .. t_n\, f_n \\ \Sigma \vdash\ w_1\, p_1 : t_1 \quad .. \quad \Sigma \vdash\ w_n\, p_n : t_n\end{array}}{\Sigma \vdash\ H\,[p \mapsto C\,(f_1 = w_1\, p_1\ ;\ ..\ ;\ f_n = w_n\, p_n)]}$$

and the definitions of well-typed stack and well-typed configuration (denoted $\Sigma\,;\,\Gamma \vdash\ H \mid S$) follow accordingly:

$$\frac{\begin{array}{l}\Gamma(x) = t \\ \Sigma \vdash\ w\, p : t \\ x \notin dom(F) \\ \Sigma\,;\,\Gamma \vdash\ F\end{array}}{\Sigma\,;\,\Gamma \vdash\ F\,[x \mapsto w\, p]} \qquad \frac{\begin{array}{l}\Sigma\,;\,\Gamma \vdash\ F \\ \Gamma \vdash\ s\end{array}}{\Sigma\,;\,\Gamma \vdash\ H \mid S} \qquad \frac{\Sigma\,;\,\Gamma \vdash\ H \mid S}{\Sigma\,;\,\Gamma \vdash\ H \mid \langle\, F \mid s \,\rangle\, S}$$

(we omit the trivial rules for empty heap and stack).

THEOREM 1 (Preservation). *If* $\Sigma\,;\,\Gamma \vdash\ H \mid S$ *and* $H \mid S \longrightarrow H' \mid S'$, *then there exist* Σ' *and* Γ' *such that* $\Sigma,\, \Sigma'\,;\,\Gamma, \Gamma' \vdash\ H' \mid S'$.

Given a well-typed program, it is easy to see that the initial configuration of the program is well-typed. This guarantees that variables with a concrete type C will only point to unwrapped objects which are instances of class C: it is safe to optimize accesses to such variables at compile time.

We can also show that no type-related run-time errors can arise from operations on variables which have a concrete type. We say that a configuration $H \mid \langle\, F \mid s \,\rangle\, S$ is *stuck* if s is non-empty and no reduction rule applies; stuck configurations capture the state just before a run-time error.

THEOREM 2 (Progress). *If a well-typed configuration* $\Sigma\,;\,\Gamma \vdash\ H \mid \langle\, F \mid s \,\rangle\, S$ *is stuck, that is* $H \mid \langle\, F \mid s \,\rangle\, S \not\longrightarrow$, *then the statement* s *is of the form* $x = y.m(y_1 .. y_n)\,;\,s'$ *and* $\Gamma(y)$ *is* $\mathbf{dyn}$ *or* ***like*** C *for some* C, *or* s *is of the form* $x = (C)\, y\,;\,s'$ *and* $F(y) = w\, p$ *with* $\textbf{\textit{ptype}}(H, p) \not<: C$.

4.2 Compilation

Run-time wrappers reflect the static view that a variable has of an object. The run-time invariant guarantees that all the stack-values associated to a given variable will have the same wrapper, and that this wrapper depends only on the static type of the variable. The

[RED_CALL_TARGET]

$$\frac{F(y) = p \qquad \mathbf{ptype}(H,\, p) = C \qquad \mathbf{mbody}(m,\, C) = x_1 \,..\, x_n \,.\, s_0 \,;\, \mathbf{return}\, x_0 \qquad F(y_1) = p_1 \quad .. \quad F(y_n) = p_n}{H \mid \langle\, F \mid x = y \,@\, m\,(y_1 \,..\, y_n)\,;\, s \,\rangle\, S \;\longrightarrow\; H \mid \langle\, [\,]\,[x_1 \mapsto p_1 \,..\, x_n \mapsto p_n]\,[\mathit{this} \mapsto p] \mid s_0 \,;\, \mathbf{return}\, x_0 \,\rangle\, \langle\, F \mid x = \mathit{ret}\,;\, s \,\rangle\, S}$$

[RED_CALL_LIKE_TARGET]

$$\frac{\begin{array}{l} F(y) = p \qquad \mathbf{ptype}(H,\, p) = D \qquad \mathbf{mbody}(m,\, D) = x_1 \,..\, x_n \,.\, s_0 \,;\, \mathbf{return}\, x_0 \\ \mathbf{mtype}(m,\, C) = t_1 \,..\, t_n \,\to\, t \qquad \mathbf{mtype}(m,\, D) = t'_1 \,..\, t'_n \,\to\, t' \qquad \forall i \,.\, t_i <: t'_i \;\vee\; t'_i = \mathbf{dyn} \\ (\mathbf{concr}(t) \;\wedge\; \mathbf{concr}(t')) \;\Rightarrow\; t' <: t \qquad F(y_1) = p_1 \quad .. \quad F(y_n) = p_n \end{array}}{H \mid \langle\, F \mid x = y \,@\, (\mathbf{like} C)\, m\,(y_1 \,..\, y_n)\,;\, s \,\rangle\, S \;\longrightarrow\; H \mid \langle\, [\,]\,[x_1 \mapsto p_1 \,..\, x_n \mapsto p_n]\,[\mathit{this} \mapsto p] \mid s_0 \,;\, \mathbf{return}\, x_0 \,\rangle\, \langle\, F \mid x = \mathit{ret}\,;\, s \,\rangle\, S}$$

[RED_CALL_DYN_TARGET]

$$\frac{\begin{array}{l} F(y) = p \qquad \mathbf{ptype}(H,\, p) = C \qquad \mathbf{mbody}(m,\, C) = x_1 \,..\, x_n \,.\, s_0 \,;\, \mathbf{return}\, x_0 \\ \mathbf{mtype}(m,\, C) = t_1 \,..\, t_n \,\to\, t \qquad F(y_1) = p_1 \quad .. \quad F(y_n) = p_n \\ \forall i \,.\, \mathbf{concr}(t_i) \;\Rightarrow\; \mathbf{ptype}(H,\, p_i) = D \;\wedge\; D <: t_i \end{array}}{H \mid \langle\, F \mid x = y \,@\, (\mathbf{dyn})\, m\,(y_1 \,..\, y_n)\,;\, s \,\rangle\, S \;\longrightarrow\; H \mid \langle\, [\,]\,[x_1 \mapsto p_1 \,..\, x_n \mapsto p_n]\,[\mathit{this} \mapsto p] \mid s_0 \,;\, \mathbf{return}\, x_0 \,\rangle\, \langle\, F \mid x = \mathit{ret}\,;\, s \,\rangle\, S}$$

Figure 5. Dynamic semantics of method dispatch in the compiled language

$$\frac{d = [\![\, \Gamma\,(y) \,]\!]}{[\![\, \Gamma,\; x = y \,.\, m\,(y_1 \,..\, y_n) \,]\!] \;\triangleq\; x = y \,@\, d\; m\,(y_1 \,..\, y_n)} \qquad \frac{}{[\![\, \Gamma,\; F\,[x_1 \mapsto w_1\; p_1 \,..\, x_n \mapsto w_n\; p_n] \,]\!] \;\triangleq\; [\![\, \Gamma,\; F \,]\!]\,[x_1 \mapsto p_1 \,..\, x_n \mapsto p_n]}$$

$$\frac{}{[\![\, \Gamma,\; H \mid S \,]\!] \;\triangleq\; [\![\, \Gamma,\; H \,]\!] \mid [\![\, \Gamma,\; S \,]\!]} \qquad \frac{}{[\![\, \Gamma,\; H\,[p \mapsto C\,(f_1 = w_1\; p_1\,;\, ..\,;\, f_n = w_n\; p_n)] \,]\!] \;\triangleq\; [\![\, \Gamma,\; H \,]\!]\,[p \mapsto C\,(f_1 = p_1\,;\, ..\,;\, f_n = p_n)]}$$

$$\frac{}{[\![\, \Gamma,\; \langle\, F_1 \mid s_1 \,\rangle \,..\, \langle\, F_n \mid s_n \,\rangle \,]\!] \;\triangleq\; \langle\, [\![\, \Gamma,\; F_1 \,]\!] \mid [\![\, \Gamma,\; s_1 \,]\!] \,\rangle \,..\, \langle\, [\![\, \Gamma,\; F_n \,]\!] \mid [\![\, \Gamma,\; s_n \,]\!] \,\rangle}$$

Figure 6. Compilation of method invocation and of configurations

dynamic semantics relied on wrappers to determine the correct reduction rule for method invocation: since the wrapper information can be derived from the static types, it is possible to determine for each method invocation the right behavior statically.

We can then define the three different dispatchers for method invocation, identified by a *dispatch label*, denoted d, which is either empty, or (**like** C), or ($\mathbf{dyn}$). With an abuse of notation, we use the same syntax for wrappers and dispatch labels, and rely on the function $[\![t]\!]$ described above to compute the appropriate dispatch label for a given type. These dispatchers implement the three reduction rules for method invocation. A method invocation can then be compiled down to the invocation of the right dispatcher for the given static type, and wrappers can be erased from stack-values altogether.

Consider the *target* language defined by the grammar below:

s	::=	statements
	skip	skip
	$s_1 \,;\, s_2$	sequence
	$\mathit{this} \,.\, f = x$	field update
	$x = \mathit{this} \,.\, f$	field read
	$x = y \,@\, d\; m\,(y_1 \,..\, y_n)$	method dispatch
	$x = \mathbf{new}\; C\,(y_1 \,..\, y_n)$	object creation
	$x = y$	copy
	$x = (t)\; y$	cast

In the semantics of the target language, stack-values are always unwrapped pointers: the stack simply associates variables to pointers. The reduction rules for method dispatch are reported in Figure 5; the reduction rules for the other constructs are inherited from the source language simply by erasing all the wrappers.

The *compile* function, denoted $[\![-]\!]$, transforms well-typed source statements into target statements, and more generally well-typed source configurations into target configurations. The compile function, described in Figure 6, is the identity on statements except for method invocation. Method invocation is compiled into the invocation of the appropriate dispatch function, according to the static type of the variable pointing to the object. Compilation of configurations compiles all the statements in all the stack-frames, and discards all the wrappers.

We can show a simulation result between the behavior of well-typed source configurations and the behavior of compiled configurations.

THEOREM 3 (Compilation). *Let* $\Sigma\,;\, \Gamma \;\vdash\; H \mid S$ *be a well-typed source configuration':*

1. *if* $H \mid S \;\longrightarrow\; H' \mid S'$, *then* $[\![\, \Gamma,\; H \mid S \,]\!] \;\longrightarrow\; [\![\, \Gamma,\; H' \mid S' \,]\!]$;
2. *conversely, if* $[\![\, \Gamma,\; H \mid S \,]\!] \;\longrightarrow\; H'' \mid S''$, *then there exists a well-typed source configuration* $\Sigma'\,;\, \Gamma' \;\vdash\; H' \mid S'$ *such that* $H \mid S \;\longrightarrow\; H' \mid S'$ *and* $[\![\, \Gamma',\; H' \mid S' \,]\!] = H'' \mid S''$.

The Thorn implementation is built upon this *wrapper-less* compilation strategy.

Generics. Like types extends nicely to a language that features bounded parametric polymorphism. Following FGJ, let X range over type variables and let concrete types $CN \;::=\; C\langle T1..Tn \rangle$, non-variable types $N \;::=\; CN \mid \mathbf{like}\; CN \mid \mathbf{dyn}$ and types $T \;::=\; N \mid X$. The key design decision is to restrict bounds in class definitions to concrete types:

$$\mathbf{class}\; C\,\langle\, X_1 \,\triangleleft\, CN_1 \,..\, X_k \,\triangleleft\, CN_k \,\rangle \,\triangleleft\, N\,\{\, \mathit{fds}\,;\, \mathit{mds} \,\}$$

where $\triangleleft$ abbreviates **extends**. If the programmer specifies a bound different from *Object*, like in *List*$\langle X \triangleleft \mathit{Foo} \rangle$, then the parameter can only be instantiated by concrete types and the usual FGJ type guarantees are recovered. If the type *Object* is instead specified as a bound, as in *List*$\langle X \triangleleft \mathit{Object} \rangle$, then the parameter can be instantiated

with any type, including **dyn** or **like** C: This guarantees ease of reuse of the *List* class.

5. Experience with Program Evolution

We report on our experience using the proposed type system. We implemented support for like types in our Thorn compiler which runs on top of a standard JVM. Method calls go through a dispatching function that is used to implement dispatch in the presence of multiple inheritance which the JVM does not support. What dispatch function to route a call through is determined by the type information available at the call-site. The concretely typed function simply handles lookup as it assumes that the run-time types of arguments are correct. The untyped function additionally performs run-time type checks for arguments to any concretely typed parameters. A call to a method `m` on a like typed receiver `x` goes trough a dispatching function that checks that `x` has an `m` with the correct types before proceeding. As a consequence of this design, calls with varying degree of typing are handled differently. Type checks are performed not at the call-site but as part of the dispatching function in the receiver object. Consequently, adding type information to some class does not require recompilation of client code to take advantage of the type checking. Run-time type checks will be carried out as part of the untyped dispatching function, and just like in e.g., Java, changing concrete types in the interface can break client code that was compiled assuming other types in the interface.

5.1 Types In Libraries

Thorn's libraries constitute a first interesting test case. To document design intents, the initial library implementation included comments that described the expected type of the functions. From these comments, we refactored libraries to use a mixture of like types and concrete types. The choice of appropriate type annotations for the interfaces of a class calls for a compromise between flexibility vs. safety and performance. Most of Thorn's libraries have like typed interfaces for maximal flexibility. Most return types are either like typed or dynamically typed. This is unsurprising since like types primarily provide local guarantees. Return values that were locally created generally have a known (concrete) type.

When adding types to untyped code, we found that it is key that the effects of adding the types do not propagate "too far." As an example, consider the following two classes, declared in separate files:

```
class A { def p(x) = println(x); }
class B { a = A();   def q(s) { a.p(s) } }
```

At a later date, the class `A` is replaced with a typed one, obtaining:

```
class A { def p(x: String) = println(x); }
```

Despite the type annotation, the signatures of the untyped and like typed dispatching functions in `A` are unchanged. The first additionally performs a type test on the `x` argument to make sure it is a `String`. Thus, old bytecode generated from the untyped code in `B` will still work with `A` without recompilation. Notably, concretely typed and like typed code will call a dispatching function that does not need to test run-time types of arguments.

Thorn supports a notion of pure classes that create immutable objects. A pure class is a functional immutable data type and many of the standard library data types are pure, `Int`, `Float`, `String` etc. The increase the flexibility, we could, although we have not implemented this, allow value classes to be automatically augmented with a parallel, like typed version, e.g.,:

```
class Int: Value {
  def +(x: Int): Int = ...
}
```

can be compiled into

```
class Int: Value {
  def +(x: Int): Int = ...
  def +(x: like Int): like Int = ...
}
```

where the second method *overloads* the first to allow + etc. to be called on any argument. This facilitates interaction between typed and untyped code, and is safe since pure classes do not modify state. Notably, `x+"foo"` when `x: Int` is still rejected statically. Following this practise would allows us to annotate most basic data types in the standard library with concrete types for speed while enjoying the flexibility of like types.

5.2 Refactoring an Untyped Program

We ported a dynamic program to Thorn along with its libraries and gradually added type annotations to it. The application we chose was Pwyky [25] a simple wiki of about 1,000 lines of Python. Pwyky relies on a generic parser module that was also ported (another 1,000 lines). This allowed us to investigate the interaction between library and user code annotations. The application is representative of scripting language code and relies on patterns that are inherently hard to type, such as using the same variable for values of different types depending on some run-time test. The ported program, "Thyky," is about the same size as the initial Python program, including libraries (2,000 LOC). Once we had an untyped version of Thyky running, we set out to gradually add type annotations to it. To illustrate this process, consider the function `upos` in the `ParserBase` class. The function `upos` is called when moving from a position `i` in the parsed string to some position `j` to update the fields `lineno` and `offset` of the `ParserBase`:

```
class ParserBase(var lineno, var offset, var rawdata) {
  def upos(i, j) {
    if (i >= j) return j;
    nlines = count(rawdata.slice(i, j), "\n");
    if (nlines != 0) {
      lineno := lineno + nlines; offset := j;
    } else offset := offset + j-i;
    return j;
  }
}
```

As a first step, we added like type annotations in a naive and straightforward way. The result was the following:

```
class ParserBase(var lineno: like Int,
                 var offset: like Int,
                 var rawdata: like String) {
  def upos(i: like Int, j: like Int): like Int {
    if (i >= j) return j;
    nlines: Int = count(rawdata.slice(i, j), "\n");
    ... # identical
}
```

Even if the types are simple data types, there is a rationale behind the choice of **like** types. `ParserBase` was the first class we annotated. This class is extended by the class `HTMLParser` and in turn by the class `Wikify` which at that time were still untyped: concrete types would have caused a number of implicit type tests to be inserted and a large number of warnings, since methods in `ParserBase` were called with untyped arguments. With the like type annotations, the type checker is now able to verify that code in `ParserBase` respects the declared types for `lineno`, `offset` and `rawdata`. The variable `nlines` is concretely typed: the `count` function returns an `Int` and, since `nlines` is internal to `upos`, there is no extra need for flexibility here. We added like type annotations to `HTMLParser` and all the code in `Thyky` in the same fashion. At this point we had an untyped and a like-type annotated version of

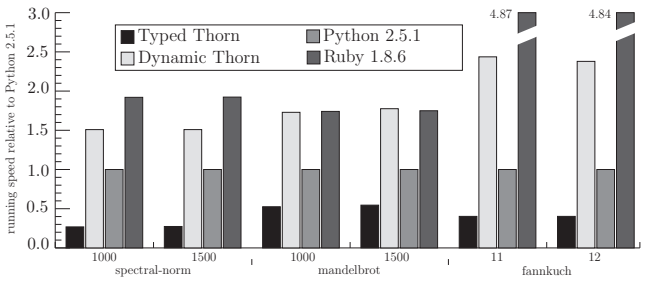

Figure 7. Performance comparison between Typed Thorn, dynamic Thorn, Python 2.5.1, Ruby 1.8.6 normalized on the Python timings. Typed Thorn notably runs the benchmarks between 2x and 4x the speed of Python.

each file; it was possible to compile and link the annotated version of one file against the untyped version of the other, and all the eight possible combinations worked as expected.

Following the annotations above, we attempted to harden the type annotations of classes such as `ParserBase` by turning the like-type annotations into concrete types. This often amounted to removing the **`like`** keywords from field declarations, while we kept the **`like`** type annotations for the arguments of function `upos` to allow calls to `upos` from untyped contexts without forcing run-time type tests and to retain the flexibility of dynamic typing. This meant that we had to rewrite the assignment to the, now concretely typed, `offset` field to make a type test on `j`, written as follows in Thorn:

```
offset := (Int) j orelse Int(j); # cast or covert to int
```

This line of code tries to cast `j` to an `Int` and if failing, tries to create a new integer value from `j`. (The **`orelse`** keyword executes its RHS if the LHS throws an exception.)

Our exercise revealed a bug in the original Python program that had survived the port to Thorn. In the code below, the variable `s` always contains a string at run-time, but was used in the following test in Python:

```
if (s < 10): area = s[0:pos+10]
else: area = s[pos-10:pos+10]
```

The test (comparison on strings and integers) is nonsensical, but nevertheless valid Python code, and always returns false. As soon as we added a type annotation to `s`, our type-checker caught the problem.

5.3 Optimizing Thorn

To demonstrate the value of concrete types, we present timing data from running three simple benchmarks ported from [13]. The original code was ported straight from Python and thus did not have any type annotations (reported as dynamic Thorn). We subsequently added type annotations to parameters to the critical functions. Running the programs side by side, we observed significant speed ups in the typed version. To give some perspective to these numbers, we present our timed runs in relation to the C implementation of Python (2.5.1) and also include the C implementation of Ruby (1.8.6), two relatively similar class-based object-oriented scripting languages. The data is presented in Figure 7. It should be noted that all the library code used by our Thorn programs is untyped. Typed version of the libraries are currently being written and should further improve performance. As shown in Figure 7, Typed Thorn runs the benchmarks between 2x and 4x faster than Python and about 3x and 6x faster than dynamic Thorn. The Ruby implementation is the slowest by far and is outperformed by a factor 7x to 12x by Typed Thorn.

Dissecting Spectral-Norm. As demonstrated in Figure 7, adding type annotations to programs in Thorn can cause significant speed-ups. Let us look at the spectral-norm benchmark [13] in additional detail. We naively translated the existing Python implementation into Thorn. Inspecting the code, we found the following frequently executed function:

```
def a(i, j) =
  1.0 / (((i + j) * (i + j + 1) >> 1) + i + 1);
```

With boxed Thorn primitives this line of code creates 8 new instances of `Int` or `Float` causing the method to execute slowly. The compiled Thorn code for this function is 87 byte code instructions of which 8 are `invokeinterface`. Adding concrete type annotations to the method's arguments brought the number of objects created down to 1, the (untyped) return value:

```
def a(i: Int, j: Int) = ...
```

Moreover, the produced bytecode for the calculation is equivalent to that of Java—18 instructions. This speed-up should not come as a surprise. After all, we have added concrete type annotations to allow the compiler to generate the optimized code for 32-bit integer values. But we note that with a traditional gradual typing system, it would not be possible to achieve this due to the need to account for wrappers (or structural subtyping). Now, let's examine what would happen if we typed the arguments with like types:

```
def a(i: like Int, j: like Int) =
  1.0 / (((i + j) * (i + j + 1) >> 1) + i + 1);
```

As the methods + and >> in class `Int` are annotated to accept **`like`** `Int` and return `Int` (a reasonable choice with respect to interaction with mixed-typed code), the highlighted additions above would be method calls, and the rest optimized into operations on primitive values. Notably, no unboxing of primitive values and no new creation of boxed integer *objects* are needed. Clearly, the like typed approach produces more efficient bytecode than the untyped.

6. Conclusions

We presented a type system designed to allow the gradual integration of values of dynamic and static types in the same programming language. Our design departs from the majority of previous work which takes an existing dynamic language as a starting point and insists that the type system be somehow backwards compatible with legacy untyped code. In our view, the drawback of those works is that the static type system is necessarily weak and fails to rule out run-time errors or permit compiler optimizations. Our proposal puts the dynamic and static parts of a program on equal footing. While dynamically typed code is unconstrained, we guarantee that statically typed code does not experience run-time type errors. By separating (semi)-structural like types from concrete types, we are able to treat the latter more strictly and as a consequence apply compiler optimizations to the generated code. Like types interact very well with untyped code, in particular, adding like type annotation will never cause working code to fail due to type errors.

Choosing nominal subtyping for the statically typed part of our design is in line with all modern object-oriented languages. But our decision of reusing type names without requiring structural subtyping for like types is more controversial. We argue that it is in line with the design philosophy of scripting languages: namely to minimize programmer effort. Like types do not require the scripting language programmer to declare new types while a program is being migrated from untyped to typed, instead they let them reuse existing types even if these types are only an approximation of the "right" ones. As such, a program with like type annotations already requires more "finger typing" (the pejorative term for inserting

type information used by the scripting community) than completely untyped code. But the additional effort is small and optional as it is always possible to interact with a like typed library without writing a single type annotation and the library will still enjoy some safety and speed-up. Like types are good for documentation and traceability. Although they impose weaker constraints on behavior than in a language such as Java, like typed code will be forced to evolve as the referenced types evolve. A consequence of the definition of like types is that exactly what subset of the operations of a type is used within a method is not visible on the outside. From a dynamic typing perspective, this is positive as it decreases coupling and makes code more modular. This is similar to the Smalltalk pattern of encoding type information in variable names [5] and essentially the same reasoning that is used by programmers in object-oriented scripting languages such as Ruby and Python, except that like types give machine-checked hints to go on.

The proposed type system is being co-designed with a new programming language called Thorn. The advantage of co-designing the type system with the language is that we can focus on key issues without having to fight corners cases of the language definition as would have been the case if we had picked Java or C# as a starting point. Some simplifications that we have allowed ourselves included ruling out method overloading on parameter types (typically supported in statically typed languages) as well as addition of fields and methods (typically supported in dynamically typed languages). Nevertheless we believe that one could add like types and **dyn** to other static languages and obtain much of the same benefits we have outlined in this paper.

Acknowledgments. We thank the entire Thorn team: Brian Burg, Gregor Richards, Bard Bloom, Nate Nystrom, and John Field. This work was partially supported by ONR grant N140910754 and ANR grant ANR-06-SETI-010-02.

References

[1] Amal Ahmed, Robert Bruce Findler, Jacob Matthews, and Philip Wadler. Blame for all. In *Script to Program Evolution (STOP)*, 2009.

[2] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Type inference for polymorphic methods in Java-like languages. In *Italian Conference on Theoretical Computer Science (ICTCS)*, 2007.

[3] Christopher Anderson and Sophia Drossopoulou. BabyJ: From object based to class based programming via types. *Electronic Notes in Theoretical Computer Science*, 82(7), 2003.

[4] John Aycock. Aggressive type inference. In *International Python Conference*, 2000.

[5] Kent Beck. *Smalltalk: Best Practice Patterns*. Prentice-Hall, 1997.

[6] Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. Thorn–robust, concurrent, extensible scripting on the JVM. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2009.

[7] Gilad Bracha. Pluggable type systems. *OOPSLA04, Workshop on Revival of Dynamic Languages*, 2004.

[8] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 1993.

[9] Patrick Camphuijsen, Jurriaan Hage, and Stefan Holdermans. Soft typing PHP. Technical report, Utrecht University, 2009.

[10] Luca Cardelli. Structural Subtyping and the Notion of Power Type. In *Symposium on Principles of Programming Languages (POPL)*, 1988.

[11] Robert Cartwright. User-defined data types as an aid to verifying LISP programs. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 228–256, 1976.

[12] Robert Cartwright and Mike Fagan. Soft Typing. In *Conference on Programming language design and implementation (PLDI)*, 1991.

[13] The Computer Language Benchmarks Game. `http://shootout.alioth.debian.org/`.

[14] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming (ICFP)*, 2002.

[15] Cormac Flanagan. Hybrid type checking. In *Symposium on Principles of Programming Languages (POPL)*, 2006.

[16] Cormac Flanagan. ValleyScript: It's like static typing. Technical report, UC Santa Cruz, 2007.

[17] Michael Furr, Jong hoon An, Jeffrey Foster, and Michael Hicks. Static type inference for ruby. In *Symposium in Applied Computing (SAC)*, 2009.

[18] Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained interoperability through mirrors and contracts. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 231–245, 2005.

[19] David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. In *Trends in Functional Programming (TFP)*, 2007.

[20] Alex Holkner and James Harland. Evaluating the dynamic behaviour of Python applications. In *Australasian Computer Science Conference (ACSC)*, 2009.

[21] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3), 2001.

[22] Adobe Systems Inc. ActionScript 3.0 Language and Components Reference, 2008.

[23] Giovanni Lagorio and Elena Zucca. Just: Safe unknown types in Java-like languages. *Journal of Object Technology*, 6(2), 2007.

[24] Sven-Olof Nyström. A soft-typing system for Erlang. In *Erlang Workshop*, 2003.

[25] Sean B. Palmer. Pwyky (A Python Wiki).

[26] Jeremy Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object Oriented Programming (ECOOP)*, 2007.

[27] Jeremy Siek and Philip Wadler. Threesomes, with and without blame. In *Script to Program Evolution (STOP)*, 2009.

[28] Jeremy G. Siek. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*, 2006.

[29] Jeremy G. Siek, Ronald Garcia, and Walid Taha. Exploring the design space of higher-order casts. In *European Symposium on Programming (ESOP)*, 2009.

[30] Jeremy G. Siek and Manish Vachharajani. Gradual typing with unification-based inference. In *Symposium on Dynamic languages (DLS)*, 2008.

[31] Ed Stephenson. Perl Runs Sweden's Pension System. O'Reilly Media, 2001.

[32] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: From scripts to programs. In *Symposium on Dynamic languages (DLS)*, 2006.

[33] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *Symposium on Principles of Programming Languages (POPL)*, 2008.

[34] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming (ESOP)*, 2009.

[35] Ulf Wiger. Four-fold increase in productivity and quality. In *Workshop on Formal Design of Safety Critical Embedded Systems*, 2001.

[36] Andrew K. Wright and Robert Cartwright. A practical soft type system for Scheme. In *Conference on LISP and Functional programming*, pages 250–262, 1994.