Transitioning from Structural to Nominal Code with Efficient Gradual Typing

Technical Report

FABIAN MUEHLBOECK, IST Austria, Austria
ROSS TATE, Cornell University, United States of America

Gradual typing is a principled means for mixing typed and untyped code. But typed and untyped code often exhibit different programming patterns. There is already substantial research investigating gradually giving types to code exhibiting typical untyped patterns, and some research investigating gradually removing types from code exhibiting typical typed patterns. This paper investigates how to extend these established gradual-typing concepts to give formal guarantees not only about how to change types as code evolves but also about how to change such programming patterns as well.

In particular, we explore mixing untyped "structural" code with typed "nominal" code in an object-oriented language. But whereas previous work only allowed "nominal" objects to be treated as "structural" objects, we also allow "structural" objects to dynamically acquire certain nominal types, namely interfaces. We present a calculus that supports such "cross-paradigm" code migration and interoperation in a manner satisfying both the static and dynamic gradual guarantees, and demonstrate that the calculus can be implemented efficiently.

CCS Concepts: • **General and reference** → **Performance**; • **Software and its engineering** → Formal language definitions; *Object oriented languages*; **Multiparadigm languages**; **Interoperability**.

Additional Key Words and Phrases: Gradual Typing, Gradual Guarantee, Nominal, Structural, Call Tags

ACM Reference Format:
Fabian Muehlboeck and Ross Tate. 2021. Transitioning from Structural to Nominal Code with Efficient Gradual Typing: Technical Report. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 127 (October 2021), 70 pages. https://doi.org/10.1145/3485504

1 INTRODUCTION

Many typed object-oriented languages are implemented markedly differently than untyped object-oriented languages. For example, many compilers for typed object-oriented languages determine the memory layouts of class instances at compile time, and similarly translate field accesses to offset memory loads during compilation; whereas objects in untyped object-oriented languages are often represented as hashtables, with the compiler translating field accesses to key-value lookups in object hashtables. While an object has a number of *structural* properties, like what fields and methods it has, many typed object-oriented languages use *nominal* type systems, permitting the compiler to use the name of a type to establish and rely upon memory-layout invariants while also abstracting such low-level details from the programmer. Beyond types, these implementation considerations can prompt typed languages to use entirely different or restricted *expressions* for specifying and constructing objects. For example, whereas untyped languages often allow one to allocate a "structural" object by manually specifying its structure through explicit fields and

Authors' addresses: Fabian Muehlboeck, IST Austria, Klosterneuburg, Austria, fabian.muehlboeck@ist.ac.at; Ross Tate, Computer Science, Cornell University, Ithaca, New York, United States of America, ross@cs.cornell.edu.

2475-1421/2021/10-ART127
https://doi.org/10.1145/3485504

methods, typed languages often require one to allocate only "nominal" objects of some nominal class, deriving the object's structure from the fields and methods of the class.

These considerations have prompted us to explore a calculus and compiler for an object-oriented language with which we examine the following two research questions:

(1) How can one extend gradual-typing concepts to make strong guarantees about bridging untyped and typed languages when there are more substantial differences between the two than just the absence or presence of type annotations?
(2) What design considerations and implementation techniques can one employ to maintain the performance of typed object-oriented languages using compile-time memory layout of typed nominal objects and method tables while also providing principled and efficient interoperation with untyped structural objects?

More concretely, we explore how to design gradually typed object-oriented languages whose programs can transition between untyped structural and typed nominal "paradigms" while incurring low overheads and still ensuring strong desirable properties, such as the gradual guarantees [Siek et al. 2015a] and soundness, that major mixed-typed industry languages like TypeScript, Flow, Hack, and C# [Bierman et al. 2010] do not offer. Wrigstad et al. [2010] did preliminary investigation in this space with `like` types in Thorn, but `like` types are limited in that they only let code treat nominal objects as their structural counterpart, providing no way for structural objects to be treated as their nominal counterpart. While our system also limits how objects can cross the paradigm boundary, we exploit the heavy use of interfaces in typed nominal code to provide a way for structural objects to masquerade as nominal objects. In particular, unlike Nom [Muehlboeck and Tate 2017], we do not require objects to be allocated with a nominal type in order to cross the boundary—untyped code can still use flexible object manipulation to create objects that can still cross the boundary into typed code. In doing so, we provide substantially more interoperation between structural and nominal code.

In this paper, we make the following contributions. In Section 2, we illustrate what transitioning between structural and nominal paradigms within a code base could look like and how it relates to existing concepts in gradual typing. In Section 3, we provide a calculus—MonNom—wherein untyped structural code can be interwoven with typed nominal code. In Section 4, we specify a **generalized precision relation** that can bridge more substantial differences than type annotations, along with a **static gradual guarantee** that formally describes the kinds of transitions our calculus supports. In Section 5, we define a semantics for our calculus along with a **dynamic gradual guarantee** ensuring that such supported transitions do not change the run-time behavior of programs (assuming inserted type annotations correctly classify the existing behavior of the program). In Section 6, we examine the semantics in more detail and discuss their connection to **implementation techniques** we employed to preserve common optimizations of typed code (such as determining memory layouts at compile time) while also providing low-overhead interoperation with untyped code. In Section 7, we demonstrate that these techniques indeed perform well: performance typically *improves* proportionate to implementation effort when using our recommended migration strategy, and even if one strays from this strategy the worst-case overheads stay under 25%.

2 OVERVIEW

Gradual typing is roughly the ability to mix typed and untyped program components together within the same program [Gronski et al. 2006; Matthews and Findler 2007; Siek and Taha 2006; Tobin-Hochstadt and Felleisen 2006]. In this work, we broaden this to support mixing not just differences in typing discipline but also differences in paradigms that, at least in object-oriented languages, often coincide with differences in typing discipline. As an example, consider Figure 1,

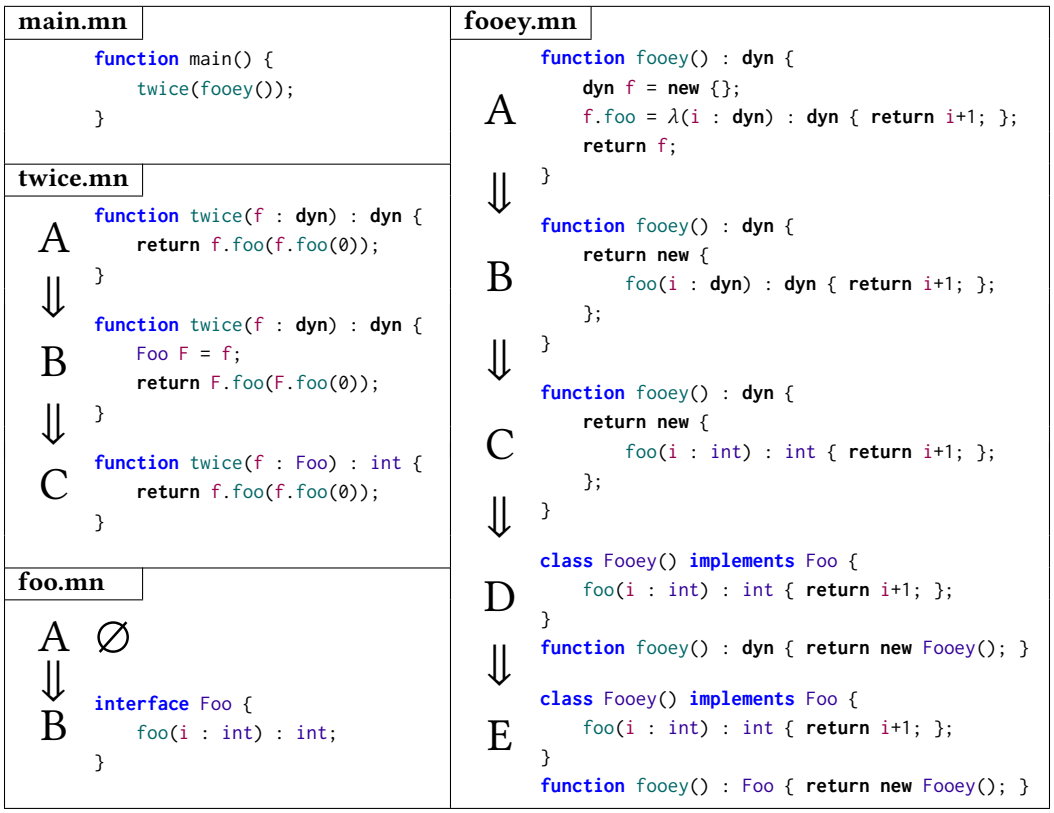

Fig. 1. An example program setup with different variants for the files **twice.mn**, **fooey.mn**, and **foo.mn**

which provides multiple variations of two different program components, functions `twice` and `fooey`, connected together in **main.mn**. One application of gradual typing is code migration, and so the variations are connected by arrows indicating how we would expect a program component to be migrated through the variations. The transition `fooey`-**B** $\implies$ `fooey`-**C** is a traditional transition in gradual typing, simply replacing occurrences of the "dynamic" type `dyn` with "static" types—in this case `int`. But the subsequent transition `fooey`-**C** $\implies$ `fooey`-**D** is beyond traditional gradual typing as it replaces an allocation of a structural object with an allocation of a newly defined nominal class—moving from a structural paradigm to a nominal paradigm. This work focuses on how to support and benefit from such mixing of paradigms as well as typing disciplines.

2.1 Code Migration and Gradual Guarantees

Gradual typing is often motivated in terms of code migration [Campora et al. 2017; Greenman and Felleisen 2018; Tobin-Hochstadt and Felleisen 2006; Tobin-Hochstadt et al. 2017]. That is, one might want to add types to an untyped codebase in order to increase confidence in its correctness, improve its maintainability and/or extensibility, and/or accelerate its run-time performance. Adding types throughout the entire program all at once can be an overwhelming undertaking, and so gradual typing is designed to enable one to do so bit by bit, i.e. gradually.

Given that the codebase presumably has an existing user base and possibly other codebases that depend on it, one would like to be assured that the migration to (correct/intended) types

does not break backwards compatibility, either with respect to compilability with other programs (i.e. *static* backwards compatibility) or with respect to its run-time behavior (i.e. *dynamic* backwards compatibility). To this end, the static and dynamic gradual guarantees were defined [Siek et al. 2015a], which ensure that adding "correct" type annotations to a program will preserve, respectively, its typeability and its run-time behavior.

These gradual guarantees are defined in terms of a *precision* relation ($\sqsubseteq$), where the left-hand program component is considered to be more (rather than less) "precise" than the right-hand component. Traditionally the only difference between more precise and less precise programs is that the more precise program has "static" types (such as `int`) at some syntactic positions where the less precise program has "dynamic" types (such as **`dyn`**). So, as an example, `fooey`-**C** would be considered more precise than `fooey`-**B**. The static and dynamic gradual guarantees then state that all program components related by the given precision relation type-check and execute identically when "all goes well", and that "problems" occur for the less precise program only when "problems" occur for the more precise program (reflecting the fact that untyped programs are more dynamic/flexible than typed programs).

We emphasize that these gradual guarantees are parameterized by a precision relation. Although this precision relation traditionally just relates syntactic constructs componentwise, plus a $\tau \sqsubseteq$ **dyn** rule for type refinement, one *can* relate more kinds of programs. For example, New et al. [2019] explored incorporating β and η equivalences into the precision relation in order deduce semantic theorems in addition to syntactic theorems. In this paper, we explore incorporating the correspondences between structural object-oriented constructs and nominal object-oriented constructs into the precision relation. For example, the structural counterpart to allocating a new instance of a nominal class is allocating a record with the same fields and methods that were defined in the class. As such, we consider `fooey`-**D** to be more precise than `fooey`-**C** (provided interface `Foo` has been defined in **foo.mn**). And, more generally, every transition in Figure 1 corresponds to our precision relation (modulo syntactic conveniences such as semicolons). Since our calculus—MonNom—satisfies the static and dynamic gradual guarantees (Theorems 4.1 and 5.3), this in turn guarantees that code can be migrated according to the transitions in Figure 1.

2.2 Mixing Paradigms

Because the precision relation is defined componentwise on most program constructs, the gradual guarantees require gradually typed languages to support mixing typed and untyped program components together in the same program. For example, these guarantees ensure that if the combination of `twice`-**C** and `fooey`-**E** works correctly (which it does), then so does the combination of `twice`-**A** and `fooey`-**E** despite the fact that the two use completely different forms of method invocation: the former looks up the method implementation according to a fixed offset within an interface method table whereas the latter performs a dictionary lookup using the method's name. But more challengingly, the combination of `twice`-**C** and `fooey`-**A** must also work correctly despite the fact that it means that a structural record dynamically extended with a field holding a functional value must be treated as if it implemented a nominal interface requiring a method of the same name whose implementation is expected to be found at a fixed offset within an interface method table. The former was possible in both Thorn and Nom, but not the latter, and that is due to the above implementation challenges that we have determined how to address efficiently in this work.

These mixed programs are often viewed as simply intermediate states on the way to the end goal of fully typed programs. But in this work we view many of these mixed programs as the end goal themselves. This is because we are bridging paradigms, and each paradigm is better suited to different tasks, so a mixed program is able to match each program component to the paradigm that best fits its purpose. Furthermore, the **dyn** type can be more than simply lack of type annotations: it

is also a means to explicitly circumvent the limitations of the static type system. That is, rather than **dyn** being just a type that has yet to be determined, we also view **dyn** as potentially conveying the programmer's intent to reason about the dynamics of the program beyond what the type system is capable of expressing. For example, a closure can be cast to an interface representing a function from $\mathbb{Z}$ to $\mathbb{Z}$ and then back to **dyn** and, unlike most (sound) gradually typed systems, our calculus permits the resulting cast value to still be supplied, say, $\mathbb{R}$s and to return $\mathbb{R}$s when invoked from untyped code with the understanding that the programmer may be well aware that the dynamics of the program guarantees that this particular function also operates on other numerical values even if the static type system is unaware of or unable to express that fact.

2.3 Changing Hierarchies

Consider the combination `twice`-**C** with `foo`-**B**. Because this is well typed, the static gradual guarantee requires that any less precise combination must also be well typed. Because `twice` and `foo` are separate components, a traditional componentwise precision relation would consider the `twice`-**C** with `foo`-**A** combination to be less precise. But clearly this combination should be ill-typed as `twice`-**C** references an undefined type *name*, an issue that does not arise in structural type systems and did not arise in Nom [Muehlboeck and Tate 2017] wherein the nominal hierarchy was fixed. This introduces a new theoretical challenge to formulating our guarantees.

We address this challenge by defining our precision relation on program components within the context of their respective nominal hierarchies. For example, our variant of the standard rule $\tau \sqsubseteq \tau'$ requires τ' to at least be valid in the nominal hierarchy of the less precise (i.e. right-hand) program.

This necessary change turns out to offer some useful flexibility. In particular, the precision relation on *expressions* must now be parameterized by the respective nominal hierarchies, and as such we can use differences in these hierarchies to guide how we bridge differences in expressions and thereby manage how we much we mix the structural and nominal paradigms. In particular, we allow class-instance allocations in the more precise program to refine structural-object allocations in the less precise program only when the class is declared in the more precise nominal hierarchy (and so prescribes a structure) *but not* in the less precise nominal hierarchy. This allows the compiler to assume the fields of class instances lie at fixed offsets within the instance, while also permitting developers to change a structural record into a new class provided that they simultaneously change all other structural records (if any) corresponding to the class at the same time. As such, we can support the `fooey`-**C** $\implies$ `fooey`-**D** transition in Figure 1.

2.4 Casts, Coercions, and Run-Time Overhead

In this work we focus on (sound) gradual typing—as opposed to (unsound) optional typing—meaning that type annotations, on say variables, actually provide dynamic guarantees, say about what values those variables can hold. Sound gradual typing is often associated with performance issues, often causing multiple factors of overhead, but in this work we *rely* on sound gradual typing to provide good performance, such as to guarantee that a class field can always be found at a fixed offset.

The performance problems with gradual typing are due to the casts that are inserted at the boundary points between typed and untyped code in order to dynamically enforce the contracts expected by typed code. These casts are particularly problematic in the case of types enforcing higher-order contracts, such as functions and objects. In many gradually typed implementations, such casts require one to create a proxy for the cast value that performs the relevant casts on all inputs to and outputs from the value. The `sieve` (of Eratosthenes) benchmark was designed to highlight this problem and was found to incur over 100x overhead in Typed Racket [Takikawa et al. 2016]. And while works such as Grift [Kuhlenschmidt et al. 2019] have proposed using

"space-efficent" coercions [Herman et al. 2010] to mitigate this issue, Feltey et al. [2018] found that—while their analagous "collapsible contracts" did address *some* pathological benchmarks—many benchmarks were unaffected by the technique. Indeed, Grift found no difference between the two techniques for `sieve` with coarse-grained mixing, and while Grift's overhead for `sieve` is much improved compared to Typed Racket (down to 3–4x slowdown), that seems to be due to low-level implementation improvements rather than high-level strategy improvements, and it still leaves unacceptable overhead for the benchmark.

On the other hand, there are some benchmarks where Grift performs extremely well. These benchmarks tend to be numerical, specifically on floating-point values. This seems to be because Grift uses type information not just to eliminate dynamic checks but furthermore to change how values are represented. In particular, untyped Grift represents 64-bit floating-point numbers as *boxed* values allocated in the heap, whereas typed Grift represents them as *unboxed* values. Thus untyped numerical programs are constantly boxing and unboxing the values they compute with whereas typed numerical programs have few heap allocations. This connects to our observation that untyped object-oriented languages tend to be "structural" whereas typed object-oriented languages tend to be "nominal"—performance can improve significantly when type information affects *representation*. But whereas Grift gets this benefit for first-order values like numbers, we want this benefit even for higher-order values. Our calculus achieves this through two means.

For interfaces, we utilize the fact that interface-method dispatch is often already implemented using a form of indirection required to support multiple interface inheritance. So we hijack this existing indirection to make it appear as if structural objects implement arbitrary interfaces.

Classes, on the other hand, often implement field access and method dispatch through fixed offsets. That is, they use type information to optimize object representation. We cannot mimic this like we can interface methods, and adding proxies would require typed code to check whether an object is a proxy before every field access or method dispatch. Given that one benefit of types in our setting is specifically performance, we do not want to add such overhead to the high-performance path. So, while we allow structural objects to be coerced to interfaces, we do not allow them to be coerced to classes, consciously trading off some interoperation flexibility to maintain performance (while still providing a migration path through the means discussed above).

The result of this consideration of tradeoffs is that we can still support higher-order benchmarks such as `sieve`, since the function type corresponds to an interface, but with much improved run-time performance. In particular, most of our configurations of mixing typing disciplines and paradigms exhibit *better* run-time performance than fully untyped code—as one would like to see from gradually typed languages—and even in our worst-case configurations the overheads are only percentages rather than factors. This is achieved without any program analysis (beyond simple type-checking), heuristics, speculation, or dynamic (re-)compilation, and as such there can be high confidence our techniques would scale to large, complex programs without hard-to-predict performance cliffs. Of course, incorporating advanced techniques such as the speculative optimization used by Richards et al. [2017] or the interprocedural program analysis used by Moy et al. [2021] could improve the performance of our language even further.

3 MONNOM

Our calculus, MonNom, is built around a nominally typed, object-oriented core, which on its own already supports a version of gradual typing similar to Nom [Muehlboeck and Tate 2017]. However, MonNom also supports structural records and lambdas. But the structural types one would normally expect for these records and lambdas are not reflected in MonNom's type system; instead, they are simply typed using the special **dyn** type from the gradual portion of MonNom's core. On its own, the structural part of the calculus behaves like major untyped object-oriented languages such

Class Name C Interface Name I Field Name f Variable Name x

Program	$\mathcal{P} ::= \langle \mathcal{H} \mid \mathcal{S} \mid \mathcal{I} \mid e \rangle$	(Notation:	$\mathcal{P} = \langle \mathcal{H}_\mathcal{P} \mid \mathcal{S}_\mathcal{P} \mid \mathcal{I}_\mathcal{P} \mid e_\mathcal{P} \rangle$)
Hierarchy	$\mathcal{H} ::= \varnothing \mid \mathcal{H}; N \leq I, \ldots$	Nominal Type	$N ::= C \mid I$
Signature	$\mathcal{S} ::= \varnothing \mid \mathcal{S}; N\{ms; \ldots\} \mid \mathcal{S}; C(\vec{\tau})\{f : \tau; \ldots\}$		
Implementation	$\mathcal{I} ::= \varnothing \mid \mathcal{I}; x : C(\Gamma)\{f := e; \ldots \mid mb; \ldots\}$		
Type	$\tau ::= N \mid \mathbb{B} \mid \textbf{dyn}$	Method Name	$m ::= f \mid \lambda$
Type List	$\vec{\tau} ::= \varnothing \mid \vec{\tau}, \tau$	Method Signature	$s ::= (\vec{\tau}) : \tau$
Type Context	$\Gamma ::= \varnothing \mid \Gamma, x : \tau$	Method Body	$b ::= (\Gamma) \mapsto e : \tau$
Expression	$e ::= x \mid \texttt{let}\ \langle \Gamma \rangle := \langle e, \ldots \rangle\ \texttt{in}\ e \mid \texttt{false} \mid \texttt{true} \mid e == e$		
	$\mid\ e.f \mid e.f := e \mid e(e, \ldots)$		
	$\mid\ \texttt{new}\ C(e, \ldots) \mid \texttt{new}\ \lambda\langle b \rangle \mid \texttt{new}\ x := \{f := e; \ldots \mid mb; \ldots\}$		

Fig. 2. Grammar

as Python and JavaScript. For example, every record has a dictionary for fields that are added at run time, and fields that contain lambdas can also be treated just like methods and called directly. Unlike Nom, no type annotations are necessary even on lambdas.

More importantly, MonNom provides a rich model of interaction between its nominal and structural parts: values originating in nominal code can interact with and flow into structural code, and vice versa. Moreover, we extend the static and dynamic gradual guarantees [Siek et al. 2015a] to cover the transition from structural to nominal code. The gradual guarantee is the key property of well-behaved program migration—it allows us to refine the parts of our example program in Figure 1 along the lineages depicted there and know that the behavior of those programs will stay the same. While MonNom takes care to align concepts between the structural and nominal paradigms and typing disciplines, it does effectively contain multiple languages plus their interactions, and consequently its formalism is too large to fit into this paper. The full formalism can be found in Appendix A. Here we will elaborate upon only the key concepts.

3.1 Grammar

Figure 2 shows the syntax of MonNom. The first part of the figure specifies the structure of a MonNom program $\mathcal{P}$. A program has four components. The last component is essentially the `main` expression of the program. But that expression exists in the context of the first three components, which altogether define a traditional nominal class and interface hierarchy. In a real language, and in our actual implementation, these three components would be defined via one syntactic unit, but for the purposes of characterizing and proving the formal guarantees of MonNom we found it useful to separate them. The first component $\mathcal{H}$ defines the inheritance hierarchy between classes and interfaces. (Note that the calculus distinguishes class names C and interface names I syntactically, using N when either is applicable. Thus the syntax of $\mathcal{H}$ indicates that only interfaces can be inherited in the calculus, although this is only for simplicity as our implementation also supports single inheritance of classes.) The second component $\mathcal{S}$ defines the signatures of class and interface methods and of class constructors and fields. The third component $\mathcal{I}$ specifies how class fields are initialized from their constructor parameters, and how class methods are implemented (where the variable x represents **`this`** or **`self`** in method bodies). So, in short, given a program $\mathcal{P}$, the hierarchy $\mathcal{H}_\mathcal{P}$ establishes the program's space of types; the signature $\mathcal{S}_\mathcal{P}$ provides the information necessary for type-checking; and the implementation $\mathcal{I}_\mathcal{P}$ defines the overall execution of the program, with the expression $e_\mathcal{P}$ specifying how to kick off the execution.

$$\boxed{\mathcal{H} \vdash \tau \leq \tau} \qquad \frac{}{\mathcal{H} \vdash \tau \leq \tau} \qquad \frac{N \leq I_1, \ldots \in \mathcal{H} \qquad \mathcal{H} \vdash I_i \leq \tau}{\mathcal{H} \vdash N \leq \tau}$$

Fig. 3. Nominal ($\leq$) Subtyping

$$\boxed{\vdash \tau \sim \tau} \qquad \frac{}{\vdash \tau \sim \tau} \qquad \frac{}{\vdash \tau \sim \mathbf{dyn}} \qquad \frac{}{\vdash \mathbf{dyn} \sim \tau} \qquad \boxed{\vdash \tau \sqsubseteq \tau} \qquad \frac{}{\vdash \tau \sqsubseteq \tau} \qquad \frac{}{\vdash \tau \sqsubseteq \mathbf{dyn}}$$

Fig. 4. Consistency ($\sim$) and Precision ($\sqsubseteq$)

The second part of Figure 2 specifies types and type contexts, as well as method names and signatures. The types are either nominal, primitive (which in the calculus is just the booleans $\mathbb{B}$, but in our implementation includes 64-bit signed integers and 64-bit IEEE-754 floating-point numbers), and the dynamic type **dyn**. Note that MonNom has no structural types for records or lambdas—they are simply assigned the dynamic type **dyn**. Method names are either field names or the special method name λ that allows objects to be invokable directly, just like a lambda closure.

The last part specifies expressions. The first line contains a number of different standard expressions whose meaning should be unsurprising. Note, though, that the equality operator in MonNom checks for equality of references. This means that, in order to satisfy the gradual guarantee, we must ensure that transitions preserve the behavior of object identities. Furthermore, although not formally discussed in this paper, we ensure this equality has the property that returning `true` guarantees identical behavior, i.e. `if` $x == x'$ `then` e_t `else` e_f is semantically equivalent to `if` $x == x'$ `then` $e_t[x' \mapsto x]$ `else` e_f, which we found to conflict with devices such as the "chaperone" references used by Typed Racket [Tobin-Hochstadt and Felleisen 2006].

The second line contains field access and mutation, along with application (which invokes λ-methods). Note that (named) method invocation is expressed in the grammar as application of arguments to the result of a field access. In typed code, the type of the receiver distinguishes the two, but untyped code has no such disambiguator. Indeed, MonNom's type system has a special rule for this particular combination, and one challenge was figuring how to design a semantics and implementation that could handle the combined case efficiently in typed settings while still satisfying the gradual guarantee with respect to untyped settings.

The third line of expressions contains the constructors for class instances, lambdas, and records, in that order. Lambdas only specify their signature and implementation and (not so importantly) have no self-reference. Records feature a variable x that, like classes, is used to represent **`this`** or **`self`** in method bodies. Records also have fields and method implementations—we can implement these more efficiently when specified as part of the initial record allocation rather than added after the allocation, and our precision relation connects the two ways of constructing records.

3.2 Type System

MonNom's type system is built around subtyping. Without taking gradual typing into account, the *nominal* subtyping relation for MonNom is defined in Figure 3. Since we omit generics in the calculus, this relation is extremely simple.

Besides being object-oriented, MonNom is also gradually typed. At the core of gradual typing is typically what is called the *consistency* relation, shown in Figure 4. Intuitively, two types are consistent with each other if there is a way to replace the occurrences of **dyn** on both sides such that the resulting types are identical (this again becomes more interesting with generics, but even then is straightforward). Consistency is then used to relax typing rules, expressing the fundamental

$$\boxed{\mathcal{H} \vdash \tau \lhd \tau} \quad \frac{\mathcal{H} \vdash \tau \leq \tau' \qquad \vdash \tau' \sim \tau''}{\mathcal{H} \vdash \tau \lhd \tau''} \qquad \boxed{\mathcal{H} \vdash \tau \blacktriangleleft \tau} \quad \frac{\mathcal{H} \vdash \tau \leq \tau' \qquad \vdash \tau' \sqsubseteq \tau''}{\mathcal{H} \vdash \tau \blacktriangleleft \tau''}$$

Fig. 5. Optimistic ($\lhd$) and Pessimistic ($\blacktriangleleft$) Subtyping

$$\boxed{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \downarrow \tau} \qquad \frac{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \uparrow \tau' \qquad \mathcal{H} \vdash \tau' \lhd \tau}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \downarrow \tau}$$

$$\boxed{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \uparrow \tau} \qquad \frac{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \uparrow \tau \quad \mathcal{S} \vdash \tau.f : \tau_f}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e.f \uparrow \tau_f} \qquad \frac{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \uparrow \tau \quad \mathcal{S} \vdash \tau.\lambda(\tau_1, \ldots) : \tau_\lambda \quad \forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e_i \downarrow \tau_i}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e(e_1, \ldots) \uparrow \tau_\lambda} \qquad \frac{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \uparrow \tau \quad \mathcal{S} \vdash \tau.f(\tau_1, \ldots) : \tau_f \quad \forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e_i \downarrow \tau_i}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e.f(e_1, \ldots) \uparrow \tau_f}$$

$$\frac{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash b}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash \mathsf{new}\ \lambda\langle b \rangle \uparrow \mathbf{dyn}} \qquad \frac{\begin{array}{c}\forall i, i'.\ f_i = f_{i'} \Rightarrow i = i' \qquad \forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e_i \uparrow \tau_i \qquad \nexists i, i'.\ f_i = m_{i'} \\ \forall i, i'.\ m_i = m_{i'} \Rightarrow i = i' \qquad \forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Gamma, x : \mathbf{dyn} \vdash b_i\end{array}}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash \mathsf{new}\ x := \{f_1 := e_1; \ldots \mid m_1 b_1; \ldots\} \uparrow \mathbf{dyn}}$$

Fig. 6. Expression Typing (selected rules of $\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \uparrow \tau$ from Figure A.8)

optimism underlying gradual typing that a program is worth executing if it is plausible that the types of the eventual run-time values could match.

The gradual guarantee depends on an asymmetric variant of consistency, called *precision*, shown in Figure 4. Intuitively, one type is more precise than another if the two can be made equal by replacing occurrences of **dyn** in the less precise type. Despite what the standard notation $\vdash \tau \sqsubseteq \tau'$ might suggest, the left-hand type τ is more (not less) precise than the right-hand type τ'. As we will see in Section 4, this concept can be extended to whole programs and provides a formal means to talk about program migration from less to more precise programs.

In type-checking, subtyping commonly plays a role similar to consistency—both relax what would otherwise be a type-unification requirement. A language featuring both subtyping and consistency needs to combine them, and Siek and Taha [2007] showed early in the history of gradual typing that the right way of doing that is what they called *consistent* subtyping, but which we call *optimistic*[1] subtyping, defined in Figure 5. The optimism in optimistic subtyping is useful for static type-checking. At run time, however, we care about safety and thus not about whether two types *could* plausibly match, but whether the values of one are guaranteed to belong to the other. Pessimistic subtyping, also defined in Figure 5, corresponds to this latter relationship.

Figure 6 shows the most interesting expression-typing rules of MonNom. MonNom uses bidirectional type-checking. For example, application and method invocation *synthesize* ($\uparrow$) the type of the receiver, use that synthesized type to lookup the corresponding signature, and then *check* ($\downarrow$) that the arguments have the expected parameter types. Note that the lookup judgements $\mathcal{S} \vdash \tau.f : \tau_f$ and $\mathcal{S} \vdash \tau.ms$ (defined in Figure A.7) account for **dyn** receivers—so that they return **dyn** and, in the latter case, accept all **dyn** parameters—which means that some expressions can be type-checked in multiple ways: as a dynamically typed field access followed by a dynamically typed application, or as a dynamically typed method invocation. Such **dyn** receivers arise in MonNom from more than

[1]We use the nomenclature of Nom [Muehlboeck and Tate 2017] with respect to optimistic and pessimistic subtyping.

$$\boxed{\vdash \mathcal{P}} \qquad \frac{\vdash \mathcal{H} \qquad \mathcal{H} \vdash \mathcal{S} : \mathcal{H} \qquad \mathcal{H} \mid \mathcal{S} \vdash \mathcal{I} : \mathcal{S} \qquad \mathcal{H} \mid \mathcal{S} \mid \varnothing \vdash e \downarrow \mathbb{B}}{\vdash \langle \mathcal{H} \mid \mathcal{S} \mid \mathcal{I} \mid e \rangle}$$

Fig. 7. Program Typing (selected judgements from Figure A.5)

just omitted type annotations because, as mentioned before, structural objects such as lambdas and records are assigned type **dyn** rather than a structural type.

Figure 7 specifies when a MonNom program is considered valid. In particular, the inheritance hierarchy must be valid ($\vdash \mathcal{H}$). Then the signature must provide method signatures for every class and interface as well as constructor and field signatures for every class in a manner that respects inheritance of methods ($\mathcal{H} \vdash \mathcal{S} : \mathcal{H}$). The implementation must provide, for every class, a constructor initializing every field as well as a body for each method in accordance with the signature ($\mathcal{H} \mid \mathcal{S} \vdash \mathcal{I} : \mathcal{S}$). And lastly, the main expression must have Boolean type. The definition of these respective judgements is deferred to Appendix A.2.2 because there is nothing surprising for readers familiar with Java-like languages (though some might find the details of method inheritance with gradual types interesting).

4 TRANSITION

Now that we have an overview of the syntax and type system the MonNom programmer works within, we can more formally discuss the primary goal of the paper: guaranteeing a transition path from untyped structural code to typed nominal code. Our calculus and language are designed to enable programmers to replace a record or lambda with a corresponding (new) class and be assured that the change will preserve the behavior of the program. Similarly, programmers should be able to define interfaces describing how objects are used and transparently replace dynamic types with those new interfaces.

As we discussed earlier, key to reasoning about this transition is the precision relation. The precision relation ($\sqsubseteq$) indicates the left component is more precise than the right component. The gradual guarantee essentially says that more precise components can always be relaxed to less precise variants and the only effect on behavior would be reduction in errors. The static gradual guarantee is specifically about compile-time behavior, whereas the dynamic gradual guarantee is specifically about run-time behavior. These guarantees ensure that no surprises happen when transitioning an imprecise program to a more precise variant; the only changes that can occur are errors that can arise from incorrectly describing the invariants (e.g. types) of the program, say by annotating a variable with a type that does not accurately describe the expressions/values that get assigned to that variable.

In most existing work, the precision relation is defined on types and trivially lifted to expressions. Since we also want to model transitioning from structural to nominal code, our notion of precision is more general. Because we make structural code untyped, the static gradual guarantee is relatively uninteresting for this work. As such, here we tend to focus on the dynamic gradual guarantee, which is what requires us to develop principled run-time interactions between structural and nominal code.

Recall the example in Figure 1. In that example, one transition replaces a record with an instance of a new class. This change adds a class to the hierarchy and changes an expression in the code in a way that is not semantically equivalent because class instances are more restricted in how they can be used. Yet we can still provide a gradual guarantee between these programs, meaning we can guarantee the less precise program using a record can be used wherever the more precise program using a new class *would* work. In particular, we can guarantee that such a replacement

$$\boxed{\vdash \mathcal{H} \sqsubseteq \mathcal{H}} \qquad \frac{\vdash \mathcal{H} \sqsubseteq \mathcal{H}' \qquad \forall i'.\exists i.\ I_i = I'_{i'} \qquad \forall i, I'.\ \mathcal{H} \vdash I_i \leq I' \wedge \mathcal{H}' \vdash I' \implies \exists i'.\ \mathcal{H}' \vdash I'_{i'} \leq I'}{\vdash \mathcal{H}; N \leq I_1, \ldots \sqsubseteq \mathcal{H}'; N \leq I'_1, \ldots}$$

$$\frac{}{\vdash \varnothing \sqsubseteq \varnothing}$$

$$\boxed{\vdash \mathcal{P} \sqsubseteq \mathcal{P}} \qquad \frac{\vdash \mathcal{H}_{\mathcal{P}} \sqsubseteq \mathcal{H}_{\mathcal{P}'} \qquad \vdash \mathcal{S}_{\mathcal{P}} \sqsubseteq \mathcal{S}_{\mathcal{P}'} \qquad \mathcal{P} \sqsubseteq \mathcal{H}_{\mathcal{P}'} \vdash \mathcal{I}_{\mathcal{P}} \sqsubseteq \mathcal{I}_{\mathcal{P}'} \qquad \mathcal{P} \sqsubseteq \mathcal{H}_{\mathcal{P}'} \vdash e_{\mathcal{P}} \sqsubseteq e_{\mathcal{P}'}}{\vdash \mathcal{P} \sqsubseteq \mathcal{P}'}$$

Fig. 8. Program Precision (selected judgements from Figure A.9)

works provided all the interfaces the record gets treated as having are explicitly inherited by the new class, and all values assigned to fields in the record match the types of the fields of the class.

There is a novel technical caveat regarding our repeated emphasis on the fact that the class is new. Technically speaking, the dynamic gradual guarantee requires that *reducing* precision does not change program behavior. So one would expect that replacing an allocation of a class C with a record allocation with the same methods and fields would not change behavior, but this is not quite true. In particular, whereas casting an instance of class C to the type C would succeed, casting a corresponding record would fail in MonNom because we only allow interfaces to be imposed upon structural objects. Of course, if the type C occurs nowhere in the less precise program, i.e. C is *new* in the more precise program, then this difference in behavior is unobservable. This means that it is critical to incorporate the fact that, as code is migrated, so is the nominal hierarchy, and precision needs to account for these changes in the nominal hierarchy even as it reasons about more local changes throughout the program. A key contribution of this work is showing how we can adapt the existing notion of precision to account for this new dimension of code migration.

4.1 Program Precision

As mentioned, reasoning about precision in MonNom first requires reasoning about the nominal hierarchy, in particular the inheritance hierarchy. The rules for precision between inheritance hierarchies are shown in Figure 8. These rules permit the more precise program to have more nominal types than the less precise program, while also ensuring that—when comparing two nominal types that *are* present in both programs—nominal subtyping coincides in both programs. Thus one can transition a program by introducing a new nominal type so long as one also simultaneously adds that nominal type to the relevant inheritance clauses. The simultaneity is important because, in MonNom, class instances cannot be cast to interfaces they do not explicitly declare, which in turn means MonNom can generate all interface-method handlers for the class at compile time (as is standard for typed object-oriented languages) and can reject subtypings not explicit in the hierarchy (which is important for efficient/decidable type-checking/casts).

The rules for precision between signatures and implementations are elided here because, besides the ability of the more precise signature/implementation to provide details for interfaces and classes that are not present in the less precise signature/implementation, precision is simply defined componentwise in the obvious manner. More importantly, the reader should observe that the precision relation for expressions (and consequently implementations) is parameterized by the more precise program and *just* the inheritance hierarchy of the less precise program. The latter parameter enables the precision relation to determine which classes in the more precise program are not in scope for the less precise expression, and the former parameter then enables the precision relation to determine which structural definitions in the less precise expression correspond to the implementations of new classes in the more precise program, as described in more detail next.

Extensibility $\chi ::= \mathsf{fix} \mid \mathsf{ext}$

$\boxed{\mathcal{P} \sqsubseteq \mathcal{H} \vdash e \sqsubseteq e}$

$$\frac{\forall i.\ \mathcal{H}' \vdash \tau_i' \qquad \forall i.\ \vdash \tau_i \sqsubseteq \tau_i' \qquad \forall i.\ \mathcal{P} \sqsubseteq \mathcal{H}' \vdash e_i \sqsubseteq e_i' \qquad \mathcal{P} \sqsubseteq \mathcal{H}' \vdash e[x_1 \mapsto x_1', \ldots] \sqsubseteq e'}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash \mathtt{let}\ \langle x_1 : \tau_1, \ldots\rangle := \langle e_1, \ldots\rangle\ \mathtt{in}\ e \sqsubseteq \mathtt{let}\ \langle x_1' : \tau_1', \ldots\rangle := \langle e_1', \ldots\rangle\ \mathtt{in}\ e'}$$

$$\frac{x \text{ is not free in } b \qquad \mathcal{P} \sqsubseteq \mathcal{H}' \vdash x := \{\mid \lambda b\} \sqsubseteq e' : \chi'}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash \mathsf{new}\ \lambda\langle b\rangle \sqsubseteq e'} \qquad \frac{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash x := \{f_1 := e_1; \ldots \mid m_1 b_1; \ldots\} \sqsubseteq e' : \mathsf{ext}}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash \mathsf{new}\ x := \{f_1 := e_1; \ldots \mid m_1 b_1; \ldots\} \sqsubseteq e'}$$

$$\frac{\begin{array}{c}\neg\, \mathcal{H}' \vdash C \qquad \forall i.\ \mathcal{H}' \vdash \tau_i' \qquad x : C(x_1 : \tau_1, \ldots)\{f_1 := e_{f;1}; \ldots \mid m_1 b_1; \ldots\} \in \mathcal{I}_{\mathcal{P}} \qquad \forall i.\ \vdash \tau_i \sqsubseteq \tau_i' \\ \forall i.\ \mathcal{P} \sqsubseteq \mathcal{H}' \vdash e_i \sqsubseteq e_i' \qquad \mathcal{P} \sqsubseteq \mathcal{H}' \vdash x := \{f_1 := e_{f;1}; \ldots \mid m_1 b_1[x_1 \mapsto x_1', \ldots]; \ldots\} \sqsubseteq e' : \chi'\end{array}}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash \mathsf{new}\ C(e_1, \ldots) \sqsubseteq \mathtt{let}\ \langle x_1' : \tau_1', \ldots\rangle := \langle e_1', \ldots\rangle\ \mathtt{in}\ e'}$$

$$\frac{\mathcal{H}' \vdash \tau' \qquad \vdash \tau \sqsubseteq \tau' \qquad \mathcal{H}_{\mathcal{P}} \vdash \tau \leq \tau_x \qquad \mathcal{P} \sqsubseteq \mathcal{H}' \vdash e_x \sqsubseteq e_x' \qquad \mathcal{P} \sqsubseteq \mathcal{H}' \vdash e[x \mapsto x'] \sqsubseteq e'}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash \mathtt{let}\ \langle x_x : \tau\rangle := \langle e_x\rangle\ \mathtt{in}\ \mathtt{let}\ \langle x : \tau_x\rangle := \langle x_x\rangle\ \mathtt{in}\ e \sqsubseteq \mathtt{let}\ \langle x' : \tau'\rangle := \langle e_x'\rangle\ \mathtt{in}\ e'}$$

$\boxed{\mathcal{P} \sqsubseteq \mathcal{H} \vdash x := \{f := e; \ldots \mid mb; \ldots\} \sqsubseteq e : \chi}$

$$\frac{x \text{ is not free in } b \qquad \mathcal{P} \sqsubseteq \mathcal{H}' \vdash b \sqsubseteq b'}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash x := \{\mid \lambda b\} \sqsubseteq \mathsf{new}\ \lambda\langle b'\rangle : \mathsf{fix}}$$

$$\frac{\forall i.\ \mathcal{P} \sqsubseteq \mathcal{H}' \vdash e_i[x \mapsto x'] \sqsubseteq e_i' \qquad \forall i.\ \mathcal{P} \sqsubseteq \mathcal{H}' \vdash b_i[x \mapsto x'] \sqsubseteq b_i'}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash x := \{f_1 := e_1; \ldots \mid m_1 b_1; \ldots\} \sqsubseteq \mathsf{new}\ x' := \{f_1 := e_1'; \ldots \mid m_1 b_1'; \ldots\} : \mathsf{ext}}$$

$$\frac{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash x := \{f_1 := e_1; \ldots \mid m_1 b_1; \ldots\} \sqsubseteq e' : \mathsf{ext} \qquad \mathcal{P} \sqsubseteq \mathcal{H}' \vdash e_f \sqsubseteq e_f'}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash x := \{f_1 := e_1; \ldots; f := e_f \mid m_1 b_1; \ldots\} \sqsubseteq e'.f := e_f' : \mathsf{ext}}$$

$$\frac{x \text{ is not free in } b \qquad \mathcal{P} \sqsubseteq \mathcal{H}' \vdash x := \{f_1 := e_1; \ldots; f := \mathsf{new}\ \lambda\langle b\rangle \mid m_1 b_1; \ldots\} \sqsubseteq e' : \chi'}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash x := \{f_1 := e_1; \ldots \mid m_1 b_1; \ldots; f b\} \sqsubseteq e' : \chi'}$$

Fig. 9. Expression Precision (selected rules from Figure A.10)

4.2 Expression Precision

Our example in Figure 1 showed that code migration can change expressions beyond just type annotations. Some of these differences are within a paradigm and some are across paradigms. Figure 9 shows the five most novel rules of our precision relation that formally reasons about such migration patterns.

The first rule is mostly straightforward. The key detail to note is that the types that occur in the less precise `let` expression are required to be valid in the provided less precise hierarchy. In prior works, the set of valid types was the same across more precise and less precise programs, and as such if a type was valid in the more precise program it was necessarily valid in the less precise program, making this requirement hold automatically. But our guarantees explicitly reason about changes in the nominal hierarchy, and as such this requirement needs to be made explicit.

The three new rules make use of the judgement $\mathcal{P} \sqsubseteq \mathcal{H}' \vdash x := \{f_1 := e_1; \ldots \mid m_1 b_1; \ldots\} \sqsubseteq e' : \chi'$, which indicates primarily that e' is an expression that will allocate an object containing the given fields and methods (with less precise initializers and bodies). The variable x indicates what variable denotes **`this`** or **`self`** in the method bodies, and the extensibility χ' indicates whether the structural object defined by e' is extensible with additional fields after allocation. Its first selected rule indicates

that a lambda expression can be used whenever there are no fields, only a λ method (with no self-reference), and the value does not need to be extensible. Its second selected rule indicates that a record can always be used and will be extensible. Its third selected rule permits a field to be added *after* the object is created (provided the object is extensible). And its fourth selected rule permits a method (with no self-reference) to instead be represented by a field initialized to a functional structural object. Thus this judgement captures the flexibility of structural objects.

Using this judgement, the first new rule indicates that a lambda expression can be relaxed to any structural object with a corresponding λ method (including a record), and the second new rule indicates that a record expression can be relaxed to any extensible structural object with the corresponding fields and methods.

The final new rule is one of the main contributions of the paper. It says that a class-instance allocation can be relaxed to a structural-object allocation with the same fields and methods of the class *provided* the class is new, i.e. *not* found in the less precise nominal hierarchy. This indicates that, in MonNom, the transition `fooey`-**C** $\implies$ `fooey`-**D** in Figure 1—solidifying a structural record into a nominal class—preserves the behavior of the program (provided the extensibility of the record was no longer needed, and the fields and methods were ascribed the appropriate types). It also indicates that, in MonNom, any place where a class instance would work (aside from casts to that same class) will also work with a lambda or record with the same structure as the class instance *despite* their lack of nominal structure (and, in particular, their lack of declared nominal interfaces). Thus this rule, in combination with the gradual guarantee, captures both the cross-paradigm migration and interoperation guarantees of MonNom.

The final expression-precision rule basically encodes a theorem of the system with regards to nominal subtyping. Consider the case where τ and τ_x respectively equal τ' and τ'_x, so that this rule applies when τ is a nominal subtype of τ_x. With this precision rule, the static gradual guarantee implies that restricting the type of a variable to a nominal subtype that its expression necessarily belongs to—such as in the less precise program of the final rule—always improves the typeability of the program (i.e. fewer compile-time errors). Similarly, the dynamic gradual guarantee implies that doing so always improves the executability of the program (i.e. fewer run-time errors). We will refer to these properties respectively as *static subsumption* and *dynamic subsumption*, which are closely related to the Liskov substitution principle [Liskov and Wing 1994].

4.3 The Static Gradual Guarantee

MonNom ensures a compile-time guarantee about transitioning between structural and nominal code, which is our adaptation of the static gradual guarantee [Siek et al. 2015a].

Theorem 4.1 (Static Gradual Guarantee). *For all programs satisfying* $\vdash \mathcal{P} \sqsubseteq \mathcal{P}'$, *if* $\vdash \mathcal{P}$, $\vdash \mathcal{H}_{\mathcal{P}'}$, *and* $\mathcal{H}_{\mathcal{P}'} \vdash \mathcal{S}_{\mathcal{P}'} : \mathcal{H}_{\mathcal{P}'}$ *hold, then so does* $\vdash \mathcal{P}'$.

This theorem (whose proof is in Appendix A.3.3) ensures that if a program is well-typed then any less precise (i.e. more dynamically typed) program is necessarily also well-typed (provided at least its inheritance hierarchy and signature are well-formed[2]). In particular, according to our definition of precision, this implies that a program is well-typed if it is possible to add classes and interfaces, replace all records and lambdas with allocations of classes, and replace all occurrences of **dyn** with nominal types such that the resulting program would be well-typed. In other words, any program with a viable path towards being statically well-typed is guaranteed to be gradually well-typed, even if that path requires changing structural code to nominal code.

[2]The issue is that signatures of methods in sub-classes/interfaces and super-classes/interfaces need to be compatible with each other, so if one decides to relax a program by replacing some type in a method signature with **dyn** in some interface, then one must make sure to similarly relax the method's signature in inheriting classes and interfaces.

Observation $o ::= \texttt{false} \mid \texttt{true} \mid \infty \mid \textbf{error}$

$$\boxed{\vdash \mathcal{P} \twoheadrightarrow o} \qquad \frac{\vdash \mathcal{P} \rightsquigarrow \check{\mathcal{P}} \qquad \check{e}_1 = \check{e}_{\check{\mathcal{P}}} \qquad H_1 = \varnothing \qquad \forall i < n.\ \check{\mathcal{P}} \vdash H_i \mid \check{e}_i \longrightarrow H_{i+1} \mid \check{e}_{i+1} \qquad e_n = v \qquad o = v}{\vdash \mathcal{P} \twoheadrightarrow o}$$

$$\frac{\vdash \mathcal{P} \rightsquigarrow \check{\mathcal{P}} \qquad \check{e}_1 = \check{e}_{\check{\mathcal{P}}} \qquad H_1 = \varnothing \qquad \forall i < n.\ \check{\mathcal{P}} \vdash H_i \mid \check{e}_i \longrightarrow H_{i+1} \mid \check{e}_{i+1} \qquad \check{\mathcal{P}} \vdash H_n \mid \check{e}_n \longrightarrow \textbf{error}}{\vdash \mathcal{P} \twoheadrightarrow \textbf{error}} \qquad \frac{\vdash \mathcal{P} \rightsquigarrow \check{\mathcal{P}} \qquad \check{e}_1 = \check{e}_{\check{\mathcal{P}}} \qquad H_1 = \varnothing \qquad \forall i \in \mathbb{N}.\ \check{\mathcal{P}} \vdash H_i \mid \check{e}_i \longrightarrow H_{i+1} \mid \check{e}_{i+1}}{\vdash \mathcal{P} \twoheadrightarrow \infty}$$

Fig. 10. Program Semantics

Lowered

Program	$\check{\mathcal{P}} ::= \langle \mathcal{H} \mid \mathcal{S} \mid \check{\mathcal{I}} \mid \check{e} \rangle$	Method Body	$\check{b} ::= (\Gamma) \mapsto \check{e} : \tau$
Type Context	$\check{\Gamma} ::= \varnothing \mid \check{\Gamma}, x$	Implementation	$\check{\mathcal{I}} ::= \varnothing \mid \check{\mathcal{I}}; x : C(\check{\Gamma})\{f := \check{e}; \ldots \mid m\check{b}\}$
Expression	$\check{e} ::= x \mid \texttt{let}\ \langle \check{\Gamma} \rangle := \langle \check{e}, \ldots \rangle\ \texttt{in}\ \check{e} \mid \texttt{false} \mid \texttt{true} \mid \check{e} == \check{e}$		
	$\mid\ \check{e}.f^{\delta} \mid \check{e}.f^{\delta} := \check{e} \mid \check{e}.m(\check{e}, \ldots)^{\delta}$		
	$\mid\ \texttt{new}\ C(\check{e}, \ldots) \mid \texttt{new}\ \lambda\langle \check{b} \rangle \mid \texttt{new}\ x := \{f := \check{e}; \ldots \mid m\check{b}; \ldots\}$		
	$\mid\ \ell \mid \langle \ell.f \rangle \mid C(v, \ldots)\{\check{e}, \ldots\} \mid \texttt{cast}^{\gamma}\ \check{e}\ \texttt{to}\ \tau \mid \texttt{impose}^{\gamma}\ \ell.m\ \texttt{on}\ \check{e}$		
Location	ℓ	Value	$v ::= \texttt{false} \mid \texttt{true} \mid \ell \mid \langle \ell.f \rangle$
Guard Mode	$\gamma ::= \varnothing \mid \textbf{dyn}$	Dispatch Mode	$\delta ::= \langle \tau \rangle$

Fig. 11. Lowered Grammar

5 SEMANTICS

The semantics of MonNom was carefully designed to simultaneously ensure our transition guarantee and to enable an efficient implementation. Before going into some of the more detailed rules, Figure 10 provides the overall structure of how we formalize our semantics. First, the program is lowered. Second, the program state is initialized to the lowering of the main expression of the program and to the empty heap. Lastly, the program state is repeatedly reduced until either arriving at some (Boolean) value, erring, or simply running ad infinitum, each of which we consider to be the observable semantics of the program.

5.1 Lowering

We define the semantics of MonNom by translating (surface) programs to lowered programs. The grammar of these lowered programs is shown in Figure 11. Note that there is no change to the inheritance hierarchy or the signature when lowering; the only changes are to expressions and components that depend on expressions (such as the implementation).

The primary change to expressions is in the second row, where we make field access and method invocation explicitly distinct (translating application to λ-invocation), and attach *dispatch modes* to each of the forms of type-directed object-access/mutation in order to keep track of the type of the receiver the program was compiled with. Our implementation uses this type information to determine how the invocation should be implemented: a v-table lookup, an interface-method lookup, or a structural-dictionary lookup. The dispatch mode is even semantically relevant, as it affects the casting behavior of the invocation, though MonNom's dynamic-subsumption property guarantees that using the receiver's synthesized type imposes the least-restrictive casts compared to any of its nominal supertypes.

$$\boxed{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \downarrow \tau \rightsquigarrow \check{e}} \qquad \frac{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \rightsquigarrow \check{e}}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \downarrow \tau \rightsquigarrow \mathsf{cast}^{\mathbf{dyn}}\ \check{e}\ \mathsf{to}\ \tau}$$

$$\boxed{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \rightsquigarrow \check{e}}$$

$$\frac{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \uparrow \tau \qquad \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \rightsquigarrow \check{e}}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e.f \uparrow \check{e}.f^{\langle \tau \rangle}} \qquad \frac{\begin{array}{c}\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \uparrow \tau \qquad \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \rightsquigarrow \check{e} \\ \mathcal{S} \vdash \tau.f : \tau_f \qquad \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e_f \downarrow \tau_f \rightsquigarrow \check{e}_f\end{array}}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e.f := e_f \rightsquigarrow \check{e}.f^{\langle \tau \rangle} := \check{e}_f}$$

$$\frac{\begin{array}{c}\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \uparrow \tau \\ \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \rightsquigarrow \check{e} \qquad \mathcal{S} \vdash \tau.\lambda(\tau_1, \ldots) : \tau_\lambda \\ \forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e_i \downarrow \tau_i \rightsquigarrow \check{e}_i\end{array}}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e(e_1, \ldots) \rightsquigarrow \check{e}.\lambda(\check{e}_1, \ldots)^{\langle \tau \rangle}} \qquad \frac{\begin{array}{c}\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \uparrow \tau \\ \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \rightsquigarrow \check{e} \qquad \mathcal{S} \vdash \tau.f(\tau_1, \ldots) : \tau_f \\ \forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e_i \downarrow \tau_i \rightsquigarrow \check{e}_i\end{array}}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e.f(e_1, \ldots) \rightsquigarrow \check{e}.f(\check{e}_1, \ldots)^{\langle \tau \rangle}}$$

$$\boxed{\vdash \mathcal{P} \rightsquigarrow \check{\mathcal{P}}} \qquad \frac{\mathcal{H} \mid \mathcal{S} \vdash \mathcal{I} \rightsquigarrow \check{\mathcal{I}} \qquad \mathcal{H} \mid \mathcal{S} \mid \varnothing \vdash e \downarrow \mathbb{B} \rightsquigarrow \check{e}}{\vdash \langle \mathcal{H} \mid \mathcal{S} \mid \mathcal{I} \mid e \rangle \rightsquigarrow \langle \mathcal{H} \mid \mathcal{S} \mid \check{\mathcal{I}} \mid \check{e} \rangle}$$

Fig. 12. Lowering (selected rules of $\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \rightsquigarrow \check{e}$ from Figure A.17)

$$\begin{array}{llll} \text{Heap} & H & ::= & \varnothing \mid H; \ell \mapsto h \\ \text{Heap Value} & h & ::= & C(v, \ldots)\{v, \ldots\} \mid \lambda_\iota b \mid \{f \mapsto_\mu v; \ldots \mid mb; \ldots\}_\iota \end{array} \qquad \begin{array}{llll} \text{Mark} & \mu & ::= & \mathsf{init} \mid \mathsf{mut} \\ \text{Imposition} & \iota & ::= & \varnothing \mid \iota, I \end{array}$$

Fig. 13. Heap Grammar

The second change is the various constructs added in the final row. Most of these arise during execution. The only one that arises during lowering is `cast`. The cast is labeled with a *guard mode* indicating whether or not the cast can err; unguarded (∅) casts cannot err whereas guarded (**dyn**) casts can.

Figure 12 presents the most interesting rules for lowering programs. There are two judgements for lowering expressions: one corresponding to when the typing rules *checked* for a particular type, and one corresponding to when the typing rules *synthesized* the type from the expression. For lowering checked expressions, one simply inserts a guarded cast to the expected type. For lowering unchecked expressions, note that the synthesized type of the receiver is used as the dispatch mode of the object-access/mutation operations. Note, also, that some expressions can be lowered to either a field access followed by a λ-invocation or to simply a named method invocation. In MonNom, we ensure that the non-determinism of lowering does not affect the program's observable semantics. (The proof is in Appendix A.9.5, which also provides a more precise theorem that ensures specifically lowering is observationally deterministic without requiring reduction to be observationally deterministic.)

Theorem 5.1 (Determinism). *For all programs satisfying* $\vdash \mathcal{P}$*, any two observations satisfying* $\vdash \mathcal{P} \twoheadrightarrow o$ *and* $\vdash \mathcal{P} \twoheadrightarrow o'$ *are necessarily equal.*

5.2 The Heap

After lowering, during execution, MonNom uses a heap, both due to the stateful nature of its objects, and to implement gradual typing efficiently. As shown in Figure 13, the heap is a (partial) mapping of locations to heap values, which are class instances, lambda closures, or records. Note that record fields have a marking μ that indicates whether the field has its initial value or has been

$$\boxed{\check{\mathcal{P}} \mid H \vdash v.m \rightsquigarrow_\gamma \check{b}} \qquad \frac{\ell \mapsto \lambda_\iota \check{b} \in H}{\check{\mathcal{P}} \mid H \vdash \ell.\lambda \rightsquigarrow_{\mathbf{dyn}} \check{b}} \qquad \frac{\ell \mapsto \{\ldots \mid m_1 \check{b}_1; \ldots\}_\iota \in H}{\check{\mathcal{P}} \mid H \vdash \ell.m_i \rightsquigarrow_{\mathbf{dyn}} \check{b}_i}$$

$$\frac{\ell \mapsto C(v_1, \ldots)\{\ldots\} \in H \qquad x : C(x_1, \ldots)\{\ldots \mid m_1 \check{b}_1; \ldots\} \in \check{\mathcal{I}}_{\check{\mathcal{P}}}}{\check{\mathcal{P}} \mid H \vdash \ell.m_i \rightsquigarrow_\varnothing \check{b}_i[x \mapsto \ell, x_1 \mapsto v_1, \ldots]} \qquad \frac{\check{\mathcal{P}} \mid H \vdash \ell.f \rightsquigarrow_\gamma \check{b}}{\check{\mathcal{P}} \mid H \vdash \langle \ell.f \rangle.\lambda \rightsquigarrow_\gamma \check{b}}$$

$$\boxed{\check{\mathcal{P}} \mid H \rightarrow H \vdash \ell.m \rightsquigarrow_\gamma \check{b}} \qquad \ell \mapsto \{f_1 \mapsto_{\mu_1} v_1; \ldots; f \mapsto_{\mathsf{init}} v; f_1' \mapsto_{\mu_1'} v_1'; \ldots \mid m_1 \check{b}_1; \ldots\}_\iota \in H \quad \check{\mathcal{P}} \mid H \vdash v.\lambda \rightsquigarrow_\gamma \check{b}$$

$$\frac{\check{\mathcal{P}} \mid H \vdash \ell.m \rightsquigarrow_\gamma \check{b}}{\check{\mathcal{P}} \mid H \rightarrow H \vdash \ell.m \rightsquigarrow_\gamma \check{b}} \qquad \frac{H' = H[\ell \mapsto \{f_1 \mapsto_{\mu_1} v_1; \ldots; f \mapsto_{\mathsf{init}} v; f_1' \mapsto_{\mu_1'} v_1'; \ldots \mid m_1 \check{b}_1; \ldots\}_\iota]}{\check{\mathcal{P}} \mid H \rightarrow H' \vdash \ell.f \rightsquigarrow_{\mathbf{dyn}} \check{b}}$$

Fig. 14. Heap Semantics (selected judgements from Figure A.18)

mutated, the purpose of which we will explain in Section 5.6. More importantly, lambda closures and records have a list of "imposed" interfaces that is initially empty but is expanded as they get cast to interfaces during execution. This list guarantees the heap value has any λ-method expected by these interfaces (whereas named methods are checked on demand), and it affects how the values returned by methods of the heap value are cast.

The semantics use various judgements for operating on the heap. In Figure 14, we show only the two more interesting judgements. The first judgement is for fetching the body of a value's method. The second judgement is for fetching the body of a (typed) location's method (potentially from a field). The details of their rules will be discussed in Section 5.6.

5.3 Values

Our calculus has only four kinds of values, though one of those—locations—indirectly represents three kinds of heap values. Up until now we have not discussed $\langle \ell.f \rangle$, which denotes a bound method. These arise from the need for a class or record with a method needing to be more precise than a record with a field containing a functional value, combined with the need for an untyped field access followed by application to be semantically equivalent to an untyped method invocation. Thus the value $\langle \ell.f \rangle$ denotes the functional value resulting from accessing a method as if it were an (untyped) field (as shown in the first rule in Figure 17). In order for transitions to preserve the behavior of (==), it is important that two separate untyped field accesses of the same method return the same value, which is why we formalize bound methods as a value rather than a heap value. That said, in our implementation we implement these as a variant of lambda closures—using other techniques to ensure uniqueness—that are artificially restricted to disallow casting in order to be faithful to the calculus.

5.4 Casts

MonNom has two casting operators: $\texttt{cast}^\gamma\ \check{e}\ \texttt{to}\ \tau$ and $\texttt{impose}^\gamma\ \ell.m\ \texttt{on}\ \tau$. The first operator uses the first judgement in Figure 15 to cast the value to the expected type, and the second operator looks up the impositions on the given ℓ, then uses the third judgement in Figure 15 to determine the applicable return types the imposed interfaces expect the method m to have, and finally casts the value according to those types using the first judgement (as shown in the relevant rule in Figure 17).

The first two rules of casting simply check to see if the value already has the expected types. The second two rules specify a form of *monotonic* casts [Siek et al. 2015b; Vitousek et al. 2017]—checking if the *location* is compatible with the expected type, and then modifying the *location*'s state in the

$$\boxed{\check{\mathcal{P}} \mid H \to H \vdash v : \vec{\tau}}$$

$$\frac{}{\check{\mathcal{P}} \mid H \to H \vdash v : \varnothing} \qquad \frac{\check{\mathcal{P}} \mid H \to H' \vdash v : \vec{\tau} \qquad \check{\mathcal{P}} \mid H' \vdash v : \tau}{\check{\mathcal{P}} \mid H \to H' \vdash v : \vec{\tau}, \tau}$$

$$\frac{\check{\mathcal{P}} \mid H \to H' \vdash \ell : \vec{\tau} \quad \ell \mapsto \lambda_\iota \check{b} \in H' \quad H'' = H'[\ell \mapsto \lambda_{\iota,I} \check{b}]}{\check{\mathcal{P}} \mid H \to H'' \vdash \ell : \vec{\tau}, I} \qquad \frac{\check{\mathcal{P}} \mid H \to H' \vdash \ell : \vec{\tau} \quad \ell \mapsto \{f_1 \mapsto_{\mu_1} v_1; \ldots \mid m_1 \check{b}_1; \ldots\}_\iota \in H' \quad \forall s.\, \mathcal{S}_{\check{\mathcal{P}}} \vdash I.\lambda s \implies \exists i.\, m_i = \lambda \quad H'' = H'[\ell \mapsto \{f_1 \mapsto_{\mu_1} v_1; \ldots \mid m_1 \check{b}_1; \ldots\}_{\iota,I}]}{\check{\mathcal{P}} \mid H \to H'' \vdash \ell : \vec{\tau}, I}$$

$$\boxed{\check{\mathcal{P}} \mid H \vdash v : \tau}$$

$$\frac{}{\check{\mathcal{P}} \mid H \vdash v : \mathbf{dyn}} \qquad \frac{}{\check{\mathcal{P}} \mid H \vdash \texttt{false} : \mathbb{B}} \qquad \frac{}{\check{\mathcal{P}} \mid H \vdash \texttt{true} : \mathbb{B}} \qquad \frac{\ell \mapsto C(\ldots)\{\ldots\} \in H \quad \mathcal{H}_{\check{\mathcal{P}}} \vdash C \blacktriangleleft \tau}{\check{\mathcal{P}} \mid H \vdash \ell : \tau} \qquad \frac{\mathcal{H}_{\check{\mathcal{P}}} \mid H \vdash \ell \mapsto \iota \quad \mathcal{H}_{\check{\mathcal{P}}} \vdash \iota \blacktriangleleft \tau}{\check{\mathcal{P}} \mid H \vdash \ell : \tau}$$

$$\boxed{\mathcal{S} \vdash (\iota).m : \vec{\tau}}$$

$$\frac{}{\mathcal{S} \vdash (\varnothing).m : \varnothing} \qquad \frac{\mathcal{S} \vdash (\iota).m : \vec{\tau} \qquad \mathcal{S} \vdash I.m(\ldots) : \tau}{\mathcal{S} \vdash (\iota, I).m : \vec{\tau}, \tau} \qquad \frac{\mathcal{S} \vdash (\iota).m : \vec{\tau} \qquad \nexists s.\, \mathcal{S} \vdash I.ms}{\mathcal{S} \vdash (\iota, I).m : \vec{\tau}}$$

Fig. 15. Cast Semantics (selected judgements from Figure A.19)

heap to permanently impose the expected type upon it. Note that in MonNom only interface types can be imposed in this manner—structural objects cannot be cast to class types. Also note that MonNom does not impose interface types upon class instances—class instances are restricted to just the explicitly inherited interfaces, both enabling their implementations to be optimized for that specific set of interfaces and enabling casting of nominal values to be implemented using efficient nominal subtyping.

When imposing an interface upon a structural object, our semantics does not check for presence or compatibility of all methods expected by the interface. We make this choice for two reasons. First, besides migration, one application we aim to serve in this work is facilitating prototyping and testing, wherein it is common for the developer to want to implement only the subset of the methods expected by the interface that is actually needed for the focused task at hand (without having to provide tedious stubs for the unneeded methods). Second, eagerly checking all these methods is costly and offers no performance when using the method-invocation techniques we discuss in Section 6.2. However, we do eagerly check that if the interface provides a λ-method then so does the structural object. The distinguished nature of that name enables some performance optimizations by using an optimized memory layout for all values with such a method, and so to be memory safe our implementation needs to eagerly check that the value has a λ-method and therefore an optimized memory layout, as we will discuss in Section 6.4.

5.5 Errors and Safety

The MonNom calculus makes a distinction between getting stuck and erring. In particular, while an error does arise from getting stuck, it can only happen in specific expressions where the implementation knows to explicitly check for such conditions. We formalize this semantics for errors in Figure 16.

Notice that nearly every potentially erroneous redex involves either a dynamic dispatch mode or a dynamic guard mode, reflecting the fact that statically typed code and unguarded casts should not err. There is, however, one exception: invocation of *named* methods of *interfaces*. This is due to the fact that our casts of structural objects to interfaces do not check for the presence or compatibility

$$\text{Potentially Erroneous Redex } \varepsilon ::= v.f^{\langle \mathbf{dyn} \rangle} \mid v.f^{\langle \mathbf{dyn} \rangle} := v \mid \ell.f(v,\ldots)^{\langle I \rangle} \mid v.m(v,\ldots)^{\langle \mathbf{dyn} \rangle} \mid \mathtt{cast}^{\mathbf{dyn}}\ v\ \mathtt{to}\ N \mid \mathtt{impose}^{\mathbf{dyn}}\ \ell.m\ \mathtt{on}\ v$$

$$\boxed{\check{\mathcal{P}} \vdash H \mid \check{e} \longrightarrow \mathbf{error}} \qquad \frac{\nexists H', \check{e}'.\ \check{\mathcal{P}} \vdash H \mid \varepsilon \longrightarrow H' \mid \check{e}'}{\check{\mathcal{P}} \vdash H \mid E[\varepsilon] \longrightarrow \mathbf{error}}$$

Fig. 16. Error Semantics (where Evaluation Contexts E are defined in Figure A.20)

of named methods. Nonetheless, we can implement this without adding significant dynamic checks to typed invocation by using the technique described in Section 6.2.

By distinguishing erring from getting stuck, we can formalize the type safety of MonNom.

Theorem 5.2 (Safety). *For any program satisfying* $\vdash \mathcal{P}$*, there exists an observation satisfying* $\vdash \mathcal{P} \twoheadrightarrow o$*.*

This theorem (whose proof is in Appendix A.8.1) indicates that well-typed programs only get stuck if they reach a potentially erroneous redex. We cannot ensure that well-typed programs do not err due to the presence of dynamic typing in MonNom, though one can show that the "statically typed" subset of MonNom would be free of errors.

5.6 Invocation

Being an object-oriented language, method invocation is a core feature of MonNom. It is important to remember that every invocation has two sides: the caller and the callee. In MonNom, lowering takes care of the caller side by determining the dispatch mode to use and inserting casts of the arguments to the parameter types expected by that dispatch mode. Thus the three rules for invocation shown in Figure 17 focus on the callee side.

Each of these three rules uses three different (but closely related) judgements to operate on the heap (recall Figure 14) in order to lookup the body of the given method. The first rule for untyped (i.e. $\langle \mathbf{dyn} \rangle$) invocation simply looks up a body of a *method* directly provided by the receiver. The second rule for untyped invocation handles the case where the receiver directly provides a *field* of the given name, extracting the method body from the λ-method implementation provided by that field's value. Together these two cases ensure that the two ways to invoke a method—directly, or applying to the result of a field access—work equivalently in *un*typed code. The rule for *typed* (i.e. $\langle N \rangle$) invocation uses a judgement that effectively combines these two cases, but with an extra step: if the implementation is provided by a (record) field, the field is checked to have never been mutated. Common implementations of and optimizations for method invocation in major nominal object-oriented languages assume methods are immutable, so this extra step dynamically asserts that those assumptions are valid for the structural interoperation at hand. In particular, whereas with the untyped expression $e.f(e_1,\ldots)$ one determines the value of the field before evaluating the arguments, we found it useful for typed method invocation to lookup the method while simultaneously supplying its *already* evaluated arguments, and so for these to be equivalent—so that the dynamic gradual guarantee holds—we need the field to not have been mutated in the interim.

After the method body has been looked up, it is supplied the respective arguments, casting them to the types expected by the method body. For untyped invocation, these casts are always guarded, where for typed invocation they are unguarded if the receiver was a class instance. Furthermore, the value returned by the call is checked to be compatible with the expected return type. For untyped invocation, this check is a no-op since the caller is untyped (though still having the no-op in the

$$\boxed{\check{\mathcal{P}} \vdash H \mid \check{e} \xrightarrow{H} H \mid \check{e}}$$

$$\frac{\check{\mathcal{P}} \mid H \vdash \ell.f \rightsquigarrow_\gamma \check{b}}{\check{\mathcal{P}} \vdash H \mid \ell.f^{\langle \mathbf{dyn} \rangle} \xrightarrow{\varnothing} H \mid \langle \ell.f \rangle}$$

$$\frac{\check{\mathcal{P}} \mid H \vdash v.m \rightsquigarrow_\gamma (x_1 : \tau_1, \ldots) \mapsto \check{e} : \tau}{\check{\mathcal{P}} \vdash H \mid v.m(v_1, \ldots)^{\langle \mathbf{dyn} \rangle} \xrightarrow{\varnothing} H \mid \mathsf{cast}^{\varnothing}\ \mathtt{let}\ \langle x_1, \ldots \rangle := \langle \mathsf{cast}^{\mathbf{dyn}}\ v_1\ \mathtt{to}\ \tau_1, \ldots \rangle\ \mathtt{in}\ \check{e}\ \mathtt{to}\ \mathbf{dyn}}$$

$$\frac{\check{\mathcal{P}} \mid H \vdash \ell.f \mapsto v \qquad \check{\mathcal{P}} \mid H \vdash v.\lambda \rightsquigarrow_\gamma (x_1 : \tau_1, \ldots) \mapsto \check{e} : \tau}{\check{\mathcal{P}} \vdash H \mid \ell.f(v_1, \ldots)^{\langle \mathbf{dyn} \rangle} \xrightarrow{\varnothing} H \mid \mathsf{cast}^{\varnothing}\ \mathtt{let}\ \langle x_1, \ldots \rangle := \langle \mathsf{cast}^{\mathbf{dyn}}\ v_1\ \mathtt{to}\ \tau_1, \ldots \rangle\ \mathtt{in}\ \check{e}\ \mathtt{to}\ \mathbf{dyn}}$$

$$\frac{\check{\mathcal{P}} \mid H \to H' \vdash \ell.m \rightsquigarrow_\gamma (x_1 : \tau_1, \ldots) \mapsto \check{e} : \tau}{\check{\mathcal{P}} \vdash H \mid \ell.m(v_1, \ldots)^{\langle N \rangle} \xrightarrow{\varnothing} H' \mid \mathsf{impose}^{\gamma}\ \ell.m\ \mathtt{on}\ \mathtt{let}\ \langle x_1, \ldots \rangle := \langle \mathsf{cast}^{\gamma}\ v_1\ \mathtt{to}\ \tau_1, \ldots \rangle\ \mathtt{in}\ \check{e}}$$

$$\frac{\check{\mathcal{P}} \mid H \to H' \vdash v : \tau}{\check{\mathcal{P}} \vdash H \mid \mathsf{cast}^{\gamma}\ v\ \mathtt{to}\ \tau \xrightarrow{\varnothing} H' \mid v} \qquad \frac{\mathcal{H}_{\check{\mathcal{P}}} \mid H \vdash \ell \mapsto \iota \qquad \mathcal{S}_{\check{\mathcal{P}}} \vdash (\iota).m : \vec{\tau} \qquad \check{\mathcal{P}} \mid H \to H' \vdash v : \vec{\tau}}{\check{\mathcal{P}} \vdash H \mid \mathsf{impose}^{\gamma}\ \ell.m\ \mathtt{on}\ v \xrightarrow{\varnothing} H' \mid v}$$

$$\boxed{\check{\mathcal{P}} \vdash H \mid \check{e} \to H \mid \check{e}}$$

$$\frac{\check{\mathcal{P}} \vdash H \mid \check{e} \xrightarrow{H''} H' \mid \check{e}' \qquad \nexists \ell, h, h'.\ \ell \mapsto h \in H' \wedge \ell \mapsto h' \in H''}{\check{\mathcal{P}} \vdash H \mid \check{e} \to H'; H'' \mid \check{e}'}$$

Fig. 17. Lowered-Expression Semantics (selected rules from Figure A.20)

formal semantics facilitates the proof of the dynamic gradual guarantee). For typed invocation, if the receiver was structural then we need to check that the returned value matches the return types expected of all the interfaces that have been imposed upon the receiver. This necessarily includes the interface specified in dispatch mode, guaranteeing type safety. But it is important that we cast the value to *all* interfaces imposed on the location, as imposing *just* the interface the invocation happened to be typed with would fail to satisfy dynamic subsumption.

5.7 The Dynamic Gradual Guarantee

At last we can formally state our run-time guarantee about transitioning between structural and nominal code with a property known as the dynamic gradual guarantee [Siek et al. 2015a].

Theorem 5.3 (Dynamic Gradual Guarantee). *For all programs satisfying* $\vdash \mathcal{P}$, $\vdash \mathcal{P}'$, *and* $\vdash \mathcal{P} \sqsubseteq \mathcal{P}'$, *any observation satisfying* $\vdash \mathcal{P} \twoheadrightarrow o$ *either also satisfies* $\vdash \mathcal{P}' \twoheadrightarrow o$ *or is* **error**; *and for any observation satisfying* $\vdash \mathcal{P}' \twoheadrightarrow o$, *either* $\vdash \mathcal{P} \twoheadrightarrow o$ *also holds or* $\vdash \mathcal{P} \twoheadrightarrow$ **error** *holds.*

This theorem (whose proof is in Appendix A.9.4) ensures that well-typed more precise and less precise programs are observably equivalent except that the former can err when the latter does not. As mentioned earlier, statically typed MonNom is error free, which in turn means this theorem implies that a program does not err if it is possible to add interfaces, replace all records and lambdas with classes implementing those interfaces, and replace all occurrences of **dyn** with nominal types such that the resulting program statically type-checks. In other words, in combination with the static gradual guarantee, any program with a viable path towards being statically well-typed (and so necessarily never erring) is guaranteed to be gradually well-typed and to never err, even if that path requires changing structural code to nominal code.

Of course, this is great in theory, but in practice gradual typing has a history of significant issues with large overheads caused by the casts ensuring safety [Takikawa et al. 2016]. Next we demonstrate that the design of MonNom enables an efficient implementation.

6 IMPLEMENTATION

We have implemented MonNom with an ahead-of-time compiler using LLVM as the backend. Our implementation extends the calculus in many ways, including more primitives as well as a standard library for common data structures. Importantly, our implementation adds generics to MonNom. This makes implementing monotonic casts efficiently more challenging as we must account for type arguments in impositions (particularly in the function interfaces provided by our standard library and used in the relevant benchmarks). In the interest of space, though, here we discuss only the techniques that we think are most critical to achieving our performance for features of MonNom present in the calculus.

6.1 Value Representation

We represent primitive values differently depending on whether they are statically or dynamically typed. In particular, primitive values are unpacked when statically typed and packed or boxed when dynamically typed. For dynamically typed values, the lower two bits differentiate between (aligned) references and the two kinds of packed primitives—integers and floating-point numbers—adapting the packing technique from Chambers et al. [1989]. If the low bits are `11`, the packed value represents the signed integer resulting from arithmetic-right-shifting the value by 2. If the low bits are `01` or `10`, the packed value represents the IEEE-754 floating-point number resulting from funnel-shifting the value right by 3 (moving the third bit to be the sign bit); the effect of this is that all floats with small-magnitude exponents (i.e. signed 10-bit) can be packed rather than boxed. (When boxing floats, we memoize the statically allocated values for positive and negative zero.) This packing scheme prevents the performance of both integer-intensive and floating-point-intensive programs from degrading severely due to heap allocations in untyped and mixed programs.

6.2 Typed Method Invocation and Call Tags

One of the key challenges to implementing MonNom is supporting typed method invocations on structural objects that were cast to interfaces they were not created to support, ideally without slowing down the fast path where the receiver is typed.

Originally, we implemented interface-method dispatch using interface tables. In this strategy, the object descriptor (which also provides, say, the v-table of class methods) provides the list of interfaces implemented by the object, and in each case providing a method table pointing to the implementations of each interface method. When we cast a structural object, we extended its interface table with a newly allocated method table for the interface that was filled with stubs that would be filled on demand. However, these allocations were costly, and we found ourselves employing speculation and heuristics in an attempt to precompute these tables, which we worried might not scale well and could lead to unpredictable performance cliffs.

More recently, we developed an extension of interface-method tables [Alpern et al. 2001] that bridges this gap—an extension we refer to as *call tags*. An interface-method table provides a fixed-sized array of code pointers, and every interface method is globally assigned an index where its implementation lies in this array. But an object can implement multiple interface methods assigned to the same index. To address this conflict, when one calls the code pointer at the corresponding index, in addition to the explicit arguments of the method one also passes the identifier of the interface method. That way, if the receiver has multiple implementations corresponding to that index, it can first switch on that identifier before doing anything else, which works even if those methods have different arities, signatures, and even calling conventions.

In typed languages, the type system guarantees that the passed identifier will be recognized by the switching code. But in the untyped setting, we cannot make such a guarantee, and matters get

very complicated when one realizes that even the size of the stack frame cannot be known without recognizing that identifier. To resolve this, we made it so that the method identifier—i.e. call tag—itself provides the code pointer to jump to when the call tag is not recognized—leaving the arguments and return address untouched. In this way, the "fall-back" handler for an interface method can convert all the arguments to their untyped representations, then invoke the corresponding untyped method on the argument specifying the receiver, and cast the returned value to the types expected by the receiver's imposed interfaces.

By using call tags, we get an implementation strategy wherein the fast path for typed invocation on typed receivers is practically untouched, and the slow path has only small indirections employing the necessary coercions. Using this, we can simply have an object descriptor for each class and for each allocation site of a structural object—no need for speculation or heuristics.

6.3 Untyped Method Invocation

An untyped method invocation is semantically equivalent to a field access followed by an argument application. However, implementing one this way would be inefficient when the object directly provides a method implementation because it creates an intermediate bound-method value. But implementing it simply as a method invocation would be incorrect in the case where the method is provided by a field whose value is changed during evaluation of the arguments.

To address these problems, we implement untyped method invocation through two stages. The first stage is executed just before argument evaluation. Its job is to determine the call-tag handler *and* the receiver to use later. Then the arguments are evaluated. Afterwards, the second stage simply calls the previously determined call-tag handler with a call tag indicating the number of arguments, providing the previously determined receiver and the values of the arguments. Thus the first stage does the bulk of the work.

For the first stage, during compilation a hashtable is inlined directly into the object descriptor. This hashtable lists all of the named methods directly provided by the object as well as some of the fields directly provided by the object upon allocation. For named methods, the entry provides a pointer to the precompiled call-tag handler to use for the second stage. For selected fields, the entry provides the offset of the field within the object. Only fields with non-primitive types are listed, which both avoids fields that cannot provide a method implementation and ensures the values of listed fields are necessarily references if and only if their lower two bits are `00`. Executing the first stage involves first looking up the corresponding entry in the receiver's object descriptor. If a method entry is found, then the specified call-tag handler and the same receiver are forwarded to the second stage. If a field entry is found, then the field's value is checked to be a reference with a λ-method, in which case that λ-method handler and that field's value are forwarded to the second stage. If no entry is found and the object is a record, its additional-field dictionary is searched for the appropriate field and handled as with built-in fields. Otherwise, the invocation fails.

6.4 λ-Methods

Our implementation (and consequently casting semantics) employs special treatment for λ-methods. If an object implements a λ-method, room for a call-tag-handling function pointer is reserved at the head of the object itself, rather than in its descriptor. This prevents the need to load the object descriptor when invoking λ-methods.

On the callee side, for class instances the function switches on the corresponding call tags for all implemented interfaces, along with a special call tag for just the class, as well as the call tag of the appropriate arity that is used for untyped method invocation. Call tags proved to be helpful here because our implementation supports nominal subtypings such as `Int` $\leq$ `Object` as well as generics, and as such compatible method signatures can still represent objects differently (e.g. boxed

vs. unboxed integers); having a call tag for each interface then enables quick conversions before jumping to the main implementation of the method. (Note that we explored the option of restricting method-signature compatibility in order to ensure consistent object representation and simply use standard functions, but our experiments found that standard functions offered no performance improvements over call tags.) For structural objects, the function only switches on the (untyped) call tag of the appropriate arity, but if no match is found it jumps to the fall-back handler provided by the call tag, just as with named methods.

On the caller side, typed invocations of λ-methods simply load the function pointer from the standard offset within the object (forgoing the object descriptor entirely) and call it with the appropriate call tag. Because our casting semantics for casting to interfaces with a λ-method eagerly checks that the structural object has some such method, this operation is guaranteed to be safe even for structural objects masquerading as their imposed interfaces. For untyped invocations though, we first have to check if such a λ-method exists. One way to do so would be to put a flag in the object itself, but that would waste memory. Another way to do so would be to put a flag in the object descriptor, but the whole point of the optimization is to avoid loading from the descriptor. So we encoded a flag in the address of the descriptor. In particular, we use memory alignment to ensure that the address of the descriptor has a specific low bit set if and only if the object has a λ-method and consequently the corresponding function pointer at the standard offset. (We could have also taken advantage of the fact that alignment ensures the low bits of any object-descriptor address are always unused, but this would require adding bit-masking operations to typed code.)

6.5 Fields

Typed field-access/mutation is implemented using predetermined offsets. For untyped field access/mutation, if the field is typed then we have to account for potential casts and change in representation, and if the field belongs to a record then we need mutations of the field to mark it appropriately. Rather than use an inlined data structure listing field information in the object descriptor and a corresponding generalized read/write operation, the object descriptor provides two function pointers accepting a receiver and a field identifier. The former returns the (packed) value of identified field. The latter accepts an additional (packed) value that it updates the identified field to. For classes and records, the implementations of these functions switch to the fields known at compile time. For records, if the identifier is not recognized then the additional-fields dictionary is searched for a corresponding entry.

7 EVALUATION

7.1 Methodology

Following Takikawa et al. [2016], it is important to evaluate the performance of gradually typed programs in different stages of program evolution to see if run-time checks introduce prohibitive worst- or average-case overheads. This is often done by taking a fully typed program and removing type annotations from modules one by one (in varying order), and then evaluating program performance on each of the intermediate configurations. In the case of the work presented here, the picture is a little more complicated as we have an additional axis of migration—structural vs. nominal. Furthermore, type annotations involving class types can only be added after the corresponding records have been converted to class allocations. This means both that there are more variables to toggle and that not all these toggles are independent. In particular, from a fully typed nominal program we generate different configurations by

(1) removing type annotations from the fields, method signatures, and method bodies of a class (including static methods), except those necessary for inheriting typed methods,

(2) removing a class (and all type annotations referring to that class) entirely and replacing uses of its constructor with record or lambda expressions (preferring the latter when possible) whose method signatures and bodies are typed according to the typing discipline of the code they occur within,
(3) and/or removing an interface (and all type annotations referring to that interface) entirely.

We then run all the valid generated configurations and compare their running times. The expectation is that running times decrease while going from left to right, though the mixed configurations in the middle may suffer from overheads due to casts and cross-paradigm indirections.

The following experiments were run on an Intel® Core™ i7-8700 CPU with 16GB main memory running Windows 10 on minimal activity. For each configuration, the reported running time is the average over 10 runs (each of which were measured after first performing warm-up runs).

7.2 Benchmarks

MonNom is a new language with a unique set of features for which we built everything from the ground up. As such, there exists no corpus of programs in the language, and its standard library is rather small. We keep adding new code and benchmarks as we develop MonNom further, but for now, we present the results on three benchmark programs (the MonNom code for which can be found in Appendix B): `sieve`, `intersort`, and `float`.

7.2.1 `sieve`. `sieve` is a key benchmark originally developed by Takikawa et al. [2016] to represent a worst-case scenario for higher-order casts. It originally consisted of two heavily interacting modules that may each be typed or untyped, resulting in four configurations. Mixed configurations of the program feature a particularly large number of interactions (and therefore casts) between typed and untyped code compared to the rest of the work of the program, making it an important benchmark for gradual-typing implementations. Due to its small size, it is feasible to increase the granularity to individual classes and interfaces and vary each according to the possible variations discussed above. One of the modules contains three lambdas that Nom [Muehlboeck and Tate 2017] had to replace with classes implementing a particular interface.[3] The fully typed MonNom program is almost identical to the Nom program, except that it represents (nullary) functions with a *generic* (nullary-)function (standard-library) interface rather than an interface monomorphized to (nullary) functions returning integers. Furthermore, MonNom can also replace functional classes (implementing functional interfaces) with lambdas. Thus, in addition to the two main classes corresponding to the two original modules, we have three additional classes to generate various configurations with. Altogether, we consider 272 configurations of `sieve` for MonNom.

Due to its widespread use in the literature, `sieve` is also a good candidate to compare our results to related work, in particular with Grift [Kuhlenschmidt et al. 2019], Monotonic Grift [Kuhlenschmidt et al. 2018], Typed Racket [Tobin-Hochstadt and Felleisen 2006] (CS 8.0), HiggsCheck [Richards et al. 2017], Transient[4] Reticulated Python [Vitousek et al. 2014, 2017] (PyPy 7.3.5), and Nom [Muehlboeck and Tate 2017]. There are substantial differences between these languages and MonNom, and as such not all MonNom configurations have a good corresponding configuration in each of these languages. So, for each language, we selected MonNom configurations for which there were good

[3] In the original benchmark, there was also a temporary pair value, which MonNom could implement as either a record or class. However, we found the version of this benchmark included in the artifact for Grift did not use this pair value, instead accessing the values contained in it more directly. Through profiling, we found that the original version of this benchmark caused the many implementations to spend most of their time collecting garbage due to these temporarily created pairs. That issue is unrelated to gradual typing, so we went with the version in Grift's suite that focused more on gradual-typing-specific performance considerations. Note that the modified version is still memory-intensive, and as such memory-management techniques—particularly for short-lived objects—do have notable effects on absolute performance.
[4] We did not evaluate Monotonic Reticulated Python because we could not get it to support our benchmarks.

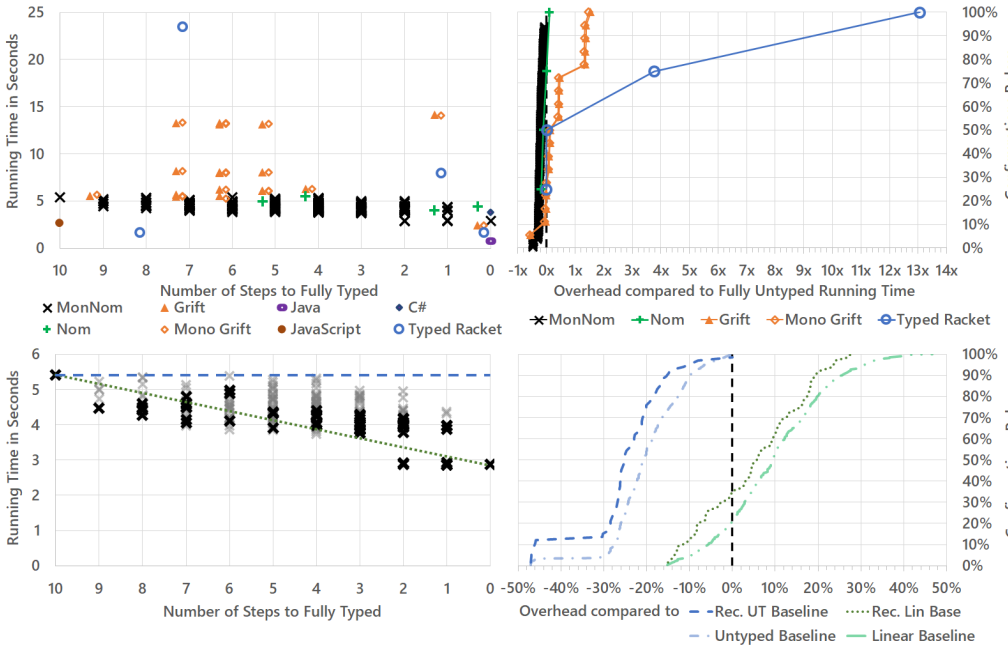

Fig. 18. Measurements for `sieve`

corresponding configurations that were also representative of how these implementations had previously been evaluated. We did the same for some major industry object-oriented languages: Java (HotSpot build 13+13), C# (CoreCLR 6.0.100-preview.7.21379.14), and JavaScript (Node.js 10.19.0).

Figure 18 shows the results of our measurements for `sieve`. The upper left-hand plot shows the absolute running times as a scatter-plot in which we group configurations by the number of steps they are away from the fully typed configuration, plotting them from left (untyped structural) to right (typed nominal). For each configuration we evaluated in other languages, we plot its measurement in the same column as its corresponding MonNom configuration.[5][6]

The upper right-hand plot shows the style of usability diagram that is common in the literature, showing the percentage of configurations that incur less than some amount of overhead compared to the fully untyped program. We see that Typed Racket has improved substantially over the years since Takikawa et al. [2016] observed more than 100x overhead on a mixed configuration of this benchmark. However, both it and Grift still incur overheads of several times the running time of the fully untyped configuration, while Nom's and MonNom's worst-case overheads are measured in percentages.

The lower left-hand plot of Figure 18 shows the numbers for the MonNom experiments in more detail. Some configurations are shown in grey while others are shown in black. The black configurations are those that follow the following "recommended" migration strategy:

[5]Though we evaluated `sieve` in HiggsCheck, its measurements are not shown. We found that the translation of `sieve` used by Richards et al. [2017] did not faithfully recreate the quantity or quality of type-boundary crossings that Takikawa et al. [2016] designed the benchmark to evaluate. When we ran HiggsCheck on our own translation, it failed to complete in any configuration due to resource exhaustion, which is why HiggsCheck is not represented in the plot even though Node.js is.
[6]Transient Reticulated Python's measurements are not shown because, though there was little relative variation across configurations, their absolutes far exceeded the upper bound of the current figure across all configurations.

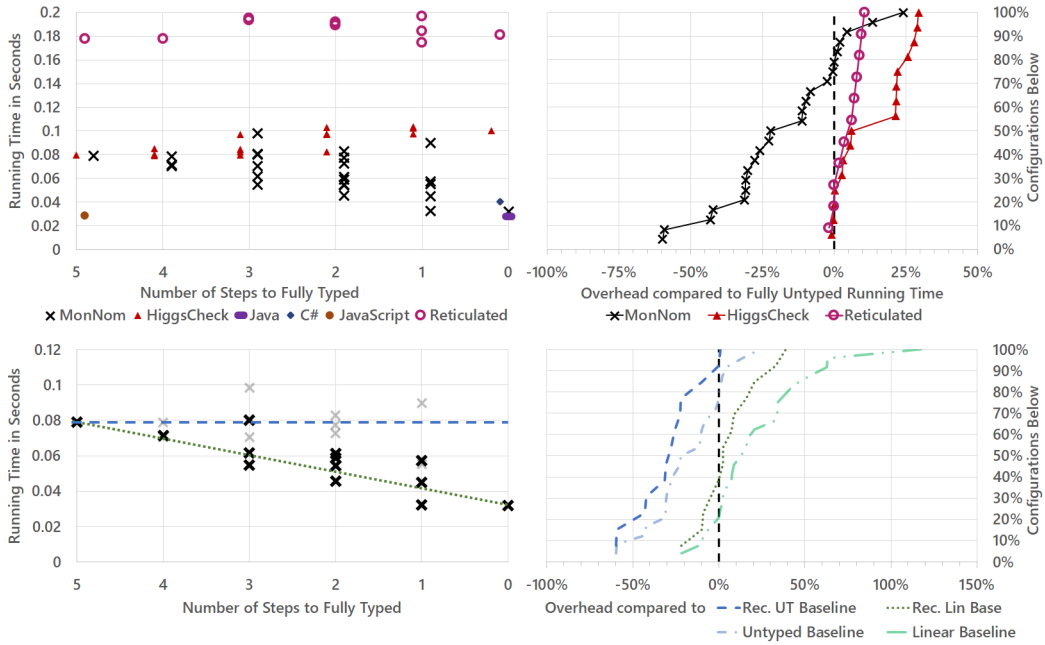

Fig. 19. Measurements for `intersort`

(1) First, if a record's fields are frequently accessed by *other* objects' methods, turn the record into a class (though not necessarily with typed fields or methods) and add the appropriate class type annotations to the other objects' methods.
(2) Second, if a class's methods call methods of another class, add type annotations to the other class before adding type annotations to this class.

The plot—and its counterparts for the other benchmarks—illustrates that this simple migration strategy for MonNom avoids the most significant pitfalls of gradual-typing overhead, and even typically leads one towards performance *improvements* proportionate to their migration effort.

The lower right-hand plot shows four sets of lines. Two lines show what percentage of "recommended" configurations exhibit a given amount of overhead, whereas another two lines show what percentage of all configurations exhibit a given amount of overhead. The blue lines plot overhead relative to the fully untyped program, i.e. the dashed blue line in the lower left-hand plot, which is the traditional metric established by Takikawa et al. [2016]. However, we believe this metric does not match one's expectation that programs should generally get faster as more types are added. As such, the green lines plot overhead relative to a linear baseline between the fully typed and fully untyped program, i.e. the dotted green line in the lower left-hand plot, which punishes failure to obtain optimizations in proportion to the migration effort put in.

7.2.2 `intersort`. While `sieve` represents a stress test for casting lambdas, `intersort` is a benchmark we designed to model a more typical scenario of specifically object-oriented code migration by sorting a data structure using just its interface methods. It consists of slightly more components than `sieve`, which we therefore group into four modules:

(1) generic interfaces for lists and bidirectional iterators,
(2) generic classes implementing those interfaces with doubly-linked lists,

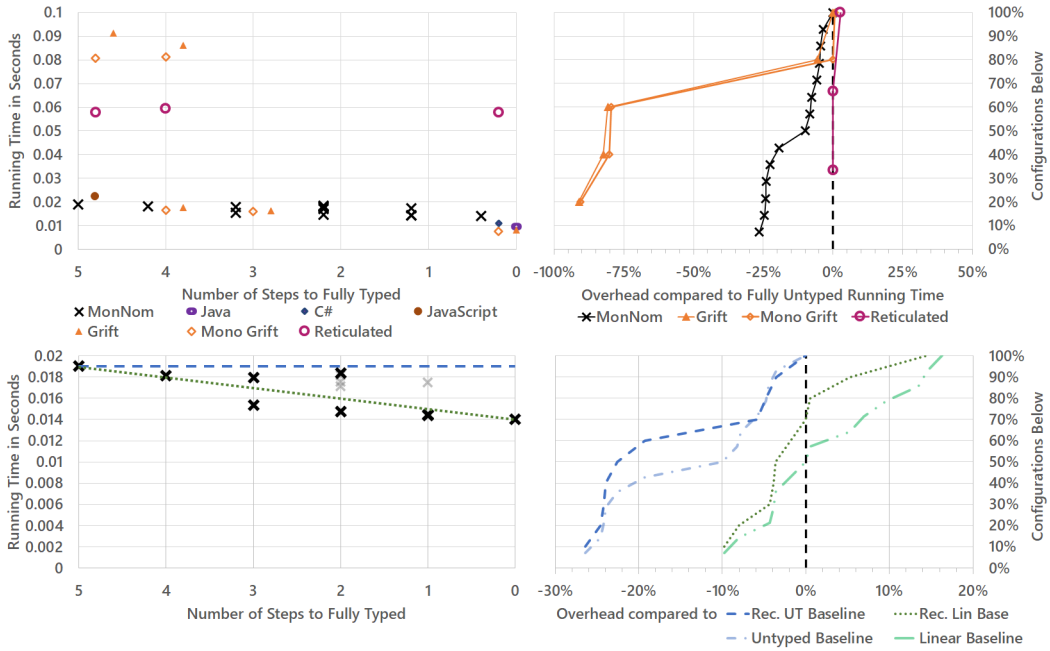

Fig. 20. Measurements for `float`

(3) an implementation of Quicksort using bidirectional iterators,
(4) and a main class generating 100,000 pseudo-random integers and then running Quicksort.

The latter two consist of only static methods and as such cannot be turned into structural objects, but the list interfaces can be removed from the program, and the classes implementing doubly-linked lists can be turned into records. We transition each module as a whole unit, resulting in 36 possible configurations.

Our measurements for `intersort` are shown in Figure 19. They illustrate that MonNom's performance generally improves proportionate to migration effort—especially when using our recommended migration strategy—despite the heavy use of classes and higher-order interfaces.

7.2.3 `float`. `float` is a benchmark we have taken from the set of benchmarks used to evaluate Nom and transient Reticulated Python [Vitousek et al. 2014, 2017]. The benchmark generates a large list of triples of floats ("points in 3D space"), iterates the list to normalize them, and then folds the list. This benchmark does not make heavy use of methods, but does make heavy use of floating-point numbers and of field accesses.

Our measurements for `float` are shown in Figure 20. For both MonNom and (both versions of) Grift, we see two cleanly separated sets of configurations. The distinguishing characterestic is whether the 3D-point data-structure is typed or not, with a notable loss when it is untyped. However, this loss is much more drastic for Grift than for MonNom. Despite being evaluated on primarily floating-point-intensive benchmarks, Grift boxes all floating-point numbers onto the heap, whereas MonNom only boxes floating-point numbers with large-magnitude exponents. We suspect this choice was critical to MonNom's success on this floating-point-intensive benchmark.

7.3 Threats to Validity

7.3.1 Overlooked Configurations. While we believe the operations above represent reasonable and realistic chunks of work that can and will often be done at once, it is possible that we missed plausible configurations with far worse running times.

7.3.2 Small Corpus. As with every new language and custom compiler/runtime, there is not any existing code that we could run and the standard library is minimal. In addition, with the larger variety of program variations and cross-program constraints on which variations are permissible, creating the benchmarks themselves is a larger task than the usual combination of fully annotated and fully unannotated files. As such, we thought it important to at least include a known worst-case benchmark (`sieve`) and a realistic benchmark (`intersort`, and possibly `float`), but our particular choices might not be sufficiently representative.

7.3.3 Usual Experimental Issues. Although benchmarks were run with consistent settings and without being affected by other concurrent processes, it is possible we failed to control for other external factors we were unaware of.

8 CONCLUSION

Nominality is a well-known device towards supporting efficient implementation, even in the context of gradual typing [Muehlboeck and Tate 2017; Wrigstad et al. 2010], and nominal class-based object-oriented systems are also a popular organizing principle in industrial programming languages. Yet both nominality and static type-checking are often seen as too onerous, in particular for the early stages of developing a program. Thus, the idea of allowing programmers to start in an untyped, structural setting and later support transforming it into a typed, nominal setting is not a new one, dating back to at least Anderson and Drossopoulou [2003]. The present paper represents the most advanced version of this idea in terms of the supported language features and overall semantic guarantees, building on years of work on gradual typing [Garcia et al. 2016; Siek and Taha 2007; Siek et al. 2015a; Tobin-Hochstadt and Felleisen 2006], while also recognizing the implementation challenges of (sound) gradual typing [Bauman et al. 2017; Greenman and Felleisen 2018; Kuhlenschmidt et al. 2019; Muehlboeck and Tate 2017; Richards et al. 2017; Roberts et al. 2019; Takikawa et al. 2016]. We presented preliminary evidence that our design can be implemented efficiently, and we showed that gradual-typing research techniques can be adapted to transition not only types but even paradigms while still providing strong guarantees and good performance.

DATA AVAILABILITY STATEMENT

Our artifact provides the source code for MonNom, the source code of each configuration used to evaluate each language, the data for each plot, and a virtual machine with everything compiled and scripts set up to reproduce all measurements [Muehlboeck and Tate 2021].

ACKNOWLEDGMENTS

We thank the reviewers for their valuable suggestions towards improving the paper. We also thank Mae Milano and Adrian Sampson, as well as the members of the Programming Languages Discussion Group at Cornell University and of the Programming Research Laboratory at Northeastern University, for their helpful feedback on preliminary findings of this work.

This material is based upon work supported in part by the National Science Foundation (NSF) through grant CCF-1350182 and the Austrian Science Fund (FWF) through grant Z211-N23 (Wittgenstein Award). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or the FWF.

REFERENCES

Bowen Alpern, Anthony Cocchi, Stephen Fink, and David Grove. 2001. Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless. In *OOPSLA*. ACM, New York, NY, USA, 108–124. https://doi.org/10.1145/504282.504291

Christopher Anderson and Sophia Drossopoulou. 2003. BabyJ: From Object Based to Class Based Programming via Types. *Electronic Notes in Theoretical Computer Science* 82, 8 (2003), 53–81. https://doi.org/10.1016/S1571-0661(04)80802-8 WOOD.

Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. 2017. Sound Gradual Typing: Only Mostly Dead. *PACMPL* 1, OOPSLA, Article 54 (2017), 24 pages. https://doi.org/10.1145/3133878

Gavin Bierman, Erik Meijer, and Mads Torgersen. 2010. Adding Dynamic Types to C#. In *ECOOP*. Springer Berlin Heidelberg, Berlin, Heidelberg, 76–100. https://doi.org/10.1007/978-3-642-14107-2_5

John Peter Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2017. Migrating Gradual Types. *PACMPL* 2, POPL, Article 15 (2017), 29 pages. https://doi.org/10.1145/3158103

Craig Chambers, David Ungar, and Elgin Lee. 1989. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *OOPSLA*. Association for Computing Machinery, New York, NY, USA, 49–70. https://doi.org/10.1145/74877.74884

Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. 2018. Collapsible Contracts: Fixing a Pathology of Gradual Typing. *PACMPL* 2, OOPSLA, Article 133 (2018), 27 pages. https://doi.org/10.1145/3276503

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *POPL*. ACM, New York, NY, USA, 429–442. https://doi.org/10.1145/2837614.2837670

Ben Greenman and Matthias Felleisen. 2018. A Spectrum of Type Soundness and Performance. *PACMPL* 2, ICFP, Article 71 (2018), 32 pages. https://doi.org/10.1145/3236766

Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. 2006. Sage: Hybrid Checking for Flexible Specifications. *Scheme and Functional Programming Workshop* 6 (2006), 93–104. http://scheme2006.cs.uchicago.edu/06-freund.pdf

David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-Efficient Gradual Typing. *Higher Order Symbol. Comput.* 23, 2 (2010), 167–189. https://doi.org/10.1007/s10990-011-9066-z

Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G Siek. 2018. An Efficient Compiler for the Gradually Typed Lambda Calculus. *Scheme and Functional Programming Workshop* 18 (2018), 19 pages. http://www.schemeworkshop.org/2018/Kuhlenschmidt_Almahallawi_Siek.pdf

Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. 2019. Toward Efficient Gradual Typing for Structural Types via Coercions. In *PLDI*. ACM, New York, NY, USA, 517–532. https://doi.org/10.1145/3314221.3314627

Barbara H. Liskov and Jeannette M. Wing. 1994. A Behavioral Notion of Subtyping. *ACM Trans. Program. Lang. Syst.* 16, 6 (Nov. 1994), 1811–1841. https://doi.org/10.1145/197320.197383

Jacob Matthews and Robert Bruce Findler. 2007. Operational Semantics for Multi-Language Programs. In *POPL*. Association for Computing Machinery, New York, NY, USA, 3–10. https://doi.org/10.1145/1190216.1190220

Cameron Moy, Phúc C. Nguyễn, Sam Tobin-Hochstadt, and David Van Horn. 2021. Corpse Reviver: Sound and Efficient Gradual Typing via Contract Verification. *PACMPL* 5, POPL, Article 53 (2021), 28 pages. https://doi.org/10.1145/3434334

Fabian Muehlboeck and Ross Tate. 2017. Sound Gradual Typing is Nominally Alive and Well. *PACMPL* 1, OOPSLA, Article 56 (2017), 30 pages. https://doi.org/10.1145/3133880

Fabian Muehlboeck and Ross Tate. 2021. Transitioning from Structural to Nominal Code with Efficient Gradual Typing: Artifact. https://doi.org/10.5281/zenodo.5518181

Max S. New, Daniel R. Licata, and Amal Ahmed. 2019. Gradual Type Theory. *PACMPL* 3, POPL, Article 15 (2019), 31 pages. https://doi.org/10.1145/3290328

Gregor Richards, Ellen Arteca, and Alexi Turcotte. 2017. The VM Already Knew That: Leveraging Compile-Time Knowledge to Optimize Gradual Typing. *PACMPL* 1, OOPSLA, Article 55 (2017), 27 pages. https://doi.org/10.1145/3133879

Richard Roberts, Stefan Marr, Michael Homer, and James Noble. 2019. Transient Typechecks Are (Almost) Free. In *ECOOP*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, Article 5, 28 pages. https://doi.org/10.4230/LIPIcs.ECOOP.2019.5

Jeremy Siek and Walid Taha. 2007. Gradual Typing for Objects. In *ECOOP*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2–27. https://doi.org/10.1007/978-3-540-73589-2_2

Jeremy G Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. *Scheme and Functional Programming Workshop* 6 (2006), 81–92. http://scheme2006.cs.uchicago.edu/13-siek.pdf

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015a. Refined Criteria for Gradual Typing. In *SNAPL*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 274–293. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. 2015b. Monotonic References for Efficient Gradual Typing. In *ESOP*. Springer Berlin Heidelberg, Berlin, Heidelberg, 432–456. https://doi.org/10.1007/978-3-662-46669-8_18

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is Sound Gradual Typing Dead?. In *POPL*. ACM, New York, NY, USA, 456–468. https://doi.org/10.1145/2837614.2837630

Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: From Scripts to Programs. In *OOPSLA*. ACM, New York, NY, USA, 964–974. https://doi.org/10.1145/1176617.1176755

Sam Tobin-Hochstadt, Matthias Felleisen, Robert Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. 2017. Migratory Typing: Ten Years Later. In *SNAPL*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, Article 17, 17 pages. https://doi.org/10.4230/LIPIcs.SNAPL.2017.17

Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and Evaluation of Gradual Typing for Python. In *DLS*. ACM, New York, NY, USA, 45–56. https://doi.org/10.1145/2661088.2661101

Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. 2017. Big Types in Little Runtime: Open-World Soundness and Collaborative Blame for Gradual Type Systems. In *POPL*. ACM, New York, NY, USA, 762–774. https://doi.org/10.1145/3009837.3009849

Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. 2010. Integrating Typed and Untyped Code in a Scripting Language. In *POPL*. ACM, New York, NY, USA, 377–388. https://doi.org/10.1145/1706299.1706343

A FULL FORMALIZATION

This appendix contains the full set of rules used in our formalization, in roughly the same sequence as they are presented in the paper, together with a detailed explanation of all the parts.

Table 1. Index of Definitions

	Signature	Name	Defined in
Grammar	C	Class Name	Figure A.1
	I	Interface Name	Figure A.1
	f	Field Name	Figure A.1
	x	Variable Name	Figure A.1
	$\mathcal{P}$	Program	Figure A.1
	$\mathcal{H}$	(Inheritance) Hierarchy	Figure A.1
	$\mathcal{S}$	Signature	Figure A.1
	$\mathcal{I}$	Implementation	Figure A.1
	N	Nominal Type	Figure A.1
	τ	Type	Figure A.1
	$\vec{\tau}$	Type List	Figure A.1
	Γ	Type Context	Figure A.1
	m	Method Name	Figure A.1
	s	Method Signature	Figure A.1
	b	Method Body	Figure A.1
	e	Expression	Figure A.1
Relations	$\vdash \tau \sim \tau$	Consistency	Figure A.2
	$\vdash \tau \sqsubseteq \tau$	Type Precision	Figure A.2
	$\mathcal{H} \vdash \tau \leq \tau$	Nominal Subtyping	Figure A.3
	$\mathcal{H} \vdash \tau \blacktriangleleft \tau$	Pessimistic Subtyping	Figure A.4
	$\mathcal{H} \vdash \tau \triangleleft \tau$	Optimistic Subtyping	Figure A.4
Typing	$\vdash \mathcal{H}$	Interface-Hierarchy Typing	Figure A.5
	$\mathcal{H} \vdash \mathcal{S} : \mathcal{H}$	Signature Typing	Figure A.5
	$\mathcal{H} \vdash s$ `extends` s	Interface Method Inheritance	Figure A.5
	$\mathcal{H} \vdash s$ `implements` s	Class Method Inheritance	Figure A.5
	$\mathcal{H} \mid \mathcal{S} \vdash \mathcal{I} : \mathcal{S}$	Implementation Typing	Figure A.5
	$\vdash \Gamma : \vec{\tau}$	Type-Context Typing	Figure A.5
	$\vdash b : s$	Method-Body Signature	Figure A.5
	$\vdash \mathcal{P}$	Program Checking	Figure A.5
	$\mathcal{H} \vdash N$	Nominal-Type Validity	Figure A.6
	$\mathcal{H} \vdash \tau$	Type Validity	Figure A.6
	$\mathcal{H} \vdash \vec{\tau}$	Type-List Validity	Figure A.6
	$\mathcal{H} \vdash \Gamma$	Type-Context Validity	Figure A.6
	$\mathcal{H} \vdash s$	Method-Signature Validity	Figure A.6
	$\Gamma \vdash x : \tau$	Variable Lookup	Figure A.7
	$\mathcal{S} \vdash N.ms$	Nominal Method Lookup	Figure A.7
	$\mathcal{S} \vdash \tau.ms$	Method Lookup	Figure A.7
	$\mathcal{S} \vdash C.f : \tau$	Class Field Lookup	Figure A.7
	$\mathcal{S} \vdash \tau.f : \tau$	Field Lookup	Figure A.7
	$\mathcal{S} \vdash C(\vec{\tau})$	Constructor Lookup	Figure A.7
	$\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \downarrow \tau$	Expression Type-Checking	Figure A.8
	$\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \uparrow \tau$	Expression Type-Synthesis	Figure A.8
	$\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash b$	Method-Body Typing	Figure A.8
Precision	χ	Extensibility	Figure A.10
	$\vdash \mathcal{H} \sqsubseteq \mathcal{H}$	Inheritance-Hierarchy Precision	Figure A.9
	$\vdash \mathcal{S} \sqsubseteq \mathcal{S}$	Signature Precision	Figure A.9
	$\vdash s \sqsubseteq s$	Method-Signature Precision	Figure A.9
	$\vdash \vec{\tau} \sqsubseteq \vec{\tau}$	Type-List Precision	Figure A.9
	$\mathcal{P} \sqsubseteq \mathcal{H} \vdash \mathcal{I} \sqsubseteq \mathcal{I}$	Implementation Precision	Figure A.9
	$\vdash \mathcal{P} \sqsubseteq \mathcal{P}$	Program Precision	Figure A.9
	$\mathcal{P} \sqsubseteq \mathcal{H} \vdash e \sqsubseteq e$	Expression Precision	Figure A.10
	$\mathcal{P} \sqsubseteq \mathcal{H} \vdash x := \{f := e; \ldots \mid mb; \ldots\} \sqsubseteq e : \chi$	Record Precision	Figure A.10
	$\mathcal{P} \sqsubseteq \mathcal{H} \vdash b \sqsubseteq b$	Method-Body Precision	Figure A.10
	$\mathcal{H} \sqsubseteq \mathcal{H} \vdash \tau :> \tau$	Subsumptive Supertyping	Figure A.11
	$\mathcal{H} \sqsubseteq \mathcal{H} \vdash \Gamma :> \Gamma$	Type-Context Subsumptive Supertyping	Figure A.11
Lowered Grammar	$\check{\mathcal{P}}$	Lowered Program	Figure A.12
	$\check{\mathcal{I}}$	Lowered Implementation	Figure A.12
	$\check{b}$	Lowered Method Body	Figure A.12
	$\check{\Gamma}$	Lowered Type Context	Figure A.12
	$\check{e}$	Lowered Expression	Figure A.12
	ℓ	Location	Figure A.12

	Signature	Name	Defined in
	v	Value	Figure A.12
	γ	Guard Mode	Figure A.12
	δ	Dispatch Mode	Figure A.12
	H	Heap	Figure A.13
	h	Heap Value	Figure A.13
	μ	Mark	Figure A.13
	ι	Imposition	Figure A.13
Lowered Typing	Σ	Heap Type	Figure A.15
	σ	Heap-Value Type	Figure A.15
	$\mathcal{H} \mid \mathcal{S} \vdash \check{\mathcal{I}} : \mathcal{S}$	Lowered-Implementation Typing	Figure A.14
	$\vdash \check{\Gamma} : \vec{\tau} \rightsquigarrow \Gamma$	Type-Context Construction	Figure A.14
	$\vdash \check{b} : s$	Lowered-Method-Body Signature	Figure A.14
	$\vdash \check{\mathcal{P}}$	Lowered-Program Typing	Figure A.14
	$\mathcal{H} \mid \mathcal{S} \vdash H : \Sigma$	Heap Typing	Figure A.15
	$\mathcal{H} \mid \mathcal{S} \mid \Sigma \vdash h : \sigma$	Heap-Value Typing	Figure A.15
	$\mathcal{H} \vdash s \lhd s$	Optimistic Method-Signature Subtyping	Figure A.15
	$\mathcal{H} \vdash \check{b} \lhd s$	Optimistic Lowered-Method-Body Typing	Figure A.15
	$\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \check{e} : \tau$	Lowered-Expression Typing	Figure A.16
	$\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \check{b} : s$	Lowered-Method-Body Typing	Figure A.16
Lowering	$\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \downarrow \tau \rightsquigarrow \check{e}$	Checked-Expression Lowering	Figure A.17
	$\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \rightsquigarrow \check{e}$	Synthesized-Expression Lowering	Figure A.17
	$\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash b \rightsquigarrow \check{b}$	Method-Body Lowering	Figure A.17
	$\vdash \Gamma \rightsquigarrow \check{\Gamma}$	Type-Context Lowering	Figure A.17
	$\mathcal{H} \mid \mathcal{S} \vdash \mathcal{I} \rightsquigarrow \check{\mathcal{I}}$	Implementation Lowering	Figure A.17
	$\vdash \mathcal{P} \rightsquigarrow \check{\mathcal{P}}$	Program Lowering	Figure A.17
Reduction and Semantics	r	Redex	Figure A.20
	E	Evaluation Context	Figure A.20
	ε	Potentially Erroneous Redex	Figure A.21
	o	Observation	Figure A.22
	$\check{\mathcal{P}} \mid H \vdash \ell.f \mapsto v$	Field Access	Figure A.18
	$\check{\mathcal{P}} \mid H \rightarrow H \vdash \ell.f := v$	Field Access	Figure A.18
	$\check{\mathcal{P}} \mid H \vdash v.m \rightsquigarrow_\gamma b$	Direct Method-Body Lookup	Figure A.18
	$\check{\mathcal{P}} \mid H \rightarrow H \vdash v.m \rightsquigarrow_\gamma b$	Indirect Method-Body Lookup	Figure A.18
	$\mathcal{H} \mid H \vdash \ell \mapsto \iota$	Imposition Fetch	Figure A.18
	$\check{\mathcal{P}} \mid H \rightarrow H \vdash v : \vec{\tau}$	Cast Reduction	Figure A.19
	$\check{\mathcal{P}} \mid H \vdash v : \tau$	Cast Checking	Figure A.19
	$\mathcal{H} \vdash \iota \blacktriangleleft \tau$	Pessimistic Imposition Subtyping	Figure A.19
	$\mathcal{S} \vdash (\iota).m : \vec{\tau}$	Imposed Method Return Types	Figure A.19
	$\check{\mathcal{P}} \vdash H \mid \check{e} \xrightarrow{H} H \mid \check{e}$	Expression Reduction	Figure A.20
	$\check{\mathcal{P}} \vdash H \mid r \xrightarrow{H} H \mid \check{e}$	Redex Reduction	Figure A.20
	$\check{\mathcal{P}} \vdash H \mid \check{e} \rightarrow H \mid \check{e}$	Reduction	Figure A.20
	$\check{\mathcal{P}} \vdash H \mid \check{e} \rightarrow \mathbf{error}$	Error Reduction	Figure A.21
	$\vdash \mathcal{P} \twoheadrightarrow o$	Observation Semantics	Figure A.22
Lowered Precision	η	Heap Correspondence	Figure A.24
	$\check{\mathcal{P}} \sqsubseteq \mathcal{H} \vdash \check{\mathcal{I}} \sqsubseteq \check{\mathcal{I}}$	Lowered-Implementation Precision	Figure A.23
	$\vdash \check{\mathcal{P}} \sqsubseteq \check{\mathcal{P}}$	Lowered-Program Precision	Figure A.23
	$\check{\mathcal{P}} \sqsubseteq \mathcal{H} \vdash H \mid \check{e} \sqsubseteq H \mid \check{e}$	Program-State Precision	Figure A.24
	$\check{\mathcal{P}} \sqsubseteq \mathcal{H} \mid \eta \vdash H \sqsubseteq H : \eta$	Heap Precision	Figure A.24
	$\check{\mathcal{P}} \sqsubseteq \mathcal{H} \mid \eta \vdash \ell \mapsto \{f \mapsto_\mu \check{e}; \ldots \mid m\check{b}; \ldots\}^\chi_\iota \sqsubseteq H \mid \ell \mapsto h : \eta$	Heap-Record Precision	Figure A.24
	$\eta \vdash v \sqsubseteq v$	Value Precision	Figure A.25
	$\check{\mathcal{P}} \sqsubseteq \mathcal{H} \mid \eta \vdash \check{e} \sqsubseteq H \mid \check{e}$	Lowered-Expression Precision	Figure A.25
	$\vdash x := \{f := \check{e}; \ldots \mid m\check{b}; \ldots\} \rightsquigarrow H \mid \check{e} : \chi$	Record Lowering	Figure A.25
	$\check{\mathcal{P}} \sqsubseteq \mathcal{H} \mid \eta \vdash \check{b} \sqsubseteq H \mid \check{b}$	Lowered-Method-Body Precision	Figure A.25
	$\check{\mathcal{P}} \sqsubseteq \mathcal{H} \vdash \delta \sqsubseteq \delta$	Dispatch-Mode Precision	Figure A.25
	$\check{\mathcal{P}} \sqsubseteq \mathcal{H} \vdash \iota \sqsubseteq \iota$	Imposition Precision	Figure A.25
	$\vdash \gamma \sqsubseteq \gamma$	Guard-Mode Precision	Figure A.25
	$\check{\mathcal{P}} \vdash H \mid \check{e} \xrightarrow{+} H \mid \check{e}$	Multi-Step Reduction	Figure A.26
	$\check{\mathcal{P}} \sqsubseteq \check{\mathcal{P}} \vdash H \mid \check{e} \xrightarrow{*} \bullet \sqsubseteq H \mid \check{e}$	Always-Eventually Refines	Figure A.26
	$\check{\mathcal{P}} \sqsubseteq \check{\mathcal{P}} \vdash H \mid \check{e} \xrightarrow{*} H \mid \check{e} \sqsubseteq \bullet$	Always-Eventually Relaxes	Figure A.26
	$\check{\mathcal{P}} \vdash H \mid \check{e} \xrightarrow{*} \varnothing$	Always-Eventually Sticks	Figure A.26

Class Name C	Interface Name I	Field Name f	Variable Name x

Program	$\mathcal{P} ::= \langle \mathcal{H} \mid \mathcal{S} \mid \mathcal{I} \mid e \rangle$	(Notation:	$\mathcal{P} = \langle \mathcal{H}_\mathcal{P} \mid \mathcal{S}_\mathcal{P} \mid \mathcal{I}_\mathcal{P} \mid e_\mathcal{P} \rangle$)
Hierarchy	$\mathcal{H} ::= \varnothing \mid \mathcal{H}; N \leq I, \ldots$	Nominal Type	$N ::= C \mid I$
Signature	$\mathcal{S} ::= \varnothing \mid \mathcal{S}; N\{ms; \ldots\} \mid \mathcal{S}; C(\vec{\tau})\{f : \tau; \ldots\}$		
Implementation	$\mathcal{I} ::= \varnothing \mid \mathcal{I}; x : C(\Gamma)\{f := e; \ldots \mid mb; \ldots\}$		
Type	$\tau ::= N \mid \mathbb{B} \mid \mathbf{dyn}$	Method Name	$m ::= f \mid \lambda$
Type List	$\vec{\tau} ::= \varnothing \mid \vec{\tau}, \tau$	Method Signature	$s ::= (\vec{\tau}) : \tau$
Type Context	$\Gamma ::= \varnothing \mid \Gamma, x : \tau$	Method Body	$b ::= (\Gamma) \mapsto e : \tau$
Expression	$e ::= x \mid \texttt{let}\ \langle \Gamma \rangle := \langle e, \ldots \rangle\ \texttt{in}\ e \mid \texttt{false} \mid \texttt{true} \mid e == e$		
	$\mid\ e.f \mid e.f := e \mid e(e, \ldots)$		
	$\mid\ \texttt{new}\ C(e, \ldots) \mid \texttt{new}\ \lambda\langle b \rangle \mid \texttt{new}\ x := \{f := e; \ldots \mid mb; \ldots\}$		

Fig. A.1. Grammar (repeat of Figure 2)

A.1 Grammar

A MonNom program specifies a nominal hierarchy of classes and interfaces. Although the implementation supports a syntax wherein each class/interface is specified in its entirety in its own file, the calculus formalizes MonNom by splitting the nominal hierarchy into three parts: the (inheritance) hierarchy $\mathcal{H}$, the siganture $\mathcal{S}$, and the implementation $\mathcal{I}$. This breaks up the recursion inherent in the surface syntax, making many of our theorems easier. For example, although expression typing depends on the nominal hierarchy, it only depends on the inheritance hierarchy and the signatures; separating the implementation out makes it easier, then, to prove the static and dynamic gradual guarantees.

$$\boxed{\vdash \tau \sim \tau} \qquad \frac{}{\vdash \tau \sim \tau} \qquad \frac{}{\vdash \tau \sim \mathbf{dyn}} \qquad \frac{}{\vdash \mathbf{dyn} \sim \tau} \qquad \boxed{\vdash \tau \sqsubseteq \tau} \qquad \frac{}{\vdash \tau \sqsubseteq \tau} \qquad \frac{}{\vdash \tau \sqsubseteq \mathbf{dyn}}$$

Fig. A.2. Consistency ($\sim$) and Precision ($\sqsubseteq$) (repeat of Figure 4)

$$\boxed{\mathcal{H} \vdash \tau \leq \tau} \qquad \frac{}{\mathcal{H} \vdash \tau \leq \tau} \qquad \frac{N \leq I_1, \ldots \in \mathcal{H} \qquad \mathcal{H} \vdash I_i \leq \tau}{\mathcal{H} \vdash N \leq \tau}$$

Fig. A.3. Nominal Subtyping (repeat of Figure 3)

$$\boxed{\mathcal{H} \vdash \tau \lhd \tau} \quad \frac{\mathcal{H} \vdash \tau \leq \tau' \qquad \vdash \tau' \sim \tau''}{\mathcal{H} \vdash \tau \lhd \tau''} \qquad \boxed{\mathcal{H} \vdash \tau \blacktriangleleft \tau} \quad \frac{\mathcal{H} \vdash \tau \leq \tau' \qquad \vdash \tau' \sqsubseteq \tau''}{\mathcal{H} \vdash \tau \blacktriangleleft \tau''}$$

Fig. A.4. Optimistic ($\lhd$) and Pessimistic ($\blacktriangleleft$) Subtyping (repeat of Figure 5)

A.2 Typing

A.2.1 Consistency, Precision, and Subtyping. The combination of gradual typing and nominal subtyping in one language means that there are many interesting relationships between types. That is, one can adjust whether inheritance is used and how dynamism is introduced or eliminated, and each of these combinations has some interesting utility.

If we ignore inheritance and just focus on dynamism, then we get the consistency ($\sim$) and precision ($\sqsubseteq$) relations. The consistency relation holds when there is some way to instantiate occurrences of **dyn** in both types in order to make the two types identical. The precision relation holds when there is some way to instantiate occurrences of **dyn** in only the right-hand type in order to make the two types identical. That is, $\tau \sqsubseteq \tau'$ holds when τ' is "more dynamic" than τ.

If we instead ignore dynamism and focus on inheritance, then we get the nominal subtyping relation ($\leq$). The nominal subtyping relation holds when either the two types are the same or the class or interface type on the left can be repeatedly replaced with some inherited interface type to arrive at exactly the type on the right.

Lastly, we can combine both dynamism and inheritance. Pessimistic subtyping ($\blacktriangleleft$) has the property that any value satisfying the contract of the left-hand type is guaranteed to satisfy the contract of the right-hand type. Optimistic subtyping ($\lhd$) conceptually (although—with generics—not precisely) holds whenever there is some way to instantiate occurrences of **dyn** in both types in order to make the left inherit the right.

A.2.2 Typing the Nominal Hierarchy. The typing rules for the nominal hierarchy (and programs) are shown in Figure A.5. They make use of judgements for validity and lookup provided in Figure A.6 and Figure A.7, respectively.

An inheritance hierarchy is well-formed if inheritance is well-founded and every class or interface is declared it most once.

A signature is well-formed if

- all types in it are valid according to the inheritance hierarchy,
- the method signatures are provided for every interface and class in the inheritance hierarchy,
- the constructor type and the field types are provided for every class in the inheritance hierarchy,
- there are no overlapping field/method names within any interface or class, and

$$\boxed{\vdash \mathcal{H}} \qquad \frac{}{\vdash \varnothing} \qquad \frac{\vdash \mathcal{H} \qquad \neg\, \mathcal{H} \vdash N \qquad \forall i.\ \mathcal{H} \vdash I_i}{\vdash \mathcal{H}; N \leq I_1, \ldots}$$

$$\boxed{\mathcal{H} \vdash \mathcal{S} : \mathcal{H}} \qquad \frac{}{\mathcal{H}_0 \vdash \varnothing : \varnothing}$$

$$\frac{\begin{array}{c}\mathcal{H}_0 \vdash \mathcal{S} : \mathcal{H} \qquad \forall i, i'.\ m_i = m_{i'} \implies i = i' \qquad \forall i.\ \mathcal{H}_0 \vdash s_i \\ \forall i, i', s.\ \mathcal{S} \vdash I_i.m_{i'}s \implies \mathcal{H}_0 \vdash s_{i'}\ \texttt{extends}\ s \qquad \forall i, m, s.\ \mathcal{S} \vdash I_i.ms \implies \exists i'.\ m_{i'} = m\end{array}}{\mathcal{H}_0 \vdash \mathcal{S}; I\{m_1 s_1; \ldots\} : \mathcal{H}; I \leq I_1, \ldots}$$

$$\frac{\begin{array}{c}\mathcal{H}_0 \vdash \mathcal{S} : \mathcal{H} \qquad \forall i, i'.\ m_i = m_{i'} \implies i = i' \qquad \forall i.\ \mathcal{H}_0 \vdash s_i \\ \forall i, i', s.\ \mathcal{S} \vdash I_i.m_{i'}s \implies \mathcal{H}_0 \vdash s_{i'}\ \texttt{implements}\ s \qquad \forall i, m, s.\ \mathcal{S} \vdash I_i.ms \implies \exists i'.\ m_{i'} = m \\ \nexists i, i'.\ m_i = f_{i'} \qquad \mathcal{H}_0 \vdash \vec{\tau} \qquad \forall i, i'.\ f_i = f_{i'} \implies i = i' \qquad \forall i.\ \mathcal{H}_0 \vdash \tau_i\end{array}}{\mathcal{H}_0 \vdash \mathcal{S}; C\{m_1 s_1; \ldots\}; C(\vec{\tau})\{f_1 : \tau_1; \ldots\} : \mathcal{H}; C \leq I_1, \ldots}$$

$$\boxed{\mathcal{H} \vdash s\ \texttt{extends}\ s} \qquad \frac{\forall i.\ \mathcal{H} \vdash \tau_i' \leq \tau_i \qquad \mathcal{H} \vdash \tau \leq \tau'}{\mathcal{H} \vdash (\tau_1, \ldots) : \tau\ \texttt{extends}\ (\tau_1', \ldots) : \tau'}$$

$$\boxed{\mathcal{H} \vdash s\ \texttt{implements}\ s} \qquad \frac{\forall i.\ \mathcal{H} \vdash \tau_i' \blacktriangleleft \tau_i \qquad \mathcal{H} \vdash \tau \leq \tau'}{\mathcal{H} \vdash (\tau_1, \ldots) : \tau\ \texttt{implements}\ (\tau_1', \ldots) : \tau'}$$

$$\boxed{\mathcal{H} \mid \mathcal{S} \vdash \mathcal{I} : \mathcal{S}} \qquad \frac{}{\mathcal{H}_0 \mid \mathcal{S}_0 \vdash \varnothing : \varnothing} \qquad \frac{\mathcal{H}_0 \mid \mathcal{S}_0 \vdash \mathcal{I} : \mathcal{S}}{\mathcal{H}_0 \mid \mathcal{S}_0 \vdash \mathcal{I} : \mathcal{S}; I\{\ldots\}}$$

$$\frac{\begin{array}{c}\mathcal{H}_0 \mid \mathcal{S}_0 \vdash \mathcal{I} : \mathcal{S} \\ \vdash \Gamma : \vec{\tau} \qquad \forall i.\ \mathcal{H}_0 \mid \mathcal{S}_0 \mid \Gamma \vdash e_i \downarrow \tau_i \qquad \forall i.\ \vdash b_i : s_i \qquad \forall i.\ \mathcal{H}_0 \mid \mathcal{S}_0 \mid \Gamma, x : C \vdash b_i\end{array}}{\mathcal{H}_0 \mid \mathcal{S}_0 \vdash \mathcal{I}; x : C(\Gamma)\{f_1 \coloneqq e_1; \ldots \mid m_1 b_1; \ldots\} : \mathcal{S}; C\{m_1 s_1; \ldots\}; C(\vec{\tau})\{f_1 : \tau_1; \ldots\}}$$

$$\boxed{\vdash \Gamma : \vec{\tau}} \qquad \frac{}{\vdash \varnothing : \varnothing} \qquad \frac{\vdash \Gamma : \vec{\tau}}{\vdash \Gamma, x : \tau : \vec{\tau}, \tau} \qquad \boxed{\vdash b : s} \qquad \frac{\vdash \Gamma : \vec{\tau}}{\vdash (\Gamma) \mapsto e : \tau : (\vec{\tau}) : \tau}$$

$$\boxed{\vdash \mathcal{P}} \qquad \frac{\vdash \mathcal{H} \qquad \mathcal{H} \vdash \mathcal{S} : \mathcal{H} \qquad \mathcal{H} \mid \mathcal{S} \vdash \mathcal{I} : \mathcal{S} \qquad \mathcal{H} \mid \mathcal{S} \mid \varnothing \vdash e \downarrow \mathbb{B}}{\vdash \langle \mathcal{H} \mid \mathcal{S} \mid \mathcal{I} \mid e \rangle}$$

Fig. A.5. Program Typing (extension of Figure 7)

$$\boxed{\mathcal{H} \vdash N \qquad \mathcal{H} \vdash \tau} \qquad \frac{}{\mathcal{H}; N \leq \ldots \vdash N} \qquad \frac{\mathcal{H} \vdash N}{\mathcal{H}; N' \leq \ldots \vdash N} \qquad \frac{}{\mathcal{H} \vdash \mathbb{B}} \qquad \frac{}{\mathcal{H} \vdash \mathbf{dyn}}$$

$$\boxed{\mathcal{H} \vdash \vec{\tau}} \qquad \frac{}{\mathcal{H} \vdash \varnothing} \qquad \frac{\mathcal{H} \vdash \vec{\tau} \qquad \mathcal{H} \vdash \tau}{\mathcal{H} \vdash \vec{\tau}, \tau}$$

$$\boxed{\mathcal{H} \vdash \Gamma} \qquad \frac{}{\mathcal{H} \vdash \varnothing} \qquad \frac{\mathcal{H} \vdash \Gamma \qquad \nexists \tau'.\ \Gamma \vdash x : \tau' \qquad \mathcal{H} \vdash \tau}{\mathcal{H} \vdash \Gamma, x : \tau}$$

$$\boxed{\mathcal{H} \vdash s} \qquad \frac{\mathcal{H} \vdash \vec{\tau} \qquad \mathcal{H} \vdash \tau}{\mathcal{H} \vdash (\vec{\tau}) : \tau}$$

Fig. A.6. Type Validation

- every interface and class provides at least the methods that any inherited interface provides, and the corresponding signature of each such method either `extends` or `implements` that of the inherited method signatures.

$$\boxed{\Gamma \vdash x : \tau}$$

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma \vdash x : \tau}{\Gamma, x' : \tau' \vdash x : \tau}$$

$$\boxed{\mathcal{S} \vdash N.ms \qquad \mathcal{S} \vdash \tau.ms}$$

$$\frac{}{\mathcal{S}; N\{m_1 s_1; \ldots\} \vdash N.m_i s_i} \qquad \frac{\mathcal{S} \vdash N.ms}{\mathcal{S}; N'\{\ldots\} \vdash N.ms} \qquad \frac{\mathcal{S} \vdash N.ms}{\mathcal{S}; C(\ldots)\{\ldots\} \vdash N.ms} \qquad \frac{}{\mathcal{S} \vdash \mathbf{dyn}.m(\mathbf{dyn}, \ldots) : \mathbf{dyn}}$$

$$\boxed{\mathcal{S} \vdash C.f : \tau \qquad \mathcal{S} \vdash \tau.f : \tau}$$

$$\frac{}{\mathcal{S}; C(\ldots)\{f_1 : \tau_1; \ldots\} \vdash C.f_i : \tau_i} \qquad \frac{\mathcal{S} \vdash C.f : \tau}{\mathcal{S}; N\{\ldots\} \vdash C.f : \tau} \qquad \frac{\mathcal{S} \vdash C.f : \tau}{\mathcal{S}; C'(\ldots)\{\ldots\} \vdash C.f : \tau} \qquad \frac{}{\mathcal{S} \vdash \mathbf{dyn}.f : \mathbf{dyn}}$$

$$\boxed{\mathcal{S} \vdash C(\vec{\tau})}$$

$$\frac{}{\mathcal{S}; C(\vec{\tau})\{\ldots\} \vdash C(\vec{\tau})} \qquad \frac{\mathcal{S} \vdash C(\vec{\tau})}{\mathcal{S}; N\{\ldots\} \vdash C(\vec{\tau})} \qquad \frac{\mathcal{S} \vdash C(\vec{\tau})}{\mathcal{S}; C'(\ldots)\{\ldots\} \vdash C(\vec{\tau})}$$

Fig. A.7. Type Lookup

An implementation is well-formed if field initializers and method bodies are provided for every class, with types and signatures in correspondence with the class signature.

We expect most of the above to be straightforward with one exception: method inheritance.

Method Inheritance. When inheriting a method from a superinterface, the signature of that method provided by the sub-class/interface must be compatible with that provided by the superinterface. One nuanced contribution of this work is determining what the compatibility should be, so here we explain why `extends` and `interface` are defined as they are.

The easier case to explain is compatibility of return types. For one, for safety it is important that any value returned by the sub-class/interface be valid value to return for the superinterface. This requires the sub-return-type to be at least a pessimistic supertype of the super-return-type. But we also want MonNom to satisfy static subsumption (and be easy to type-check). As such, using the sub-class/interface to determine the method's signature should return a type that type-checks in at least all the contexts where using the superinterface to determine the method's signature would type-check. Pessimistic subtyping does not guarantee that: C is a pessimistic subtype of **dyn** but type-checks in far fewer contexts than **dyn** (per the static gradual guarantee). In MonNom, nominal subtyping is the strongest relation that satisfies the safety and the subsumption properties. Furthermore, it has the added benefit that there is no need change representations (e.g. boxed versus unboxed) between nominal supertypes, making for a simpler and more efficient implementation.

For method parameter types, we still have the safety requirement, but no longer the subsumption requirement. This is why `implements` requires only pessimistic supertyping (note the contravariance). That also enables classes to have all **dyn** parameters so that MonNom supports classes with untyped implementations of typed methods (although the return type still has to be a nominal subtype of the interface return types, the method implementation only needs to check against it optimistically). But `implements` is only used when *classes* inherit methods.

When *interfaces* inherit methods, they must use the more restrictive `extends` judgement. The reason is that one can cast structural objects to interfaces, and when doing so the method bodies

are checked to optimistically satisfy the method signatures. (Technically this is only done for the λ-method, but we felt that it would be better if requirements for method inheritance were consistent across all method names.) For reasons we will illustrate shortly, in order to ensure dynamic subsumption it is important that if a cast to a subinterface succeeds then so must a cast to any superinterface, and as such if a method implementation provided by a structural object optimistically satisfies a subinterface's method signature then it must also optimistically satisfy the superinterface's method signature (if there is any). Once again, pessimistic supertyping does not guarantee this: the method body might require a particular class as an input, and requiring just **dyn** as input in the subinterface's signature would be optimistically compatible, whereas requiring a class as input in the superinterface's signature would not be optimistically compatible if the two classes were unrelated, despite **dyn** being a pessimistic supertype of any class. And, once again, nominal supertyping does satisfy both this and the safety property (as does nominal subtyping for return types). This is why interfaces must use `extends` rather than `implements` in order to satisfy dynamic subsumption. (Note that statically typed object-oriented languages generally already require nominal supertyping/subtyping in method inheritance, so although this is more restrictive than one might expect for gradual typing, it is no more restrictive than languages without gradual typing.)

Now, in the above, we stated that dynamic subsumption requires that if a cast to a subinterface succeeds then the cast of the same value to any superinterface must succeed as well. To see why, consider the case where that a method's receiver's type might require some interface as input to the method. By contravariance of method inheritance, some subtype of that method's receiver's type might require only some superinterface as input, which would cause lowering to insert a different cast. Dynamic subsumption requires that if the former method invocation succeeds then so must the latter, and as such a successful cast to the (sub-)interface in the former must imply a successful cast to the superinterface in the latter. This illustrates the subtle interactions between subtyping and gradual typing that we have managed to address in MonNom.

A.2.3 Typing Expressions. Figure A.8 shows the typing rules for expressions. The key detail to note is where type-checking versus type-synthesis is used. In particular, it is critical that field access, field assignment, and method invocation use type-synthesis rather than type-checking for the receiver's type. If they used type-checking, the receiver's type could always be simply **dyn**, in which case all field accesses, field assignments, and method invocations would type-check (supposing their constituents were at least valid regardless of context). That said, adding subtyping rules with respect to nominal subtyping to both of these judgements would be a conservative extension to type-checking due to the considerations in Section A.2.2, which contributes to MonNom's static subsumption property.

$\boxed{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \downarrow \tau}$

$$\frac{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \uparrow \tau' \qquad \mathcal{H} \vdash \tau' \lhd \tau}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \downarrow \tau}$$

$\boxed{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \uparrow \tau}$

$$\frac{\Gamma \vdash x : \tau}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash x \uparrow \tau} \qquad \frac{\forall i.\ \mathcal{H} \vdash \tau_i \qquad \forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e_i \downarrow \tau_i \qquad \mathcal{H} \mid \mathcal{S} \mid \Gamma, x_1 : \tau_1, \ldots \vdash e \uparrow \tau}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash \texttt{let}\ \langle x_1 : \tau_1, \ldots \rangle := \langle e_1, \ldots \rangle\ \texttt{in}\ e \uparrow \tau}$$

$$\frac{}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash \texttt{false} \uparrow \mathbb{B}} \qquad \frac{}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash \texttt{true} \uparrow \mathbb{B}}$$

$$\frac{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e_1 \uparrow \tau_1 \quad \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e_2 \uparrow \tau_2}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e_1 == e_2 \uparrow \mathbb{B}} \qquad \frac{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \uparrow \tau \quad \mathcal{S} \vdash \tau.f : \tau_f}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e.f \uparrow \tau_f} \qquad \frac{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \uparrow \tau \quad \mathcal{S} \vdash \tau.f : \tau_f \quad \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e_f \downarrow \tau_f}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e.f := e_f \uparrow \tau}$$

$$\frac{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \uparrow \tau \quad \mathcal{S} \vdash \tau.\lambda(\tau_1, \ldots) : \tau_\lambda \quad \forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e_i \downarrow \tau_i}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e(e_1, \ldots) \uparrow \tau_\lambda} \qquad \frac{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \uparrow \tau \quad \mathcal{S} \vdash \tau.f(\tau_1, \ldots) : \tau_f \quad \forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e_i \downarrow \tau_i}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e.f(e_1, \ldots) \uparrow \tau_f}$$

$$\frac{\mathcal{S} \vdash C(\tau_1, \ldots) \qquad \forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e_i \downarrow \tau_i}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash \texttt{new}\ C(e_1, \ldots) \uparrow C} \qquad \frac{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash b}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash \texttt{new}\ \lambda\langle b \rangle \uparrow \mathbf{dyn}}$$

$$\frac{\begin{array}{c}\forall i, i'.\ f_i = f_{i'} \implies i = i' \qquad \forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e_i \uparrow \tau_i \\ \nexists i, i'.\ f_i = m_{i'} \qquad \forall i, i'.\ m_i = m_{i'} \implies i = i' \qquad \forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Gamma, x : \mathbf{dyn} \vdash b_i\end{array}}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash \texttt{new}\ x := \{f_1 := e_1; \ldots \mid m_1 b_1; \ldots\} \uparrow \mathbf{dyn}}$$

$\boxed{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash b}$

$$\frac{\mathcal{H} \vdash \Gamma_b \qquad \mathcal{H} \vdash \tau_b \qquad \mathcal{H} \mid \mathcal{S} \mid \Gamma, \Gamma_b \vdash e_b \downarrow \tau_b}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash (\Gamma_b) \mapsto e_b : \tau_b}$$

Fig. A.8. Expression Typing (extension of Figure 6)

$$\boxed{\vdash \mathcal{H} \sqsubseteq \mathcal{H}}$$

$$\frac{}{\vdash \varnothing \sqsubseteq \varnothing} \qquad \frac{\vdash \mathcal{H} \sqsubseteq \mathcal{H}'}{\vdash \mathcal{H}; N \le I_1, \ldots \sqsubseteq \mathcal{H}'} \qquad \frac{\vdash \mathcal{H} \sqsubseteq \mathcal{H}' \qquad \forall i'. \exists i.\ I_i = I'_{i'} \qquad \forall i, I'.\ \mathcal{H} \vdash I_i \le I' \wedge \mathcal{H}' \vdash I' \implies \exists i'.\ \mathcal{H}' \vdash I'_{i'} \le I'}{\vdash \mathcal{H}; N \le I_1, \ldots \sqsubseteq \mathcal{H}'; N \le I'_1, \ldots}$$

$$\boxed{\vdash \mathcal{S} \sqsubseteq \mathcal{S}}$$

$$\frac{}{\vdash \varnothing \sqsubseteq \varnothing} \qquad \frac{\vdash \mathcal{S} \sqsubseteq \mathcal{S}'}{\vdash \mathcal{S}; N\{\ldots\} \sqsubseteq \mathcal{S}'} \qquad \frac{\vdash \mathcal{S} \sqsubseteq \mathcal{S}'}{\vdash \mathcal{S}; C(\ldots)\{\ldots\} \sqsubseteq \mathcal{S}'}$$

$$\frac{\vdash \mathcal{S} \sqsubseteq \mathcal{S}' \qquad \forall i.\ \vdash s_i \sqsubseteq s'_i}{\vdash \mathcal{S}; N\{m_1 s_1; \ldots\} \sqsubseteq \mathcal{S}'; N\{m_1 s'_1; \ldots\}} \qquad \frac{\vdash \mathcal{S} \sqsubseteq \mathcal{S}' \qquad \vdash \vec{\tau} \sqsubseteq \vec{\tau}' \qquad \forall i.\ \vdash \tau_i \sqsubseteq \tau'_i}{\vdash \mathcal{S}; C(\vec{\tau})\{f_1 : \tau_1; \ldots\} \sqsubseteq \mathcal{S}'; C(\vec{\tau}')\{f_1 : \tau'_1; \ldots\}}$$

$$\boxed{\vdash s \sqsubseteq s}$$

$$\frac{\vdash \vec{\tau} \sqsubseteq \vec{\tau}' \qquad \vdash \tau \sqsubseteq \tau'}{\vdash (\vec{\tau}) : \tau \sqsubseteq (\vec{\tau}') : \tau'}$$

$$\boxed{\vdash \vec{\tau} \sqsubseteq \vec{\tau}}$$

$$\frac{}{\vdash \varnothing \sqsubseteq \varnothing} \qquad \frac{\vdash \vec{\tau} \sqsubseteq \vec{\tau}' \qquad \vdash \tau \sqsubseteq \tau'}{\vdash \vec{\tau}, \tau \sqsubseteq \vec{\tau}', \tau'}$$

$$\boxed{\mathcal{P} \sqsubseteq \mathcal{H} \vdash \mathcal{I} \sqsubseteq \mathcal{I}}$$

$$\frac{}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash \varnothing \sqsubseteq \varnothing} \qquad \frac{\neg\, \mathcal{H}' \vdash C \qquad \mathcal{P} \sqsubseteq \mathcal{H}' \vdash \mathcal{I} \sqsubseteq \mathcal{I}'}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash \mathcal{I}; x : C(\ldots)\{\ldots \mid \ldots\} \sqsubseteq \mathcal{I}'}$$

$$\frac{\forall i.\ \vdash \tau_i \sqsubseteq \tau'_i \qquad \forall i.\ \mathcal{P} \sqsubseteq \mathcal{H}' \vdash e_i \sqsubseteq e'_i \qquad \forall i.\ \mathcal{P} \sqsubseteq \mathcal{H}' \vdash b_i[x \mapsto x', x_1 \mapsto x'_1, \ldots] \sqsubseteq b'_i \qquad \forall i.\ \mathcal{H}' \vdash \tau'_i \qquad \mathcal{P} \sqsubseteq \mathcal{H}' \vdash \mathcal{I} \sqsubseteq \mathcal{I}'}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash \mathcal{I}; x : C(x_1 : \tau_1, \ldots)\{f_1 \coloneqq e_1; \ldots \mid m_1 b_1; \ldots\} \sqsubseteq \mathcal{I}'; x' : C(x'_1 : \tau'_1, \ldots)\{f_1 \coloneqq e'_1; \ldots \mid m_1 b'_1; \ldots\}}$$

$$\boxed{\vdash \mathcal{P} \sqsubseteq \mathcal{P}}$$

$$\frac{\vdash \mathcal{H}_{\mathcal{P}} \sqsubseteq \mathcal{H}_{\mathcal{P}'} \qquad \vdash \mathcal{S}_{\mathcal{P}} \sqsubseteq \mathcal{S}_{\mathcal{P}'} \qquad \mathcal{P} \sqsubseteq \mathcal{H}_{\mathcal{P}'} \vdash \mathcal{I}_{\mathcal{P}} \sqsubseteq \mathcal{I}_{\mathcal{P}'} \qquad \mathcal{P} \sqsubseteq \mathcal{H}_{\mathcal{P}'} \vdash e_{\mathcal{P}} \sqsubseteq e_{\mathcal{P}'}}{\vdash \mathcal{P} \sqsubseteq \mathcal{P}'}$$

Fig. A.9. Program Precision (extension of Figure 8)

A.3 Precision

A.3.1 Program Precision. Now we get to the core contribution of our formalization: the precision relations. We start with program precision, shown in Figure A.9. The only interesting aspects of these rules were already discussed in Section 4.1.

A.3.2 Expression Precision. The rules for expression precision are shown in Figure A.10. Again, the only interesting rules were already discussed in Section 4.2. Note that here we should an additional rule for record precision, where the last rule permits reordering of methods. One implication of the record-precision rules is that they can change the order in which expressions are evaluated: field initializers can be changed to execute after the record is created, and emulating methods with lambda-valued fields can create lambda closures that would not exist in the original program. In all cases, the reordered and new expressions can easily be determined to not affect the execution of the program because they necessarily occur in isolation. For example, whether the record field is initialized before or after the record is created is unobservable because the record location is

Extensibility $\chi ::= \mathsf{fix} \mid \mathsf{ext}$

$\boxed{\mathcal{P} \sqsubseteq \mathcal{H} \vdash e \sqsubseteq e}$

$$\frac{}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash x \sqsubseteq x}$$

$$\frac{\forall i.\ \mathcal{H}' \vdash \tau_i' \qquad \forall i.\ \vdash \tau_i \sqsubseteq \tau_i' \qquad \forall i.\ \mathcal{P} \sqsubseteq \mathcal{H}' \vdash e_i \sqsubseteq e_i' \qquad \mathcal{P} \sqsubseteq \mathcal{H}' \vdash e[x_1 \mapsto x_1', \ldots] \sqsubseteq e'}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash \mathtt{let}\ \langle x_1 : \tau_1, \ldots \rangle := \langle e_1, \ldots \rangle\ \mathtt{in}\ e \sqsubseteq \mathtt{let}\ \langle x_1' : \tau_1', \ldots \rangle := \langle e_1', \ldots \rangle\ \mathtt{in}\ e'}$$

$$\frac{}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash \mathtt{false} \sqsubseteq \mathtt{false}} \qquad \frac{}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash \mathtt{true} \sqsubseteq \mathtt{true}}$$

$$\frac{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash e_1 \sqsubseteq e_1' \quad \mathcal{P} \sqsubseteq \mathcal{H}' \vdash e_2 \sqsubseteq e_2'}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash e_1 == e_2 \sqsubseteq e_1' == e_2'} \qquad \frac{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash e \sqsubseteq e'}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash e.f \sqsubseteq e'.f} \qquad \frac{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash e \sqsubseteq e' \quad \mathcal{P} \sqsubseteq \mathcal{H}' \vdash e_f \sqsubseteq e_f'}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash e.f := e_f \sqsubseteq e'.f := e_f'}$$

$$\frac{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash e \sqsubseteq e' \qquad \forall i.\ \mathcal{P} \sqsubseteq \mathcal{H}' \vdash e_i \sqsubseteq e_i'}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash e(e_1, \ldots) \sqsubseteq e'(e_1', \ldots)}$$

$$\frac{\forall i.\ \mathcal{P} \sqsubseteq \mathcal{H}' \vdash e_i \sqsubseteq e_i'}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash \mathsf{new}\ C(e_1, \ldots) \sqsubseteq \mathsf{new}\ C(e_1', \ldots)} \qquad \frac{x \text{ is not free in } b \qquad \mathcal{P} \sqsubseteq \mathcal{H}' \vdash x := \{\mid \lambda b\} \sqsubseteq e' : \chi'}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash \mathsf{new}\ \lambda\langle b \rangle \sqsubseteq e'}$$

$$\frac{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash x := \{f_1 := e_1; \ldots \mid m_1 b_1; \ldots\} \sqsubseteq e' : \mathsf{ext}}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash \mathsf{new}\ x := \{f_1 := e_1; \ldots \mid m_1 b_1; \ldots\} \sqsubseteq e'}$$

$$\frac{\begin{array}{c}\neg\ \mathcal{H}' \vdash C \qquad \forall i.\ \mathcal{H}' \vdash \tau_i' \qquad x : C(x_1 : \tau_1, \ldots)\{f_1 := e_{f;1}; \ldots \mid m_1 b_1; \ldots\} \in \mathcal{I}_{\mathcal{P}} \qquad \forall i.\ \vdash \tau_i \sqsubseteq \tau_i' \\ \forall i.\ \mathcal{P} \sqsubseteq \mathcal{H}' \vdash e_i \sqsubseteq e_i' \qquad \mathcal{P} \sqsubseteq \mathcal{H}' \vdash x := \{f_1 := e_{f;1}; \ldots \mid m_1 b_1[x_1 \mapsto x_1', \ldots]; \ldots\} \sqsubseteq e' : \chi'\end{array}}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash \mathsf{new}\ C(e_1, \ldots) \sqsubseteq \mathtt{let}\ \langle x_1' : \tau_1', \ldots \rangle := \langle e_1', \ldots \rangle\ \mathtt{in}\ e'}$$

$$\frac{\mathcal{H}' \vdash \tau' \qquad \vdash \tau \sqsubseteq \tau' \qquad \mathcal{H}_{\mathcal{P}} \vdash \tau \leq \tau_x \qquad \mathcal{P} \sqsubseteq \mathcal{H}' \vdash e_x \sqsubseteq e_x' \qquad \mathcal{P} \sqsubseteq \mathcal{H}' \vdash e[x \mapsto x'] \sqsubseteq e'}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash \mathtt{let}\ \langle x_x : \tau \rangle := \langle e_x \rangle\ \mathtt{in}\ \mathtt{let}\ \langle x : \tau_x \rangle := \langle x_x \rangle\ \mathtt{in}\ e \sqsubseteq \mathtt{let}\ \langle x' : \tau' \rangle := \langle e_x' \rangle\ \mathtt{in}\ e'}$$

$\boxed{\mathcal{P} \sqsubseteq \mathcal{H} \vdash x := \{f := e; \ldots \mid mb; \ldots\} \sqsubseteq e : \chi}$

$$\frac{x \text{ is not free in } b \qquad \mathcal{P} \sqsubseteq \mathcal{H}' \vdash b \sqsubseteq b'}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash x := \{\mid \lambda b\} \sqsubseteq \mathsf{new}\ \lambda\langle b' \rangle : \mathsf{fix}}$$

$$\frac{\forall i.\ \mathcal{P} \sqsubseteq \mathcal{H}' \vdash e_i[x \mapsto x'] \sqsubseteq e_i' \qquad \forall i.\ \mathcal{P} \sqsubseteq \mathcal{H}' \vdash b_i[x \mapsto x'] \sqsubseteq b_i'}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash x := \{f_1 := e_1; \ldots \mid m_1 b_1; \ldots\} \sqsubseteq \mathsf{new}\ x' := \{f_1 := e_1'; \ldots \mid m_1 b_1'; \ldots\} : \mathsf{ext}}$$

$$\frac{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash x := \{f_1 := e_1; \ldots \mid m_1 b_1; \ldots\} \sqsubseteq e' : \mathsf{ext} \qquad \mathcal{P} \sqsubseteq \mathcal{H}' \vdash e_f \sqsubseteq e_f'}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash x := \{f_1 := e_1; \ldots; f := e_f \mid m_1 b_1; \ldots\} \sqsubseteq e'.f := e_f' : \mathsf{ext}}$$

$$\frac{x \text{ is not free in } b \qquad \mathcal{P} \sqsubseteq \mathcal{H}' \vdash x := \{f_1 := e_1; \ldots; f := \mathsf{new}\ \lambda\langle b \rangle \mid m_1 b_1; \ldots\} \sqsubseteq e' : \chi'}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash x := \{f_1 := e_1; \ldots \mid m_1 b_1; \ldots; f b\} \sqsubseteq e' : \chi'}$$

$$\frac{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash x := \{f_1 := e_1; \ldots \mid m_1 b_1; \ldots; m' b'; mb; m_1' b_1'; \ldots\} \sqsubseteq e' : \chi'}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash x := \{f_1 := e_1; \ldots \mid m_1 b_1; \ldots; mb; m' b'; m_1' b_1'; \ldots\} \sqsubseteq e' : \chi'}$$

$\boxed{\mathcal{P} \sqsubseteq \mathcal{H} \vdash b \sqsubseteq b}$

$$\frac{\forall i.\ \mathcal{H}' \vdash \tau_i' \qquad \mathcal{H}' \vdash \tau' \qquad \forall i.\ \vdash \tau_i \sqsubseteq \tau_i' \qquad \mathcal{P} \sqsubseteq \mathcal{H}' \vdash e[x_1 \mapsto x_1', \ldots] \sqsubseteq e' \qquad \vdash \tau \sqsubseteq \tau'}{\mathcal{P} \sqsubseteq \mathcal{H}' \vdash (x_1 : \tau_1, \ldots) \mapsto e : \tau \sqsubseteq (x_1' : \tau_1', \ldots) \mapsto e' : \tau'}$$

Fig. A.10. Expression Precision (extension of Figure 9)

not accessible to the initializer. So the impact of this is more on how the semantics need to be formalized in order to easily prove the necessary commutation properties.

Instance-Private Fields. One important nuance of our calculus is that its classes effectively have instance-private fields, meaning fields whose values are (directly) accessible only to the instance they are a part of. Our calculus achieves this by permitting the arguments to the constructor to be directly accessed by the class's method bodies. Without assigning these arguments to (named) fields or returning them from methods, there is no way for even other instances of the same class to access them *even in untyped code*.

That final note about untyped code is important. By not assigning (public) names to these fields, we prevent even our gradual type system from circumventing their privacy. This is not by accident; rather, it is critical to transitioning from structural to nominal code in a way that satisfies the gradual guarantee.

Note that the method bodies in lambda and record expressions can access the local variables in their context. When these expressions are executed, the values of those variables implicitly become part of the structural object allocated in the heap. In order to transition these structural lambda and record expressions into nominal class constructions, we need some way to make those values explicit. If we were to simply designate them as named fields, those named fields would be accessible in the more-precise typed code, and per the gradual guarantee would need to be accessible with the same names in the untyped code, where they were unnamed before. So instead we make them the parameters of the class constructor, where they are accessible from the method bodies but are otherwise unnamed. Thus strong encapsulation might have a critical role in gradual typing.

A.3.3 The Static Gradual Guarantee. Now that the relevant judgements have been formalized in full, here we repeat the formal statement of our guarantee about transitioning between structural and nominal code with a property known as the static gradual guarantee, this time with proof.

Theorem 4.1 (Static Gradual Guarantee). *For all programs satisfying* $\vdash \mathcal{P} \sqsubseteq \mathcal{P}'$*, if* $\vdash \mathcal{P}$*,* $\vdash \mathcal{H}_{\mathcal{P}'}$*, and* $\mathcal{H}_{\mathcal{P}'} \vdash \mathcal{S}_{\mathcal{P}'} : \mathcal{H}_{\mathcal{P}'}$ *hold, then so does* $\vdash \mathcal{P}'$*.*

Proof. This is a corollary of Lemma A.1, below, basically stating the same guarantee for specifically expressions. Because the precision relation is defined in terms of the more-precise implementation but *not* the less-precise implementation, one can simply apply Lemma A.1 to prove that well-typedness of the more-precise implementation implies well-typedness of the corresponding expressions (and method bodies) in the less-precise implementation. The same can be done to show that well-typedness of the more-precise main expression implies well-typedness of the less-precise main expression. Altogether this guarantees that the less-precise program is well-typed if the more-precise program is. □

The following lemma makes use of *subsumptive* supertyping (across hierarchies), which we define in Figure A.11. Subsumptive supertyping is designed to have the property that the subtype is usable in all settings where the supertype is usable, as the following lemma proves.

Lemma A.1. *For all programs satisfying* $\vdash \mathcal{P}$*, any hierarchy and signature satisfying* $\vdash \mathcal{H}_{\mathcal{P}} \sqsubseteq \mathcal{H}'$*,* $\vdash \mathcal{S}_{\mathcal{P}} \sqsubseteq \mathcal{S}'$*,* $\vdash \mathcal{H}'$*, and* $\mathcal{H}' \vdash \mathcal{S}' : \mathcal{H}'$ *have the property that all expressions satisfying* $\mathcal{P} \sqsubseteq \mathcal{H}' \vdash e \sqsubseteq e'$ *also satisfy both of the following for all type contexts and types:*

$$\mathcal{H}_{\mathcal{P}} \mid \mathcal{S}_{\mathcal{P}} \mid \Gamma \vdash e \downarrow \tau \wedge \mathcal{H}_{\mathcal{P}} \sqsubseteq \mathcal{H}' \vdash \Gamma :> \Gamma' \implies \forall \tau'. \quad \vdash \tau \sqsubseteq \tau' \implies \mathcal{H}' \mid \mathcal{S}' \mid \Gamma' \vdash e' \downarrow \tau'$$
$$\mathcal{H}_{\mathcal{P}} \mid \mathcal{S}_{\mathcal{P}} \mid \Gamma \vdash e \uparrow \tau \wedge \mathcal{H}_{\mathcal{P}} \sqsubseteq \mathcal{H}' \vdash \Gamma :> \Gamma' \implies \exists \tau'. \mathcal{H}_{\mathcal{P}} \sqsubseteq \mathcal{H}' \vdash \tau :> \tau' \quad \wedge \quad \mathcal{H}' \mid \mathcal{S}' \mid \Gamma' \vdash e' \uparrow \tau'$$

$\boxed{\mathcal{H} \sqsubseteq \mathcal{H} \vdash \tau :> \tau}$

$$\frac{\mathcal{H}' \vdash \tau' \qquad \mathcal{H} \vdash \tau_{\leq} \leq \tau \qquad \vdash \tau_{\leq} \sqsubseteq \tau'}{\mathcal{H} \sqsubseteq \mathcal{H}' \vdash \tau :> \tau'}$$

$\boxed{\mathcal{H} \sqsubseteq \mathcal{H} \vdash \Gamma :> \Gamma}$

$$\frac{}{\mathcal{H} \sqsubseteq \mathcal{H}' \vdash \varnothing :> \varnothing} \qquad \frac{\mathcal{H} \sqsubseteq \mathcal{H}' \vdash \Gamma :> \Gamma' \qquad \mathcal{H} \sqsubseteq \mathcal{H}' \vdash \tau :> \tau'}{\mathcal{H} \sqsubseteq \mathcal{H}' \vdash \Gamma, x : \tau :> \Gamma', x : \tau'}$$

Fig. A.11. Subsumptive Supertyping

Proof. The two properties respectively regarding type-checking and type-synthesis are proven simultaneously by induction on the proof of $\mathcal{P} \sqsubseteq \mathcal{H}' \vdash e \sqsubseteq e'$. Each case is straightforward, though occasionally one needs to prove a simple lemma that a lookup judgement respects subsumptive supertyping or about how the various subtyping judgements interact. □

Lowered

Program	$\check{\mathcal{P}} ::= \langle \mathcal{H} \mid \mathcal{S} \mid \check{\mathcal{I}} \mid \check{e} \rangle$	Method Body	$\check{b} ::= (\Gamma) \mapsto \check{e} : \tau$
Type Context	$\check{\Gamma} ::= \varnothing \mid \check{\Gamma}, x$	Implementation	$\check{\mathcal{I}} ::= \varnothing \mid \check{\mathcal{I}}; x : C(\check{\Gamma})\{f := \check{e}; \ldots \mid m\check{b}\}$
Expression	$\check{e} ::= x \mid \texttt{let}\ \langle\check{\Gamma}\rangle := \langle\check{e}, \ldots\rangle\ \texttt{in}\ \check{e} \mid \texttt{false} \mid \texttt{true} \mid \check{e} == \check{e}$		
	$\mid\ \check{e}.f^{\delta} \mid \check{e}.f^{\delta} := \check{e} \mid \check{e}.m(\check{e}, \ldots)^{\delta}$		
	$\mid\ \texttt{new}\ C(\check{e}, \ldots) \mid \texttt{new}\ \lambda\langle\check{b}\rangle \mid \texttt{new}\ x := \{f := \check{e}; \ldots \mid m\check{b}; \ldots\}$		
	$\mid\ \ell \mid \langle \ell.f \rangle \mid C(v, \ldots)\{\check{e}, \ldots\} \mid \texttt{cast}^{\gamma}\ \check{e}\ \texttt{to}\ \tau \mid \texttt{impose}^{\gamma}\ \ell.m\ \texttt{on}\ \check{e}$		
Location	ℓ	Value	$v ::= \texttt{false} \mid \texttt{true} \mid \ell \mid \langle \ell.f \rangle$
Guard Mode	$\gamma ::= \varnothing \mid \textbf{dyn}$	Dispatch Mode	$\delta ::= \langle\tau\rangle$

Fig. A.12. Lowered Grammar (repeat of Figure 11)

Heap	H	$::=$	$\varnothing \mid H; \ell \mapsto h$	Mark	$\mu ::= \texttt{init} \mid \texttt{mut}$
Heap Value	h	$::=$	$C(v, \ldots)\{v, \ldots\} \mid \lambda_{\iota} b \mid \{f \mapsto_{\mu} v; \ldots \mid mb; \ldots\}_{\iota}$	Imposition	$\iota ::= \varnothing \mid \iota, I$

Fig. A.13. Heap Grammar (repeat of Figure 13)

A.4 Lowered Grammar

The semantics of MonNom is given by lowering to a lowered grammar, shown in Figure A.12. In doing so, type annotations are removed (except in method bodies, where they have relevance to the semantics of both invocation and casting) but complementary casts are inserted. Furthermore, method invocations are introduced as a distinct construct (with λ-invocation as a special case), and all field accesses, field mutations, and method invocations are given a dispatch mode.

The lowered grammar also has a number of new constructs that arise during execution. Bound methods result from untyped accesses of field names that turn out to be the names of methods in the object. Method invocations reduce to calls to method bodies that are guarded according to whether the invocation guarantees the argument values will belong to the parameter types. Class constructors reduce to initializing the fields before actually allocating the class instance. Casts dynamically check to see whether the value has the given type, sometimes mutating the heap to place an imposition on the value so that it does. Typed invocations on untyped receivers need to cast the value returned by the receiver to all the types that the caller might be expecting based on the method that was called and the interfaces that the receiver supposedly implements.

In the heap, there are three heap values: class instances, lambda closures, and record instances. Lambda closures and record instances each have a list of interfaces that have been imposed on them. Fields of record instances each have a mark indicating whether the field has been mutated or not.

$$\boxed{\mathcal{H} \mid \mathcal{S} \vdash \check{\mathcal{I}} : \mathcal{S}} \qquad \frac{}{\mathcal{H}_0 \mid \mathcal{S}_0 \vdash \varnothing : \varnothing} \qquad \frac{\mathcal{H}_0 \mid \mathcal{S}_0 \vdash \check{\mathcal{I}} : \mathcal{S}}{\mathcal{H}_0 \mid \mathcal{S}_0 \vdash \check{\mathcal{I}} : \mathcal{S}; I\{\ldots\}}$$

$$\frac{\mathcal{H}_0 \mid \mathcal{S}_0 \vdash \check{\mathcal{I}} : \mathcal{S} \quad \vdash \check{\Gamma} : \vec{\tau} \rightsquigarrow \Gamma \qquad \forall i.\ \mathcal{H}_0 \mid \mathcal{S}_0 \mid \varnothing \mid \Gamma \vdash \check{e}_i : \tau_i \qquad \forall i.\ \vdash \check{b}_i : s_i \qquad \forall i.\ \mathcal{H}_0 \mid \mathcal{S}_0 \mid \varnothing \mid \Gamma, x : C \vdash \check{b}_i :}{\mathcal{H}_0 \mid \mathcal{S}_0 \vdash \mathcal{I}; x : C(\check{\Gamma})\{f_1 := \check{e}_1; \ldots \mid m_1\check{b}_1; \ldots\} : \mathcal{S}; C\{m_1 s_1; \ldots\}; C(\vec{\tau})\{f_1 : \tau_1; \ldots\}}$$

$$\boxed{\vdash \check{\Gamma} : \vec{\tau} \rightsquigarrow \Gamma} \qquad \frac{}{\vdash \varnothing : \varnothing \rightsquigarrow \varnothing} \qquad \frac{\vdash \check{\Gamma} : \vec{\tau} \rightsquigarrow \Gamma}{\vdash \check{\Gamma}, x : \vec{\tau}, \tau \rightsquigarrow \Gamma, x : \tau} \qquad \boxed{\vdash \check{b} : s} \qquad \frac{\vdash \Gamma : \vec{\tau}}{\vdash (\Gamma) \mapsto \check{e} : \tau : (\vec{\tau}) : \tau}$$

$$\boxed{\vdash \check{\mathcal{P}}} \qquad \frac{\vdash \mathcal{H} \qquad \mathcal{H} \vdash \mathcal{S} : \mathcal{H} \qquad \mathcal{H} \mid \mathcal{S} \vdash \check{\mathcal{I}} : \mathcal{S} \qquad \mathcal{H} \mid \mathcal{S} \mid \varnothing \mid \varnothing \vdash \check{e} : \mathbb{B}}{\vdash \langle \mathcal{H} \mid \mathcal{S} \mid \check{\mathcal{I}} \mid \check{e} \rangle}$$

Fig. A.14. Lowered-Program Typing

$$\begin{array}{ll} \text{Heap Type} & \Sigma ::= \varnothing \mid \Sigma, \ell : \sigma \\ \text{Heap-Value Type} & \sigma ::= C \mid \{f, \ldots\}_\iota \end{array}$$

$$\boxed{\mathcal{H} \mid \mathcal{S} \vdash H : \Sigma} \qquad \frac{\forall i, i'.\ \ell_i = \ell_{i'} \implies i = i' \qquad \forall \ell : \sigma \in \Sigma.\ \exists i.\ \ell_i = \ell \wedge \mathcal{H} \mid \mathcal{S} \mid \Sigma \vdash \mathcal{H}_i : \sigma}{\mathcal{H} \mid \mathcal{S} \vdash \ell_1 \mapsto h_1; \ldots : \Sigma}$$

$$\boxed{\mathcal{H} \mid \mathcal{S} \mid \Sigma \vdash h : \sigma}$$

$$\frac{C(\tau_{x;1}, \ldots)\{f_1 : \tau_{f;1}; \ldots\} \in \mathcal{S} \quad \forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \varnothing \vdash v_{x;i} : \tau_{x;i} \quad \forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \varnothing \vdash v_{f;i} : \tau_{f;i}}{\mathcal{H} \mid \mathcal{S} \mid \Sigma \vdash C(v_{x;1}, \ldots)\{v_{f;1}, \ldots\} : C} \qquad \frac{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \varnothing \vdash \check{b} : s \qquad \forall i'.\exists i.\ \mathcal{H} \vdash I_i \le I'_{i'}}{\mathcal{H} \mid \mathcal{S} \mid \Sigma \vdash \lambda_{I_1,\ldots}\check{b} : \{\}_{I'_1,\ldots}}$$

$$\frac{\begin{array}{c}\forall i, i'.\ f_i = f_{i'} \implies i = i' \qquad \forall i, i'.\ m_i = m_{i'} \implies i = i' \\ \nexists i, i'.\ f_i = m_{i'} \qquad \forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \varnothing \vdash v_i : \tau_i \qquad \forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \varnothing \vdash \check{b}_i : s_i \\ \forall i, s.\ \mathcal{S} \vdash I_i.\lambda s \implies \exists i'.\ m_{i'} = \lambda \qquad \forall i'.\exists i.\ m_i = f'_{i'} \qquad \forall i'.\exists i.\ \mathcal{H} \vdash I_i \le I'_{i'}\end{array}}{\mathcal{H} \mid \mathcal{S} \mid \Sigma \vdash \{f_1 \mapsto_{\mu_1} v_1; \ldots \mid m_1\check{b}_1; \ldots\}_{I_1,\ldots} : \{f'_1, \ldots\}_{I'_1,\ldots}}$$

$$\boxed{\mathcal{H} \vdash s \lhd s} \qquad \frac{\forall i.\ \mathcal{H} \vdash \tau'_i \lhd \tau_i \qquad \mathcal{H} \vdash \tau \lhd \tau'}{\mathcal{H} \vdash (\tau_1, \ldots) : \tau \lhd (\tau'_1, \ldots) : \tau'} \qquad \boxed{\mathcal{H} \vdash \check{b} \lhd s} \qquad \frac{\vdash \check{b} : s \qquad \mathcal{H} \vdash s \lhd s'}{\mathcal{H} \vdash \check{b} \lhd s'}$$

Fig. A.15. Heap Typing

A.5 Lowered Typing

Although not discussed in the main body of the paper, lowered MonNom also has a type system, which is used to define the invariant that is key to our safety theorem.

A.5.1 Lowered-Program Typing. Lowered-program typing, defined in Figure A.14, is practically identical to program typing.

A.5.2 Heap Typing. Although when programs are first lowered there are no locations in the resulting lowered expressions, as the program evaluates locations and dependencies on the heap are introduced. To facilitate the proofs, lowered-expression typing is defined in terms of a heap

$$\boxed{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \check{e} : \tau} \qquad \frac{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \check{e} : \tau \qquad \mathcal{H} \vdash \tau \blacktriangleleft \tau'}{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \check{e} : \tau'}$$

$$\frac{\Gamma \vdash x : \tau}{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash x : \tau} \qquad \frac{\forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \check{e}_i : \tau_i \qquad \mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma, x_1 : \tau_1, \ldots \vdash \check{e} : \tau}{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \mathtt{let}\ \langle x_1, \ldots \rangle := \langle \check{e}_1, \ldots \rangle\ \mathtt{in}\ \check{e} : \tau}$$

$$\frac{}{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \mathtt{false} : \mathbb{B}} \qquad \frac{}{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \mathtt{true} : \mathbb{B}}$$

$$\frac{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \check{e}_1 : \tau_1 \qquad \mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \check{e}_2 : \tau_2}{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \check{e}_1 == \check{e}_2 : \mathbb{B}}$$

$$\frac{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \check{e} : \tau_\delta \qquad \mathcal{S} \vdash \tau_\delta.f : \tau}{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \check{e}.f^{\langle \tau_\delta \rangle} : \tau} \qquad \frac{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \check{e} : \tau_\delta \qquad \mathcal{S} \vdash \tau_\delta.f : \tau_f \qquad \mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \check{e}_f : \tau_f}{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash e.f^{\langle \tau_\delta \rangle} := e_f : \tau_\delta}$$

$$\frac{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \check{e} : \tau_\delta \qquad \mathcal{S} \vdash \tau_\delta.m(\tau_1, \ldots) : \tau \qquad \forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \check{e}_i : \tau_i}{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \check{e}.m(\check{e}_1, \ldots)^{\langle \tau_\delta \rangle} : \tau}$$

$$\frac{\mathcal{S} \vdash C(\tau_1, \ldots) \qquad \forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \check{e}_i : \tau_i}{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \mathtt{new}\ C(\check{e}_1, \ldots) : C} \qquad \frac{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \check{b} : s}{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \mathtt{new}\ \lambda\langle \check{b} \rangle : \mathbf{dyn}}$$

$$\frac{\begin{array}{c}\forall i, i'.\ f_i = f_{i'} \implies i = i' \qquad \forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \check{e}_i : \tau_i \\ \nexists i, i'.\ f_i = m_{i'} \qquad \forall i, i'.\ m_i = m_{i'} \implies i = i' \qquad \forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma, x : \mathbf{dyn} \vdash \check{b}_i : s_i\end{array}}{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \mathtt{new}\ x := \{f_1 := \check{e}_1; \ldots \mid m_1 \check{b}_1; \ldots\} : \mathbf{dyn}}$$

$$\frac{\ell : C \in \Sigma}{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \ell : C} \qquad \frac{\ell : \{\ldots\}_{I_1, \ldots} \in \Sigma}{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \ell : I_i}$$

$$\frac{\ell : C \in \Sigma \qquad \mathcal{S} \vdash C.fs}{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \langle \ell.f \rangle : \mathbf{dyn}} \qquad \frac{\ell : \{f_1, \ldots\}_\ldots \in \Sigma}{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \langle \ell.f_i \rangle : \mathbf{dyn}}$$

$$\frac{C(\tau_{x;1}, \ldots)\{f_1 : \tau_{f;1}; \ldots\} \in \mathcal{S} \qquad \forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash v_i : \tau_{x;i} \qquad \forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \check{e}_i : \tau_{f;i}}{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash C(v_1, \ldots)\{\check{e}_1, \ldots\} : C}$$

$$\frac{\mathcal{H} \vdash \tau \qquad \mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \check{e} : \tau}{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \mathtt{cast}^{\varnothing}\ \check{e}\ \mathtt{to}\ \tau : \tau} \qquad \frac{\mathcal{H} \vdash \tau \qquad \mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \check{e} : \tau'}{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \mathtt{cast}^{\mathbf{dyn}}\ \check{e}\ \mathtt{to}\ \tau : \tau}$$

$$\frac{\ell : C \in \Sigma \qquad \mathcal{S} \vdash C.m(\ldots) : \tau \qquad \mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \check{e} : \tau}{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \mathtt{impose}^{\varnothing}\ \ell.m\ \mathtt{on}\ \check{e} : \tau} \qquad \frac{\ell : \{\ldots\}_{I_1, \ldots} \in \Sigma \qquad \mathcal{S} \vdash I_i.m(\ldots) : \tau \qquad \mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \check{e} : \tau'}{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \mathtt{impose}^{\mathbf{dyn}}\ \ell.m\ \mathtt{on}\ \check{e} : \tau}$$

$$\boxed{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash \check{b} : s} \qquad \frac{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma, \Gamma' \vdash \check{e} : \tau}{\mathcal{H} \mid \mathcal{S} \mid \Sigma \mid \Gamma \vdash (\Gamma') \mapsto \check{e} : \tau : (\Gamma') : \tau}$$

Fig. A.16. Lowered-Expression Typing

type abstracting the specific state of the heap. Figure A.15 introduces heap types, as well as the rules for typing heaps. These are defined in terms of heap-value types, which indicate whether the location represents a class instance or a structural object with certain named methods and imposed interfaces.

A.5.3 Lowered-Expression Typing. Lowered-expression typing is defined in Figure A.16. The rules are overall unsurprising. They are parameterized by the heap type in order to type locations and bound methods. The first rule guarantees subsumption with respect to *pessimistic* subyping, illustrating that lowered-expression typing is pessimistically typed rather than optimistically typed, which is why there is no need to distinguish between checking and synthesis. Similarly, dispatch modes—rather than just the receiver's type—are used to check field access, field mutation, and method invocation.

The most important detail to notice is that guarded and unguarded operations are type-checked differently. The reason is that guarded operations are permitted to err and so can have rather lax requirements on their inputs. On the other hand, unguarded operations must ensure that the operation can make progress and so must check that their inputs are guaranteed to satisfy the requisite conditions.

$$\boxed{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \downarrow \tau \rightsquigarrow \check{e}} \qquad \frac{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \rightsquigarrow \check{e}}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \downarrow \tau \rightsquigarrow \mathsf{cast}^{\mathbf{dyn}}\ \check{e}\ \mathsf{to}\ \tau}$$

$$\boxed{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \rightsquigarrow \check{e}}$$

$$\frac{\Gamma \vdash x : \tau}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash x \rightsquigarrow x} \qquad \frac{\forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e_i \downarrow \tau_i \rightsquigarrow \check{e}_i \qquad \mathcal{H} \mid \mathcal{S} \mid \Gamma, x_1 : \tau_1, \ldots \vdash e \uparrow \check{e}}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash \mathsf{let}\ \langle x_1 : \tau_1, \ldots \rangle := \langle e_1, \ldots \rangle\ \mathsf{in}\ e \rightsquigarrow \mathsf{let}\ \langle x_1, \ldots \rangle := \langle \check{e}_1, \ldots \rangle\ \mathsf{in}\ \check{e}}$$

$$\frac{}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash \mathsf{false} \rightsquigarrow \mathsf{false}} \qquad \frac{}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash \mathsf{true} \rightsquigarrow \mathsf{true}}$$

$$\frac{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e_1 \rightsquigarrow \check{e}_1 \qquad \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e_2 \rightsquigarrow \check{e}_2}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e_1 == e_2 \rightsquigarrow \check{e}_1 == \check{e}_2}$$

$$\frac{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \uparrow \tau \qquad \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \rightsquigarrow \check{e}}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e.f \uparrow \check{e}.f^{\langle \tau \rangle}} \qquad \frac{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \uparrow \tau \qquad \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \rightsquigarrow \check{e} \qquad \mathcal{S} \vdash \tau.f : \tau_f \qquad \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e_f \downarrow \tau_f \rightsquigarrow \check{e}_f}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e.f := e_f \rightsquigarrow \check{e}.f^{\langle \tau \rangle} := \check{e}_f}$$

$$\frac{\begin{array}{c}\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \uparrow \tau \\ \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \rightsquigarrow \check{e} \qquad \mathcal{S} \vdash \tau.\lambda(\tau_1, \ldots) : \tau_\lambda \\ \forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e_i \downarrow \tau_i \rightsquigarrow \check{e}_i\end{array}}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e(e_1, \ldots) \rightsquigarrow \check{e}.\lambda(\check{e}_1, \ldots)^{\langle \tau \rangle}} \qquad \frac{\begin{array}{c}\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \uparrow \tau \\ \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e \rightsquigarrow \check{e} \qquad \mathcal{S} \vdash \tau.f(\tau_1, \ldots) : \tau_f \\ \forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e_i \downarrow \tau_i \rightsquigarrow \check{e}_i\end{array}}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e.f(e_1, \ldots) \rightsquigarrow \check{e}.f(\check{e}_1, \ldots)^{\langle \tau \rangle}}$$

$$\frac{\mathcal{S} \vdash C(\tau_1, \ldots) \qquad \forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e_i \downarrow \tau_i \rightsquigarrow \check{e}_i}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash \mathsf{new}\ C(e_1, \ldots) \uparrow \mathsf{new}\ C(\check{e}_1, \ldots)} \qquad \frac{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash b \rightsquigarrow \check{b}}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash \mathsf{new}\ \lambda\langle b \rangle \rightsquigarrow \mathsf{new}\ \lambda\langle \check{b} \rangle}$$

$$\frac{\forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e_i \rightsquigarrow \check{e}_i \qquad \forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Gamma, x : \mathbf{dyn} \vdash b_i \rightsquigarrow \check{b}_i}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash \mathsf{new}\ x := \{f_1 := e_1; \ldots \mid m_1 b_1; \ldots\} \rightsquigarrow \mathsf{new}\ x := \{f_1 := \check{e}_1; \ldots \mid m_1 \check{b}_1; \ldots\}}$$

$$\boxed{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash b \rightsquigarrow \check{b}} \qquad \boxed{\vdash \Gamma \rightsquigarrow \check{\Gamma}}$$

$$\frac{\mathcal{H} \mid \mathcal{S} \mid \Gamma, \Gamma_b \vdash e_b \downarrow \tau_b \rightsquigarrow \check{e}}{\mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash (\Gamma_b) \mapsto e_b : \tau_b \rightsquigarrow (\Gamma_b) \mapsto \check{e} : \tau_b} \qquad \frac{}{\vdash \varnothing \rightsquigarrow \varnothing} \qquad \frac{\vdash \Gamma \rightsquigarrow \check{\Gamma}}{\vdash \Gamma, x : \tau \rightsquigarrow \check{\Gamma}, x}$$

$$\boxed{\mathcal{H} \mid \mathcal{S} \vdash \mathcal{I} \rightsquigarrow \check{\mathcal{I}}} \qquad \frac{}{\mathcal{H} \mid \mathcal{S} \vdash \varnothing \rightsquigarrow \varnothing}$$

$$\frac{\mathcal{H} \mid \mathcal{S} \vdash \mathcal{I} \rightsquigarrow \check{\mathcal{I}} \qquad \vdash \Gamma \rightsquigarrow \check{\Gamma} \qquad \forall i.\ \mathcal{S} \vdash C.f_i : \tau_i \qquad \forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash e_i \downarrow \tau_i \rightsquigarrow \check{e}_i \qquad \forall i.\ \mathcal{H} \mid \mathcal{S} \mid \Gamma \vdash b_i \rightsquigarrow \check{b}_i}{\mathcal{H} \mid \mathcal{S} \vdash \mathcal{I}; x : C(\Gamma)\{f_1 := e_1; \ldots \mid m_1 b_1; \ldots\} \rightsquigarrow \check{\mathcal{I}}; x : C(\check{\Gamma})\{f_1 := \check{e}_1; \ldots \mid m_1 \check{b}_1; \ldots\}}$$

$$\boxed{\vdash \mathcal{P} \rightsquigarrow \check{\mathcal{P}}} \qquad \frac{\mathcal{H} \mid \mathcal{S} \vdash \mathcal{I} \rightsquigarrow \check{\mathcal{I}} \qquad \mathcal{H} \mid \mathcal{S} \mid \varnothing \vdash e \downarrow \mathbb{B} \rightsquigarrow \check{e}}{\vdash \langle \mathcal{H} \mid \mathcal{S} \mid \mathcal{I} \mid e \rangle \rightsquigarrow \langle \mathcal{H} \mid \mathcal{S} \mid \check{\mathcal{I}} \mid \check{e} \rangle}$$

Fig. A.17. Lowering (extension of Figure 12)

A.6 Lowering

MonNom programs are given a semantics by lowering them to lowered programs using the judgements in Figure A.17. Most of these rules have already been discussed in Section 5.1, and there

is nothing particularly interesting among the remaining rules. More important are the following properties of lowering.

Lemma A.2. *For any program satisfying* $\vdash \mathcal{P}$ *there exists a lowered program satisfying* $\vdash \mathcal{P} \rightsquigarrow \check{\mathcal{P}}$.

Proof. The corresponding lemma for totality of lowering exceptions is easily proven by induction on the proof of well-typedness. Totality of lowering programs is then a trivial corollary. □

Lemma A.3. *Any programs and lowered programs satisfying both* $\vdash \mathcal{P}$ *and* $\vdash \mathcal{P} \rightsquigarrow \check{\mathcal{P}}$ *furthermore satisfy* $\vdash \check{\mathcal{P}}$.

Proof. The corresponding lemma for type-preservation of lowering exceptions is easily proven by induction on the proof of lowering. Type-preservation of lowering programs is then a trivial corollary. □

Combined, these lemmas guarantee that any well-typed program lowers into a well-typed lowered program, and as such lowering is a reliable means for giving programs semantics. Note, though, that there may be multiple such lowerings. In general, this could lead to unwanted non-determinism, but in Section A.9.5 we will show that all such lowerings are necessarily semantically equivalent.

$$\boxed{\check{\mathcal{P}} \mid H \vdash \ell.f \mapsto v}$$

$$\frac{\ell \mapsto C(\ldots)\{v_1, \ldots\} \in H \qquad C(\ldots)\{f_1 : \tau_1; \ldots\} \in \mathcal{S}_{\check{\mathcal{P}}}}{\check{\mathcal{P}} \mid H \vdash \ell.f_i \mapsto v_i} \qquad \frac{\ell \mapsto \{f_1 \mapsto_{\mu_1} v_1; \ldots \mid \ldots\}_\iota \in H}{\check{\mathcal{P}} \mid H \vdash \ell.f_i \mapsto v_i}$$

$$\boxed{\check{\mathcal{P}} \mid H \to H \vdash \ell.f := v}$$

$$\frac{\begin{array}{c}\ell \mapsto C(v_{x;1}, \ldots)\{v_{f;1}, \ldots, v, v'_{f;1}, \ldots\} \in H \qquad C(\ldots)\{f_1 : \tau_1; \ldots; f : \tau; f'_1 : \tau'_1; \ldots\} \in \mathcal{S}_{\check{\mathcal{P}}} \\ \check{\mathcal{P}} \mid H \to H' \vdash v : \tau \qquad H'' = H'[\ell \mapsto C(v_{x;1}, \ldots)\{v_{f;1}, \ldots, v', v'_{f;1}, \ldots\}]\end{array}}{\check{\mathcal{P}} \mid H \to H'' \vdash \ell.f := v'}$$

$$\frac{\begin{array}{c}\ell \mapsto \{f_1 \mapsto_{\mu_1} v_1; \ldots; f \mapsto_{\mu} v; f'_1 \mapsto_{\mu'_1} v'_1 \mid m_1 \check{b}_1; \ldots\}_\iota \in H \\ H' = H[\ell \mapsto \{f_1 \mapsto_{\mu_1} v_1; \ldots; f \mapsto_{\mathsf{mut}} v'; f'_1 \mapsto_{\mu'_1} v'_1 \mid m_1 \check{b}_1; \ldots\}_\iota]\end{array}}{\check{\mathcal{P}} \mid H \to H' \vdash \ell.f := v'}$$

$$\frac{\begin{array}{c}\ell \mapsto \{f_1 \mapsto_{\mu_1} v_1; \ldots \mid m_1 \check{b}_1; \ldots\}_\iota \in H \\ \nexists i.\ f = f_i \qquad \nexists i.\ f = m_i \qquad H' = H[\ell \mapsto \{f_1 \mapsto_{\mu_1} v_1; \ldots; f \mapsto_{\mathsf{init}} v' \mid m_1 \check{b}_1; \ldots\}_\iota]\end{array}}{\check{\mathcal{P}} \mid H \to H' \vdash \ell.f := v'}$$

$$\boxed{\check{\mathcal{P}} \mid H \vdash v.m \rightsquigarrow_\gamma \check{b}}$$

$$\frac{\ell \mapsto C(v_1, \ldots)\{\ldots\} \in H \qquad x : C(x_1, \ldots)\{\ldots \mid m_1 \check{b}_1; \ldots\} \in \check{\mathcal{I}}_{\check{\mathcal{P}}}}{\check{\mathcal{P}} \mid H \vdash \ell.m_i \rightsquigarrow_\varnothing \check{b}_i[x \mapsto \ell, x_1 \mapsto v_1, \ldots]} \qquad \frac{\ell \mapsto \lambda_\iota \check{b} \in H}{\check{\mathcal{P}} \mid H \vdash \ell.\lambda \rightsquigarrow_{\mathbf{dyn}} \check{b}}$$

$$\frac{\ell \mapsto \{\ldots \mid m_1 \check{b}_1; \ldots\}_\iota \in H}{\check{\mathcal{P}} \mid H \vdash \ell.m_i \rightsquigarrow_{\mathbf{dyn}} \check{b}_i} \qquad \frac{\check{\mathcal{P}} \mid H \vdash \ell.f \rightsquigarrow_\gamma \check{b}}{\check{\mathcal{P}} \mid H \vdash \langle \ell.f \rangle.\lambda \rightsquigarrow_{\mathbf{dyn}} \check{b}}$$

$$\boxed{\check{\mathcal{P}} \mid H \to H \vdash \ell.m \rightsquigarrow_\gamma \check{b}} \qquad \begin{array}{c}\ell \mapsto \{f_1 \mapsto_{\mu_1} v_1; \ldots; f \mapsto_{\mathsf{init}} v; f'_1 \mapsto_{\mu'_1} v'_1; \ldots \mid m_1 \check{b}_1; \ldots\}_\iota \in H \\ \check{\mathcal{P}} \mid H \vdash v.\lambda \rightsquigarrow_\gamma \check{b}\end{array}$$

$$\frac{\check{\mathcal{P}} \mid H \vdash \ell.m \rightsquigarrow_\gamma \check{b}}{\check{\mathcal{P}} \mid H \to H \vdash \ell.m \rightsquigarrow_\gamma \check{b}} \qquad \frac{H' = H[\ell \mapsto \{f_1 \mapsto_{\mu_1} v_1; \ldots; f \mapsto_{\mathsf{init}} v; f'_1 \mapsto_{\mu'_1} v'_1; \ldots \mid m_1 \check{b}_1; \ldots\}_\iota]}{\check{\mathcal{P}} \mid H \to H' \vdash \ell.f \rightsquigarrow_{\mathbf{dyn}} \check{b}}$$

$$\boxed{\mathcal{H} \mid H \vdash \ell \mapsto \iota}$$

$$\frac{\ell \mapsto C(\ldots)\{\ldots\} \in H \qquad C \leq I_1, \ldots \in \mathcal{H}}{\mathcal{H} \mid H \vdash \ell \mapsto I_1, \ldots} \qquad \frac{\ell \mapsto \lambda_\iota \check{b} \in H}{\mathcal{H} \mid H \vdash \ell \mapsto \iota} \qquad \frac{\ell \mapsto \{\ldots \mid \ldots\}_\iota \in H}{\mathcal{H} \mid H \vdash \ell \mapsto \iota}$$

Fig. A.18. Heap Semantics (extension of Figure 14)

A.7 Reduction

After lowering a program, its semantics is determined by repeatedly reducing the main expression of the program (accummulating a heap in the process).

A.7.1 Heap Semantics. Expression reduction utilizes a number of operations on the heap, defined in Figure A.18. These judgements are used to get the value of a field, set the value of a field, direcly lookup the body of a method, indirectly lookup the body of a method (freezing the corresponding field if appropriate), and fetch the list of interfaces imposed upon a value (or list of interfaces implemented by the class the object is an instance of). Besides what was already discussed in

$$\boxed{\check{\mathcal{P}} \mid H \to H \vdash v : \vec{\tau}}$$

$$\frac{}{\check{\mathcal{P}} \mid H \to H \vdash v : \varnothing} \qquad \frac{\check{\mathcal{P}} \mid H \to H' \vdash v : \vec{\tau} \qquad \check{\mathcal{P}} \mid H' \vdash v : \tau}{\check{\mathcal{P}} \mid H \to H' \vdash v : \vec{\tau}, \tau}$$

$$\frac{\check{\mathcal{P}} \mid H \to H' \vdash \ell : \vec{\tau} \quad \ell \mapsto \lambda_\iota \check{b} \in H' \quad H'' = H'[\ell \mapsto \lambda_{\iota,I} \check{b}]}{\check{\mathcal{P}} \mid H \to H'' \vdash \ell : \vec{\tau}, I} \qquad \frac{\check{\mathcal{P}} \mid H \to H' \vdash \ell : \vec{\tau} \quad \ell \mapsto \{f_1 \mapsto_{\mu_1} v_1; \ldots \mid m_1 \check{b}_1; \ldots\}_\iota \in H' \quad \forall s.\, \mathcal{S}_{\check{\mathcal{P}}} \vdash I.\lambda s \implies \exists i.\, m_i = \lambda \quad H'' = H'[\ell \mapsto \{f_1 \mapsto_{\mu_1} v_1; \ldots \mid m_1 \check{b}_1; \ldots\}_{\iota,I}]}{\check{\mathcal{P}} \mid H \to H'' \vdash \ell : \vec{\tau}, I}$$

$$\boxed{\check{\mathcal{P}} \mid H \vdash v : \tau}$$

$$\frac{}{\check{\mathcal{P}} \mid H \vdash v : \mathbf{dyn}} \qquad \frac{}{\check{\mathcal{P}} \mid H \vdash \texttt{false} : \mathbb{B}} \qquad \frac{}{\check{\mathcal{P}} \mid H \vdash \texttt{true} : \mathbb{B}} \qquad \frac{\ell \mapsto C(\ldots)\{\ldots\} \in H \quad \mathcal{H}_{\check{\mathcal{P}}} \vdash C \blacktriangleleft \tau}{\check{\mathcal{P}} \mid H \vdash \ell : \tau} \qquad \frac{\mathcal{H}_{\check{\mathcal{P}}} \mid H \vdash \ell \mapsto \iota \quad \mathcal{H}_{\check{\mathcal{P}}} \vdash \iota \blacktriangleleft \tau}{\check{\mathcal{P}} \mid H \vdash \ell : \tau}$$

$$\boxed{\mathcal{H} \vdash \iota \blacktriangleleft \tau}$$

$$\frac{}{\mathcal{H} \vdash \varnothing \blacktriangleleft \mathbf{dyn}} \qquad \frac{\mathcal{H} \vdash \iota \blacktriangleleft \tau}{\mathcal{H} \vdash \iota, I \blacktriangleleft \tau} \qquad \frac{\mathcal{H} \vdash I \blacktriangleleft \tau}{\mathcal{H} \vdash \iota, I \blacktriangleleft \tau}$$

$$\boxed{\mathcal{S} \vdash (\iota).m : \vec{\tau}}$$

$$\frac{}{\mathcal{S} \vdash (\varnothing).m : \varnothing} \qquad \frac{\mathcal{S} \vdash (\iota).m : \vec{\tau} \qquad \mathcal{S} \vdash I.m(\ldots) : \tau}{\mathcal{S} \vdash (\iota, I).m : \vec{\tau}, \tau} \qquad \frac{\mathcal{S} \vdash (\iota).m : \vec{\tau} \qquad \nexists s.\, \mathcal{S} \vdash I.ms}{\mathcal{S} \vdash (\iota, I).m : \vec{\tau}}$$

Fig. A.19. Cast Semantics (extension of Figure 15)

Section 5.2, the only detail to note is that the rule for setting the value of a class field first casts the value to the type of the class field.

A.7.2 Cast Semantics. MonNom's casting semantics are shown in Figure A.19. Only one judgement is new compared to Figure 15, and its definition is straightforward.

A.7.3 Lowered-Expression Semantics. Figure A.20 provides the lowered-expression semantics for MonNom. These reduction semantics take an expression, split it uniquely into an evaluation context of some redex, and then reduce the redex within the evaluation context. The upper judgement specifies an "allocated" heap, which the lower judgement then incorporates into the overall heap. In particular, the upper judgements before-and-after heaps have mappings for the same locations, with the latter preserving the type of the former. The reduced expression may reference locations in the "allocated" heap, but these locations are conceptually fresh, and as such one could rename all the new locations in the allocated heap and propagate the renaming throughout the allocated heap and the reduced expression and the result would be another viable reduction for the judgement. This separation of the allocated heap facilitates many of the proofs, particularly those that require reordering of independent operations (to accommodate field initializations being moved to after record allocation).

A.7.4 Error Semantics. As already discussed in Section 5.5, MonNom distinguishes between getting stuck and erring. This is formalized in Figure A.21, which is simply a repeat of Figure 16.

$$
\begin{array}{lrcl}
\text{Redex} & r & ::= & \texttt{let}\ \langle \check{\Gamma} \rangle := \langle v, \ldots \rangle\ \texttt{in}\ \check{e} \mid v == v \mid v.f^{\delta} \mid v.f^{\delta} := v \mid v.m(v,\ldots)^{\delta} \\
& & \mid & \texttt{new}\ C(v,\ldots) \mid \texttt{new}\ \lambda\langle \check{b} \rangle \mid \texttt{new}\ x := \{f := v; \ldots \mid m\check{b}; \ldots\} \\
& & \mid & C(v,\ldots)\{v,\ldots\} \mid \texttt{cast}^{\gamma}\ v\ \texttt{to}\ \tau \mid \texttt{impose}^{\gamma}\ \ell.m\ \texttt{on}\ v \\
\text{Evaluation Context} & E & ::= & \bullet \mid \texttt{let}\ \langle \check{\Gamma} \rangle := \langle v, \ldots, E, \check{e}, \ldots \rangle\ \texttt{in}\ \check{e} \mid E == \check{e} \mid v == E \\
& & \mid & E.f^{\delta} \mid E.f^{\delta} := \check{e} \mid v.f^{\delta} := E \mid E.m(\check{e},\ldots)^{\delta} \mid v.m(v,\ldots,E,\check{e},\ldots)^{\delta} \\
& & \mid & \texttt{new}\ C(v,\ldots,E,\check{e},\ldots) \mid \texttt{new}\ x := \{f := v; \ldots; f := E; f := \check{e}; \ldots \mid mb; \ldots\} \\
& & \mid & C(v,\ldots)\{v,\ldots,E,\check{e},\ldots\} \mid \texttt{cast}^{\gamma}\ E\ \texttt{to}\ \tau \mid \texttt{impose}^{\gamma}\ \ell.m\ \texttt{on}\ E
\end{array}
$$

$$\boxed{\check{\mathcal{P}} \vdash H \mid \check{e} \xrightarrow{H} H \mid \check{e} \qquad \check{\mathcal{P}} \vdash H \mid r \xrightarrow{H} H \mid \check{e}}$$

$$\frac{\check{\mathcal{P}} \vdash H \mid r \xrightarrow{H''} H' \mid \check{e}'}{\check{\mathcal{P}} \vdash H \mid E[r] \xrightarrow{H''} H' \mid E[\check{e}']} \qquad \frac{}{\check{\mathcal{P}} \vdash H \mid \texttt{let}\ \langle x_1, \ldots \rangle := \langle v_1, \ldots \rangle\ \texttt{in}\ \check{e} \xrightarrow{\varnothing} H \mid \check{e}[x_1 \mapsto v_1, \ldots]}$$

$$\frac{}{\check{\mathcal{P}} \vdash H \mid v == v \xrightarrow{\varnothing} H \mid \texttt{true}} \qquad \frac{v_1 \neq v_2}{\check{\mathcal{P}} \vdash H \mid v_1 == v_2 \xrightarrow{\varnothing} H \mid \texttt{false}}$$

$$\frac{\check{\mathcal{P}} \mid H \vdash \ell.f \mapsto v}{\check{\mathcal{P}} \vdash H \mid \ell.f^{\delta} \xrightarrow{\varnothing} H \mid v} \qquad \frac{\check{\mathcal{P}} \mid H \vdash \ell.f \rightsquigarrow_{\gamma} \check{b}}{\check{\mathcal{P}} \vdash H \mid \ell.f^{\langle \mathbf{dyn} \rangle} \xrightarrow{\varnothing} H \mid \langle \ell.f \rangle} \qquad \frac{\check{\mathcal{P}} \mid H \rightarrow H' \vdash \ell.f := v}{\check{\mathcal{P}} \vdash H \mid \ell.f^{\delta} := v \xrightarrow{\varnothing} H' \mid \ell}$$

$$\frac{\check{\mathcal{P}} \mid H \vdash v.m \rightsquigarrow_{\gamma} (x_1 : \tau_1, \ldots) \mapsto \check{e} : \tau}{\check{\mathcal{P}} \vdash H \mid v.m(v_1,\ldots)^{\langle \mathbf{dyn} \rangle} \xrightarrow{\varnothing} H \mid \texttt{cast}^{\varnothing}\ \texttt{let}\ \langle x_1, \ldots \rangle := \langle \texttt{cast}^{\mathbf{dyn}}\ v_1\ \texttt{to}\ \tau_1, \ldots \rangle\ \texttt{in}\ \check{e}\ \texttt{to}\ \mathbf{dyn}}$$

$$\frac{\check{\mathcal{P}} \mid H \vdash \ell.f \mapsto v \qquad \check{\mathcal{P}} \mid H \vdash v.\lambda \rightsquigarrow_{\gamma} (x_1 : \tau_1, \ldots) \mapsto \check{e} : \tau}{\check{\mathcal{P}} \vdash H \mid \ell.f(v_1,\ldots)^{\langle \mathbf{dyn} \rangle} \xrightarrow{\varnothing} H \mid \texttt{cast}^{\varnothing}\ \texttt{let}\ \langle x_1, \ldots \rangle := \langle \texttt{cast}^{\mathbf{dyn}}\ v_1\ \texttt{to}\ \tau_1, \ldots \rangle\ \texttt{in}\ \check{e}\ \texttt{to}\ \mathbf{dyn}}$$

$$\frac{\check{\mathcal{P}} \mid H \rightarrow H' \vdash \ell.m \rightsquigarrow_{\gamma} (x_1 : \tau_1, \ldots) \mapsto \check{e} : \tau}{\check{\mathcal{P}} \vdash H \mid \ell.m(v_1,\ldots)^{\langle N \rangle} \xrightarrow{\varnothing} H' \mid \texttt{impose}^{\gamma}\ \ell.m\ \texttt{on}\ \texttt{let}\ \langle x_1, \ldots \rangle := \langle \texttt{cast}^{\gamma}\ v_1\ \texttt{to}\ \tau_1, \ldots \rangle\ \texttt{in}\ \check{e}}$$

$$\frac{x : C(x_1,\ldots)\{f_1 := \check{e}_1; \ldots \mid \ldots\} \in \check{\mathcal{I}}_{\check{\mathcal{P}}}}{\check{\mathcal{P}} \vdash H \mid \texttt{new}\ C(v_1,\ldots) \xrightarrow{\varnothing} H \mid C(v_1,\ldots)\{\check{e}_1[x_1 \mapsto v_1, \ldots], \ldots\}}$$

$$\frac{H'' = \ell \mapsto \lambda_{\varnothing}\check{b}}{\check{\mathcal{P}} \vdash H \mid \texttt{new}\ \lambda\langle \check{b} \rangle \xrightarrow{H''} H \mid \ell} \qquad \frac{H'' = \ell \mapsto \{f_1 \mapsto_{\mathsf{init}} v_1; \ldots \mid m_1 b_1[x \mapsto \ell]; \ldots\}_{\varnothing}}{\check{\mathcal{P}} \vdash H \mid \texttt{new}\ x := \{f_1 := v_1; \ldots \mid m_1 b_1; \ldots\} \xrightarrow{H''} H \mid \ell}$$

$$\frac{H'' = \ell \mapsto C(v_1,\ldots)\{v_1',\ldots\}}{\check{\mathcal{P}} \vdash H \mid C(v_1,\ldots)\{v_1',\ldots\} \xrightarrow{H''} H \mid \ell} \qquad \frac{\check{\mathcal{P}} \mid H \rightarrow H' \vdash v : \tau}{\check{\mathcal{P}} \vdash H \mid \texttt{cast}^{\gamma}\ v\ \texttt{to}\ \tau \xrightarrow{\varnothing} H' \mid v}$$

$$\frac{\mathcal{H}_{\check{\mathcal{P}}} \mid H \vdash \ell \mapsto \iota \qquad \mathcal{S}_{\check{\mathcal{P}}} \vdash (\iota).m : \vec{\tau} \qquad \check{\mathcal{P}} \mid H \rightarrow H' \vdash v : \vec{\tau}}{\check{\mathcal{P}} \vdash H \mid \texttt{impose}^{\gamma}\ \ell.m\ \texttt{on}\ v \xrightarrow{\varnothing} H' \mid v}$$

$$\boxed{\check{\mathcal{P}} \vdash H \mid \check{e} \rightarrow H \mid \check{e}} \qquad \frac{\check{\mathcal{P}} \vdash H \mid \check{e} \xrightarrow{H''} H' \mid \check{e}' \qquad \nexists \ell, h, h'.\ \ell \mapsto h \in H' \wedge \ell \mapsto h' \in H''}{\check{\mathcal{P}} \vdash H \mid \check{e} \rightarrow H'; H'' \mid \check{e}'}$$

Fig. A.20. Lowered-Expression Semantics (extension of Figure 17)

$$\text{Potentially Erroneous Redex } \varepsilon ::= v.f^{\langle \mathbf{dyn} \rangle} \mid v.f^{\langle \mathbf{dyn} \rangle} := v \mid \ell.f(v,\ldots)^{\langle I \rangle} \mid v.m(v,\ldots)^{\langle \mathbf{dyn} \rangle} \mid \mathtt{cast}^{\mathbf{dyn}}\ v\ \mathtt{to}\ N \mid \mathtt{impose}^{\mathbf{dyn}}\ \ell.m\ \mathtt{on}\ v$$

$$\boxed{\check{\mathcal{P}} \vdash H \mid \check{e} \longrightarrow \mathbf{error}} \qquad \frac{\nexists H', \check{e}'.\ \check{\mathcal{P}} \vdash H \mid \varepsilon \longrightarrow H' \mid \check{e}'}{\check{\mathcal{P}} \vdash H \mid E[\varepsilon] \longrightarrow \mathbf{error}}$$

Fig. A.21. Error Semantics (repeat of Figure 16)

Observation $o ::= \texttt{false} \mid \texttt{true} \mid \infty \mid \textbf{error}$

$$\boxed{\vdash \mathcal{P} \twoheadrightarrow o} \qquad \frac{\vdash \mathcal{P} \rightsquigarrow \check{\mathcal{P}} \qquad \check{e}_1 = \check{e}_{\check{\mathcal{P}}} \qquad H_1 = \varnothing \qquad \forall i < n.\ \check{\mathcal{P}} \vdash H_i \mid \check{e}_i \longrightarrow H_{i+1} \mid \check{e}_{i+1} \qquad \check{e}_n = v \qquad o = v}{\vdash \mathcal{P} \twoheadrightarrow o}$$

$$\frac{\vdash \mathcal{P} \rightsquigarrow \check{\mathcal{P}} \qquad \check{e}_1 = \check{e}_{\check{\mathcal{P}}} \qquad H_1 = \varnothing \qquad \forall i < n.\ \check{\mathcal{P}} \vdash H_i \mid \check{e}_i \longrightarrow H_{i+1} \mid \check{e}_{i+1} \qquad \check{\mathcal{P}} \vdash H_n \mid \check{e}_n \longrightarrow \textbf{error}}{\vdash \mathcal{P} \twoheadrightarrow \textbf{error}} \qquad \frac{\vdash \mathcal{P} \rightsquigarrow \check{\mathcal{P}} \qquad \check{e}_1 = \check{e}_{\check{\mathcal{P}}} \qquad H_1 = \varnothing \qquad \forall i \in \mathbb{N}.\ \check{\mathcal{P}} \vdash H_i \mid \check{e}_i \longrightarrow H_{i+1} \mid \check{e}_{i+1}}{\vdash \mathcal{P} \twoheadrightarrow \infty}$$

Fig. A.22. Program Semantics (repeat of Figure 10)

A.8 Semantics

The semantics of MonNom are repeated in Figure A.22. A program is lowered and then reduced repeatedly. One observes either a resulting value, an explicit error, or an infinite divergence.

A.8.1 Type Safety.

Theorem 5.2 (Safety). *For any program satisfying $\vdash \mathcal{P}$, there exists an observation satisfying $\vdash \mathcal{P} \twoheadrightarrow o$.*

Proof. Lemma A.2 guarantees that there exists some lowered program $\check{\mathcal{P}}$ that is a lowering of $\mathcal{P}$. Lemma A.3 guarantees that that lowering is well-typed. In particular, that implies that $\check{e}_{\check{\mathcal{P}}}$ (and $\check{e}_0$) has type $\mathbb{B}$. The following Lemma A.4 then guarantees that the expression is either a Boolean value (a valid observation), errs (a valid observation), or reduces. And Lemma A.5 ensures that if it reduces then the result must still have type $\mathbb{B}$. This establishes an invariant of a $\mathbb{B}$-typed expression, so we can repeat the process until either an observation occurs or it reduces forever (which is also a valid observation). Thus the program necessarily has some observable semantics (and does not get stuck without also erring). □

Lemma A.4. *For any lowered program, heap, heap type, lowered expression, and type satisfying $\vdash \check{\mathcal{P}}, \mathcal{H}_{\check{\mathcal{P}}} \mid \mathcal{S}_{\check{\mathcal{P}}} \vdash H : \Sigma$, and $\mathcal{H}_{\check{\mathcal{P}}} \mid \mathcal{S}_{\check{\mathcal{P}}} \mid \Sigma \mid \varnothing \vdash \check{e} : \tau$, (exactly) one of the following holds:*

- *the lowered expression $\check{e}$ is in fact a value v,*
- *the lowered expression errs, i.e. $\check{\mathcal{P}} \vdash H \mid \check{e} \longrightarrow \textbf{error}$ holds, or*
- *the lowered expression reduces, i.e. there exists a heap H' and lowered expression $\check{e}'$ such that $\check{\mathcal{P}} \vdash H \mid \check{e} \longrightarrow H' \mid \check{e}'$ holds.*

Proof. Proven by induction on the proof that $\check{e}$ is well-typed. Each case is straightforward, sometimes employing simple lemmas about the heap or the like. □

Lemma A.5. *For any lowered program, heaps, heap type, lowered expressions, and type satisfying $\vdash \check{\mathcal{P}}, \mathcal{H}_{\check{\mathcal{P}}} \mid \mathcal{S}_{\check{\mathcal{P}}} \vdash H : \Sigma, \mathcal{H}_{\check{\mathcal{P}}} \mid \mathcal{S}_{\check{\mathcal{P}}} \mid \Sigma \mid \varnothing \vdash \check{e} : \tau$, and $\check{\mathcal{P}} \vdash H \mid \check{e} \longrightarrow H' \mid \check{e}'$, there exists a heap type Σ' such that $\mathcal{H}_{\check{\mathcal{P}}} \mid \mathcal{S}_{\check{\mathcal{P}}} \vdash H' : \Sigma'$ and $\mathcal{H}_{\check{\mathcal{P}}} \mid \mathcal{S}_{\check{\mathcal{P}}} \mid \Sigma' \mid \varnothing \vdash \check{e}' : \tau$ both hold.*

Proof. Proven by induction on the proof that $\check{e}$ is well-typed. Each case is straightforward, sometimes employing simple lemmas about the heap or substitution or the like. □

$$\boxed{\check{\mathcal{P}} \sqsubseteq \mathcal{H} \vdash \check{\mathcal{I}} \sqsubseteq \check{\mathcal{I}}}$$

$$\frac{}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \vdash \varnothing \sqsubseteq \varnothing} \qquad \frac{\neg\, \mathcal{H}' \vdash C \qquad \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \vdash \check{\mathcal{I}} \sqsubseteq \check{\mathcal{I}}'}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \vdash \check{\mathcal{I}}; x : C(\ldots)\{\ldots \mid \ldots\} \sqsubseteq \check{\mathcal{I}}'}$$

$$\frac{\forall i.\ \mathcal{H}' \vdash \tau_i' \qquad \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \vdash \check{\mathcal{I}} \sqsubseteq \check{\mathcal{I}}' \qquad \forall i.\ \vdash \tau_i \sqsubseteq \tau_i' \qquad \forall i.\ \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \varnothing \vdash \check{e}_i \sqsubseteq \varnothing \mid \check{e}_i' \qquad \forall i.\ \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \varnothing \vdash \check{b}_i[x \mapsto x', x_1 \mapsto x_1', \ldots] \sqsubseteq \varnothing \mid \check{b}_i'}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \vdash \check{\mathcal{I}}; x : C(x_1 : \tau_1, \ldots)\{f_1 := \check{e}_1; \ldots \mid m_1 \check{b}_1; \ldots\} \sqsubseteq \check{\mathcal{I}}'; x' : C(x_1' : \tau_1', \ldots)\{f_1 := \check{e}_1'; \ldots \mid m_1 \check{b}_1'; \ldots\}}$$

$$\boxed{\vdash \check{\mathcal{P}} \sqsubseteq \check{\mathcal{P}}}$$

$$\frac{\vdash \mathcal{H}_{\check{\mathcal{P}}} \sqsubseteq \mathcal{H}_{\check{\mathcal{P}}'} \qquad \vdash \mathcal{S}_{\check{\mathcal{P}}} \sqsubseteq \mathcal{S}_{\check{\mathcal{P}}'} \qquad \check{\mathcal{P}} \sqsubseteq \mathcal{H}_{\check{\mathcal{P}}'} \vdash \mathcal{I}_{\check{\mathcal{P}}} \sqsubseteq \mathcal{I}_{\check{\mathcal{P}}'} \qquad \check{\mathcal{P}} \sqsubseteq \mathcal{H}_{\check{\mathcal{P}}'} \mid \varnothing \vdash \check{e}_{\check{\mathcal{P}}} \sqsubseteq \varnothing \mid \check{e}_{\check{\mathcal{P}}'}}{\vdash \check{\mathcal{P}} \sqsubseteq \check{\mathcal{P}}'}$$

Fig. A.23. Lowered-Program Precision

A.9 Lowered Precision

Just as there is a lowered type system not discussed in the paper, there is also a lowered precision relation not discussed in the paper. This precision relation establishes the relationship between lowered programs that is critical to our theorems.

A.9.1 Lowered-Program Precision. The rules for lowered-program precision, Figure A.23, are the obvious analog of those for program precision in Figure A.9.

A.9.2 Heap Precision. In Figure A.24 we present the rules for heap precision. Notably, this figure introduces the concept of a *heap correspondence* η. This correspondence indicates which locations and bound methods in the more-precise heap correspond to which locations in the less-precise heap. The rules are designed so that this correspondence as *at most* one-to-one. The "at most" qualifier is important because there can be many locations in the less-precise heap that correspond to no location in the more-precise heap—rather they represent a method in a more-precise object that has been implemented as a field in a corresponding less-precise object. This machinery is achieved by the judgement $\check{\mathcal{P}} \sqsubseteq \mathcal{H} \mid \eta \vdash \ell \mapsto \{f \mapsto_\mu \check{e}; \ldots \mid m\check{b}; \ldots\}_t^\chi \sqsubseteq H' \mid \ell' \mapsto h' : \eta$. In this judgement, ℓ and ℓ' are the corresponding locations, and h' is a structural heap value containing (relaxations of) the prescribed fields and methods. Sometimes h' might represent one of the prescribed methods using instead a field whose value is a location in the heap H', with an appropriate corresponding heap value. This judgement also takes care of addressing issues with reordering that can arise from delaying a field initialization until after a record has been allocated, interspersed with using fields to implement methods.

A.9.3 Lowered-Expression Precision. The lowered-expression precision relation is defined in the three-page Figure A.25. The first judgement simply uses the heap correspondence to correlate values. The second judgement, on the other hand, enumerates all of the correspondences between lowered expressions that arise either directly from the lowering process itself or indirectly from reduction. The heap in the judgement represents locations that have been allocated in the less-precise program that have no corresponding location in the more-precise program either because, in particular, evaluation is in the midst of constructing a record where the more-precise program uses a method but the less-precise program uses a field whose initialization has already been evaluated.

Besides these nuances with records, there are a few other rules to take note of. First, there is the rule relating a field access followed by a λ-invocation to an untyped named-method invocation, and

Heap Correspondence $\eta ::= \varnothing \mid \eta, \ell \sqsubseteq \ell \mid \eta, \langle \ell.f \rangle \sqsubseteq \ell$

$$\boxed{\check{\mathcal{P}} \sqsubseteq \mathcal{H} \vdash H \mid \check{e} \sqsubseteq H \mid \check{e}}$$

$$\frac{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \vdash H_1; H_3; H_2; H_4 \mid \check{e} \sqsubseteq H' \mid \check{e}'}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \vdash H_1; H_2; H_3; H_4 \mid \check{e} \sqsubseteq H' \mid \check{e}'} \qquad \frac{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \vdash H \mid \check{e} \sqsubseteq H_1'; H_3'; H_2'; H_4' \mid \check{e}'}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \vdash H \mid \check{e} \sqsubseteq H_1'; H_2'; H_3'; H_4' \mid \check{e}'}$$

$$\frac{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash H \sqsubseteq H' : \eta \qquad \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e} \sqsubseteq H'' \mid \check{e}'}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \vdash H \mid \check{e} \sqsubseteq H'; H'' \mid \check{e}'}$$

$$\boxed{\check{\mathcal{P}} \sqsubseteq \mathcal{H} \mid \eta \vdash H \sqsubseteq H : \eta}$$

$$\frac{}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta_0 \vdash H \sqsubseteq H' : \varnothing} \qquad \frac{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta_0 \vdash H_1 \sqsubseteq H_1' : \eta_1 \qquad \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta_0 \vdash H_2 \sqsubseteq H_2' : \eta_2}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta_0 \vdash H_1; H_2 \sqsubseteq H_1'; H_2' : \eta_1, \eta_2}$$

$$\frac{\forall i.\ \eta_0 \vdash v_{x;i} \sqsubseteq v'_{x;i} \qquad \forall i.\ \eta_0 \vdash v_{f;i} \sqsubseteq v'_{f;i}}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta_0 \vdash \ell \mapsto C(v_{x;1}, \ldots)\{v_{f;1}, \ldots\} \sqsubseteq \ell' \mapsto C(v'_{x;1}, \ldots)\{v'_{f;1}, \ldots\} : \ell \sqsubseteq \ell'}$$

$$\frac{\begin{array}{c} C \le I_1, \ldots \in \mathcal{H}_{\check{\mathcal{P}}} \qquad x : C(x_1, \ldots)\{f_1 := e_1; \ldots \mid m_1 \check{b}_1; \ldots\} \in \mathcal{I}_{\check{\mathcal{P}}} \\ \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta_0 \vdash \ell \mapsto \{f_1 \mapsto_{\mathsf{init}} v_1; \ldots \mid m_1 \check{b}_1[x \mapsto \ell]; \ldots\}^{\mathsf{fix}}_{I_1, \ldots} \sqsubseteq H' \mid \ell' \mapsto h' : \eta \end{array}}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta_0 \vdash \ell \mapsto C(v_{x;1}, \ldots)\{v_{f;1}, \ldots\} \sqsubseteq H'; \ell' \mapsto h' : \eta, \ell \sqsubseteq \ell'}$$

$$\frac{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta_0 \vdash \ell \mapsto \{\mid \lambda \check{b}\}^{\mathsf{fix}}_{\iota} \sqsubseteq H' \mid \ell' \mapsto h' : \eta}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta_0 \vdash \ell \mapsto \lambda_\iota \check{b} \sqsubseteq H'; \ell' \mapsto h' : \eta, \ell \sqsubseteq \ell'}$$

$$\frac{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta_0 \vdash \ell \mapsto \{f_1 \mapsto_{\mu_1} v_1; \ldots \mid m_1 \check{b}_1; \ldots\}^{\mathsf{ext}}_{\iota} \sqsubseteq H' \mid \ell' \mapsto h' : \eta}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta_0 \vdash \ell \mapsto \{f_1 \mapsto_{\mu_1} v_1; \ldots \mid m_1 \check{b}_1; \ldots\}_{\iota} \sqsubseteq H'; \ell' \mapsto h' : \eta, \ell \sqsubseteq \ell'}$$

$$\boxed{\check{\mathcal{P}} \sqsubseteq \mathcal{H} \mid \eta \vdash \ell \mapsto \{f \mapsto_{\mu} \check{e}; \ldots \mid m\check{b}; \ldots\}^{\chi}_{\iota} \sqsubseteq H \mid \ell \mapsto h : \eta}$$

$$\frac{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta_0 \vdash \check{b} \sqsubseteq H' \mid \check{b}' \qquad \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \vdash \iota \sqsubseteq \iota'}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta_0 \vdash \ell \mapsto \{\mid \lambda \check{b}\}^{\mathsf{fix}}_{\iota} \sqsubseteq H' \mid \ell' \mapsto \lambda_{\iota'} \check{b}' : \varnothing}$$

$$\frac{\begin{array}{c} \forall i.\ \eta_0 \vdash v_i \sqsubseteq v'_i \\ \forall i.\ \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta_0 \vdash \check{b}_i \sqsubseteq H'_i \mid \check{b}'_i \qquad \forall i.\ \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta_0 \vdash \ell_{m;i} \mapsto \{\mid \lambda b_{m;i}\}^{\mathsf{fix}}_{\varnothing} \sqsubseteq \varnothing \mid \ell'_{m;i} \mapsto h'_{m;i} : \varnothing \\ \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \vdash \iota \sqsubseteq \iota' \qquad H' = H'_1; \ldots; \ell'_{m;1} \mapsto h'_{m;1}; \ldots \end{array}}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta_0 \vdash \begin{array}{c} \ell \mapsto \{f_1 \mapsto_{\mu_1} v_1; \ldots \mid m_1 \check{b}_1; \ldots; f_{m;1} \check{b}_{m;1}; \ldots\}^{\chi}_{\iota} \\ \sqsubseteq \\ H' \mid \ell' \mapsto \{f_1 \mapsto_{\mu_1} v'_1; \ldots; f_{m;1} \mapsto_{\mathsf{init}} \ell'_{m;1}; \ldots \mid m_1 \check{b}'_1; \ldots\}_{\iota'} \end{array} : \langle \ell.f_{m;1} \rangle \sqsubseteq \ell'_{m;1}, \ldots}$$

$$\frac{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta_0 \vdash \ell \mapsto \{f_1 \mapsto_{\mu_1} v_1; \ldots \mid m_1 \check{b}_1; \ldots; m'\check{b}'; m\check{b}; m'_1 \check{b}'_1; \ldots\}^{\chi}_{\iota} \sqsubseteq H' \mid \ell' \mapsto h' : \eta}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta_0 \vdash \ell \mapsto \{f_1 \mapsto_{\mu_1} v_1; \ldots \mid m_1 \check{b}_1; \ldots; m\check{b}; m'\check{b}'; m'_1 \check{b}'_1; \ldots\}^{\chi}_{\iota} \sqsubseteq H' \mid \ell' \mapsto h' : \eta}$$

$$\frac{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta_0 \vdash \ell \mapsto \{f_1 \mapsto_{\mu_1} v_1; \ldots; f' \mapsto_{\mu'} v'; f \mapsto_{\mu} v; f'_1 \mapsto_{\mu'_1} v'_1; \ldots \mid m_1 \check{b}_1; \ldots\}^{\chi}_{\iota} \sqsubseteq H' \mid \ell' \mapsto h' : \eta}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta_0 \vdash \ell \mapsto \{f_1 \mapsto_{\mu_1} v_1; \ldots; f \mapsto_{\mu} v; f' \mapsto_{\mu'} v'; f'_1 \mapsto_{\mu'_1} v'_1; \ldots \mid m_1 \check{b}_1; \ldots\}^{\chi}_{\iota} \sqsubseteq H' \mid \ell' \mapsto h' : \eta}$$

Fig. A.24. Heap Precision

$\boxed{\eta \vdash v \sqsubseteq v}$

$$\frac{}{\eta \vdash \texttt{false} \sqsubseteq \texttt{false}} \qquad \frac{}{\eta \vdash \texttt{true} \sqsubseteq \texttt{true}} \qquad \frac{\ell \sqsubseteq \ell' \in \eta}{\eta \vdash \ell \sqsubseteq \ell'} \qquad \frac{\ell \sqsubseteq \ell' \in \eta}{\eta \vdash \langle \ell.f \rangle \sqsubseteq \langle \ell'.f \rangle} \qquad \frac{\langle \ell.f \rangle \sqsubseteq \ell' \in \eta}{\eta \vdash \langle \ell.f \rangle \sqsubseteq \ell'}$$

$\boxed{\check{\mathcal{P}} \sqsubseteq \mathcal{H} \mid \eta \vdash \check{e} \sqsubseteq H \mid \check{e}}$

$$\frac{}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash x \sqsubseteq \varnothing \mid x} \qquad \frac{\eta \vdash v \sqsubseteq v'}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash v \sqsubseteq \varnothing \mid v'}$$

$$\frac{\forall i.\ \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e}_i \sqsubseteq H_i' \mid \check{e}_i' \qquad \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e}[x_1 \mapsto x_1', \ldots] \sqsubseteq H' \mid \check{e}'}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \texttt{let}\ \langle x_1, \ldots \rangle := \langle \check{e}_1, \ldots \rangle\ \texttt{in}\ \check{e} \sqsubseteq H_1'; \ldots; H' \mid \texttt{let}\ \langle x_1', \ldots \rangle := \langle \check{e}_1', \ldots \rangle\ \texttt{in}\ \check{e}'}$$

$$\frac{\begin{array}{c}\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e}_1 \sqsubseteq H_1' \mid \check{e}_1' \\ \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e}_2 \sqsubseteq H_2' \mid \check{e}_2'\end{array}}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e}_1 == \check{e}_2 \sqsubseteq H_1'; H_2' \mid \check{e}_1' == \check{e}_2'} \qquad \frac{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e} \sqsubseteq H' \mid \check{e}' \qquad \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \vdash \delta \sqsubseteq \delta'}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e}.f^{\delta} \sqsubseteq H' \mid \check{e}'.f^{\delta'}}$$

$$\frac{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e} \sqsubseteq H' \mid \check{e}' \qquad \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \vdash \delta \sqsubseteq \delta' \qquad \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e}_f \sqsubseteq H_f' \mid \check{e}_f'}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e}.f^{\delta} := \check{e}_f \sqsubseteq H'; H_f' \mid \check{e}'.f^{\delta'} := \check{e}_f'}$$

$$\frac{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e} \sqsubseteq H' \mid \check{e}' \qquad \forall i.\ \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e}_i \sqsubseteq H_i' \mid \check{e}_i' \qquad \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \vdash \delta \sqsubseteq \delta'}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e}.m(\check{e}_1, \ldots)^{\delta} \sqsubseteq H'; H_1'; \ldots \mid \check{e}'.m(\check{e}_1', \ldots)^{\delta'}}$$

$$\frac{\forall i.\ \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e}_i \sqsubseteq H_i' \mid \check{e}_i'}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \texttt{new}\ C(\check{e}_1, \ldots) \sqsubseteq H_1'; \ldots \mid \texttt{new}\ C(\check{e}_1', \ldots)}$$

$$\frac{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{b} \sqsubseteq \check{b}' \mid H_\lambda' \qquad x \text{ is not free in } \check{b}' \qquad \vdash x := \{\mid \lambda \check{b}' \} \rightsquigarrow H' \mid \check{e}' : \chi'}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \texttt{new}\ \lambda\langle \check{b} \rangle \sqsubseteq H_\lambda'; H' \mid \check{e}'}$$

$$\frac{\forall i.\ \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e}_i \sqsubseteq H_{f;i}' \mid \check{e}_i' \qquad \forall i.\ \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{b}_i \sqsubseteq H_{m;i}' \mid \check{b}_i' \qquad \vdash x := \{f_1 := \check{e}_1; \ldots \mid m_1 \check{b}_1; \ldots\} \rightsquigarrow H' \mid \check{e}' : \text{ext}}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \texttt{new}\ x := \{f_1 := \check{e}_1; \ldots \mid m_1 \check{b}_1; \ldots\} \sqsubseteq H_{f;1}'; \ldots; H_{m;1}'; \ldots; H' \mid \check{e}'}$$

$$\frac{C(\tau_{x;1}, \ldots)\{f_1 : \tau_{f;1}; \ldots\} \in \mathcal{S} \qquad \forall i.\ \eta \vdash v_i \sqsubseteq v_i' \qquad \forall i.\ \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e}_i \sqsubseteq H_i' \mid \check{e}_i'}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash C(v_1, \ldots)\{\check{e}_1, \ldots\} \sqsubseteq H_1'; \ldots \mid C(v_1', \ldots)\{\check{e}_1', \ldots\}}$$

$$\frac{\mathcal{H}' \vdash \tau' \qquad \vdash \gamma \sqsubseteq \gamma' \qquad \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e} \sqsubseteq H' \mid \check{e}' \qquad \mathcal{H}_{\check{\mathcal{P}}} \vdash \tau \blacktriangleleft \tau'}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \texttt{cast}^{\gamma}\ \check{e}\ \texttt{to}\ \tau \sqsubseteq H' \mid \texttt{cast}^{\gamma'}\ \check{e}'\ \texttt{to}\ \tau'}$$

$$\frac{\vdash \gamma \sqsubseteq \gamma' \qquad \eta \vdash \ell \sqsubseteq \ell' \qquad \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e} \sqsubseteq H' \mid \check{e}'}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \texttt{impose}^{\gamma}\ \ell.m\ \texttt{on}\ \check{e} \sqsubseteq H' \mid \texttt{impose}^{\gamma'}\ \ell'.m\ \texttt{on}\ \check{e}'}$$

Fig. A.25. Lowered-Expression Precision (Part I)

similarly there is the rule related a named method invocation to an untyped field access followed by an untyped λ-invocation. Second, there is the rule relating `impose` to a trivial `cast` so that the result of reducing typed invocations correlates with the result of reducing untyped invocations. Third, the final rule arises from supporting dynamic subsumption, where the unnecessary bind and cast to a nominal supertype can be disregarded.

$$\frac{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e} \sqsubseteq H' \mid \check{e}' \qquad \forall i.\ \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e}_i \sqsubseteq H'_i \mid \check{e}'_i}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e}.f^{\delta_f}.\lambda(\check{e}_1, \ldots)^{\delta_\lambda} \sqsubseteq H'; H'_1; \ldots \mid \check{e}'.f(\check{e}'_1, \ldots)^{\langle \mathbf{dyn} \rangle}}$$

$$\frac{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e} \sqsubseteq H' \mid \check{e}' \qquad \forall i.\ \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e}_i \sqsubseteq H'_i \mid \check{e}'_i}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e}.f(\check{e}_1, \ldots)^{\delta} \sqsubseteq H'; H'_1; \ldots \mid \check{e}'.f^{\langle \mathbf{dyn} \rangle}.\lambda(\check{e}'_1, \ldots)^{\langle \mathbf{dyn} \rangle}}$$

$$\frac{\begin{array}{c}\neg\, \mathcal{H}' \vdash C \qquad \forall i.\ \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e}_i \sqsubseteq H'_i \mid \check{e}'_i \\ x : C(x_1, \ldots)\{f_1 \coloneqq \check{e}_{f;1}; \ldots \mid m_1 \check{b}_1; \ldots\} \in \check{\mathcal{I}}_{\check{\mathcal{P}}} \qquad \forall i.\ \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e}_{f;i}[x_1 \mapsto x'_1, \ldots] \sqsubseteq H'_{f;i} \mid \check{e}'_{f;i} \\ \forall i.\ \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{b}_i[x_1 \mapsto x'_1, \ldots] \sqsubseteq H'_{m;i} \mid \check{b}'_i \qquad \vdash x \coloneqq \{f_1 \coloneqq \check{e}'_{f;1}; \ldots \mid m_1 \check{b}'_1; \ldots\} \rightsquigarrow H' \mid \check{e}' : \chi\end{array}}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \mathsf{new}\, C(\check{e}_1, \ldots) \sqsubseteq H'_1; \ldots; H'_{f;1}; \ldots; H'_{m;1}; \ldots; H' \mid \mathtt{let}\ \langle x'_1, \ldots \rangle \coloneqq \langle \check{e}'_1, \ldots \rangle\ \mathtt{in}\ \check{e}'}$$

$$\frac{\begin{array}{c}\neg\, \mathcal{H}' \vdash C \qquad \forall i.\ \eta \vdash v_i \sqsubseteq v'_i \\ x : C(x_1, \ldots)\{f_1 \coloneqq \check{e}_{f;1}; \ldots \mid m_1 \check{b}_1; \ldots\} \in \check{\mathcal{I}}_{\check{\mathcal{P}}} \qquad \forall i.\ \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e}_i \sqsubseteq H'_{f;i} \mid \check{e}'_i \\ \forall i.\ \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{b}_i[x_1 \mapsto v_1, \ldots] \sqsubseteq H'_{m;i} \mid \check{b}'_i \qquad \vdash x \coloneqq \{f_1 \coloneqq \check{e}'_{f;1}; \ldots \mid m_1 \check{b}'_1; \ldots\} \rightsquigarrow H' \mid \check{e}' : \chi\end{array}}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash C(v_1, \ldots)\{\check{e}_1, \ldots\} \sqsubseteq H'_{f;1}; \ldots; H'_{m;1}; \ldots; H' \mid \check{e}'}$$

$$\frac{\eta \vdash \ell \sqsubseteq \ell' \qquad \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e} \sqsubseteq H' \mid \check{e}'}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \mathsf{impose}^{\gamma}\ \ell.m\ \mathsf{on}\ \check{e} \sqsubseteq H' \mid \mathsf{cast}^{\gamma'}\ \check{e}'\ \mathsf{to}\ \mathbf{dyn}}$$

$$\frac{\begin{array}{c}\mathcal{H}' \vdash \tau' \qquad \vdash \gamma \sqsubseteq \gamma' \qquad \mathcal{H}_{\check{\mathcal{P}}} \vdash \tau \blacktriangleleft \tau' \\ \mathcal{H}_{\check{\mathcal{P}}} \vdash \tau \blacktriangleleft \tau_x \qquad \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e}_x \sqsubseteq H'_x \mid \check{e}'_x \qquad \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e}[x \mapsto x'] \sqsubseteq H' \mid \check{e}'\end{array}}{\begin{array}{r}\mathtt{let}\ \langle x_x \rangle \coloneqq \langle \mathsf{cast}^{\gamma}\ e_x\ \mathtt{to}\ \tau \rangle\ \mathtt{in}\ \mathtt{let}\ \langle x \rangle \coloneqq \langle \mathsf{cast}^{\gamma_x}\ x_x\ \mathtt{to}\ \tau_x \rangle\ \mathtt{in}\ \check{e} \\ \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \qquad \sqsubseteq \\ H'_x; H' \mid \mathtt{let}\ \langle x' \rangle \coloneqq \langle \mathsf{cast}^{\gamma'}\ e'_x\ \mathtt{to}\ \tau' \rangle\ \mathtt{in}\ \check{e}'\end{array}}$$

$$\boxed{\vdash x \coloneqq \{f \coloneqq \check{e}; \ldots \mid m\check{b}; \ldots\} \rightsquigarrow H \mid \check{e} : \chi}$$

$$\frac{x \text{ is not free in } \check{b}}{\vdash x \coloneqq \{\mid \lambda\check{b}\} \rightsquigarrow \varnothing \mid \mathsf{new}\, \lambda\langle\check{b}\rangle : \mathsf{fix}}$$

$$\frac{}{\vdash x \coloneqq \{f_1 \coloneqq \check{e}_1; \ldots \mid m_1\check{b}_1; \ldots\} \rightsquigarrow \varnothing \mid \mathsf{new}\, x' \coloneqq \{f_1 \coloneqq \check{e}_1; \ldots \mid m_1\check{b}_1[x \mapsto x']; \ldots\} : \mathsf{ext}}$$

$$\frac{\vdash x \coloneqq \{f_1 \coloneqq \check{e}_1; \ldots \mid m_1\check{b}_1; \ldots\} \rightsquigarrow H \mid \check{e} : \mathsf{ext}}{\vdash x \coloneqq \{f_1 \coloneqq \check{e}_1; \ldots; f \coloneqq \check{e}_f \mid m_1\check{b}_1; \ldots\} \rightsquigarrow H \mid \check{e}.f^{\langle \mathbf{dyn} \rangle} \coloneqq \check{e}_f : \mathsf{ext}}$$

$$\frac{\begin{array}{c}x \text{ is not free in } \check{b} \\ \vdash x \coloneqq \{\mid \lambda\check{b}\} \rightsquigarrow H_f \mid \check{e}_f : \chi_f \qquad \vdash x \coloneqq \{f_1 \coloneqq \check{e}_1; \ldots; f \coloneqq \check{e}_f \mid m_1\check{b}_1; \ldots\} \rightsquigarrow H \mid \check{e} : \chi\end{array}}{\vdash x \coloneqq \{f_1 \coloneqq \check{e}_1; \ldots \mid m_1\check{b}_1; \ldots; f\check{b}\} \rightsquigarrow H_f; H \mid \check{e} : \chi}$$

$$\frac{\vdash x \coloneqq \{f_1 \coloneqq e_1; \ldots \mid m_1\check{b}_1; \ldots; m'\check{b}'; m\check{b}; m'_1\check{b}'_1; \ldots\} \rightsquigarrow H \mid e : \chi}{\vdash x \coloneqq \{f_1 \coloneqq e_1; \ldots \mid m_1\check{b}_1; \ldots; m\check{b}; m'\check{b}'; m'_1\check{b}'_1; \ldots\} \rightsquigarrow H \mid e : \chi}$$

$$\frac{x \text{ is not free in } \check{b}}{\vdash x \coloneqq \{\mid \lambda\check{b}\} \rightsquigarrow \ell \mapsto \lambda_{\varnothing}\check{b} \mid \ell : \mathsf{fix}}$$

$$\frac{}{\vdash x \coloneqq \{f_1 \coloneqq v_1; \ldots \mid m_1\check{b}_1; \ldots\} \rightsquigarrow \ell \mapsto \{f_1 \mapsto_{\mathsf{init}} v_1; \ldots \mid m_1\check{b}_1[x \mapsto \ell]; \ldots\}_{\varnothing} \mid \ell : \mathsf{ext}}$$

Fig. A.25. Lowered-Expression Precision (Part II)

$$\boxed{\check{\mathcal{P}} \sqsubseteq \mathcal{H} \mid \eta \vdash \check{b} \sqsubseteq H \mid \check{b}}$$

$$\frac{\mathcal{H}' \vdash \tau' \qquad \forall i.\ \vdash \tau_i \sqsubseteq \tau_i' \qquad \forall i.\ \mathcal{H}' \vdash \tau_i' \qquad \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash \check{e}[x_1 \mapsto x_1', \ldots] \sqsubseteq H' \mid \check{e}' \qquad \vdash \tau \sqsubseteq \tau'}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \mid \eta \vdash (x_1 : \tau_1, \ldots) \mapsto \check{e} : \tau \sqsubseteq H' \mid (x_1' : \tau_1', \ldots) \mapsto \check{e}' : \tau'}$$

$$\boxed{\check{\mathcal{P}} \sqsubseteq \mathcal{H} \vdash \delta \sqsubseteq \delta}$$

$$\frac{\mathcal{H}_{\check{\mathcal{P}}} \sqsubseteq \mathcal{H}' \vdash \tau :> \tau'}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \vdash \langle \tau \rangle \sqsubseteq \langle \tau' \rangle}$$

$$\boxed{\check{\mathcal{P}} \sqsubseteq \mathcal{H} \vdash \iota \sqsubseteq \iota}$$

$$\frac{}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \vdash \iota \sqsubseteq \varnothing} \qquad \frac{\mathcal{H}' \vdash I' \qquad \check{\mathcal{P}} \sqsubseteq \mathcal{H}' \vdash I_1, \ldots \sqsubseteq \iota' \qquad \mathcal{H}_{\check{\mathcal{P}}} \vdash I_i \leq I'}{\check{\mathcal{P}} \sqsubseteq \mathcal{H}' \vdash I_1, \ldots \sqsubseteq \iota', I'}$$

$$\boxed{\vdash \gamma \sqsubseteq \gamma} \qquad \frac{}{\vdash \gamma \sqsubseteq \gamma} \qquad \frac{}{\vdash \gamma \sqsubseteq \mathbf{dyn}}$$

Fig. A.25. Lowered-Expression Precision (Part III)

A.9.4 The Dynamic Gradual Guarantee. We repeat the statement of the dynamic gradual guarantee MonNom ensures, this time with proof.

Theorem 5.3 (Dynamic Gradual Guarantee). *For all programs satisfying* $\vdash \mathcal{P}$, $\vdash \mathcal{P}'$, *and* $\vdash \mathcal{P} \sqsubseteq \mathcal{P}'$, *any observation satisfying* $\vdash \mathcal{P} \twoheadrightarrow o$ *either also satisfies* $\vdash \mathcal{P}' \twoheadrightarrow o$ *or is* **error**; *and for any observation satisfying* $\vdash \mathcal{P}' \twoheadrightarrow o$, *either* $\vdash \mathcal{P} \twoheadrightarrow o$ *also holds or* $\vdash \mathcal{P} \twoheadrightarrow$ **error** *holds.*

Proof. By the following Lemma A.6, the two programs necessarily lower to related lowered programs. Furthermore, by the previous Lemma A.3, these lowered programs are necessarily typed, and so by Lemma A.5 the err if and only if they get stuck.

By the following Lemma A.7, if the main expression of the more-precise program steps then, after possibly a few more steps, so does the main expression of the less-precise program, eventually reaching a state where the two are related again. Thus if the more-precise program reduces to a value or steps forever, then the less-precise program must likewise reduce to the same value or step forever. But if the more-precise program gets stuck, although it necessarily errs, the less-precise program might still continue to reduce.

On the other side, by the following Lemma A.8, if the main expression of the less-precise program steps then, after possibly a few more steps, so does the main expression of the more-precise program, eventually reaching a state where the two are related again—unless it gets stuck. Thus if the less-precise program reduces to a value or steps forever, then the more-precise program must likewise reduce to the same value or step forever—unless it gets stuck, i.e. errs. □

Lemma A.6. *All programs and lowered programs satisfying* $\vdash \mathcal{P} \sqsubseteq \mathcal{P}'$, $\vdash \mathcal{P} \rightsquigarrow \check{\mathcal{P}}$, *and* $\vdash \mathcal{P}' \rightsquigarrow \check{\mathcal{P}}'$ *furthermore satisfy* $\vdash \check{\mathcal{P}} \sqsubseteq \check{\mathcal{P}}'$.

Proof. One can easily prove the corresponding lemma for lowering related expressions by induction on the proof of expression precision. The proof for lowering related programs is a straightforward corollary of that lemma. □

The following lemmas make use of the judgements defined in Figure A.26. The issue is that more-precise and less-precise expressions do not reduce in lock-step. For example, when one uses a named method invocation where the other uses a field access followed by a λ-invocation, the former takes one step where the latter takes two. Figure A.26 defines judgements to express that two related states will always eventually synchronize again. The first judgement simply steps a state one

$$\boxed{\check{\mathcal{P}} \vdash H \mid \check{e} \xrightarrow{+} H \mid \check{e}}$$

$$\frac{\check{\mathcal{P}} \vdash H \mid \check{e} \to H' \mid \check{e}'}{\check{\mathcal{P}} \vdash H \mid \check{e} \xrightarrow{+} H' \mid \check{e}'} \qquad \frac{\check{\mathcal{P}} \vdash H \mid \check{e} \to H' \mid \check{e}' \qquad \check{\mathcal{P}} \vdash H' \mid \check{e}' \xrightarrow{+} H'' \mid \check{e}''}{\check{\mathcal{P}} \vdash H \mid \check{e} \xrightarrow{+} H'' \mid \check{e}''}$$

$$\boxed{\check{\mathcal{P}} \sqsubseteq \check{\mathcal{P}} \vdash H \mid \check{e} \xrightarrow{*} \bullet \sqsubseteq H \mid \check{e}}$$

$$\frac{\check{\mathcal{P}}' \vdash H' \mid \check{e}' \xrightarrow{+} H_2' \mid \check{e}_2' \qquad \check{\mathcal{P}} \sqsubseteq \mathcal{H}_{\check{\mathcal{P}}'} \vdash H \mid \check{e} \sqsubseteq H_2' \mid \check{e}_2'}{\check{\mathcal{P}} \sqsubseteq \check{\mathcal{P}}' \vdash H \mid \check{e} \xrightarrow{*} \bullet \sqsubseteq H' \mid \check{e}'}$$

$$\frac{\begin{array}{c}\exists H_2, \check{e}_2.\ \check{\mathcal{P}} \vdash H \mid \check{e} \to H_2 \mid \check{e}_2 \\ \forall H_2, \check{e}_2.\ \check{\mathcal{P}} \vdash H \mid \check{e} \to H_2 \mid \check{e}_2 \implies \check{\mathcal{P}} \sqsubseteq \check{\mathcal{P}}' \vdash H_2 \mid \check{e}_2 \xrightarrow{*} \bullet \sqsubseteq H' \mid \check{e}'\end{array}}{\check{\mathcal{P}} \sqsubseteq \check{\mathcal{P}}' \vdash H \mid \check{e} \xrightarrow{*} \bullet \sqsubseteq H' \mid \check{e}'}$$

$$\boxed{\check{\mathcal{P}} \sqsubseteq \check{\mathcal{P}} \vdash H \mid \check{e} \xrightarrow{*} H \mid \check{e} \sqsubseteq \bullet}$$

$$\frac{\check{\mathcal{P}} \vdash H \mid \check{e} \xrightarrow{+} H_2 \mid \check{e}_2 \qquad \check{\mathcal{P}} \sqsubseteq \mathcal{H}_{\check{\mathcal{P}}'} \vdash H_2 \mid \check{e}_2 \sqsubseteq H' \mid \check{e}'}{\check{\mathcal{P}} \sqsubseteq \check{\mathcal{P}}' \vdash H' \mid \check{e}' \xrightarrow{*} H \mid \check{e} \sqsubseteq \bullet}$$

$$\frac{\begin{array}{c}\exists H_2', \check{e}_2'.\ \check{\mathcal{P}}' \vdash H' \mid \check{e}' \to H_2' \mid \check{e}_2' \\ \forall H_2', \check{e}_2'.\ \check{\mathcal{P}}' \vdash H' \mid \check{e}' \to H_2' \mid \check{e}_2' \implies \check{\mathcal{P}} \sqsubseteq \check{\mathcal{P}}' \vdash H_2' \mid \check{e}_2' \xrightarrow{*} H \mid \check{e} \sqsubseteq \bullet\end{array}}{\check{\mathcal{P}} \sqsubseteq \check{\mathcal{P}}' \vdash H' \mid \check{e}' \xrightarrow{*} H \mid \check{e} \sqsubseteq \bullet}$$

$$\boxed{\check{\mathcal{P}} \vdash H \mid \check{e} \xrightarrow{*} \varnothing}$$

$$\frac{\nexists v.\ \check{e} = v \qquad \forall H_2, \check{e}_2.\ \check{\mathcal{P}} \vdash H \mid \check{e} \to H_2 \mid \check{e}_2 \implies \check{\mathcal{P}} \vdash H_2 \mid \check{e}_2 \xrightarrow{*} \varnothing}{\check{\mathcal{P}} \vdash H \mid \check{e} \xrightarrow{*} \varnothing}$$

Fig. A.26. Eventual Refinement

or more times. The second judgement, *Always-Eventually Refines*, steps the state until it is in a state that is recognized as more precise than a state reachable by the less-precise state in one or more steps. The third judgement, *Always-Eventually Relaxes*, does the same with the precision relation essentially flipped. Finally, the last judgement, *Always-Eventually Sticks*, describes states that will always eventually get stuck. Each of these judgements is inductive (rather than coinductive), which avoids the possibility of one program diverging without the other ever matching up or getting stuck.

Lemma A.7. *For all lowered programs, heaps, heap correspondences, and lowered expressions satisfying* $\vdash \check{\mathcal{P}}$, $\vdash \check{\mathcal{P}}'$, *and* $\check{\mathcal{P}} \sqsubseteq \mathcal{H}_{\check{\mathcal{P}}'} \vdash H_1 \mid \check{e}_1 \sqsubseteq H_1' \mid \check{e}_2'$, *the following implication holds:*

$$\forall H_2, \check{e}_2.\ \check{\mathcal{P}} \vdash H_1 \mid \check{e}_1 \to H_2 \mid \check{e}_2 \implies \check{\mathcal{P}} \sqsubseteq \check{\mathcal{P}}' \vdash H_2 \mid \check{e}_2 \xrightarrow{*} \bullet \sqsubseteq H_1' \mid \check{e}_1'$$

Proof. One can do induction on the proof of $\check{\mathcal{P}} \sqsubseteq \mathcal{H}_{\check{\mathcal{P}}'} \vdash H_1 \mid \check{e}_1 \sqsubseteq H_1' \mid \check{e}_2'$ to extract proofs $\check{\mathcal{P}} \sqsubseteq \mathcal{H}_{\check{\mathcal{P}}'} \mid \eta \vdash H_\propto \sqsubseteq H_\ell' : \eta$ and $\check{\mathcal{P}} \sqsubseteq \mathcal{H}_{\check{\mathcal{P}}'} \mid \eta \vdash \check{e}_1 \sqsubseteq H_r' \mid \check{e}_1'$, where $H_\propto$ is a reordering of H_1, and $H_\ell'; H_r'$ is a reordering of H_1'. Then, by induction on the proof of $\check{\mathcal{P}} \sqsubseteq \mathcal{H}_{\check{\mathcal{P}}'} \mid \eta \vdash \check{e}_1 \sqsubseteq H_r' \mid \check{e}_1'$, it is

relatively easy to prove—despite the number of cases involved—that a reduction step of $\check{e}_1$ always eventually leads to a refinement of some multi-step reduction of $\check{e}'_1$. □

Lemma A.8. *For all lowered programs, heaps, heap correspondences, and lowered expressions satisfying* $\vdash \check{\mathcal{P}}$, $\vdash \check{\mathcal{P}}'$, *and* $\check{\mathcal{P}} \sqsubseteq \mathcal{H}_{\check{\mathcal{P}}'} \vdash H_1 \mid \check{e}_1 \sqsubseteq H'_1 \mid \check{e}'_2$, *the following implication holds:*

$$\forall H'_2, \check{e}'_2.\ \check{\mathcal{P}}' \vdash H'_1 \mid \check{e}'_1 \longrightarrow H'_2 \mid \check{e}'_2 \implies \check{\mathcal{P}} \sqsubseteq \check{\mathcal{P}}' \vdash H'_2 \mid \check{e}'_2 \xrightarrow{*} H_1 \mid \check{e}_1 \sqsubseteq \bullet \vee \check{\mathcal{P}} \vdash H_1 \mid \check{e}_1 \xrightarrow{*} \varnothing$$

Proof. One can do induction on the proof of $\check{\mathcal{P}} \sqsubseteq \mathcal{H}_{\check{\mathcal{P}}'} \vdash H_1 \mid \check{e}_1 \sqsubseteq H'_1 \mid \check{e}'_2$ to extract proofs $\check{\mathcal{P}} \sqsubseteq \mathcal{H}_{\check{\mathcal{P}}'} \mid \eta \vdash H_\propto \sqsubseteq H'_\ell : \eta$ and $\check{\mathcal{P}} \sqsubseteq \mathcal{H}_{\check{\mathcal{P}}'} \mid \eta \vdash \check{e}_1 \sqsubseteq H'_r \mid \check{e}'_1$, where $H_\propto$ is a reordering of H_1, and $H'_\ell; H'_r$ is a reordering of H'_1. Then, by induction on the proof of $\check{\mathcal{P}} \sqsubseteq \mathcal{H}_{\check{\mathcal{P}}'} \mid \eta \vdash \check{e}_1 \sqsubseteq H'_r \mid \check{e}'_1$, it is relatively easy to prove—despite the number of cases involved—that a reduction step of $\check{e}'_1$ either always eventually leads to a relaxation of some multi-step reduction of $\check{e}_1$ unless $\check{e}_1$ always eventually gets stuck. □

A.9.5 Determinism. Lastly, we prove that our calculus is deterministic.

Theorem 5.1 (Determinism). *For all programs satisfying* $\vdash \mathcal{P}$, *any two observations satisfying* $\vdash \mathcal{P} \twoheadrightarrow o$ *and* $\vdash \mathcal{P} \twoheadrightarrow o'$ *are necessarily equal.*

Proof. Because program precision is reflexive, a consequence of Lemma A.6 is that all lowerings of a program are refinements of each other. By Lemma A.8, that in turn implies all lowerings of a program are semantically equivalent. For the general purposes of gradual typing, that is enough. But to simplify the presentation of our calculus and results in the main body of the paper, we made a stronger claim: that the semantics of MonNom was deterministic. So, in addition to the above reasoning, we need the following Lemma A.9, from which remainder of the proof easily follows. □

Lemma A.9. *All lowered programs, heaps, and lowered expressions satisfying* $\check{\mathcal{P}} \vdash H_1 \mid \check{e}_1 \longrightarrow H_2 \mid \check{e}_2$ *and* $\check{\mathcal{P}} \vdash H_1 \mid \check{e}_1 \longrightarrow H'_2 \mid \check{e}'_2$ *furthermore satisfy* $\check{\mathcal{P}} \sqsubseteq H_2 \vdash \check{e}_2 \mid H'_2 \sqsubseteq \check{e}'_2 \mid$.

Proof. This is a straightforward proof by case. □

B BENCHMARK SOURCE CODE

This appendix contains the full source listings of the benchmarks for files that are not considered standard library material and are actively varied by our benchmark generator.

B.1 sieve

Listing 1. Main.mn—Fully Untyped

```
class Main {
    public static fun CountFrom(dyn n) : dyn {
        dyn rest = () => {
            dyn next = n + 1;
            return Main.CountFrom(next);
        } : dyn;
        return new(dyn First = n, dyn Rest = rest) { };
    }
    public static fun Sift(dyn n, dyn s) : dyn {
        dyn first = s.First;
        if ( first % n == 0 ) {
            return Main.Sift(n, let dyn fn = s.Rest in fn());
        } else {
            dyn rest = () => {
```

```
                return Main.Sift(n, let dyn fn = s.Rest in fn());
            } : dyn;
            return new(dyn First = first, dyn Rest = rest) { };
        }
    }
    public static fun Sieve(dyn s) : dyn {
        dyn first = s.First;
        dyn rest = () => {
            return Main.Sieve(Main.Sift(first, let dyn fn = s.Rest in fn()));
        } : dyn;
        return new(dyn First = first, dyn Rest = rest) { };
    }
    public static fun GetPrimes() : dyn {
        return Main.Sieve(Main.CountFrom(2));
    }
    public static fun Main() : dyn {
        dyn timer = new Timer();
        dyn prime = Stream.Get(Main.GetPrimes(), 9999);
        timer.PrintDifference();
        "\n".Print();
        prime.ToString().Print();
        "\n".Print();
    }
}
```

Listing 2. Streams.mn—Fully Untyped

```
class Stream {
    public static fun Get(dyn s, dyn n) : dyn {
        while ( n > 0 ) {
            n = n - 1;
            s = s.Rest();
        }
        return s.First;
    }
}
```

Listing 3. Main.mn—Fully Typed

```
interface IFun<T> {
    public fun this() : T;
}
class CountFromFun implements IFun<Stream> {
    private readonly Int N;
    public constructor(Int n) {
        N = n;
        super();
    }
    public fun this() : Stream {
        Int next = N + 1;
        return Main.CountFrom(next);
    }
}
```

```
class SiftFun implements IFun<Stream> {
    private readonly Int N;
    private readonly Stream S;
    public constructor(Int n, Stream s) {
        N = n;
        S = s;
        super();
    }
    public fun this() : Stream {
        return Main.Sift(N, let IFun<Stream> fn = S.Rest in fn());
    }
}
class SieveFun implements IFun<Stream> {
    private readonly Int N;
    private readonly Stream S;
    public constructor(Int n, Stream s) {
        N = n;
        S = s;
        super();
    }
    public fun this() : Stream {
        return Main.Sieve(Main.Sift(N, let IFun<Stream> fn = S.Rest in fn()));
    }
}
class Main {
    public static fun CountFrom(Int n) : Stream {
        return new Stream(n, new CountFromFun(n));
    }
    public static fun Sift(Int n, Stream s) : Stream {
        Int first = s.First;
        if ( first % n == 0 ) {
            return Main.Sift(n, let IFun<Stream> fn = s.Rest in fn());
        } else {
            return new Stream(first, new SiftFun(n, s));
        }
    }
    public static fun Sieve(Stream s) : Stream {
        Int first = s.First;
        return new Stream(first, new SieveFun(first, s));
    }
    public static fun GetPrimes() : Stream {
        return Main.Sieve(Main.CountFrom(2));
    }
    public static fun Main() : Void {
        Timer timer = new Timer();
        Int prime = Stream.Get(Main.GetPrimes(), 9999);
        timer.PrintDifference();
        "\n".Print();
        prime.ToString().Print();
        "\n".Print();
    }
}
```

Listing 4. Streams.mn—Fully Typed

```
class Stream {
    public readonly Int First;
    public readonly IFun<Stream> Rest;
    public constructor(Int first, IFun<Stream> rest) {
        First = first;
        Rest = rest;
        super();
    }
    public static fun Get(Stream s, Int n) : Int {
        while ( n > 0 ) {
            n = n - 1;
            s = s.Rest();
        }
        return s.First;
    }
}
```

B.2 intersort

Listing 5. Sort.mn—Fully Untyped

```
class Sort {
    public static fun Quicksort(dyn list) : dyn {
        dyn loIter = list.GetIterator();
        dyn hiIter = list.GetIterator();
        if ( loIter.MoveNext() ) {
            hiIter.MoveNext();
            dyn lo = 0;
            dyn hi = 0;
            while ( hiIter.MoveNext() ) {
                hi = hi + 1;
            }
            QuicksortRec(loIter, hiIter, lo, hi);
        }
    }
    private static fun QuicksortRec(dyn loIter, dyn hiIter, dyn lo, dyn hi) : dyn {
        if ( lo < hi ) {
            dyn upper = hiIter.Clone();
            dyn lower = loIter.Clone();
            dyn losize = Partition(lower, upper, hi - lo);
            QuicksortRec(loIter, upper, lo, lo + losize - 1);
            QuicksortRec(lower, hiIter, lo + losize, hi);
        }
    }
    private static fun Partition(dyn loIter, dyn hiIter, dyn distance) : dyn {
        dyn pivot = loIter.Current();
        dyn losize = 0;
        while ( true ) {
            while ( loIter.Current() < pivot ) {
                loIter.MoveNext();
                distance = distance - 1;
```

```
                losize = losize + 1;
            }
            while ( hiIter.Current() > pivot ) {
                hiIter.MovePrev();
                distance = distance - 1;
            }
            if ( distance < 0 ) {
                break 0;
            }
            dyn buffer = loIter.Current();
            loIter.SetValue(hiIter.Current());
            hiIter.SetValue(buffer);
            loIter.MoveNext();
            losize = losize + 1;
            hiIter.MovePrev();
            distance = distance - 2;
        }
        return losize;
    }
}
```

Listing 6. Main.mn—Fully Untyped

```
class Main {
    public static fun Main() : dyn {
        dyn intlist = MakeIntList();
        Main.Test(intlist);
        dyn ilistIter = intlist.GetIterator();
        ilistIter.MoveNext();
        dyn last = ilistIter.Current();
        while ( ilistIter.MoveNext() ) {
            if ( ilistIter.Current() < last ) {
                ERROR("sorting␣failed");
            }
            last = ilistIter.Current();
        }
    }
    public static fun Test(dyn list) : dyn {
        dyn timer = new Timer();
        Sort.Quicksort(list);
        timer.PrintDifference();
    }
    public static fun MakeIntList() : dyn {
        dyn head = new(dyn value = 5, dyn next = this, dyn prev = this) {};
        dyn list = new(dyn first = head, dyn Size = 1) {
            public fun Add(dyn val) : dyn {
                dyn newNode = new(dyn value = val, dyn next = this, dyn prev = this) {};
                newNode.prev = this.first.prev;
                newNode.next = this.first;
                this.first.prev = newNode;
                newNode.prev.next = newNode;
                this.Size = this.Size + 1;
```

```
            }
            public fun GetIterator() : dyn {
                dyn iter = this.MakeIterator(this.first);
                return iter;
            }
            public fun GetSize() : dyn {
                return this.Size;
            }
            public fun MakeIterator(dyn node) : dyn {
                dyn self = this;
                return new(dyn currentNode = node, dyn parent = self) {
                    public fun MovePrev() : dyn {
                        if ( this.currentNode == parent.first ) {
                            return false;
                        }
                        this.currentNode = this.currentNode.prev;
                        return true;
                    }
                    public fun MoveNext() : dyn {
                        if ( this.currentNode.next == this.parent.first ) {
                            return false;
                        }
                        this.currentNode = this.currentNode.next;
                        return true;
                    }
                    public fun Current() : dyn {
                        return this.currentNode.value;
                    }
                    public fun SetValue(dyn val) : dyn {
                        this.currentNode.value = val;
                    }
                    public fun Clone() : dyn {
                        return this.parent.MakeIterator(this.currentNode);
                    }
                };
            }
        };
        dyn i = 0;
        while ( i < 100000 ) {
            dyn num = (i * 163841 + 176081) % 122251;
            list.Add(num);
            i = i + 1;
        }
        return list;
    }
}
```

Listing 7. List.mn—Fully Typed

```
interface IIterator<T> {
    public fun Current() : T;
    public fun Clone() : IIterator<T>;
```

```
    public fun MoveNext() : Bool;
    public fun MovePrev() : Bool;
    public fun SetValue(T val) : Void;
}
interface IList<T> {
    public fun GetIterator() : IIterator<T>;
    public fun GetSize() : Int;
    public fun Add(T val) : Void;
}
```

Listing 8. Sort.mn—Fully Typed

```
class Sort {
    public static fun Quicksort(IList<Int> list) : Void {
        IIterator<Int> loIter = list.GetIterator();
        IIterator<Int> hiIter = list.GetIterator();
        if ( loIter.MoveNext() ) {
            hiIter.MoveNext();
            Int lo = 0;
            Int hi = 0;
            while ( hiIter.MoveNext() ) {
                hi = hi + 1;
            }
            QuicksortRec(loIter, hiIter, lo, hi);
        }
    }
    private static fun QuicksortRec(IIterator<Int> loIter, IIterator<Int> hiIter, Int lo, Int hi)
     : Void {
        if ( lo < hi ) {
            IIterator<Int> upper = hiIter.Clone();
            IIterator<Int> lower = loIter.Clone();
            Int losize = Partition(lower, upper, hi - lo);
            QuicksortRec(loIter, upper, lo, lo + losize - 1);
            QuicksortRec(lower, hiIter, lo + losize, hi);
        }
    }
    private static fun Partition(IIterator<Int> loIter, IIterator<Int> hiIter, Int distance) :
     Int {
        Int pivot = loIter.Current();
        Int losize = 0;
        while ( true ) {
            while ( loIter.Current() < pivot ) {
                loIter.MoveNext();
                distance = distance - 1;
                losize = losize + 1;
            }
            while ( hiIter.Current() > pivot ) {
                hiIter.MovePrev();
                distance = distance - 1;
            }
            if ( distance < 0 ) {
                break 0;
```

```
            }
            Int buffer = loIter.Current();
            loIter.SetValue(hiIter.Current());
            hiIter.SetValue(buffer);
            loIter.MoveNext();
            losize = losize + 1;
            hiIter.MovePrev();
            distance = distance - 2;
        }
        return losize;
    }
}
```

Listing 9. ListImpl.mn—Fully Typed

```
class List<T> implements IList<T> {
    private ListNode first;
    private Int Size;
    public constructor(T val) {
        Size = 1;
        first = new ListNode(val);
        super();
    }
    public fun Add(T val) : Void {
        ListNode newNode = new ListNode(val);
        newNode.prev = this.first.prev;
        newNode.next = this.first;
        this.first.prev = newNode;
        newNode.prev.next = newNode;
        this.Size = this.Size + 1;
    }
    public fun GetIterator() : IIterator<T> {
        IIterator<T> iter = this.MakeIterator(this.first);
        return iter;
    }
    public fun GetSize() : Int {
        return this.Size;
    }
    class ListNode {
        public constructor(T val) {
            value = val;
            next = this;
            prev = this;
            super();
        }
        public T value;
        public ListNode next;
        public ListNode prev;
    }
    public fun MakeIterator(List<T>. ListNode node) : IIterator<T> {
        List<T> self = this;
        return new IteratorImpl(self, node);
```

```
    }
    class IteratorImpl implements IIterator<T> {
        private ListNode currentNode;
        private List<T> parent;
        public constructor(List<T> list, ListNode node) {
            currentNode = node;
            parent = list;
            super();
        }
        public fun MovePrev() : Bool {
            if ( this.currentNode == parent.first) {
                return false;
            }
            this.currentNode = this.currentNode.prev;
            return true;
        }
        public fun MoveNext() : Bool {
            if ( this.currentNode.next == this.parent.first) {
                return false;
            }
            this.currentNode = this.currentNode.next;
            return true;
        }
        public fun Current() : T {
            return this.currentNode.value;
        }
        public fun SetValue(T val) : Void {
            this.currentNode.value = val;
        }
        public fun Clone() : IIterator<T> {
            return this.parent.MakeIterator(this.currentNode);
        }
    }
}
```

Listing 10. Main.mn—Fully Typed

```
class Main {
    public static fun Main() : Void {
        IList<Int> intlist = MakeIntList();
        Main.Test(intlist);
        IIterator<Int> ilistIter = intlist.GetIterator();
        ilistIter.MoveNext();
        Int last = ilistIter.Current();
        while ( ilistIter.MoveNext() ) {
            if ( ilistIter.Current() < last ) {
                ERROR("sorting_failed");
            }
            last = ilistIter.Current();
        }
    }
    public static fun Test(IList<Int> list) : Void {
```

```
        Timer timer = new Timer();
        Sort.Quicksort(list);
        timer.PrintDifference();
    }

    public static fun MakeIntList() : IList<Int> {
        IList<Int> list = new List<Int>(5);
        Int i = 0;
        while ( i < 100000 ) {
            Int num = (i * 163841 + 176081) % 122251;
            list.Add(num);
            i = i + 1;
        }
        return list;
    }
}
```

B.3 float

Listing 11. Float.mn—Fully Untyped

```
class Main {
    public static fun Maximize(dyn points) : dyn {
        dyn next = points.Get(0);
        foreach (dyn point in Enumerable.From<dyn>(points, 1) ) {
            next = next.Maximize(point);
        }
        return next;
    }
    public static fun Benchmark(dyn n) : dyn {
        dyn points = Enumerable.ToList<dyn>(Enumerable.Map<Int, dyn>([0..n:1], (dyn i) => {
            dyn f = i + 0.0;
            dyn s = Math.Sin(f);
            return new(dyn x = s, dyn y = Math.Cos(f) * 3, dyn z = ( s * s ) / 2) {
                public fun Normalize() : dyn {
                    dyn norm = Math.Sqrt(x * x + y * y + z * z);
                    x = x / norm;
                    y = y / norm;
                    z = z / norm;
                }
                public fun Maximize(dyn other) : dyn {
                    if ( x < other.x ) {
                        x = other.x;
                    }
                    if ( y < other.y ) {
                        y = other.y;
                    }
                    if ( z < other.z ) {
                        z = other.z;
                    }
                    return this;
                }
```

```
                public fun Print() : dyn {
                    "<".Print();
                    x.ToString().Print();
                    ",␣".Print();
                    y.ToString().Print();
                    ",␣".Print();
                    z.ToString().Print();
                    ">␣".Print();
                }
            };
        } : dyn));
        foreach ( dyn point in points ) {
            point.Normalize();
        }
        return Maximize(points);
    }
    public static fun Main() : dyn {
        dyn timer = new Timer();
        dyn result = Benchmark(100000);
        timer.PrintDifference();
        "\n".Print();
        result.Print();
    }
}
```

Listing 12. Float.mn—Fully Typed

```
class Point {
    public Float x;
    public Float y;
    public Float z;
    public constructor(Float f) {
        Float s = Math.Sin(f);
        x = s;
        y = Math.Cos(f) * 3;
        z = ( s * s ) / 2;
        super();
    }
    public fun Normalize() : Void {
        Float norm = Math.Sqrt(x * x + y * y + z * z);
        x = x / norm;
        y = y / norm;
        z = z / norm;
    }
    public fun Maximize(Point other) : Point {
        if ( x < other.x ) {
            x = other.x;
        }
        if ( y < other.y ) {
            y = other.y;
        }
        if ( z < other.z ) {
```

```
            z = other.z;
        }
        return this;
    }
    public fun Print() : Void {
        "<".Print();
        x.ToString().Print();
        ",␣".Print();
        y.ToString().Print();
        ",␣".Print();
        z.ToString().Print();
        ">␣".Print();
    }
}
class PointMapFun implements Fun<Int, Point> {
    public constructor() {
        super();
    }
    public fun this(Int i) : Point {
        return new Point(i + 0.0);
    }
}
class Main {
    public static fun Maximize(ArrayList<Point> points) : Point {
        Point next = points.Get(0);
        foreach ( Point point in Enumerable.From<Point>(points, 1) ) {
            next = next.Maximize(point);
        }
        return next;
    }
    public static fun Benchmark(Int n) : Point {
        ArrayList<Point> points = Enumerable.ToList<Point>(Enumerable.Map<Int, Point>([0..n:1],
     new PointMapFun()));
        foreach ( Point point in points ) {
            point.Normalize();
        }
        return Maximize(points);
    }
    public static fun Main() : Void {
        Timer timer = new Timer();
        Point result = Benchmark(100000);
        timer.PrintDifference();
        "\n".Print();
        result.Print();
    }
}
```