A Monadic Framework for Name Resolution in Multi-phased Type Checkers

Casper Bach Poulsen
c.b.poulsen@tudelft.nl
Delft University of Technology
Netherlands

Aron Zwaan
a.s.zwaan@tudelft.nl
Delft University of Technology
Netherlands

Paul Hübner*
paul.huebner@googlemail.com
Delft University of Technology
Netherlands

Abstract

An important aspect of type checking is name resolution—i.e., determining the types of names by resolving them to a matching declaration. For most languages, we can give typing rules that define name resolution in a way that abstracts from what order different units of code should be checked in. However, implementations of type checkers in practice typically use multiple phases to ensure that declarations of resolvable names are available before names are resolved. This gives rise to a gap between typing rules that abstract from order of type checking and multi-phased type checkers that rely on explicit ordering.

This paper introduces techniques that reduce this gap. First, we introduce a monadic interface for phased name resolution which detects and rejects type checking runs with name resolution phasing errors where names were wrongly resolved because some declarations were not available when they were supposed to be. Second, building on recent work by Gibbons et al., we use applicative functors to compositionally map abstract syntax trees onto (phased) monadic computations that represent typing constraints. These techniques reduce the gap between type checker implementations and typing rules in the sense that (1) both are given by compositional mappings over abstract syntax trees, and (2) type checker cases consist of computations that roughly correspond to typing rule premises, except these are composed using monadic combinators. We demonstrate our approach by implementing type checkers for Mini-ML with Damas-Hindley-Milner type inference, and LM, a toy module language with a challenging import resolution policy.

CCS Concepts: • **Theory of computation** → **Program analysis**; **Type structures**; *Algebraic semantics*; • **Software and its engineering** → **Compilers**; *Semantics*.

---

*Author's current affiliation is KTH Royal Institute of Technology, Sweden

Keywords: stable name resolution, scope graph, phasing, applicative functor composition, type checker

1 Introduction

Most modern programming languages have a mutual dependency between typing and name resolution. For example, consider the following program in a language with modules:

```
1  module A {
2      import B
3      def f: Int = 1
4      def g: Int = h
5  }
6  module B {
7      def h: Int = A.f + 2
8  }
```

The named reference h (line 4) must resolve to the declaration in B, and A.f (line 7) must resolve to the declaration in A. This raises the question: in what order should we check the modules A and B such that we can determine that all named references indeed resolve to declarations of the right type?

For most type checkers in practice, the answer is to use multiple phases. For the module language above we can first analyze the overall module structure, associate types with each declared name, and about which names are reachable via declared imports. In a subsequent phase, this information is used to verify that named references on the right hand side of **def**s resolve to declarations of the right type.

In contrast, it is common for typing rules to abstract from phasing concerns. For example, the typing rules for Featherweight Java [15] and related calculi [3, 10, 20, 29] use class tables and abstract from how and when class table entries are constructed.

However, for type checker implementations, it is important to construct and query name binding information (e.g., in a *symbol table* [1] or a *scope graph* [28]) in the correct order. Attempting to resolve a name in a wrongly phased manner can lead to subtle bugs. For example, consider the following program with a nested module, where the reference x can

be resolved to an imported definition and a definition in the enclosing module:

```
1  module C {
2    def x: Int = 3
3    module D {
4      import E
5      def y: Int = x
6    }
7  }
8  module E {
9    def x: Int = 4
10 }
```

Since the nested module C imports E, the reference to x on line 5 could resolve to either the declaration in C (line 2) or E (line 9). Before we can resolve the right hand sides of **def**s and the x on line 5, we should *first* resolve the E import reference (line 4). This way, we know that the declarations in E are reachable from D. However, a wrongly phased type checker could fail to resolve imports before type checking the right hand side of **def**s. In this case, x on line 5 would resolve to the declaration on line 3. Since the intended semantics of our language is that declarations from imports shadow declarations from the lexical context, this *silently resolves names to the wrong declarations.*

This paper presents abstractions for multi-phased type checking that prevent such subtle errors. We contribute: (1) a new interface of effectful operations for creating and querying name binding information, using scope graphs [28, 33, 37, 38]; and (2) techniques that use *applicative functors* [9, 17, 24] to map abstract syntax trees (ASTs) to compact, explicitly phased, and effectful operations for type checking.

These contributions build on and extend previous work. Our use of applicative functors builds on the work of Gibbons et al. [9], and our scope graph operations are inspired by Rouvoet et al. [33]. A key feature of our operations is that they detect phasing errors during type checking and rule out subtle phasing errors such as the one above. The Statix language [33] provides this guarantee in a different way, via a static ownership type discipline and a sound (but incomplete) query scheduling algorithm. As we show in §5, our approach supports language features which Statix does not.

Most programming language implementations resolve names in multiple phases. For example, Haskell has a relatively simple module system that uses two phases [11, §2.3.2]. Scala combines a range of sophisticated name binding features such as inheritance, import statements, traits, type members, *dependent object types* [2], and *multi-staging* [22] in the MetaML tradition [35]. Languages such as Java, C#, Kotlin, and Rust also have multi phase type checking.

Our operations also require computations to run in a phased order. A naive approach to implementing this ordering is to traverse ASTs in multiple passes. However, such passes add syntactic overhead compared to typing rules that abstract from such phasing, as is common for typing rules

that use scope graphs [33, 38, 39]. We reduce the syntactic overhead of type checker implementations by compositionally mapping AST nodes onto monadic, multi-phased computations, using generic combinators for implementing the required phase ordering. This makes our type checker implementations more compact than explicitly phased implementations, akin to how monadic parser combinators [14] make parsers more compact than recursive descent parsers.

Our focus is on detecting phasing errors and on compactness. We believe our approach is not fundamentally at odds with efficiency but exploring this is left to future work. For now, our type checker implementations are likely have a subpar performance compared with direct style type checkers.

We make the following technical contributions:

- We present (in §3) a monadic interface of operations for designing phased type checkers, using scope graphs. The operations dynamically detect and report name resolution phasing errors during type checking.
- Building on techniques for multi-phasing from Gibbons et al. [9] and Kidney and Wu [17], we present (in §4) generic combinators for multi-phased computation where later phases may depend on values from prior ones. In §4.5, we discuss how these techniques make type checker implementations more compact and more closely related to typing rules.
- We validate and evaluate our approach (in §5) by considering two case studies: a type checker for Mini-ML that uses Damas-Hindley-Milner type inference, and a type checker for a subset of the LM language due to Neron et al. [28].

The paper is structured as follows. §2 gives an overview of the problem and our solution. Then, §3 and §4 describe the implementation of our scope graph operations and techniques for phased computation using applicative functors. §5 describes case studies, §6 related work, and §7 concludes.

The framework and case studies are available in an artifact [32]. The abstractions in this paper are implemented in Haskell, and familiarity with Haskell is assumed.

2　The Multi-phased Name Resolution Problem and its Solution

Type checking generally requires producing name binding information in multiple phases. How do we represent such name binding information in typing rules and in compilers?

Typing rules most often use *type environments* that map names to types. However, few such specifications model the semantics of, e.g., modules or classes. In practice, compilers traditionally use *symbol tables* [1]. The details of symbol table implementations differ from language to language but a symbol table generally represents a "scope". It stores the declared names of a scope, and (typically) the types of each name. By linking symbol tables to other symbol tables [1, §2.7], we can represent which scopes are reachable from the

```
1  module F {
2    import G
3    def i: Int = j
4  }
5  module G {
6    import H
7  }
8  module H {
9    def j: Int = 5
10 }
```
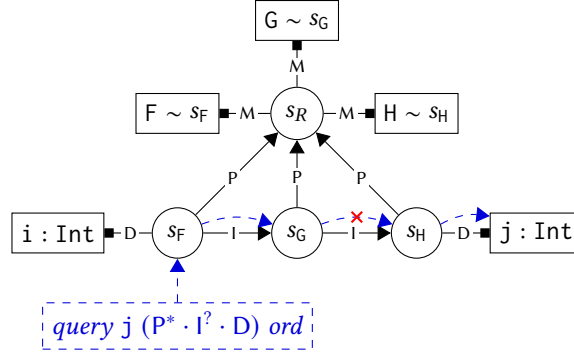

Figure 1. Name reachability example

```
1  module C {
2    def x: Int = 3
3    module D {
4      import E
5      def y: Int = x
6    }
7  }
8  module E {
9    def x: Int = 4
10 }
```
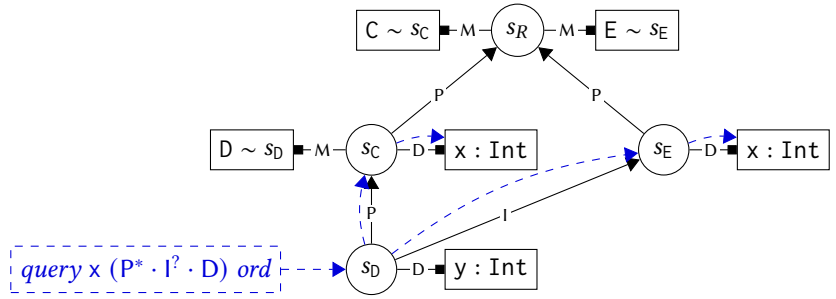

Figure 2. Name shadowing example

current scope; for example, names in the lexical context or names in imported modules. Compilers resolve names by traversing reachable symbol table entries and links.

Scope graphs are a mathematical model of name binding and name resolution that can be used as a stand-in replacement for both symbol tables and and type environments. A type environment typically represents the set of *visible* names, whereas symbol tables represent the set of *reachable* names and model visibility as a search through these. We can think of a type environment as a "flattened" symbol table resulting from applying the visibility search procedure. The scope graph analogue to searching a symbol table is resolving a *name resolution query*. Thus scope graphs can replace both symbol tables and type environments.

Following Visser and co-authors [28, 33, 37–39], we can use scope graphs to define both typing rules and type checker implementations. In this section we give an introduction to scope graphs, the problem with phased name resolution, and how we solve the problem using a new set of monadic operations for scope graph construction.

2.1 Scope Graphs as a Model of Name Resolution

A scope graph is a data structure that represents the scopes and declarations of a program. Scopes (nodes in the scope graph) are conceptually similar to symbol tables, in that each scope is associated with declarations, and each scope may be connected to other scopes via *directed, labeled edges*.

Names are resolved by traversing edges in the scope graph and inspecting declarations. With symbol tables, the name resolution policy is given by a language specific algorithm that traverses tables. *Scope graph queries* succinctly define such traversals and name resolution policies. We illustrate how scope graphs and queries provide a declarative model of *reachability* and *visibility* (i.e., *shadowing*).

Reachability. A declaration is *reachable* if we can follow directed edges through the graph to reach it. For example, consider the program and scope graph in fig. 1. The program (left) has three modules that transitively import each other: F imports G and G imports H. On the right is its scope graph. There are four scopes, denoted by circles. s_R represents the "root scope" of the program, which contains declarations (labeled —■ arrows from scopes) for each of the three modules. These declarations associate module names with their scopes. For example, $\boxed{C \sim s_C}$ associates C with scope s_C. Module scopes have declarations for each module member. Member declarations associate names with types; e.g., i : Int in s_F. Labels on declaration edges indicates the kind of declaration: D for module members (**def**s); M for modules. Labels on edges between scopes indicates the scoping relation: P for lexical parent relations; I for import relations.

Named references are resolved by *querying* the scope graph. For example, the dashed blue box connected to s_F is a query for the named reference j (line 3). This name is

passed to the first argument of the query, which ensures only declarations with name j are matched. As the dashed blue edges show, it is possible to follow labeled edges to reach a j declaration. However, this path does not reflect the intended import semantics. The regex $P^* \cdot I^? \cdot D$ of the *query* says that a *valid* path has zero or more lexical parent edges, and *at most one import edge*. The shown path has two import steps so it does not match the query. The path would match if the query allowed transitive imports; e.g., $P^* \cdot I^* \cdot D$. The third argument of the query (*ord*) is an *ordering relation on paths*, which defines the *visibility semantics* of queries.

Visibility. The example from the introduction is repeated in fig. 2 (left). Its scope graph is on the right. The reference to x on line 5 can resolve to either x on line 2 or line 9. Which we prefer depends on the visibility semantics, given by an ordering relation on paths. This ordering decides which of the two blue paths (both valid according to the query reachability regex) shadows the other. Any type of ordering is possible, but a partial order on labels ($- < - \subseteq Label \times Label$) is sufficient for many languages.[1] For the example in fig. 2:

1. If $P < I$ then we prefer declarations reachable via the lexical context over via imports. A step-wise comparison of the paths in the figure gives precedence to the path through s_C, and the declaration on line 2 shadows the one on line 9.
2. If $I < P$ then we prefer declarations reachable via imports over via the lexical context. A step-wise comparison gives precedence to the path through scope s_E, and the declaration on line 9 shadows line 2.
3. If neither $P \nless I$ nor $I \nless P$, then neither declaration is preferred, and the x reference on line 5 is ambiguous.

2.2 The Multi-phased Name Resolution Problem

Scope graphs (like symbol tables) are data structures containing name binding information. The question is: how do type checkers build this data structure in a way that guarantees all relevant information is available before querying? A key challenge of guaranteeing this is that, to build some parts of the data structure, we need to query it (i.e., resolve names). For example, to construct the import edge between s_D and s_E in fig. 2, we must first resolve E (line 4). As discussed in the introduction, failing to construct this import edge before resolving x on line 6 causes our type checker to subtly fail. The next section summarizes how we address this challenge.

2.3 A Monadic Solution to the Multi-phased Name Resolution Problem

We introduce monadic operations for scope graph construction and querying that implicitly check that queries are *stable*; i.e., new edges and declarations do not change the results

---

[1]Some languages need a more general path ordering. For example, the MiniStatix specification of Scala compares full paths: https://github.com/MetaBorgCube/scala.mstx#scala-precedence-as-a-path-order

of previously executed queries. We illustrate query stability by example shortly. First, using M as the type of our monad for scope graph construction, our operations are:
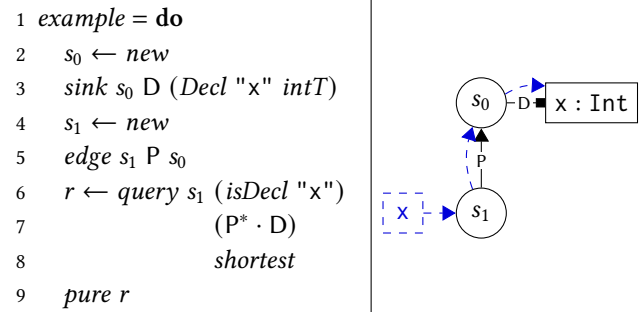
$$new \quad :: M\ Scope$$
$$edge \quad :: Scope \rightarrow Label \rightarrow Scope \rightarrow M\ ()$$
$$sink \quad :: Scope \rightarrow Label \rightarrow Decl \quad \rightarrow M\ ()$$
$$query :: Scope \rightarrow (Decl \rightarrow Bool) \rightarrow RegEx\ Label$$
$$\rightarrow (Path\ Label\ Decl \rightarrow Path\ Label\ Decl \rightarrow Bool)$$
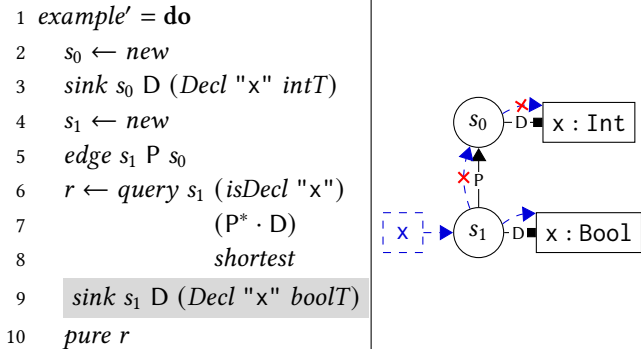$$\rightarrow M\ [Path\ Label\ Decl]$$

Here *new* creates a new scope, *edge s l s'* creates an *l*-labeled edge between scopes, *sink s l d* creates an *l*-labeled edge to a declaration (i.e., a node with no direct outgoing edges), and *query s dm re ord* resolves declarations matching the predicate *dm* starting in scope *s*, using the reachability regex *re*, and ordering paths according to *ord*. The operations are parameterized by the types of *Label* and *Decl* while a *Path* is a sequence of labeled steps between scopes ending in a *Decl*.

To illustrate what it means for a query to be stable, consider the following example and its scope graph:

```
1  example = do
2    s₀ ← new
3    sink s₀ D (Decl "x" intT)
4    s₁ ← new
5    edge s₁ P s₀
6    r ← query s₁ (isDecl "x")
7              (P* · D)
8              shortest
9    pure r
```


Here *isDecl* "x" matches a *Decl* named "x", and *shortest* prefers shorter paths, ignoring labels. The function *pure* :: $\forall a.\ a \rightarrow M\ a$ used on line 9 is a "pure" computation which returns a value as result without any side effects.

Extending the program with the declaration on line 9 below, the query on line 6 gives the same result. However, this result is *not a valid resolution in the final scope graph!*

```
1   example' = do
2     s₀ ← new
3     sink s₀ D (Decl "x" intT)
4     s₁ ← new
5     edge s₁ P s₀
6     r ← query s₁ (isDecl "x")
7               (P* · D)
8               shortest
9     sink s₁ D (Decl "x" boolT)
10    pure r
```


In multi-phased type checkers, a query whose result changes depending on when it is run may give rise to subtle name resolution errors. Our operations avoid this by checking that queries are stable and raising an error if they are not.

The following law says that a query is stable if its order of execution is independent from any (possibly graph constructing) computation m:

$$\begin{pmatrix} \mathbf{do}\ m \\ \quad query\ s\ dm\ re\ ord \end{pmatrix} \equiv \begin{pmatrix} \mathbf{do}\ xs \leftarrow query\ s\ dm\ re\ ord \\ m \\ pure\ xs \end{pmatrix}$$

As *example* and *example′* show, this law does not hold in general. However, our operations do satisfy the following law where $- \equiv_\perp -$ holds if both sides agree *or* if either side raises a *stability error*, indicating that a query result may have been violated:

$$\begin{pmatrix} \mathbf{do}\ m \\ \quad query\ s\ dm\ re\ ord \end{pmatrix} \equiv_\perp \begin{pmatrix} \mathbf{do}\ ys \leftarrow query\ s\ dm\ re\ ord \\ m \\ pure\ ys \end{pmatrix}$$

§3 describes how our monad guarantees this property.

In summary, our monadic operations detect and reject type checker runs with phasing errors but do not statically guarantee their absence. Type checker engineers must therefore phase type checking in a way that avoids such errors. One solution is to implement multiple phases using multiple AST traversals. A more compact solution which we describe in §4 is to use a single pass to map AST nodes onto phased computations.

3 Monadic Scope Graph Construction

We consider how the operations discussed in the previous section construct scope graphs, and how they detect and reject programs with stability errors.

3.1 An Interface for Scope Graph Construction

Below is a type class that captures this monadic interface discussed in §2.3:[2]

class $Monad\ m \Rightarrow SG\ l\ d\ m \mid m \rightarrow l\ d$ **where**
$\quad new\ \ ::\ m\ Scope$
$\quad edge\ \ ::\ Scope \rightarrow l \rightarrow Scope \rightarrow m\ ()$
$\quad sink\ \ \ ::\ Scope \rightarrow l \rightarrow d \rightarrow m\ ()$
$\quad query ::\ Scope \rightarrow (d \rightarrow Bool) \rightarrow RegEx\ l$
$\quad\quad\quad\quad \rightarrow (Path\ l\ d \rightarrow Path\ l\ d \rightarrow Bool) \rightarrow m\ [Path\ l\ d]$

One of our core contributions is that we provide an instance of this interface. The instance we provide in the code accompanying this paper [32] is defined using a Haskell embedding of *algebraic effects and handlers* [30]. The benefit of defining our instance in this way is that it is easy to compose the effects summarized by the *SG* interface above with other effects. For example, the case studies in §§5.1 and 5.2 make use of auxiliary effects for unification (used for type inference), emitting errors and backtracking. However, the details of embedding algebraic effects and handlers in Haskell are

---

[2]The $\mid m \rightarrow l\ d$ part indicates a functional dependency. That means that l and d should be determined by m. I.e., for any given m, there may be at most one pair of l and d such that a type class instance of $SG\ l\ d\ m$ exists.

beyond the scope of this paper. We summarize at a high level how our code implements the operations and invite readers to consult the code for further details.

Representing Scope Graphs. Our operations construct new nodes, edges, and sinks in scope graphs given by the following record type:

type $Scope = Int$
data $Graph\ l\ d$
$\quad = Graph\ \{\ scopes :: Scope$
$\quad\quad\quad\quad\quad , edges\ \ :: Scope \rightarrow l \rightarrow [\,Scope\,]$
$\quad\quad\quad\quad\quad , sinks\ \ :: Scope \rightarrow l \rightarrow [\,d\,]\,\}$

We use integers to represent scopes such that we have an infinite supply of fresh scopes. Edges in the graph are given by a (curried) mapping from scope-label pairs to a (possibly empty) list of target scopes. Declarations (or sinks) are similarly defined. The implementation of the operations in *SG* threads *Graph*s through in a stateful manner.

A naive implementation of *new*, *edge*, and *sink* would simply update the graph, and *query* would simply traverse the current graph to find matching paths. However, this naive implementation would suffer from the query stability problem discussed in the introduction and §2.2.

Detecting Stability Errors. Our implementation detects and reports stability violations; i.e., additions to the scope graph that cause an earlier query to give a different result. A simple but expensive way (in terms of runtime) to implement this check is to cache *every* query made during type checking, and then re-run every query when adding sinks or edges. Our operations implement a less expensive approach.

We associate each scope with residual queries which precisely define the traversals that have started from that scope in the past. When adding a new edge or declaration we execute that traversal over the new edge, as though the past query was run on the newly extended graph, and check that the query result remains unchanged. To illustrate, consider the scope graph on the left and the (truncated) residual queries for s_0 and s_1 on the right:
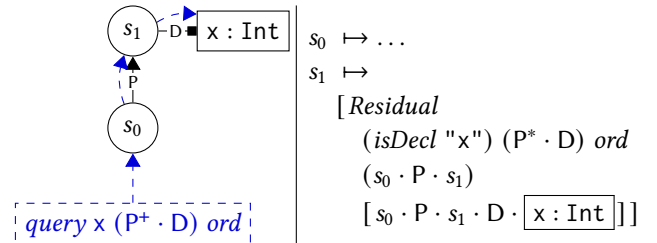
The first three arguments of *Residual* represents the state of the query that traversed scope s_1 to resolve the blue path. Since the query already traversed a P edge, the second argument is the *derivative* [4] of the original regex with respect to P. The fourth argument $(s_0 \cdot P \cdot s_1)$ is the path leading from the source scope (s_0) to the current (s_1). The last argument is the final result of the original query at the time it was run.

Say we extend scope ① with a new P-labeled edge to a new scope ②, as shown below:
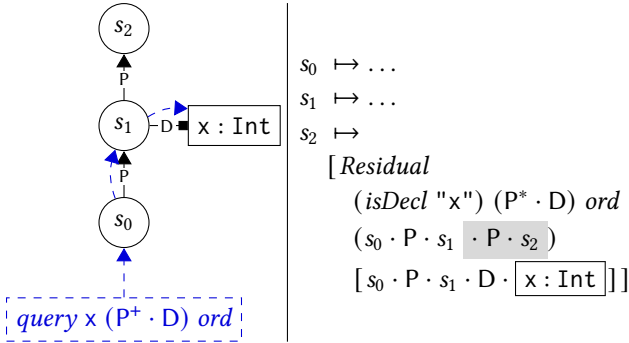
In doing so, we enact the residual queries of s_1 which in turn associates a new residual query with ②. The residual for ② is the same as for ① except that the path leading from the source scope to the current scope now has an additional step, highlighted in gray.

Now say we extend ② with a new x declaration. The *sink* operation first adds this declaration to the scope graph and then enacts the residual queries of ②. This enactment yields a valid path since $s_0 \cdot P \cdot s_1 \cdot P \cdot s_2 \cdot D \cdot \boxed{\texttt{x:Int}}$ matches the (original) regex. This path is compared with the original set of paths using the ordering *ord*. If the path is not shadowed by any of the original paths in the residual, a changed query result is detected, and we raise a stability error.

Over-Approximating Stability Errors. The stability error detection described above is precise. However, that can make it hard to write tests that expose phasing errors. An approach that over-approximates stability but detects common phasing anti-patterns is sometimes desirable. The code artifact accompanying this paper implements both the precise stability error detection discussed above and the over-approximation we describe next. Let us revisit the example from before.
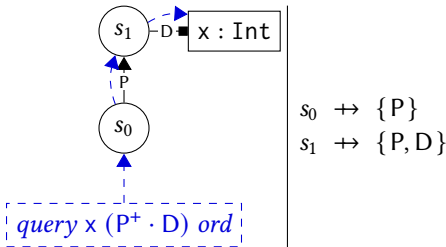
The *head set* (i.e., the set of characters that words accepted by the regex start with) of the query regex $P^+ \cdot D$ is {P}, so all outgoing P edges of ⓪ are traversed by the query. If we add a new P edge to ⓪ later, new declarations may become reachable which may give rise to stability errors. Our over-approximation of stability errors prevents this possibility by "closing" scope ⓪ under P. ① is closed under both P and D because the head set of the query upon reaching ① is {P, D}. Attempting to add an edge or sink of a label that a scope is closed under raises a stability error.

Discussion. The code accompanying this paper contains implementations of both of the stability error strategies described above. While the over-approximating approach is more coarse grained, it enforces a certain programming discipline: we should only resolve names via a scope once all of the edges that the query may traverse have been added. Failing to follow this principle causes our interface to raise (potentially over-approximate) stability errors. In our experience, this helpfully pinpoints dubious phasing patterns in multi-phased type checkers.

Our implementation of both strategies satisfy the query stability law discussed in §2.3; i.e., for any $m, s, d, r,$ and o:

$$\textbf{do } m; query\ s\ d\ r\ o \equiv_\perp \textbf{do } xs \leftarrow query\ s\ d\ r\ o; m; pure\ xs\ (*)$$

3.2 Explicitly Phased Type Checking

The previous section discussed how our monadic operations detect and reject stability errors. Let us consider how we can use these operations to define multi-phased type checkers for a simple module language whose abstract syntax is:

data *MDec* = *Import String* | *Def String Ty Expr*
        | *Module String* [*MDec*]
data *Expr* = *Ident String* | *Lit Int* | *Tru* | *Fals*
data *Ty* = *IntT* | *BoolT*

MDec defines module member declarations (**import**s, **def**s, and **module**s), *Expr* expressions, and *Ty* types. For simplicity, expressions can only be identifiers, integer literals, or Boolean literals.

We can define an explicitly phased type checker for this language that uses three phases:

$top :: SG\ Label\ Decl\ m \Rightarrow MDec \rightarrow m\ ()$
$top\ m = \textbf{do }s \leftarrow new$
          $(q_1, q_2) \leftarrow modules\ m\ s$
          $imports\ q_1$
          $members\ q_2$

$modules :: SG\ Label\ Decl\ m \Rightarrow MDec \rightarrow Scope$
          $\rightarrow m\ ([\,(Scope, String)\,], [\,(Scope, (Ty, Expr))\,])$
$imports :: SG\ Label\ Decl\ m \Rightarrow [\,(Scope, String)\,] \rightarrow m\ ()$
$members :: SG\ Label\ Decl\ m \Rightarrow [\,(Scope, (Ty, Expr))\,]$
          $\rightarrow m\ ()$

Here *top* orchestrates three phases which do the following. *modules* takes as input an *MDec* and its scope, and elaborates it into two working lists. As part of this elaboration, **def** declarations are added to their corresponding scopes, scopes are created for nested modules, and lexical parent (P) edges connect these module scopes to their lexical parents. The first working list represents import to be added to that scope, which is generally only possible once we know all module names. The second represents expressions to type check in that scope, which is generally only possible once all names

are imported. These working lists are processed in separate phases, using the *imports* and *members* functions.

The explicit phasing in *top* above uses three traversals: one over the abstract syntax to turn it into two working lists which we traverse next. A more compact alternative that does not use artificial intermediate working lists, is to map abstract syntax nodes onto phased computations that check well typedness, in a single traversal. We show how next.

4 Applicative Phasing and Its Application to Scope Graph Construction

As recently demonstrated by Gibbons et al. [9] and Kidney and Wu [17], applicative functors provides a useful abstraction for phased computation. In §4.1 we recall the concept of applicative functors. Next (§§4.2 and 4.3), we describe how and why we build on and adapt the techniques of Gibbons et al. [9]. Then §4.4 shows how to implement type checkers using applicative functors. Finally (in §4.5) we compare a case from a type checker written in this style with its corresponding typing rule.

4.1 Applicative Functors

Applicative functors are a standard feature in many Haskell libraries. These libraries usually use the *Applicative* type class [24].[3] We use the following alternative but equivalent category theory inspired interface [24]:[4]

class *Functor f* $\Rightarrow$ *Monoidal f* **where**
  unit :: f ()
  ($\star$) :: $f\ a \to f\ b \to f\ (a, b)$

As we will see, the *Monoidal* interface is well-suited for defining phased computations, and we use that instead of the *Applicative* interface. If we think of f as a computation, the *unit* operation represents a pure computation returning a unit value whereas $\star$ combines two computations. The operations are subject to the following laws where $(f \times g)\ (x, y) = (f\ x, g\ y)$ and *assoc* $(a, (b, c)) = ((a, b), c)$:

$$fmap\ (f \times g)\ (m_1 \star m_2) \equiv (fmap\ f\ m_1) \star (fmap\ g\ m_2)$$
$$fmap\ snd\ (unit \star m) \equiv m$$
$$fmap\ fst\ (m \star unit) \equiv m$$
$$fmap\ assoc\ (m_1 \star (m_2 \star m_3)) \equiv (m_1 \star m_2) \star m_3$$

Next, we consider how to use $\star$ to compose multi-phased computations.

4.2 Functor Composition and Phasing

In the module language in §3.2, nested modules pose a phasing challenge. The challenge is that, before we can resolve imports, we need to know the names of all modules. Thus, phase 1 first creates the scopes of all modules; and only then

---

[3]https://hackage.haskell.org/package/base-4.18.0.0/docs/Control-Applicative.html
[4]In categorical terms, an applicative functor is a *strong lax monoidal functor*.

do we, in phase 2, resolve named imports. §3.2 used multiple traversals to implement this phasing. With applicative functors we can use a single traversal that returns a *phased computation* directly. But what is a phased computation?

The answer that Gibbons et al. [9] give to this question is *Day convolutions*. Briefly summarized, a Day convolution *Day f g a* consists of a pair of two applicative functors, f and g, which represent two distinct phases. The idea is to construct phased computations using two functions:

phase1 :: (*Monoidal f*, *Monoidal g*) $\Rightarrow$ $f\ a \to$ *Day f g a*
phase2 :: (*Monoidal f*, *Monoidal g*) $\Rightarrow$ $g\ a \to$ *Day f g a*

Because Day convolutions are applicative functors themselves, computations can then be freely combined, in any order, using the $\star$ operation. In particular, the following holds for any $m_1 :: f\ a$ and $m_2 :: g\ b$ where f, g are applicative functors:

$$phase1\ m_1 \star phase2\ m_2 \equiv fmap\ twist\ (\ phase2\ m_2 \\ \star\ phase1\ m_1)$$

where *twist* $(x, y) = (y, x)$

An attractive property of Day convolutions is that, in order to run a phased computation *Day f g a*, we only need to assume that f and g are themselves applicative functors. This means that Day convolutions can be used to phase a general class of computations, including monads since (in Haskell) all monads are applicative functors.

However, Day convolutions generally do not allow using of results from prior computations in subsequent ones, which is a common pattern in multi-phased type checkers. For example, if we infer the module type (associating member names to types, which we represent as a *Scope*) in a prior computation, we want a subsequent computation to use this module type to check that expressions in module members are well typed. That is, for two applicative functors f and g, we want to phase $m :: f\ Scope$ and $k :: Scope \to g\ a$ where the *Scope* to pass to k is the one that m computes.

Consider how we might try to write this using only *phase1*, *phase2*, and the applicative functor product $\star$:

canWeDoThis = phase1 m $\star$ *phase2* (k ??)

This will not work: $\star$ combines m and k in a way that their computations are independent, so we cannot fill in ?? with the result of m in this way. In fact, applicative functors generally do not allow the use of results from prior computations in the definition of subsequent ones, so we cannot in general write this program using only *phase1*, *phase2*, and $\star$.

Instead, we could use the operations of m and k to pass information along from a prior computation to a subsequent one. For example, if m has operations for outputting values and k has operations that read such values as input, we can wire the outputs from m to inputs of k. For some applications, such wiring is natural; for example, the phased solution to the *repmin* problem considered by Gibbons et al. [9]. We

might use a similar scheme for our type checkers, but then we need to label the scopes produced in a prior phase so that we can retrieve the intended module type by dereferencing the correct label in subsequent phases.

Instead of relying on such a labeling scheme, we use functor composition and monads. This lets us use Haskell functions—specifically, the two combinators we introduce in §4.3—to wire outputs to inputs such that (1) we do not have to invent a labeling scheme for labeling outputs produced in prior phases and unlabeling them in subsequent ones, and (2) the underlying monads do not need to support operations for output and input of labeled data. The downside is that we rely on binding combinators other than monadic bind. On the other hand, our combinators rely on standard machinery (functor composition and monadic bind), and provide a typed interface that helps enforce phase consistency.

The combinators we will introduce in the next section are based on *functor composition*; i.e.:

newtype $(f \circ g)\ a = Comp\ \{\ getComp :: f\ (g\ a)\ \}$

We use composed functors $f \circ g$ to represent phased computations where f computations run in the first phase, and g in the second. We will exploit that g is nested inside f since this makes it possible for the g computation to *directly* depend on the values produced by the outer f computation. We show how in §4.3. First, we define some auxiliary and standard[5] functor composition helper functions. First, composed functors are themselves functorial:

instance $(Functor\ f, Functor\ g) \Rightarrow Functor\ (f \circ g)$ **where**
$\quad fmap\ f\ (Comp\ x) = Comp\ (fmap\ (fmap\ f)\ x)$

Second, composed applicative functors are also applicative:

instance $(Monoidal\ f, Monoidal\ g)$
$\qquad\qquad \Rightarrow Monoidal\ (f \circ g)$ **where**
$\quad unit = Comp\ (fmap\ (const\ unit)\ unit)$
$\quad Comp\ x \star Comp\ y = Comp\ (fmap\ (uncurry\ (\star))\ (x \star y))$

This *Monoidal* instance lets us compose phased computations in any order, similarly to Day convolutions. In particular, the functions below are analogous to *phase1* and *phase2*:

$here :: (Functor\ f, Monoidal\ g) \Rightarrow f\ a \to (f \circ g)\ a$
$here\ m = Comp\ (fmap\ (\lambda x \to fmap\ (const\ x)\ unit)\ m)$
$there :: Monoidal\ f \Rightarrow g\ a \to (f \circ g)\ a$
$there\ m = Comp\ (fmap\ (const\ m)\ unit)$

The following holds for any $m_1 :: f\ a$ and $m_2 :: g\ a$ where f and g are applicative functors:

$$here\ m_1 \star there\ m_2 \equiv fmap\ twist\ (there\ m_2 \star here\ m_1) \quad (\dagger)$$

4.3 Chaining Phases

While *here* and *there* combine phased computations, they do not allow us to define phases where later phases explicitly depend on values produced by earlier phases. The following functions do:

$(\multimap) :: Functor\ f \Rightarrow f\ a \to (a \to g\ b) \to (f \circ g)\ b$
$m \multimap k = Comp\ (fmap\ k\ m)$
$(\gg\!\!\!-) :: Monad\ f \Rightarrow f\ a \to (a \to (f \circ g)\ b) \to (f \circ g)\ b$
$m \gg\!\!\!- k = Comp\ (m \ggg (getComp \circ k))$

Both functions define a two-phased computation where the second phase may depend on the first. Both $m \multimap k$ and $m \gg\!\!\!- k$ assume that m is a phase 1 computation. The difference is that $m \multimap k$ assumes that the effects in k are phase 2 computations, whereas $m \gg\!\!\!- k$ allows k to have both phase 1 and phase 2 computations. The latter uses the monadic bind of f to sequence m with the phase 1 computations in k. In the next section we illustrate how the combinators above can be used to phase computations in type checkers.

4.4 Implicitly Phased Scope Graph Construction

Using the machinery from sections 4.1 to 4.3 we can use a single traversal over ASTs to construct a phased computation representing the type checking constraints of a program. Figure 3 (left) shows the cases of a type checker for the *MDec* type of the module language from §3.2. On the right in the figure are the corresponding typing rules which we will discuss in §4.5. In addition to the *SG* monad type class from §3.1, it uses the following error monad type class:

class $Monad\ m \Rightarrow Err\ m$ **where** $err :: String \to m\ a$

The type signature on line 2 shows that the *mdec* function produces a computation in three phases where each phase has the same set of effects (m). The *Module* case of *mdec* function on line 3-7 uses $\gg\!\!\!-$ to create a module scope and declaration in phase 1, and passes the created scope to the computation which uses the *traverse* function to recursively call *mdec* to check module member declarations where:[6]

$traverse :: Monoidal\ f \Rightarrow (a \to f\ b) \to [\,a\,] \to f\ [\,b\,]$

Here *traverse* uses *Monoidal* to compose the phased computations resulting from recursively calling *mdec* in a manner that respects eq. (†) from §4.2. The use of $\gg\!\!\!-$ composes the phase 1 computation on line 4-6 with the phase 1 computations recursively created by *mdec* calls in line 7.

Lines 8-13 of fig. 3 use *there* (*here* ...) on line 8 to insert the computation on line 9-13 in phase 2. The computation resolves a named import and creates an import edge between the current scope and the resolved module scope. For simplicity, the module language and query on line 9 only

---

[5]https://hackage.haskell.org/package/base-4.18.0.0/docs/Data-Functor-Compose.html

[6]https://hackage.haskell.org/package/base-4.16.0.0/docs/Data-Traversable.html

```
1  mdec :: (SG Label Decl m, Err m)
2        ⇒ MDec → Scope → (m ∘ m ∘ m) ()

3  mdec (Module x mds) s ①  = void
4     ((do  sₘ ← new ②
5           sink s M (ModDecl x sₘ) ③
6           pure sₘ) ⊶ λsₘ →
7        (traverse (λm → mdec m sₘ) mds) ④)

8  mdec (Import x) s ⑤ = there (here (do
9     ps ← query s (isModDecl x) (P* · M) pCompare ⑥
10    case map scOf ps ⑥ of
11       [Just sᵢ] → edge s I sᵢ ⑦
12       _ → err "bad import"))

13 mdec (Def x t e) s ⑧ = void
14    ( here (sink s D (DefDecl x t)) ⑨
15    ★ there (there (expr e s t) ⑩))
```

$$\frac{\nabla s_m \quad ② \qquad s \xrightarrow{\mathrm{M}} \blacksquare \ (x \sim s_m) \quad ③ \qquad s_m \vdash mds \ \mathrm{OK} \quad ④}{s \vdash \mathbf{module} \ x \ \{\ mds\ \} \ \mathrm{OK} \quad ①}$$

$$\frac{\mathbf{query} \ s \ (isModDecl \ x) \ (\mathrm{P}^* \cdot \mathrm{M}) \ pCompare \rightsquigarrow \{(x \sim s_i)\} \quad ⑥ \qquad s \xrightarrow{\mathrm{I}} s_i \quad ⑦}{s \vdash \mathbf{import} \ x \ \mathrm{OK} \quad ⑤}$$

$$\frac{s \xrightarrow{\mathrm{D}} \blacksquare \ x : T \quad ⑨ \qquad s \vdash e : T \quad ⑩}{s \vdash \mathbf{def} \ x : T = e \ \mathrm{OK} \quad ⑧}$$
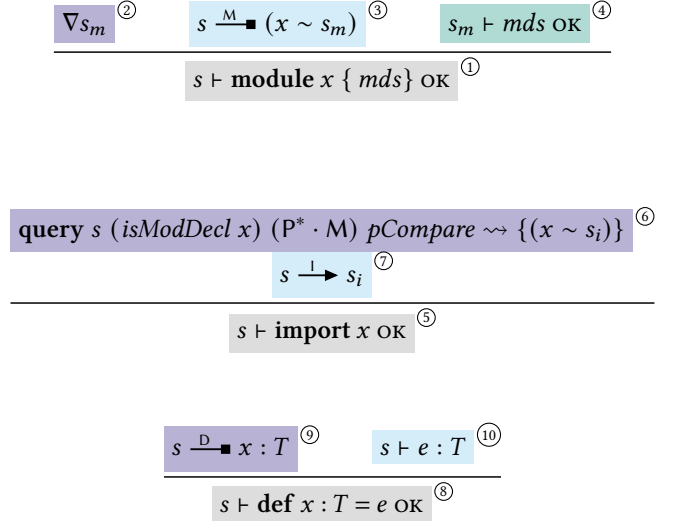
Figure 3. Representative cases of a type checker for the module language from §3.2.

allows modules to be resolved via lexical scoping; i.e., by following P edges until an M-labeled edge leading to a module declaration is found. Also for simplicity, the module language does not support qualified (module) names. Thus, module import resolution in this language is *relative* (i.e., imports are resolved starting in the scope where they occur), *unordered* (imports can occur anywhere and will be in scope for the entire module), *glob* (also known as wildcard—i.e., they import all definitions from a module), and *import insensitive* (i.e., modules cannot be resolved via import statements; only via the lexical context). In §5.2 we show how to support *import sensitive module resolution* (i.e., modules can be resolved via import statements) as well as type inference.

The final case in fig. 3 uses ★ to compose a phase 1 and a phase 3 computation which, respectively, creates a declaration for the *Def* in the current scope, and then checks the expression of the *Def*.

4.5 Correspondence to Typing Rules

The code on the left in fig. 3 defines a type checker for the rules shown on the right in the same figure. The typing rules are written in a similar style as in the work of van Antwerpen et al. [38]. The rules themselves are transcribed from the MiniStatix specification of LM due to Rouvoet et al. [33], except that the rule for imports only allows enclosing modules to be imported. In contrast, the MiniStatix specification of LM due to Rouvoet et al. [33] has support for *import sensitive module resolution* which we will show how

to type check in §5.2 The colors and numbers indicate how each premise and conclusion of the typing rules is checked in our type checker. Each case of *mdec* corresponds with one of the rules, and all premises correspond to an expression within the case. Besides the standard **do** keyword, the uncolored parts of the code either (1) phase the type checker using *here*, *there*, ★, and ⊶, (2) perform error handling, or (3) use *void* :: *Functor f* ⇒ *f a* → *f* () to discard return values to match the type signature. In contrast, in the type checker discussed in §3.2, the premises would be scattered across different functions: declarations are created by the *modules* function, whereas the right hand sides of **def**s are type checked by the *members* function.

5 Case Studies

We present two case studies (also included in the artifact [32]) that explore the expressiveness of our approach. First, §5.1 shows how Damas-Hindley-Milner type inference (i.e. Algorithm W [7]) can be implemented using our approach. Next, we extend the language from §§3.2 and 4.4 with *import sensitive* module resolution. Both case studies could not be operationalized in previous scope graph based frameworks [33, 39].

5.1 Mini-ML with Damas-Hindley-Milner Inference

According to Zwaan and van Antwerpen [39], one of the primary limitations of Statix is its lack of support for Damas-Hindley-Milner-style type inference [6]. Here we present

the first scope graph based type checker for MiniML [16], a language with let polymorphism [26]. The language has the usual λ calculus constructs, as well as let bindings and number literals/addition. Its (truncated) syntax is:

data *MiniML*
  = *Ident String*
  | *Let String MiniML MiniML*
  | $\cdots$

To infer types for MiniML we will make use of operations for generating new meta-variables and unify first-order terms given by the following syntax:

data *Term* = *Var Int* | *Term String* [*Term*]

Here we use integer indices to distinguish different variables. We use terms to represent types, and use the following smart constructors for number and function types:

type *Ty* = *Term*
numT = *Term* "Num" [ ]
funT s t = *Term* "->" [*s, t*]

The operations of the type class below generate new meta-variables (*Vars*) and unify terms:

class *Monad m* $\Rightarrow$ *Unif m* **where**
  exists :: *m Term*
  equals :: *Term* $\rightarrow$ *Term* $\rightarrow$ *m* ()
  inspect :: *Term* $\rightarrow$ *m Term*

Here *exists* creates a new meta-variable (e.g., *Var x* where *x* is a fresh integer); *equals* t_1 t_2 either unifies t_1 and t_2 or raises an error if they cannot be unified; and *inspect t* inspects a term, applying all substitutions resulting from previously performed unifications.

Polymorphic types (type *schemes*) in MiniML are given by the *Scheme* data type.

data *Scheme* = *Scheme* { *sbinds* :: [*Int*], *stype* :: *Ty* }

Using schemes as the type of declarations, our type checker uses a single phase and is given by the *mml* function whose type is shown below:

data *Label* = P | D

data *Decl* = *Decl String Scheme*

mml :: (*SG Label Decl m, Err m, Unif m*)
    $\Rightarrow$ *MiniML* $\rightarrow$ *Scope* $\rightarrow$ *Ty* $\rightarrow$ *m* ()

The function takes as input a *MiniML* expression, the current *Scope*, and the *Type* that the input expression should be checked to have. We use unification to infer the type. If the type of the input expression is not known beforehand, the *Ty* argument of *mml* is a unification variable.

The implementation of *mml* follows Algorithm W [7]. We consider two of the most interesting cases, starting with the case for variables.

```
1  mml (Ident x) s t = do
2     ps ← query s (isDecl x) (P* · D) pShortest
3     case map schemeOf ps of
4        [ sc ] → do dt ← inst sc; equals t dt
5        _ → err ("bad identifier: " + x)
6  inst :: Unif m ⇒ Scheme → m Term
```

The query on line 2 resolves the identifier. If the query succeeds, we call *inst* to instantiate the type scheme, and then unify the resulting term with the input type *t* (line 4). The (elided) implementation of *inst* substitutes the variables bound by the type scheme by fresh variables.

The other interesting case is for let bindings:

```
1  mml (Let x e body) s t = do
2     t' ← exists
3     mml e s t'
4     t'' ← inspect t'
5     st ← gen s t''
6     s' ← new
7     edge s' P s
8     sink s' D (Decl x st)
9     mml body s' t
```

Lines 2-3 introduce a fresh unification variable t' and use it to infer the type of the let bound expression *e*. Next, on lines 4-5, we first inspect the inferred type, and then call *gen* to *generalize* the type relative to the current scope *s* and create a new type scheme *st*. This scheme becomes the type of *x* declared in the sub-scope s' used to check the body of the let expression (line 6-9). Here *gen* is defined as follows:

gen :: *SG Label Decl m* $\Rightarrow$ *Scope* $\rightarrow$ *Term* $\rightarrow$ *m Scheme*
gen s t = **do**
  ps ← *query s* (*const True*) (P* · D) *noOrd*
  let fv_s = *concatMap* (*fv* $\circ$ *stype* $\circ$ *schemeOf*) *ps*
  pure (*Scheme* (*fv t* $\setminus\!\setminus$ fv_s) *t*)

It first collects all declarations in scope, using (*const True*) to match any and all declarations and *noOrd* to prevent shadowing. Then, it projects all free variables of declaration types, using the utility function *fv* :: *Ty* $\rightarrow$ [*Int*]. This is analogous to how Algorithm W [7] inspects all free variables in a type environment. Finally, it creates a scheme that quantifies over the truly free variables in *t*; i.e., variables that do not occur in types reachable from the current scope (fv_s).

Discussion. This case study shows that the lack of support for Damas-Hindley-Milner type systems in Statix is not a limitation of scope graphs. In fact, the *generalize* operation, as traditionally defined on environments, maps rather naturally to scope graphs, when given control over the order of operations and the ability to inspect terms. While the definition of *MiniML* only uses lexical scoping, our *mml*, *gen*, and *inst* functions can be extended to support non-lexical scoping (e.g., modules and imports) without significant changes.
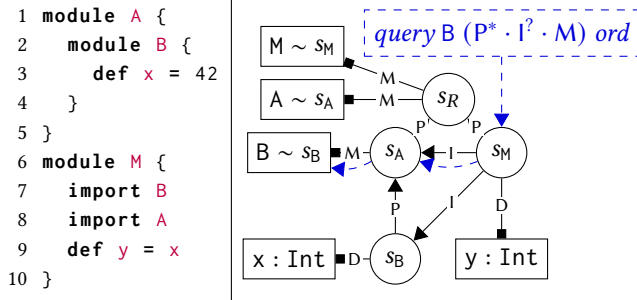
```
1  module A {
2    module B {
3      def x = 42
4    }
5  }
6  module M {
7    import B
8    import A
9    def y = x
10 }
```

Figure 4. A program and scope graph with import sensitive module resolution.
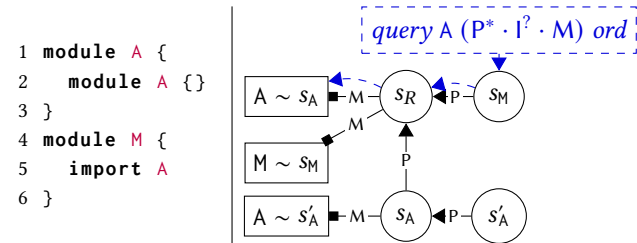
```
1  module A {
2    module A {}
3  }
4  module M {
5    import A
6  }
```

Figure 5. An ambiguous program and its partial scope graph. The program has no model, as adding an import edge from s_M to s_A contradicts its own resolution via the P edge to s_R.

5.2 Language with Modules (LM)

Our second case study is LM, a proof-of-concept language by Neron et al. [28]. LM is similar to the simple module language that we gave a type checker for in §3.2, with two important differences: (1) LM has *optional* type annotations and relies on (monomorphic) type inference; and (2) LM uses *import sensitive module resolution* (in contrast to the example in §4.4 and fig. 3).

Import Sensitive Module Resolution. Consider the example in fig. 4. The imports on line 7 and 8 are unordered and *import sensitive*, in the sense that module names can be resolved through other imports. The A import on line 8 resolves via the lexical context, to the declaration on line 1. Because of this, s_M has an import edge to s_A in the scope graph. The B import is resolved using the query shown in the figure. The regex of this query allows modules to be resolved via an import edge (that is, it is *import sensitive*). Therefore, the import edge to s_A can be used to resolve module B, resulting in an edge to s_B.

The combination of unordered imports and import sensitive module resolution has a subtle semantics in some cases. For example, consider fig. 5 (borrowed from Hübner [13]). It is possible to resolve the A import on line 5 to the A declaration on line 1 along the shown path. Because of this resolution, we should add an import edge between s_M and s_A in the scope graph. But, if we add this edge, then the query in

the figure becomes *unstable*. In LM, imports have precedence over the lexical context, so the added edge would cause the query in the figure to resolve to the (inner) A declared in s_A instead of s_R. Because no graph exists where all names can be stably resolved, the program has no model.

This subtlety illustrates a key challenge of type checking LM programs. Because module resolution is unordered and import sensitive, each import may depend on an arbitrary sequence of other imports. So how do we decide which the order imports should be resolved in?

Implementation. Figure 6 shows how we implement import resolution. The highlights on the left summarize differences from §4.4, fig. 3. These differences stem from how we deal with import resolution. Because of import sensitivity, we use an *Aggr* functor to aggregate the list of all imports to be resolved in phase 2. As shown in the type of *lm* on line 3, this functor is inserted between phase 1 and phase 2. Its definition and relevant *Monoidal* instance is:

data *Aggr r a* = *Aggr r* $(r \rightarrow a)$ **deriving** *Functor*

instance *Monoid r* $\Rightarrow$ *Monoidal* (*Aggr r*) **where**
  unit = *Aggr mempty* (*const* ())
  Aggr xs m $\star$ *Aggr ys n* =
    Aggr (*xs* $\diamond$ *ys*) ($\lambda r \rightarrow (m\ r, n\ r)$)

This *Monoidal* instance assumes that *r* is a monoid, and uses the monoidal product ($\diamond$) for aggregation. In line 11 in fig. 6 we use *Aggr* to aggregate all imports, indexed by a scope. In line 9, we bind the final aggregation to variable *is* and call *imps* to perform import sensitive module resolution.

The *imps* function on the right in fig. 6 implements import sensitive module resolution. The most challenging part of this is that we do not know the correct order in which the imports must be resolved. To compute that, we must be able to speculatively add edges and do queries, backtracking if an import fails. To implement this, we use the *anyOrder* operation:

class *Monad m* $\Rightarrow$ *AnyOrder m* **where**
  anyOrder :: [*m* (*Maybe* ())] $\rightarrow$ *m* () $\rightarrow$ *m* ()

This operation implements the following kind of error handling behavior. The first parameter is a list of computations that may fail (i.e., return *Nothing*). The operation tries to find an order to execute these computations in which (1) all computations succeed, and (2) there are no stability errors. If no such order exists, the second argument is invoked to handle the failure.

Lines 14-16 uses the *anyOrder* operation to search for a stable import resolution order. The first argument is a list of computations that resolve each import, given by *is s*, of a module. If no stable order is found, the second argument is run to raise an error.

The type checker in fig. 6 was validated using (1) 15/19 of the test cases of Rouvoet et al. [33] (we excluded four because

```
1  lm :: (SG Label Decl m, Err m, Unif  m,  AnyOrder m )
2     ⇒ LMDec → Scope
3     → (m ∘ Aggr (Scope → [String]) ∘ m ∘ m) ()
4  lm (Module x mds) s = void
5     ( (do s' ← new
6          sink s M (ModDecl x s')
7          pure s') ⧁ λs' →
8       traverse (λm → lm m s') mds
9     ★ there (Aggr (const []) id ⧁ λis → here (imps is s')) )
```

```
10  lm (Import x) s =
11      there (here (Aggr (λs' → [x | s ≡ s']) (const ())))
12  imps :: (SG Label Decl m, Err m, Unif m, AnyOrder m)
13      ⇒ (Scope → [String]) → Scope → m ()
14  imps is s = anyOrder
15     (map (λi → resolveMod i s) (is s))
16     (err "Could not resolve imports")
```

Figure 6. Representative cases of a type checker for LM.

they used qualified names, which are not included in our language), (2) seven additional test cases from Hübner [13], and (3) two new test cases for query stability corner cases. Unlike Statix, all cases pass.

Discussion. Certain optimisations over this scheme are conceivable. For example, failures can help to prune the remaining permutations. When an import query is invalidated by another import, all permutations containing imports in the same order can be filtered. Similarly, when an import does not resolve, we only need to retain permutations that have at least one unresolved import before the failing one.

While backtracking over all permutations seems expensive, it is an improvement over other approaches found in the literature. NaBL2 [37] re-resolves imports at *every reference*, even variables inside a scope, while we perform import resolution once per module. Apart from this, we are not aware of any implementation for such a module system.

6 Related Work
We discuss related work on scope graphs and phasing.

6.1 Scope Graphs
Scope graphs were originally introduced by Neron et al. [28]. In their model, imports are first-class, whereas we model them using queries and edges. Van Antwerpen et al. [37] introduce *NaBL2*, a type system specification meta-language using scope graph for name resolution. To cover more type systems, van Antwerpen et al. [38] refined the scope graph model, and embedded it in the logic language Statix [38]. We use this model in this paper. In contrast to Neron et al., this model allows interleaving of scope graph construction and querying. Statix is a declarative language, in which a model satisfying all constraints is found by constraint solving. This gives rise to a scheduling problem: when can queries be executed without later edge additions invalidating the result? The answer given by Rouvoet et al. [33] is, whenever the query does not traverse scopes that have *critical edges*; i.e., edges that give rise to new paths for the query. Since

finding critical edges is as difficult as solving the constraint program, Rouvoet et al. use *weakly critical edges*, an over-approximation of critical edges which can be inferred by a static analysis of Statix rules. In our approach, phasing is done manually, as opposed to automatically by Statix. As our case studies show, this lets us type check languages beyond what Statix supports. On the other hand, Statix supports dynamically scheduled scope graph construction which may be difficult to support in our explicitly phased approach.

Our over-approximating stability error detection from §3.1 implements the dual of weakly critical edges: instead of scheduling queries after no edges are added anymore, we prevent adding edges after a query traversed that edge. Our precise stability error detection from §3.1 detects real critical edges, which is not possible in Statix. A more in-depth overview of the evolution and application of scope graphs is given by Zwaan and van Antwerpen [39].

Casamento [5] uses scope graphs to write correct-by-construction type checkers in Agda that yield intrinsically typed syntax for languages where scope graph construction does not depend on querying a partially constructed graph.

6.2 Phasing in Other Type Checkers
Rouvoet et al. [33] observe that the module system of Rust has features comparable to our LM case study. A key difference is that enclosing modules are not reachable by default, but must be brought in scope using (e.g.) a `use super::*` statement. Hence, modules that are in (lexical) scope cannot be shadowed by modules that are imported later. Thus, newly resolved imports can only make other module references *ambiguous*. As such, Rust's module resolution does not require backtracking, but is implemented with a fix-point computation instead. In addition, it has a 'finalize' phase that checks whether the import resolution is stable.[7] The *anyOrder* operation from §5.2 performs similar checks.

The Scala 3 compiler also generalizes the notion of symbol tables, but in a different way than scope graphs do [21].

---

[7]https://github.com/rust-lang/rust/blob/55e8df2b0e3c4494b77f2431b912c51e6fe733ba/compiler/rustc_resolve/src/imports.rs#L466

Internally, different phases have different contexts. Each context carries meanings (denotations) for symbols. In this way, information (such as types) can be changed between phases, which is used to keep typing information accurate under transformations. However, *within* the single type checking phase, names are resolved in a way that looks like traditional lookups in symbol tables. Thus, there appears to be no explicit stability checking, neither within the type checking phase nor in the updates to a context that a transformation can introduce. In future research, we could investigate whether our monadic scope graph framework could be extended to track stability under (controlled) transformations.

Another common framework to write type checkers in is *attribute grammars* [19]. In this system, attributes can be associated with productions in a grammar. Attributes are evaluated using small 'functions' that can refer to attributes of other nodes. Ironically, this paradigm has made the opposite development regarding stratification as the scope graph framework has. While canonical attribute grammar execution followed a statically determined ordering of multiple traversals, later extensions (aiming to improve expressiveness) introduced dynamic scheduling to the paradigm. Some of these extensions include *reference attributes* [8, 12], which allow attributes to evaluate to references to other AST nodes; *parameterized attributes* [8] which allow passing parameters to attributes; and *collection attributes* [23] allow attributes to be a "combination of contributions from distant nodes in the abstract syntax tree". Evaluation of each of these kinds of attributes is usually done dynamically; i.e., on-demand. This gives more flexibility than our approach, at the cost of declarative appeal and (sometimes) termination guarantees. Rouvoet et al. [34, §E.1] claim that Statix' scheduling is more precise than JastAdd's collection attributes. This would imply that our framework could be able to express phasing that cannot be derived from attribute grammars using collection attributes, although examples are still to be found.

Finally, an earlier version of our library has been used to explore how to type check a Java subset [36], a Scala subset [25], type classes [27], and substructural types [18]. Our LM case study is based Hübner's work [13], with two main differences. First, we use applicative functors for phasing; second, we use back-tracking to implement import sensitive module resolution whereas Hübner uses a dedicated import resolution algorithm. We conjecture that our implementation is sound and complete w.r.t. the typing rules of LM whereas Hübner's algorithm is known to be incomplete.

6.3 (Higher-Order) Algebraic Effects and Handlers

In the code artifact accompanying the paper [32], we used a Haskell embedding of *effects and handlers* [30] to provide an implementation of the monad which the code examples in this paper leaves abstract. This allowed us to separately define and subsequently compose effects. For example, our implementation of LM composes separately defined handlers for unification operations (the *Unif* interface from §5.1) and scope graph construction operations (the *SG* interface from §3.1). To define *higher-order* effects (i.e., effects where operations can have computations as arguments, such as the *anyOrder* operation) in §5.2, we used *hefty algebras* [31] to elaborate higher-order effects into algebraic effects.

7 Conclusion

Implementing type systems for languages with complex (non-lexical) name binding features is challenging. A key challenge is that all relevant name binding information must be aggregated before a name is resolved. We showed how to address this challenge using scope graph constructing operations which dynamically detect and reject programs that fail to aggregate relevant name binding information before name resolution. Scheduling queries correctly typically requires multi-phased type checking. However, it is often possible to define typing rules that abstract from such operational phasing concerns. To make type checker implementations more compact, we used recently developed techniques from previous work on applicative functors to compositionally map program ASTs onto computations representing phased typing constraints. This yields an expressive framework for sound name resolution of complex binding structures, and reduces the gap between type checker implementations and typing rules since type checker cases consist of computations that roughly correspond to typing rule premises, except these are composed using monadic combinators.

Future Work. While our case studies show that our approach supports type systems that cannot be operationalized using Statix, it is an open question whether the reverse is true. We enforce a static number of phases, while (e.g.) constraint-based approaches support more dynamic scheduling. To the best of our knowledge, a precise characterization of static vs. dynamic phasing, and a comparison of their expressiveness, is yet to be made. For example, it is not yet clear whether a type system with simple record inference (e.g. as presented by Van Antwerpen et al. [38, §2.3]) can be ported to our framework. In addition, scheduling schemes designed using this framework might inform refinements of algorithms used in automatically scheduling systems, such as Statix. While our approach is expressive, our type checker implementations are currently not very efficient. In future work, we would like to explore a more efficient implementation of scope graph construction and querying, and explore *fusing* phases similarly to Gibbons et al. [9].

Acknowledgments

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley. https://www.worldcat.org/oclc/12285707

[2] Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science, Vol. 9600)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.). Springer, 249–272. https://doi.org/10.1007/978-3-319-30936-1_14

[3] G. M. Bierman, M. J. Parkinson, and A. M. Pitts. 2003. *MJ: An imperative core calculus for Java and Java with effects*. Technical Report UCAM-CL-TR-563. University of Cambridge.

[4] Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (1964), 481–494. https://doi.org/10.1145/321239.321249

[5] Katherine Imhoff Casamento. 2019. *Correct-by-Construction Type-checking with Scope Graphs*. Master's thesis. Portland State University. Department of Computer Science.

[6] Luís Damas. 1984. *Type assignment in programming languages*. Ph. D. Dissertation. University of Edinburgh, UK. https://hdl.handle.net/1842/13555

[7] Luís Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*, Richard A. DeMillo (Ed.). ACM Press, 207–212. https://doi.org/10.1145/582153.582176

[8] Torbjörn Ekman and Görel Hedin. 2007. The JastAdd system - modular extensible compiler construction. *Science of Computer Programming* 69, 1-3 (2007), 14–26. https://doi.org/10.1016/j.scico.2007.02.003

[9] Jeremy Gibbons, Donnacha Oisín Kidney, Tom Schrijvers, and Nicolas Wu. 2022. Breadth-First Traversal via Staging. In *Mathematics of Program Construction - 14th International Conference, MPC 2022, Tbilisi, Georgia, September 26-28, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13544)*, Ekaterina Komendantskaya (Ed.). Springer, 1–33. https://doi.org/10.1007/978-3-031-16912-0_1

[10] Robert Griesemer, Raymond Hu, Wen Kokke, Julien Lange, Ian Lance Taylor, Bernardo Toninho, Philip Wadler, and Nobuko Yoshida. 2020. Featherweight go. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 149:1–149:29. https://doi.org/10.1145/3428217

[11] Cordelia V. Hall, Kevin Hammond, Will Partain, Simon L. Peyton Jones, and Philip Wadler. 1992. The Glasgow Haskell Compiler: A Retrospective. In *Functional Programming, Glasgow 1992, Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, UK, 6-8 July 1992 (Workshops in Computing)*, John Launchbury and Patrick M. Sansom (Eds.). Springer, 62–71. https://doi.org/10.1007/978-1-4471-3215-8_6

[12] Görel Hedin. 2000. Reference Attributed Grammars. *Informatica (Slovenia)* 24, 3 (2000).

[13] Paul Hübner. 2023. *Type-Checking Modules and Imports using Scope Graphs*. Bachelor's Thesis. Delft University of Technology. http://resolver.tudelft.nl/uuid:e7f16989-9aca-4707-9d4a-74eba2adc5e4

[14] Graham Hutton and Erik Meijer. 1996. Monadic parser combinators.

[15] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (2001), 396–450. https://doi.org/10.1145/503502.503505

[16] G. Kahn. 1987. Natural semantics. In *STACS 87*, Franz J. Brandenburg, Guy Vidal-Naquet, and Martin Wirsing (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 22–39.

[17] Donnacha Oisín Kidney and Nicolas Wu. 2021. Algebras for weighted search. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. https://doi.org/10.1145/3473577

[18] Jan Knapen. 2023. *Type checker for a language with a substructural type system using scope graphs*. Bachelor's Thesis. Delft University of Technology. http://resolver.tudelft.nl/uuid:0469a179-4228-4d30-a263-8f7e17da7026

[19] Donald E. Knuth. 1968. Semantics of context-free languages. *Mathematical Systems Theory* 2, 2 (1968), 127–145.

[20] Edlira Kuci, Sebastian Erdweg, Oliver Bracevac, Andi Bejleri, and Mira Mezini. 2017. A Co-contextual Type Checker for Featherweight Java. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs, Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 18:1–18:26. https://doi.org/10.4230/LIPIcs.ECOOP.2017.18

[21] LAMP/EPFL. Accessed: 2023-05-28. Scala 3 High Level Architecture: Symbols. https://dotty.epfl.ch/docs/contributing/architecture/symbols.html.

[22] LAMP/EPFL. Accessed: 2023-05-28. Scala 3 Metaprogramming: Run-Time Multi-Stage Programming. https://dotty.epfl.ch/docs/reference/metaprogramming/staging.html.

[23] Eva Magnusson, Torbjörn Ekman, and Görel Hedin. 2009. Demand-driven evaluation of collection attributes. *Autom. Softw. Eng.* 16, 2 (2009), 291–322. https://doi.org/10.1007/s10515-009-0046-z

[24] Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *J. Funct. Program.* 18, 1 (2008), 1–13. https://doi.org/10.1017/S0956796807006326

[25] Radu Mihălăchiuţă. 2023. *Building Type Checkers Using Scope Graphs: Scope Graph-Based Type Checking for a Scala Subset*. Bachelor's Thesis. Delft University of Technology. http://resolver.tudelft.nl/uuid:a4050c81-4a1c-4cf0-b26e-7c60aa18503a

[26] Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

[27] Andreea Mocanu. 2023. *Building Type Checker Using Scope Graphs: For a Language with Type Classes*. Bachelor's Thesis. Delft University of Technology. http://resolver.tudelft.nl/uuid:64aa6cc8-9039-47b1-99f6-decf150dcab8

[28] Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2015. A Theory of Name Resolution. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9032)*, Jan Vitek (Ed.). Springer, 205–231. https://doi.org/10.1007/978-3-662-46669-8_9

[29] Johan Östlund and Tobias Wrigstad. 2010. Welterweight Java. In *Objects, Models, Components, Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6141)*, Jan Vitek (Ed.). Springer, 97–116. https://doi.org/10.1007/978-3-642-13953-6_6

[30] Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5502)*, Giuseppe Castagna (Ed.). Springer, 80–94. https://doi.org/10.1007/978-3-642-00590-9_7

[31] Casper Bach Poulsen and Cas van der Rest. 2023. Hefty Algebras: Modular Elaboration of Higher-Order Algebraic Effects. *Proc. ACM Program. Lang.* 7, POPL (2023), 1801–1831. https://doi.org/10.1145/3571255

[32] Casper Bach Poulsen, Aron Zwaan, and Paul Hübner. 2023. *Mophasco (MOnadic framework for PHAsed name resolution using SCOpe graphs)*. https://doi.org/10.5281/zenodo.8337245

[33] Arjen Rouvoet, Hendrik van Antwerpen, Casper Bach Poulsen, Robert Krebbers, and Eelco Visser. 2020. Knowing when to ask: sound

scheduling of name resolution in type checkers derived from declarative specifications. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 180:1–180:28. https://doi.org/10.1145/3428248

[34] Arjen Rouvoet, Hendrik van Antwerpen, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. 2020. Knowing When to Ask: Sound scheduling of name resolution in type checkers derived from declarative specifications (Extended Version). Zenodo. https://doi.org/10.5281/zenodo.4091445

[35] Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.* 248, 1-2 (2000), 211–242. https://doi.org/10.1016/S0304-3975(00)00053-0

[36] Omar Thabet. 2023. *Phased Type Checker For Java: A Type Checker For a Subset of Java Built On Scope Graph Semantics.* Bachelor's Thesis. Delft University of Technology. http://resolver.tudelft.nl/uuid:f05ab738-0fde-48bc-b3b3-4a8a1345db4c

[37] Hendrik van Antwerpen, Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2016. A constraint language for static semantic analysis based on scope graphs. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Martin Erwig and Tiark Rompf (Eds.). ACM, 49–60. https://doi.org/10.1145/2847538.2847543

[38] Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. 2018. Scopes as types. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 114:1–114:30. https://doi.org/10.1145/3276484

[39] Aron Zwaan and Hendrik van Antwerpen. 2023. Scope Graphs: The Story so Far. In *Eelco Visser Commemorative Symposium, EVCS 2023, April 5, 2023, Delft, The Netherlands (OASIcs, Vol. 109)*, Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 32:1–32:13. https://doi.org/10.4230/OASIcs.EVCS.2023.32