The Simple Essence of Monomorphization

MATTHEW LUTZE, Aarhus University, Denmark
PHILIPP SCHUSTER, University of Tübingen, Germany
JONATHAN IMMANUEL BRACHTHÄUSER, University of Tübingen, Germany

Monomorphization is a common implementation technique for parametric type-polymorphism, which avoids the potential runtime overhead of uniform representations at the cost of code duplication. While important as a folklore implementation technique, there is a lack of general formal treatments in the published literature. Moreover, it is commonly believed to be incompatible with higher-rank polymorphism. In this paper, we formally present a simple monomorphization technique based on a type-based flow analysis that generalizes to programs with higher-rank types, existential types, and arbitrary combinations. Inspired by algebraic subtyping, we track the flow of type instantiations through the program. Our approach only supports monomorphization up to polymorphic recursion, which we uniformly detect as cyclic flow. Treating universal and existential quantification uniformly, we identify a novel form of polymorphic recursion in the presence of existential types, which we coin polymorphic packing. We study the meta-theory of our approach, showing that our translation is type-preserving and preserves semantics step-wise.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Polymorphism**; • **Theory of computation** → **Semantics and reasoning**.

Additional Key Words and Phrases: monomorphization, polymorphic recursion, polymorphic packing, higher-rank types, existential types

ACM Reference Format:
Matthew Lutze, Philipp Schuster, and Jonathan Immanuel Brachthäuser. 2025. The Simple Essence of Monomorphization. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 116 (April 2025), 27 pages. https://doi.org/10.1145/3720472

1 Introduction

Almost every statically typed programming language features some form of parametric type polymorphism. One common strategy for implementing parametric polymorphism is *monomorphization*, used for example in C++, Rust, Go [Griesemer et al. 2020], MLton [Cejtin et al. 2000; Weeks 2006], and Futhark [Hovgaard et al. 2018]. For each instantiation of a type parameter, monomorphization creates a specialized copy of the function, which avoids the runtime cost of heap allocation for boxing and opens up the potential for additional downstream optimizations. While important as a folklore implementation strategy, formal treatments in the published literature are rare.

Moreover, it is often believed that monomorphization is inherently incompatible with type system features that go beyond simple ML-style polymorphism, such as higher-rank polymorphism. Notably, several researchers in the field have made claims to that effect, such as

> *Higher-rank types cannot be monomorphized at all.*
> — Eisenberg and Peyton Jones [2017]

Authors' Contact Information: Matthew Lutze, Aarhus University, Denmark, mlutze@cs.au.dk; Philipp Schuster, University of Tübingen, Germany, philipp.schuster@uni-tuebingen.de; Jonathan Immanuel Brachthäuser, University of Tübingen, Germany, jonathan.brachthaeuser@uni-tuebingen.de.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2475-1421/2025/4-ART116
https://doi.org/10.1145/3720472

and

> *Furthermore, monomorphization cannot be done for [...] higher-rank (first-class polymorphic) functions.*
> — Oleg Kiselyov, 2021[1]

and

> *some polymorphism simply cannot be specialized statically (polymorphic recursion, first-class polymorphism)*
> — Kennedy and Syme [2001]

manifesting the common belief that it is not possible or at least very hard to monomorphize higher-rank polymorphism. We present a novel understanding of monomorphization, which is not only easy to understand but also generalizes to higher-rank and existential polymorphism. In contrast to previous work [Griesemer et al. 2020], which also monomorphizes higher-rank polymorphism, our approach leads to a straightforward implementation and almost trivial correctness proof. The key contribution of this paper is the following simple idea:

> We use a type-based flow analysis to find monomorphic specializations.
> We refer to this approach as *flow-directed monomorphization*.

We describe the analysis and the monomorphization to which it leads in more detail in Section 2, but want to highlight a few key insights here.

- Performing a type-based flow analysis allows us to factor the process of monomorphization into three cleanly separated phases (Section 2): a total constraint generation phase that tracks the flow of types through type variables, a fallible constraint solving phase, and a total monomorphization phase.
- We identify four kinds of polymorphism: *(1)* type-parametric functions (Section 2.1), *(2)* type-parametric data types and interfaces (Section 2.2), *(3)* type-parametric constructors, *i.e.*, existential polymorphism (Section 2.3), *(4)* type-parametric methods, *i.e.*, higher-rank polymorphism (Section 2.4).
- While kinds *(1)*, *(3)*, and *(4)* occur in terms, the second kind of polymorphism only occurs in types. Constraints for *(1)*, *(3)*, and *(4)* are introduced by our typing judgment, while constraints for *(2)* are introduced by our well-formedness judgment. These two categories of flow are not mutually recursive (Section 3.3).
- We identify variants of polymorphic recursion, which impede monomorphization, for each of the above-mentioned four kinds (Section 2.5).

The intended target audience of this paper is implementers of functional languages without subtyping. Our approach supports languages with or without higher-rank polymorphism and may apply to OCaml and Haskell, though we leave this to future work.

1.1 Challenges of Monomorphization

To understand the challenges of monomorphization in general and of higher-rank polymorphism specifically, consider the following small example in SystemF on the left. We define the identity function *id* and a function *appPair*, which takes an argument f and applies it to each of its other two arguments, x and y, returning a pair. Importantly, *appPair* is a higher-rank function, as the type of f is universally quantified.

[1] https://okmij.org/ftp/Computation/typeclass.html

Example in SystemF

```
let id = λA . λa : A . a
let appPair = ΛX . ΛY .
  λf : (∀ Z . Z → Z) .
    λx : X . λy : Y .
      (f X x, f Y y)

appPair Int Bool id 123 false
```

Possible monomorphization to STLC

```
let idInt = λa : Int . a
let idBool = λa : Bool . a
let appPair =
  λfInt : (Int → Int) .
    λfBool : (Bool → Bool) .
      λx : Int . λy : Bool .
        (fInt x, fBool y)

appPair idInt idBool 123 false
```

When we attempt to convert it to an equivalent program in the simply-typed lambda calculus, we notice that it is not immediately clear how to monomorphize *appPair*. Of course, we can easily specialize X and Y to Int and Bool, respectively, but what can be done about Z (and A respectively)? Clearly, f is applied to two different types in the body of *appPair*, so specializing the type Z to one type is not a valid solution. The astute reader might have come up with a monomorphic version of this particular program. One possible approach is shown on the right, where we generate two copies of the polymorphic identity function (specialized to Int and Bool respectively) and then change *appPair* to take these two monomorphic functions as separate arguments. However, this approach changes the arity of *appPair* and, for instance, prevents us from passing *appPair* itself as a function argument without also modifying the callee. While this small example already illustrates some problems, the difficulty grows with the complexity of the program under consideration, further underlining the intuition that it is very hard to monomorphize higher-rank polymorphism.

Agreeing with many before us, inspecting the *appPair* example, we see that monomorphizing SystemF programs to STLC is indeed challenging. In this paper, we slightly shift the goalposts by:

(1) not using SystemF as the source language of monomorphization, and
(2) not using STLC as the target language of monomorphization.

This reframing is based on the observation that passing two different versions of *id* is difficult in STLC, but would be much easier if the target language supported *objects*. Instead of passing two functions, we can simply pass a single object with *two methods*.

We make the additional observation that structural typing in the source language complicates reasoning about monomorphization, artificially. Instead, if the source language expressed first-class functions nominally, it would be much easier to track the *flow of types* through these nominal types. Expressing function types in terms of nominal types is by now completely standard and common practice in languages like Scala and Java, which express the function type A → B in terms of an interface type `Function[A, B]` with a method apply : A → B.

Figure 1 presents a rendering of the example in our polymorphic source language LangPoly. Since we pass `id` as a first-class function, we represent it as an instance of the nominal interface `Id`. It is easy to see that `id.apply` must be specialized to Int and Bool; these are the types with which it is called at runtime. But how do we determine those types? For example, in order to determine which specializations are required for `A`, we simply follow the flow of types. Specifically, type `Int` flows into type parameter `X`, which flows into the type parameter `Z` on the interface `Id`. Since `id` implements the interface `Id`, all types that flow into `Z` also flow into `A`. Hence, transitively, `Int` flows into `A` and we require a specialization for it. Nominal types help us solve the problem of specializing `id` by adding two methods, one specialized to `Int` and one to `Bool`.

Example translated to LangPoly

```
trait Id {
  def apply[Z](Z): Z
}
let id = new Id {
  def apply[A](a) = a
}

def appPair[X, Y](f: Id, x: X, y: Y) =
  (f.apply[X](x), f.apply[Y](y))

appPair[Int, Bool](id, 123, false)
```

Result of monomorphization to LangMono

```
trait Id {
  def apply_Int(Int): Int
  def apply_Bool(Bool): Bool
}
let id = new Id {
  def apply_Int(a) = a
  def apply_Bool(a) = a
}

def appPair_IntBool(f: Id, x: Int, y: Bool) =
  (f.apply_Int(x), f.apply_Bool(y))

appPair_IntBool(id, 123, false)
```

Fig. 1. Introductory example presented in our source language LangPoly on the left, and the result of monomorphization to language LangMono on the right.

1.2 Monomorphizing Higher-Rank and Existential Types, and Their Combination

This flow analysis generalizes to large programs and allows us to monomorphize much more complex examples, like the one in Figure 2. Here, we define a stream parameterized by a universal element type `A` and an existential internal state type `S`. We then define a Church-encoded list of element type `D` whose `fold` operation is parametric over its result type `R`. We define an infinite stream `nats` over the natural numbers of type `Stream[Int]`. Finally, we define a `take` function to extract a prefix of a stream and a `sum` function over our Church-encoded list, which we call as follows to find the sum of the first three natural numbers.

```
sum(take[Int](nats, 3))
```

There are significant challenges to monomorphizing this program, including existential types in `Stream` and higher-rank types in `List`. Nevertheless, with the techniques described in this paper, we show that this program is monomorphizable. Moreover, it is actually not that difficult[2].

1.3 Non-goals

Although we present a type system, we are not interested in type inference or typechecking as such, but only as a means for supporting monomorphization. Therefore, we require fully type-annotated programs as input to our monomorphization. Of course, type inference could in principle be performed as a preprocessing step. Furthermore, monomorphization naturally involves whole-program analysis and code duplication; there may be complexity issues in the running time of the transformation and the size of the transformed code, but we do not consider those issues within the scope of this paper. Similarly, the code we generate might include more duplication than strictly necessary. Improving the precision of our analysis is interesting and important future work.

Finally, the subject of this work is the monomorphization of higher-rank types, and we do *not* claim to be able to monomorphize *all* programs. In particular, we identify the classes of programs which we cannot monomorphize, involving polymorphic recursion and related structures, discussed

[2]The fully monomorphized program can be found in Appendix A, submitted as supplementary material. Additionally, we make all of the examples of this and the following section available as supplementary material together with an interactive web editor that encourages further experimentation.

Type Definitions

```
enum Stream[A] {
  case Impl[ S ](state: S, next: S ⇒ Option[(A, S)])
}
trait List[D] {
  def fold[ R ](alg: Alg[D, R]): R
}
```

```
trait Alg[B, C] {
  def nil(): C
  def cons(head: B, tail: C): C
}
```

Program

```
let nats = Impl[Int](0, n ⇒ Some((n, n + 1)));

def take[E](s: Stream[E], n: Int): List[E] {
  new List[E] {
    def fold[T](alg: Alg[E, T]) { s match {
      case Impl[U](state, next) ⇒
        if (n ≤ 0) alg.nil()
        else next(state) match
          case None ⇒ alg.nil()
          case Some((value, newState)) ⇒
            let s2 = Impl[U](newState, next)
            let rest = take[E](s2, n - 1);
            alg.cons(value, rest.fold[T](alg))
    }}
  }
}
```

```
def sum(l: List[Int]): Int {
  l.fold[Int](new Alg[Int, Int] {
    def nil() { 0 }
    def cons(head, tail) {
      head + tail
    }
  })
}
```

Fig. 2. Motivational example combining the use of higher-rank polymorphism and existential types, highlighted in `gray`.

in Section 2.5. Instead, we advance the state of the art by describing a simple flow-based approach that supports a larger class of programs than commonly thought possible.

1.4 Contributions

- **Essence of Monomorphization**: We identify the *flow of types* as the guiding principle of monomorphization, which admits simple formalizations and implementations.
- **Monomorphizing higher-rank types and existentials**: The same guiding principle works for monomorphizing higher-rank types and existential types.
- **Polymorphic Packing**: Supporting both higher-rank types and existential types, we identify and discuss the problem of *polymorphic packing* as the dual to polymorphic recursion in a unifying framework.
- **Partially Streamable Monomorphization**: Our monomorphization strategy consists of three steps: gathering constraints, solving constraints, and applying the solution. Of these, only the constraint solving step requires information from the entire program to be in memory; the other steps can be performed as stream processing.
- **Calculi**: We define two languages: the polymorphic LangPoly and the corresponding monomorphic LangMono. For each, we define a semantics and type system. We further define formally a monomorphization strategy from LangPoly to LangMono.

- **Metatheory**: We demonstrate the critical properties of our transformation: it is type- and semantics-preserving. Putting the flow of types center stage, the proofs of these properties become almost trivial.
- **Implementation**: We implement monomorphization in a prototype based on LangPoly. The implementation supports all features in LangPoly, as well as practically motivated extensions, attesting to their implementability.

1.5 Limitations

- **Polymorphic Recursion**: Our monomorphization technique cannot be applied to programs featuring polymorphic recursion.
- **Whole-program Compilation**: As common for monomorphization, part of our technique requires access to the entire source of the program, meaning it cannot be performed modularly.
- **Overapproximation**: Due in part to our use of nominal types to represent higher-order and existential types, our monomorphization technique in some circumstances results in the generation of unused code.
- **Subtyping**: While our calculus includes higher-order and existential types, it does not support subtyping. Significant adaptations might be needed to apply our monomorphization technique to a language with subtyping.

1.6 Structure of the Paper

The rest of the paper is organized as follows: In Section 2, we describe our monomorphization technique through several examples. In Section 3, we present our source language LangPoly, our target language LangMono, monomorphization between them, and theorems. In Section 4, we describe our implementation, including extensions and differences from the formalism. In Section 5, we discuss related work.

2 Flow-Directed Monomorphization by Example

In this section, we provide an overview of our approach with examples of increasing complexity. We close this section with a demonstration of the limitations of our approach, providing examples that we cannot monomorphize.

2.1 Monomorphizing First-order Programs

Let us start with a simple first-order program with polymorphic function definitions. While the monomorphization of such programs is not challenging, doing so allows us to introduce the basic idea in a simple setting. Consider the following program, written in our polymorphic source language LangPoly with some minor syntactic extensions.

```
def first[A](x: A, y: A) { x }

def second[B](x: B, y: B) { first[B](y, x) }
```

```
def main() {
  second[Int](1, 2);
  second[String]("one", "two")
}
```

The example defines two polymorphic functions. Function `first` returns the first of its arguments, of type `A`. Function `second` is implemented in terms of `first` and is polymorphic in type `B`. For ease of presentation, we assume that all type variables that occur in the program are distinctly named. Finally, we define a function `main` that calls `second` at types `Int` and `String`. Abstraction as well as instantiation of types is explicit.

Our goal is to monomorphize this program. In other words, we want to transform it into an equivalent program, which does not abstract over types, by creating a copy of each polymorphic function for each type it is used at. We do so in three steps.

Step 1. Constraint Collection. Firstly, we gather a set of constraints C that captures the flow of types into type variables. We get one constraint for each function call, three in this example.

$$C \;=\; \{\, \mathsf{B} \;\sqsubseteq\; \mathsf{A},\; \mathsf{Int} \;\sqsubseteq\; \mathsf{B},\; \mathsf{String} \;\sqsubseteq\; \mathsf{B} \,\}$$

We read the inequality constraint $\mathsf{Int} \;\sqsubseteq\; \mathsf{B}$ as "type Int flows into type parameter B". At some calls, we have ground types, like Int and String flowing into the type variable of the definition, while at others, we have type variables like B flowing into it.

Step 2. Constraint Solving. Secondly, we compute a solution S of these constraints as their transitive closure. The solution maps type variables to sets of ground types.

$$S \;=\; \mathsf{A} \mapsto \{\, \mathsf{Int},\; \mathsf{String} \,\},\; \mathsf{B} \mapsto \{\, \mathsf{Int},\; \mathsf{String} \,\}$$

We want the solution S to satisfy each constraint in C. Intuitively, this ensures that for every function call at a type, that type is in the set of the corresponding type variable. In other words, we compute an upper bound on the set of ground types that might flow into each type variable.

Step 3. Specialization. Finally, given the solution, we create a copy of each polymorphic function definition for each of the types in the set of its type variable. The following is the resulting program in our monomorphic language LangMono, again with some mild syntactic extensions.

```
def first_Int(x: Int, y: Int) { x }
def first_String(x: String, y: String) { x }

def second_Int(x: Int, y: Int) {
  first_Int(y, x) }
def second_String(x: String, y: String) {
  first_String(y, x) }

def main() {
  second_Int(1, 2)
  second_String("one", "two")
}
```

We have replaced every function call with a call to the corresponding specialized function following a naming convention. The resulting program is well-typed (Theorem 3.4) and step-for-step behaves the same as the original (Theorem 3.10).

2.2 Monomorphizing Type Declarations

Another useful form of polymorphism is polymorphic type declarations. Consider the following example, where we define a polymorphic data type `Lazy` and a polymorphic interface type `Get`.

```
enum Lazy[A] {
  case Present(A)
  case Absent(Get[A])
}

trait Get[B] {
  def get(Unit): B
}

def main() {
  let x = Lazy[Int].Present(123);
  new Get[Bool] { def get(y) = false }
}
```

The `Lazy` data type has two constructors: `Present`, with a parameter of type `A`; and `Absent`, with a parameter of type `Get[A]`. The `Get` interface has one definition, `get`, with return type `B`. In the `main` function, we construct one value of each type: a `Lazy[Int]` and a `Get[Bool]`.

As before, we track the flow of types into type variables. Each use of a polymorphic type gives rise to a constraint, including the appearance of `Get[B]` in the `Absent` constructor, which gives

rise to an additional constraint associating A and B:

$$C = \{ \mathsf{A} \sqsubseteq \mathsf{B},\ \mathsf{Int} \sqsubseteq \mathsf{A},\ \mathsf{Bool} \sqsubseteq \mathsf{B} \}$$

The solution of these constraints reflects the flow of the type `Bool` through A and into B:

$$S = \mathsf{A} \mapsto \{ \mathsf{Int} \},\ \mathsf{B} \mapsto \{ \mathsf{Int},\ \mathsf{Bool} \}$$

Again, the solution indicates the set of ground types that flow into each type variable. For type declarations, this intuitively represents the set of specializations of the type that might occur at run time. For each ground type flowing into a type's type parameter, we create a specialized copy of the type. In this example, the monomorphized program has one copy of the `Lazy` type and two copies of the `Get` type.

```
enum Lazy_Int {
  case Present(Int)
  case Absent(Get_Int)
}
```

```
trait Get_Int {
  def get(): Int
}
trait Get_Bool {
  def get(): Bool
}
```

```
def main() {
  let x = Lazy_Int.Present(123);
  new Get_Bool { def get(y) = false }
}
```

We change the main function to make reference to the specialized types. The result is again a well-typed monomorphic program, behaving the same as the original.

2.3 Monomorphizing Existential Polymorphism

The languages LangPoly and LangMono that we consider both feature nominal algebraic data types. Existential types are those where a type variable is bound at a constructor. As an example, consider the program in Figure 3 that defines a data type `Showable` with a single constructor `pack`. The constructor existentially hides a type A, together with a value of type A, and a function that converts this type to `String`. Function `printShowable` unpacks the given existential package into a value `v: B` and a function `f: B ⟶ String`. It uses the function to render the value to a string and then prints the result. In `main`, we apply this function twice: once to an integer value packed together with a primitive function that converts it to a string, and once with a string packed together with a monomorphic identity function. We again monomorphize this program in three steps, where in the first two we gather constraints and compute their transitive closure. Similar to type abstractions and their applications, for every use of a constructor we generate a constraint that the type it is used at flows into the existential type variable at its definition. Additionally, we track the flow of A into B in the pattern match.

$$C = \{ \mathsf{Int} \sqsubseteq \mathsf{A},\ \mathsf{String} \sqsubseteq \mathsf{A},\ \mathsf{A} \sqsubseteq \mathsf{B} \}$$

This is one of our key ideas: types flow from constructors into the type parameters at the type definition. From there they continue to flow into the type variables at pattern matches.

$$S = \mathsf{A} \mapsto \{ \mathsf{Int},\ \mathsf{String} \},\ \mathsf{B} \mapsto \{ \mathsf{Int},\ \mathsf{String} \}$$

The solution now contains the set of all ground types that flow into the existential type variable A and transitively into B. To actually monomorphize this program we create a monomorphic constructor for each of those types. This is another one of our key ideas: we specialize existentials by creating a constructor for each ground type. Moreover, in every pattern match on this existential type, we create a copy of the clause for each created constructor. Again, the resulting program is well-typed and step-for-step behaves the same as the original. In the above example, the type of v is now monomorphized to `Int` and `String` in the two different copies, respectively.

Before Monomorphization

```
enum Showable {
  case pack[A](A, A → String)
}
def printShowable(s: Showable) {
  s match {
    case pack[B](v, f) ⇒ print(f(v))
  }
}
```

```
def main() {
  printShowable(pack[Int](5,
    x ⇒ intToString(x)))

  printShowable(pack[String]("hi",
    x ⇒ x))
}
```

After Monomorphization

```
enum Showable {
  case pack_Int(Int, Int → String)
  case pack_String(String, String → String)
}
def printShowable(s: Showable) {
  s match {
    case pack_Int(v, f) ⇒
      print(f(v))
    case pack_String(v, f) ⇒
      print(f(v))
  }
}
```

```
def main() {
  printShowable(pack_Int(5,
    x ⇒ intToString(x)))

  printShowable(pack_String("hi",
    x ⇒ x))
}
```

Fig. 3. Monomorphizing existential types.

Before Monomorphization

```
let t = new CBool {
  def choose[B](x: B, y: B) { x }
};
let f = new CBool {
  def choose[C](x: C, y: C) { y }
};
```

```
trait CBool {
  def choose[A](x: A, y: A): A
}
def main() {
  t.choose[CBool](t, f).choose[Int](1, 0)
}
```

After Monomorphization

```
let t = new CBool {
  def choose_CBool(x, y) { x }
  def choose_Int(x, y) { x }
};
let f = new CBool {
  def choose_CBool(x, y) { y }
  def choose_Int(x, y) { y }
};
```

```
trait CBool {
  def choose_CBool(x: CBool, y: CBool): CBool
  def choose_Int(x: Int, y: Int): Int
}
def main() {
  t.choose_CBool(t, f).choose_Int(1, 0)
}
```

Fig. 4. Monomorphizing higher-rank polymorphism.

2.4 Monomorphizing Higher-Rank Polymorphism

We consider languages which, in addition to algebraic data types, have nominal interface types that declare potentially polymorphic methods, and objects that implement them. Those polymorphic methods correspond to higher-rank polymorphism. In the example program in Figure 4, modeling Church-encoded Booleans, we define the interface `CBool` with a single polymorphic method `choose`. We then implement an object `t` where the method chooses its first argument, and an object `f` where the method chooses its second argument. Finally, we invoke the method `choose` to choose between the two objects, and invoke `choose` on the result to choose between one and zero. We use the same three steps to monomorphize this program. For every invocation of a polymorphic method, we track the flow of the type it is used at into the type parameter of the method in the interface. Additionally, we track the flow of these type variables into the method definitions.

$$C \;=\; \{\, \mathsf{CBool} \,\sqsubseteq\, \mathsf{A},\ \mathsf{Int} \,\sqsubseteq\, \mathsf{A},\ \mathsf{A} \,\sqsubseteq\, \mathsf{B},\ \mathsf{A} \,\sqsubseteq\, \mathsf{C} \,\}$$

In this example, the solution S is surprisingly simple. Note that the example contains indirect control flow, and that we instantiate a type variable with a polymorphic type.

$$S \;=\; \mathsf{A} \mapsto \{\, \mathsf{CBool},\ \mathsf{Int} \,\},\ \mathsf{B} \mapsto \{\, \mathsf{CBool},\ \mathsf{Int} \,\},\ \mathsf{C} \mapsto \{\, \mathsf{CBool},\ \mathsf{Int} \,\}$$

To monomorphize, we create a new method in the interface for each ground type in the solution set of its type parameter. This is another key idea: we specialize polymorphic functions by creating a monomorphic method for each ground type. Accordingly, in every implementation of the interface, we create copies of every polymorphic method for each of those types. Variables `x` and `y` thus have types `CBool` and `Int` in the respective specialization. Finally, we invoke the specialized methods following a naming convention.

2.5 Four Kinds of Polymorphic Recursion

Clearly, there are classes of programs that we cannot monomorphize. Thanks to our uniform treatment of the four different kinds of polymorphism, we identify four kinds of polymorphic recursion, which manifest in cyclic flow, and which we detect through a type flow analysis.

2.5.1 Polymorphically Recursive Functions. The first such class is a standard limitation of monomorphization: *polymorphic recursion* [Mycroft 1984]. While well-known, we illustrate how it manifests and how we detect it. For polymorphic recursion to cause an infinite set of specializations, at least one polymorphic type, such as `Box`, is required:

```
enum Box[B] {
  case Wrap(B)
}
```

```
def f[A](x: A) {
  f[Box[A]](Wrap(x)); x
}
```

```
def main() {
  f[Int](5)
}
```

The constraints we gather are $\{\, \mathsf{Box[A]} \,\sqsubseteq\, \mathsf{A},\ \mathsf{Int} \,\sqsubseteq\, \mathsf{A} \,\}$. Solving these, we reach an intermediate solution $\mathsf{A} \mapsto \{\, \mathsf{Box[A]},\ \mathsf{Int} \,\}$ and notice A occurs under a type constructor in its own set. As usual, this would lead to unbounded specializations: Int, Box[Int], Box[Box[Int]], and so on. We use a simple type flow analysis to detect and rule out such cases.

2.5.2 Polymorphically Recursive Types. Types themselves may also exhibit polymorphic recursion, even in the absence of polymorphic functions. A classic example is balanced binary trees.

```
enum Tree[A] {
  case Leaf(A)
  case Branch(Tree[Two[A]]) }
```

```
enum Two[B] {
  case Both(B, B)
}
```

```
def main() {
  Tree[Int].Leaf(123)
}
```

The constraints arising from this program are $\{ \text{Two}[A] \sqsubseteq A, A \sqsubseteq B, \text{Int} \sqsubseteq A \}$. This example presents a problem for monomorphization, as it requires an infinite number of type specializations, manifested as the recursive constraint $\text{Two}[A] \sqsubseteq A$. Again, a type flow analysis identifies this problem, and we do not attempt to monomorphize it.

2.5.3 Polymorphically Recursive Methods. Recursive functions are not the only way to cause polymorphic recursion. A more indirect way to achieve the same situation is through interfaces and polymorphic methods, such as `Forall` in the following example:

```
trait Forall {
  def rec[A](f: Forall, a: A): A
}
def main() { x.rec[Int](x, 0) }
```

```
let x = new Forall {
  def rec[B](f: Forall, a: B) {
    f.rec[Box[B]](f, Wrap(a)); a
}};
```

Here, we gather constraints $\{ A \sqsubseteq B, \text{Box}[B] \sqsubseteq A, \text{Int} \sqsubseteq A \}$. While the program does not involve recursive functions directly, we still get recursive constraints, which we again recognize as such with a type flow analysis.

2.5.4 Polymorphic Packing. Finally, and more surprisingly, we also gather recursive constraints in the following example involving an existential type `Dynamic`, but again no term-level recursion.

```
enum Dynamic {
  case Hide[A](A)
}
```

```
def main() {
  let x = Hide[Int](0);
  match x { case Hide[B](x: B) ⇒ Hide[Box[B]](Wrap(x)) }
}
```

For this example, we gathered constraints $\{ A \sqsubseteq B, \text{Box}[B] \sqsubseteq A, \text{Int} \sqsubseteq A \}$ – exactly the same as in the previous example. We refer to this variant of polymorphic recursion, which is introduced by repacking an existential type, as *polymorphic packing* to set it apart from the other variants, which actually require an unbounded number of types at runtime. All three forms have in common that they introduce a non-trivial cycle in the gathered constraints.

While we have identified the problem of polymorphic packing, at this time, we do not know its practical implications. So far, we can only speculate about possible mitigations, and leave a deeper investigation to future work. In some cases, for instance in the last example, programs could still be monomorphized with a more precise static analysis. Perhaps it is even possible to fully monomorphize programs even in the presence of polymorphic recursion and polymorphic packing. At the moment, however, we do not have a definitive answer to this question.

Having introduced our approach by example, in the next section, we formally present our polymorphic source language LangPoly, our monomorphic target language LangMono, monomorphization from the former into the latter, and theorems about this translation.

3 Formalization

We start our formal presentation by describing the syntax, typing, and semantics of two languages: LangPoly, our polymorphic source language, and LangMono, our monomorphic target language. We then describe monomorphization as a translation from the source language to the target language. Monomorphization takes well-typed programs to well-typed programs (Theorem 3.4), and preserves their semantics step-for-step (Theorem 3.10).

3.1 Source Language

Our source language LangPoly features top-level functions, nominal data types (**enum**), and nominal interface types (**trait**). Top-level functions and types, data type constructors, and methods are polymorphic, expressing ordinary rank-1 polymorphism, existential polymorphism, and higher-rank polymorphism, respectively. The calculus is designed to admit a concise presentation of our approach to monomorphization, and we thus purposefully omit other features, such as first-class functions and local function definitions. It is not difficult to express first-class functions using interfaces and lift local function definitions to the top level before monomorphizing them.

Terms:

Programs	P	$::=$	$f \mapsto \{\, [\alpha](x : \tau) \Rightarrow t \,\}, \ldots$
Terms	t	$::=$	v
		$\mid$	x
		$\mid$	$\textbf{let}\ x = t;\ t$
		$\mid$	$f[\tau](x)$
		$\mid$	$c[\tau](x)$
		$\mid$	$x\ \textbf{match}\ T[\tau]\ \{\ c[\alpha](x) \Rightarrow t, \ldots\ \}$
		$\mid$	$\textbf{new}\ T[\tau]\ \{\ m[\alpha](x) \Rightarrow t, \ldots\}$
		$\mid$	$x.m[\tau](x)$
Values	v	$::=$	$0 \mid \mathsf{true} \mid \text{"hello"} \mid \ldots$

Types:

Types	τ	$::=$	$\mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{String} \mid \ldots$
		$\mid$	α
		$\mid$	$T[\tau]$
Ground Types	ρ	$::=$	$\mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{String} \mid \ldots$
		$\mid$	$T[\rho]$
Environment Types	Γ	$::=$	$\Gamma,\ x : \tau$
		$\mid$	$\Gamma,\ \alpha$
		$\mid$	$\emptyset$
Function Types	Δ	$::=$	$f : \forall\, \alpha\, .\, \tau \to \tau, \ldots$
Signatures	Σ	$::=$	$\textbf{enum}\ T[\alpha]\ \{\ c[\alpha](\tau), \ldots\},\ \textbf{trait}\ T[\alpha]\ \{\ m[\alpha](\tau) : \tau, \ldots\} \ldots$

Names:

Term Variables	$x \in \mathsf{x},\ \mathsf{y},\ \mathsf{k}, \ldots$	Type Names	$T \in \mathsf{List},\ \mathsf{Func} \ldots$
Type Variables	$\alpha \in \mathsf{A},\ \mathsf{B},\ \mathsf{C}, \ldots$	Constructor Names	$c \in \mathsf{pack},\ \mathsf{cons}, \ldots$
Function Names	$f \in \mathsf{f},\ \mathsf{g},\ \mathsf{h}, \ldots$	Method Names	$m \in \mathsf{apply},\ \mathsf{next}, \ldots$

Fig. 5. Syntax of the polymorphic source language LangPoly.

The syntax of LangPoly is defined in Figure 5. A program P is a map from function identifiers f to their implementations $[\alpha](x : \tau) \Rightarrow t$. For simplicity, each function consists of exactly one type parameter α, one type-annotated term parameter $x : \tau$, and the body of the function t. The syntax of terms t includes the usual constructs of primitive values v (including integers, Booleans, strings,

and so on), variables (ranged over by x), and let-bindings. To simplify our formalization and the proofs, without loss of generality, we present the calculus in a normal form similar to fine-grain call-by-value [Levy 1999], where arguments to calls, scrutinees, and receivers are required to be variables. This way, the treatment of composite terms is confined to let-bindings, as usual with this form of presentation. Function application, denoted $f[\tau](x)$, applies the polymorphic function f to the type argument τ and argument x. The pattern matching expression x_0 **match** $T[\tau_0]$ $\{ c[\alpha](x) \Rightarrow t, \dots \}$ matches on x_0. In case c of data type T, the type argument of c will be bound to α and the argument of c will be bound to x in the body t. The constructor call $c[\tau](x)$ creates a data value of case c with type argument τ and argument x. Method invocation $x_0.m[\tau](x)$ calls method m on the object x_0, applying it to the type argument τ and argument x. Objects are constructed using **new** $T[\tau_0]$ $\{ m[\alpha](x) \Rightarrow t, \dots \}$, creating a new instance of the interface T with implementations for methods with names m. Each method binds a type parameter α and a parameter x in its body t. Types are either primitive types (*e.g.*, Int), type variables α, or type constructors T applied to a type τ. Ground types ρ are the subset of types that do not contain type variables. As usual, the type environment Γ maps variables to their types $(\Gamma,\ x : \tau)$ or brings type variables $(\Gamma,\ \alpha)$ into scope. It is defined inductively in order to preserve the scoping information between types and type variables. The function environment Δ is a map from function identifiers f to their polymorphic types $\forall\, \alpha\, .\, \tau_1 \to \tau_2$. The signature environment Σ maps data types (**enum**) and interfaces (**trait**) to a list of their constructors and methods, respectively.

3.1.1 *Typing.* The typing rules for our polymorphic source language LangPoly are shown in Figure 6. The typing judgments for terms and programs are defined with respect to signature environment Σ and function environment Δ, which are globally defined. Types can be recursive and mutually recursive through Σ. As usual, in all rules, we silently assume standard well-formedness of the typing context and well-formedness of all types with respect to the signature environment Σ and typing context Γ, meaning that all mentioned type variables must be in scope.

Rule Literal represents the set of rules addressing constant values. These, along with Variable, and Let are entirely standard. Rule Call types a function application $f[\tau](x_1)$ by looking up the function f in the function environment Δ, instantiating the type parameter α with the provided type τ, requiring the argument x to match the instantiated parameter type $\tau_1[\alpha \mapsto \tau]$, and typing the result against the instantiated return type $\tau_2[\alpha \mapsto \tau]$. Rule Match types a pattern-match expression x_0 **match** $T[\tau_0]$ $\{ c[\alpha](x) \Rightarrow t, \dots \}$ by looking up the definition of data type T in the signature environment Σ, requiring that the variable x_0 be mapped to $T[\tau_0]$ in the type environment, and typing each branch under an environment where x is typed as the argument to the constructor, instantiated with α and τ_0. Each branch must be typed against the same τ, which is the result type of the match expression. Rule Construct types a constructor call $c[\tau](x_1)$ by looking up the data type T in the signature environment Σ and requiring that x_1 be mapped in the type environment Γ to the type of the constructor argument, instantiated with the type argument τ and τ_0. The result type is the constructed data type $T[\tau_0]$. Rule Invoke types a method invocation expression $x_0.m[\tau](x_1)$ by requiring that x_0 be mapped to an object type $T[\tau_0]$ in the type environment, looking up T in the signature environment, and requiring that x_1 be mapped in the type environment Γ to the method argument type instantiated with the type argument τ and τ_0. Rule New types object creation **new** $T[\tau_0]$ $\{ m[\alpha](x) \Rightarrow t, \dots \}$ by looking up the interface T in the signature environment and typing each method body under an environment where x is typed as the argument to the method, instantiated with α and τ_0. Each method's body type must match the respective signature's return type, instantiated with α and τ_0. The result type is the annotated interface $T[\tau_0]$.

A program is well-typed if all function definitions therein are well-typed in accordance with function environment Δ, through which functions can be recursive and mutually recursive.

Typing of Terms $\boxed{\Sigma \mid \Delta \mid \Gamma \vdash t : \tau}$

$$\frac{}{\Sigma \mid \Delta \mid \Gamma \vdash \mathsf{true} : \mathsf{Bool}}\ [\textsc{Literal}] \qquad \frac{\Gamma(x) = \tau}{\Sigma \mid \Delta \mid \Gamma \vdash x : \tau}\ [\textsc{Variable}]$$

$$\frac{\Sigma \mid \Delta \mid \Gamma \vdash t_0 : \tau_0 \quad \Sigma \mid \Delta \mid \Gamma, x_0 : \tau_0 \vdash t : \tau}{\Sigma \mid \Delta \mid \Gamma \vdash \textbf{let}\ x_0 = t_0;\ t : \tau}\ [\textsc{Let}]$$

$$\frac{\Delta(f) = \forall\, \alpha\, .\, \tau_1 \to \tau_2 \quad \Gamma(x_1) = \tau_1\ [\alpha \mapsto \tau]}{\Sigma \mid \Delta \mid \Gamma \vdash f[\tau](x_1) : \tau_2\ [\alpha \mapsto \tau]}\ [\textsc{Call}]$$

$$\frac{\textbf{enum}\ T[\alpha_0]\ \{\ c[\alpha_1](\tau_1),\ \ldots\ \} \in \Sigma \quad \Gamma(x_1) = \tau_1[\alpha_0 \mapsto \tau_0,\ \alpha_1 \mapsto \tau]}{\Sigma \mid \Delta \mid \Gamma \vdash c[\tau](x_1) : T[\tau_0]}\ [\textsc{Construct}]$$

$$\frac{\begin{array}{l}\textbf{enum}\ T[\alpha_0]\ \{\ c[\alpha_1](\tau_1),\ \ldots\ \} \in \Sigma \qquad \Gamma(x_0) = T[\tau_0] \\ \Sigma \mid \Delta \mid \Gamma,\ \alpha,\ x : \tau_1[\alpha_0 \mapsto \tau_0,\ \alpha_1 \mapsto \alpha] \vdash t : \tau \qquad \ldots\end{array}}{\Sigma \mid \Delta \mid \Gamma \vdash x_0\ \textbf{match}\ T[\tau_0]\ \{\ c[\alpha](x) \Rightarrow t,\ \ldots\ \} : \tau}\ [\textsc{Match}]$$

$$\frac{\begin{array}{l}\textbf{trait}\ T[\alpha_0]\ \{\ m[\alpha_1](\tau_1) : \tau_2,\ \ldots\ \} \in \Sigma \\ \Sigma \mid \Delta \mid \Gamma,\ \alpha,\ x : \tau_1[\alpha_0 \mapsto \tau_0,\ \alpha_1 \mapsto \alpha] \vdash t : \tau_2[\alpha_0 \mapsto \tau_0,\ \alpha_1 \mapsto \alpha] \quad \ldots\end{array}}{\Sigma \mid \Delta \mid \Gamma \vdash \textbf{new}\ T[\tau_0]\ \{\ m[\alpha](x) \Rightarrow t,\ \ldots\ \} : T[\tau_0]}\ [\textsc{New}]$$

$$\frac{\textbf{trait}\ T[\alpha_0]\ \{\ m[\alpha_1](\tau_1) : \tau_2,\ \ldots\ \} \in \Sigma \qquad \Gamma(x_0) = T[\tau_0] \qquad \Gamma(x_1) = \tau_1[\alpha_0 \mapsto \tau_0,\ \alpha_1 \mapsto \tau]}{\Sigma \mid \Delta \mid \Gamma \vdash x_0.m[\tau](x_1) : \tau_2[\alpha_0 \mapsto \tau_0,\ \alpha_1 \mapsto \tau]}\ [\textsc{Invoke}]$$

Typing of Programs $\boxed{\Sigma \mid \Delta \vdash P : \Delta}$

$$\frac{\Sigma \mid \Delta \mid \alpha,\ x : \tau_1 \vdash t : \tau_2 \quad \ldots}{\Sigma \mid \Delta \vdash f \mapsto \{\ [\alpha](x : \tau_1) \Rightarrow t\ \},\ \ldots : f : \forall\, \alpha\, .\, \tau_1 \to \tau_2,\ \ldots}\ [\textsc{Program}]$$

Fig. 6. Typing rules for the polymorphic source language LangPoly.

Syntax:

Machines	M	$::=$	$\langle\, t \mid K \mid E \mid P\, \rangle$
Contexts	K	$::=$	$\{\ x \Rightarrow t,\ E\ \} :: K \ \mid\ \bullet$
Environments	E	$::=$	$E,\ x \mapsto v \ \mid\ \bullet$
Values	v	$::=$	$\ldots$
		$\mid$	$c[\rho](v)$
		$\mid$	$(E,\ \{\ m[\alpha](x) \Rightarrow t,\ \ldots\})$

Stepping relation:

(con)	$\langle\, c[\rho](x_1) \mid K \mid E \mid P\, \rangle$	$\to$	$\langle\, c[\rho](E(x_1)) \mid K \mid E \mid P\, \rangle$
(mat)	$\langle\, x_0\ \textbf{match}\ T[\rho_0]\ \{\ c[\alpha](x) \Rightarrow t,\ \ldots\ \} \mid K \mid E \mid P\, \rangle$ where $E(x_0) = c[\rho](v)$	$\to$	$\langle\, t[\alpha \mapsto \rho] \mid K \mid E,\ x \mapsto v \mid P\, \rangle$
(new)	$\langle\, \textbf{new}\ T[\rho_0]\ \{\ m[\alpha](x) \Rightarrow t,\ \ldots\} \mid K \mid E \mid P\, \rangle$	$\to$	$\langle\, (E,\ \{\ m[\alpha](x) \Rightarrow t,\ \ldots\}) \mid K \mid E \mid P\, \rangle$
(inv)	$\langle\, x_0.m[\rho](x_1) \mid K \mid E \mid P\, \rangle$ where $E(x_0) = (E_0,\ \{\ m[\alpha](x) \Rightarrow t,\ \ldots\})$	$\to$	$\langle\, t[\alpha \mapsto \rho] \mid K \mid E_0,\ x \mapsto E(x_1) \mid P\, \rangle$

Fig. 7. Abstract machine for LangPoly with selected stepping rules that illustrate the flow of types at runtime.

3.1.2 Semantics. We define the semantics of LangPoly in terms of the abstract machine in Figure 7. The machine is a 4-tuple $\langle\, t \mid K \mid E \mid P \,\rangle$ consisting of a term t, a context K corresponding to a stack of let-bindings, an environment E mapping variables to their values, and a program P containing the global function definitions. We extend values v to additionally include runtime values such as constructors containing their evaluated arguments and objects containing their respective closure environments. Note that we can syntactically differentiate between constructor calls $c[\rho](x)$ and constructed values $c[\rho](v)$. The final state of the abstract machine is $\langle\, v \mid \bullet \mid E \mid P \,\rangle$. Maybe surprisingly, our abstract machine semantics is environment-based for term-variables and substitution-based for type-variables. Type substitutions at runtime exactly correspond to the flow of types that we want to capture with our constraints. While they are operationally irrelevant, having a source language with explicit type annotations and substituting types at runtime simplifies proving properties of monomorphization. Figure 7 also lists definitions of selected stepping rules that illustrate this runtime flow of types. For existential types, we can observe how in rule *(con)* the type ρ flows from the constructor call to the value stored in the environment. Importantly, the constructed value $c[\rho](v)$ *closes* over the type ρ. Later, when pattern matching in rule *(mat)*, we extract ρ and use it to substitute for α, hence continuing the flow within t. Dually, for universal types, rule *(new)* constructs an object where the implementation is still parametric in α. Later, in rule *(inv)*, when invoking method m, the type argument ρ flows into the body t. Since terms are closed with respect to types at runtime, we only ever substitute type parameters by ground types ρ. As we will see, this property is important to establish a connection to the monomorphized variant, which also only mentions ground types.

3.2 Target Language

Our monomorphic target language LangMono (Figure 8) is like our source language LangPoly, but without polymorphism. As a monomorphic language, it thus only includes ground types ρ. Neither functions, types, constructors, nor methods take type parameters. Instead, their identifiers are indexed by a ground type, as in $f_\rho,\ T_\rho,\ c_\rho,\ m_\rho$. In our formalization, these ground types are part of the identifier name and uniquely determine the respective monomorphic variant. In practice, more advanced mapping schemes or naming conventions must be used. By abuse of notation, we use the isomorphic sets of ground types ρ of LangPoly and of LangMono interchangeably.

3.2.1 Typing. Figure 9 defines the typing rules for our target language LangMono. The type system is almost the same as for LangPoly, save for polymorphism. Typing contexts Γ now map variables to ground types, and functions, constructors, and methods all have monomorphic types.

3.2.2 Semantics. Like for LangPoly, the semantics of LangMono is given in terms of an abstract machine (Figure 10). Since there are no type variables, we also never substitute any types. Consequently, instead of closing over the existential type in rule *(con)*, we use the tag for the respective monomorphic constructor. While in the source language, pattern matching clauses had the shape $c[\alpha](x) \Rightarrow t$, binding type parameter α, in the monomorphized variant, pattern matches need to be prepared to handle the monomorphized constructor tag $c_\rho(x) \Rightarrow t$. As we show in the remainder of this section, our translation ensures that this is the case.

3.3 Monomorphization

Like already demonstrated by example in Section 2, we perform monomorphization in three steps: (1) constraint collection, (2) constraint solving, and (3) specialization. We now go through each of these steps formally, before presenting metatheoretical properties about our translation.

Terms:

Programs	P	::=	$f_\rho \mapsto \{ (x : \rho) \Rightarrow t \}, \ldots$
Terms	t	::=	v
		|	x
		|	**let** $x = t;\ t$
		|	$f_\rho\,(x)$
		|	$c_\rho\,(x)$
		|	x **match** $T_\rho \ \{ c_\rho\,(x) \Rightarrow t, \ldots \}$
		|	**new** $T_\rho \ \{ m_\rho\,(x) \Rightarrow t, \ldots \}$
		|	$x.\,m_\rho\,(x)$
Values	v	::=	0 | true | ”hello” | ...

Types:

Ground Types	ρ	::=	Int | Bool | String | ...
		|	T_ρ
Environment Types	Γ	::=	$\Gamma,\ x : \rho$
		|	$\emptyset$
Function Types	Δ	::=	$f_\rho : \rho \rightarrow \rho, \ldots$
Signatures	Σ	::=	**enum** $T_\rho\ \{ c_\rho(\rho), \ldots\}$, **trait** $T_\rho\ \{ m_\rho(\rho) : \rho, \ldots\} \ldots$

Names:

Term Variables	x ∈ x, y, k, ...	Type Names	T_ρ ∈ List_Int, Func_Bool ...
Type Variables	*none*	Constructor Names	c_ρ ∈ pack_Int, cons_Bool, ...
Function Names	f_ρ ∈ f_Int, g_Bool ...	Method Names	m_ρ ∈ apply_Int, next_Bool, ...

Fig. 8. Syntax of the monomorphic target language LangMono, without any type parameters and polymorphism. Changes highlighted in *gray*.

3.3.1 Step 1. Constraint Collection. Our monomorphization procedure is based on tracking the flow of types into type variables. We represent this flow in terms of constraints using the syntax $\tau \sqsubseteq \alpha$, pronounced "*τ flows into α*". Figure 11 defines constraint collection by extending the typing judgment of Figure 6 to a 6-point relation with an additional output emitting constraint sets C. While written in inference style, the input to this phase is the entire typing derivation, meaning no inference is performed here. Besides the constraint generation, the rules are identical to the ones presented in Figure 6. Types flow into type variables in type applications, such as Construct and Invoke, where we emit a constraint $\tau \sqsubseteq \alpha_1$. Type variables flow into other type variables in rules Match and New, where we emit constraint $\alpha_1 \sqsubseteq \alpha$. All other rules, such as Let, simply accumulate the constraints from their subterms.

Well-formedness generates constraints. Wherever they appear in the rules, types are assumed to be well-formed. While it is not unusual to implicitly require well-formedness of types with regard to free type variables, in our formalization they emit additional constraints. Intuitively, this is because the instantiation of a polymorphic type gives rise to a flow of the type argument into the type parameter. Concretely, whenever a type $T[\tau]$ appears, there is a constraint $\tau \sqsubseteq \alpha$, where $T[\alpha]$ is a polymorphic data type or interface. For example, with explicit well-formedness requirements, rule Invoke looks like the following.

$$\frac{\begin{array}{c}\Gamma \vdash T[\tau_0] \Rightarrow C_0 \qquad \Gamma \vdash \tau_1[\alpha_0 \mapsto \tau_0,\ \alpha_1 \mapsto \tau] \Rightarrow C_1 \qquad \Gamma \vdash \tau_2[\alpha_0 \mapsto \tau_0,\ \alpha_1 \mapsto \tau] \Rightarrow C_2 \\ \textbf{trait}\ T[\alpha_0]\ \{ m[\alpha_1](\tau_1) : \tau_2, \ldots \} \in \Sigma \quad \Gamma(x_0) = T[\tau_0] \quad \Gamma(x_1) = \tau_1[\alpha_0 \mapsto \tau_0,\ \alpha_1 \mapsto \tau]\end{array}}{\Sigma \mid \Delta \mid \Gamma \vdash x_0.m[\tau](x_1) : \tau_2[\alpha_0 \mapsto \tau_0,\ \alpha_1 \mapsto \tau] \Rightarrow \{ \tau \sqsubseteq \alpha_1 \} \cup C_0 \cup C_1 \cup C_2}$$

Typing of Terms $\boxed{\Sigma \mid \Delta \mid \Gamma \vdash t : \rho}$

$$\frac{}{\Sigma \mid \Delta \mid \Gamma \vdash \mathsf{true} : \mathsf{Bool}}\ [\text{Literal}] \qquad \frac{\Gamma(x) = \rho}{\Sigma \mid \Delta \mid \Gamma \vdash x : \rho}\ [\text{Variable}]$$

$$\frac{\Sigma \mid \Delta \mid \Gamma \vdash t_0 : \rho_0 \qquad \Sigma \mid \Delta \mid \Gamma, x_0 : \rho_0 \vdash t : \rho}{\Sigma \mid \Delta \mid \Gamma \vdash \textbf{let}\ x_0 = t_0;\ t : \rho}\ [\text{Let}]$$

$$\frac{\Delta(f_\rho) = \rho_1 \to \rho_2 \qquad \Gamma(x_1) = \rho_1}{\Sigma \mid \Delta \mid \Gamma \vdash f_\rho(x_1) : \rho_0}\ [\text{Call}]$$

$$\frac{\textbf{enum}\ T_{\rho_0}\ \{\ c_\rho(\rho_1),\ \ldots\ \} \in \Sigma \qquad \Gamma(x_1) = \rho_1}{\Sigma \mid \Delta \mid \Gamma \vdash c_\rho(x_1) : T_{\rho_0}}\ [\text{Construct}]$$

$$\frac{\textbf{enum}\ T_{\rho_0}\ \{\ c_\rho(\rho_1),\ \ldots\ \} \in \Sigma \qquad \Gamma(x_0) = T_{\rho_0} \qquad \Sigma \mid \Delta \mid \Gamma, x : \rho_1 \vdash t : \rho_2 \qquad \ldots}{\Sigma \mid \Delta \mid \Gamma \vdash x_0\ \textbf{match}\ T_{\rho_0}\ \{\ c_\rho(x) \Rightarrow t,\ \ldots\ \} : \rho_2}\ [\text{Match}]$$

$$\frac{\textbf{trait}\ T_{\rho_0}\ \{\ m_\rho(\rho_1) : \rho_2,\ \ldots\ \} \in \Sigma \qquad \Sigma \mid \Delta \mid \Gamma, x : \rho_1 \vdash t : \rho_2 \qquad \ldots}{\Sigma \mid \Delta \mid \Gamma \vdash \textbf{new}\ T_{\rho_0}\ \{\ m_\rho(x) \Rightarrow t,\ \ldots\ \} : T_{\rho_0}}\ [\text{New}]$$

$$\frac{\textbf{trait}\ T_{\rho_0}\ \{\ m_\rho(\rho_1) : \rho_2,\ \ldots\ \} \in \Sigma \qquad \Gamma(x_0) = T_{\rho_0} \qquad \Gamma(x_1) = \rho_1}{\Sigma \mid \Delta \mid \Gamma \vdash x_0.\,m_\rho(x_1) : \rho_2}\ [\text{Invoke}]$$

Typing of Programs $\boxed{\Sigma \mid \Delta \vdash P : \Delta}$

$$\frac{\Sigma \mid \Delta \mid x : \rho_1 \vdash t : \rho_2 \qquad \ldots}{\Sigma \mid \Delta \vdash f_\rho \mapsto \{\ (x : \rho_1) \Rightarrow t\ \},\ \ldots : f : \rho_1 \to \rho_2,\ \ldots}\ [\text{Program}]$$

Fig. 9. Typing rules for the monomorphic target language LangMono. All involved types are now monomorphic. Differences to the polymorphic source language highlighted in *gray*.

Syntax:

Machines	M	$::=$	$\langle\, t \mid K \mid E \mid P\,\rangle$
Contexts	K	$::=$	$\{\ x \Rightarrow t,\ E\ \} :: K \ \mid\ \bullet$
Environments	E	$::=$	$E,\ x \mapsto v \ \mid\ \bullet$
Values	v	$::=$	$\ldots$
		$\mid$	$c_\rho(v)$
		$\mid$	$(E,\ \{\ m_\rho(x) \Rightarrow t,\ \ldots\})$

Stepping relation:

(con) $\langle\, c_\rho(x_1) \mid K \mid E \mid P\,\rangle \ \to\ \langle\, c_\rho(E(x_1)) \mid K \mid E \mid P\,\rangle$

(mat) $\langle\, x_0\ \textbf{match}\ T_{\rho_0}\ \{\ c_\rho(x) \Rightarrow t,\ \ldots\ \} \mid K \mid E \mid P\,\rangle \ \to\ \langle\, t \mid K \mid E,\ x \mapsto v \mid P\,\rangle$
where $E(x_0) = c_\rho(v)$

(new) $\langle\, \textbf{new}\ T_{\rho_0}\ \{\ m_\rho(x) \Rightarrow t,\ \ldots\} \mid K \mid E \mid P\,\rangle \ \to\ \langle\, (E,\ \{\ m_\rho(x) \Rightarrow t,\ \ldots\}) \mid K \mid E \mid P\,\rangle$

(inv) $\langle\, x_0.\,m_\rho(x_1) \mid K \mid E \mid P\,\rangle \ \to\ \langle\, t \mid K \mid E_0,\ x \mapsto E(x_1) \mid P\,\rangle$
where $E(x_0) = (E_0,\ \{\ m_\rho(x) \Rightarrow t,\ \ldots\})$

Fig. 10. Abstract machine for LangMono and selected stepping rules. Differences to the semantics of the polymorphic source language are highlighted in *gray*.

The purpose of well-formedness constraints like $\Gamma \vdash T[\tau_0] \Rightarrow C_0$ is twofold. As usual, they ensure that type variables are well-scoped and therefore prevent existentially bound type parameters from escaping. In our setting, they also track the flow of types into type parameters of nested types and make sure we create monomorphic variants of deeply nested types like List[List[Int]]. The well-formedness judgment $\Gamma \vdash \tau \Rightarrow C$ is defined as follows:

$$\frac{}{\Sigma \mid \Gamma \vdash \mathsf{Int} \Rightarrow \emptyset}\ [\textsc{Primitive}] \qquad \frac{\alpha \in \Gamma}{\Sigma \mid \Gamma \vdash \alpha \Rightarrow \emptyset}\ [\textsc{Polymorphic}]$$

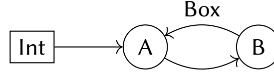
$$\frac{\Gamma \vdash \tau_0 \Rightarrow C \qquad \textbf{enum}\ T[\alpha_0]\ \{\ c[\alpha_1](\tau_1),\ \ldots\ \} \in \Sigma}{\Sigma \mid \Gamma \vdash T[\tau_0] \Rightarrow \{\ \tau_0 \sqsubseteq \alpha_0\ \} \cup C}\ [\textsc{Enum}] \qquad \frac{\Gamma \vdash \tau_0 \Rightarrow C \qquad \textbf{trait}\ T[\alpha_0]\ \{\ m[\alpha_1](\tau_1) : \tau_2,\ \ldots\ \} \in \Sigma}{\Sigma \mid \Gamma \vdash T[\tau_0] \Rightarrow \{\ \tau_0 \sqsubseteq \alpha_0\ \} \cup C}\ [\textsc{Trait}]$$

3.3.2 Step 2. Constraint Solving. In a second step, we compute a solution S of the gathered constraints C as their transitive closure. A solution $S ::= \alpha \mapsto \{\rho, \ldots\}, \ldots$ is a map from type variables to finite sets of ground types. In presence of polymorphic recursion or polymorphic packing, we might fail to find a finite solution (Section 2.5), in which case we cannot monomorphize the program. Intuitively, a solution maps each type variable to those types that transitively flow into them. This intuitive notion is captured formally in Figure 12. The judgment $S \vDash \tau \sqsubseteq \alpha$ expresses that a solution S satisfies a specific constraint $\tau \sqsubseteq \alpha$. Rule Sub indicates that a solution satisfies a ground type constraint $\rho \sqsubseteq \alpha$ if the solution maps the type variable α to a set containing the ground type ρ. Moreover, it indicates that a solution satisfies a type variable constraint $\beta \sqsubseteq \alpha$ if all types in the image of β are also in the image of α. Finally, it indicates that a solution satisfies a complex type constraint $T[\tau] \sqsubseteq \alpha$ if all types that τ stands for under this solution are in the image of α. Together, these rules capture the intuition that if a type variable flows into another, then all types that flow into the first transitively flow into the second. Figure 12 once more adjusts the typing rules for LangPoly. The presented rules are dual to the constraint rules of Figure 11, in that they determine whether a solution is valid for the given term, rather than generating constraints for it. While constraint typing is essential to algorithmically compute a monomorphization, solution typing is exclusively used to prove various properties about it. As with the constraint rules, the solution rules are identical to the typing rules, with the exception of the solutions S. Again Construct, Invoke, Match, and New all involve flow constraints. In these cases, we require that the solution satisfies the constraint from the flow of the type argument into the type parameter.

Detecting non-monomorphizable programs. Programs involving polymorphic recursion and polymorphic packing are not monomorphizable. In order to detect these properties, it is instructive to reinterpret the constraints as a graph, where vertices correspond to primitive types and type variables, and edges are labeled with (possibly empty) lists of type constructors. Formally, the edges E of the monomorphization graph of constraints C are given by the following:

$$\begin{aligned} E_1 &= \{\ \beta \xrightarrow{T_1,\ \ldots,\ T_n} \alpha \mid (T_1[\ \ldots\ T_n[\beta]\ ] \sqsubseteq \alpha) \in C\ \} \\ E_2 &= \{\ \mathsf{Int} \xrightarrow{T_1,\ \ldots,\ T_n} \alpha \mid (T_1[\ \ldots\ T_n[\mathsf{Int}]\ ] \sqsubseteq \alpha) \in C\ \} \quad \text{(for each primitive type)} \\ E &= E_1 \cup E_2 \end{aligned}$$

As an example, below is the monomorphization graph corresponding to the program featuring polymorphic packing in Section 2.5.4.



The three edges of the graph arise from the three constraints { A ⊑ B, Box[B] ⊑ A, Int ⊑ A }. The edge from B to A has a label because of the type constructor in the constraint Box[B] ⊑ A.

Intuitively, a path starting at a primitive node in the graph represents a ground type, where the type constructors on the edges are applied as the path is followed. We can formalize this as the following recursive definition:

$$\begin{aligned} type(v_0 \to \ldots \to v_i \overset{T_1, \ldots, T_n}{\to} v_{i+1}) &= T_1[\ \ldots\ T_n[type(v_0 \to \ldots \to v_i)]\ ] \\ type(v) &= v \end{aligned}$$

The solution for a particular type variable corresponds to the set of paths from primitive types to that type variable:

$$S(\alpha) \quad = \quad \{\ type(p)\ \mid p \in paths(\mathsf{Int},\ \alpha)\ \} \quad \text{(for each primitive type)}$$

Returning to the example, we see that the paths to A include Int → A and Int → A → B $\overset{\mathsf{Box}}{\to}$ A, whose corresponding types are Int and Box[Int] The set of paths to A is infinite, due to the cycle between A and B. We can already see that this poses a problem: The infinite set of paths to A corresponds to an infinite solution S(A) of types (Box[Box[Int]], Box[Box[Box[Int]]], and so on) flowing into A. We refer to any cycle in the graph containing an edge labeled with a nonempty constructor list as a *growing cycle*, which indicates an infinite solution set. A non-growing cycle can be considered "redundant": The type represented by a path containing a cycle is the same as the type represented by the same path with the cycle removed. In the example, if the Box label were *not* present, the cycle would not be growing, and the set of types flowing into A would be the finite set { Int }. The set of monomorphizable programs is the set of programs whose constraints have a finite solution – those giving rise to a monomorphization graph without a growing cycle. The four kinds of polymorphic recursion identified in Section 2.5 can all be identified in the same way, by finding the growing cycle in their monomorphization graphs.

3.3.3 Step 3. Specialization. Monomorphization is a whole-program translation of types and terms. We define it with regard to a solution S, which maps type variables α to sets of ground types ρ. This set directly specifies the monomorphic variants of functions, types, constructors, and methods.

Figure 13 defines the translation of types with respect to a solution S. For every polymorphic function in Δ, we create a number of monomorphic functions: one for each ground type ρ in $S(\alpha)$. The name of the function is f_ρ and we obtain its type by substituting ρ for α in the parameter and return types. Similarly, for every polymorphic type, based on the solution at the type variable α_0, we generate a specialized type with the name T_ρ. The definition makes use of a translation $[\![\ \tau\ ]\!]$ on ground types, which consistently translates type applications such as $T[\rho]$ to names T_ρ. As with functions, for polymorphic constructors and methods, we create a number of monomorphic ones, based on the solution at the type variable α_1.[3] It is crucial that we create variants for all ground types that we might need at runtime. While the translation is defined for any solution S, there are additional requirements when we want to guarantee that it produces a well-typed program.

Figure 13 also defines the translation of programs and terms with regard to solution S. In accordance with the translation of function types, for every polymorphic top-level function definition

[3]The monomorphization of enums may result in multiple constructors of the same name but of different types. This poses no problem in practice, as equally named constructors for different types can be distinguished by fully qualifying them.

Constraint Collection $\Sigma \mid \Delta \mid \Gamma \vdash t : \tau \Rightarrow C$

$$\frac{\Sigma \mid \Delta \mid \Gamma \vdash t_0 : \tau_0 \Rightarrow C_0 \qquad \Sigma \mid \Delta \mid \Gamma,\, x_0 : \tau_0 \vdash t : \tau \Rightarrow C}{\Sigma \mid \Delta \mid \Gamma \vdash \textbf{let}\; x_0 = t_0;\, t : \tau \Rightarrow C_0 \cup C}\;[\text{Let}]$$

$$\frac{\textbf{enum}\; T[\alpha_0]\; \{\, c[\alpha_1](\tau_1),\, \dots \,\} \in \Sigma \qquad \Gamma(x_1) = \tau_1[\alpha_0 \mapsto \tau_0,\, \alpha_1 \mapsto \tau]}{\Sigma \mid \Delta \mid \Gamma \vdash c[\tau](x_1) : T[\tau_0] \Rightarrow \{\, \tau \sqsubseteq \alpha_1 \}}\;[\text{Construct}]$$

$$\frac{\begin{array}{c}\textbf{enum}\; T[\alpha_0]\; \{\, c[\alpha_1](\tau_1),\, \dots \,\} \in \Sigma \qquad \Gamma(x_0) = T[\tau_0] \\ \Sigma \mid \Delta \mid \Gamma,\, \alpha,\, x : \tau_1[\alpha_0 \mapsto \tau_0,\, \alpha_1 \mapsto \alpha] \vdash t : \tau \Rightarrow C_1 \qquad \dots\end{array}}{\Sigma \mid \Delta \mid \Gamma \vdash x_0\; \textbf{match}\; T[\tau_0]\; \{\, c[\alpha](x) \Rightarrow t,\, \dots \,\} : \tau \Rightarrow \{\, \alpha_1 \sqsubseteq \alpha \,\} \cup C_1 \cup \dots}\;[\text{Match}]$$

$$\frac{\begin{array}{l}\textbf{trait}\; T[\alpha_0]\; \{\, m[\alpha_1](\tau_1) : \tau_2,\, \dots \,\} \in \Sigma \\ \Sigma \mid \Delta \mid \Gamma,\, \alpha,\, x : \tau_1[\alpha_0 \mapsto \tau_0,\, \alpha_1 \mapsto \alpha] \vdash t : \tau_2[\alpha_0 \mapsto \tau_0,\, \alpha_1 \mapsto \alpha] \Rightarrow C_1 \qquad \dots\end{array}}{\Sigma \mid \Delta \mid \Gamma \vdash \textbf{new}\; T[\tau_0]\; \{\, m[\alpha](x) \Rightarrow t,\, \dots \,\} : T[\tau_0] \Rightarrow \{\, \alpha_1 \sqsubseteq \alpha \,\} \cup C_1 \cup \dots}\;[\text{New}]$$

$$\frac{\textbf{trait}\; T[\alpha_0]\; \{\, m[\alpha_1](\tau_1) : \tau_2,\, \dots \,\} \in \Sigma \qquad \Gamma(x_0) = T[\tau_0] \qquad \Gamma(x_1) = \tau_1[\alpha_0 \mapsto \tau_0,\, \alpha_1 \mapsto \tau]}{\Sigma \mid \Delta \mid \Gamma \vdash x_0.m[\tau](x_1) : \tau_2[\alpha_0 \mapsto \tau_0,\, \alpha_1 \mapsto \tau] \Rightarrow \{\, \tau \sqsubseteq \alpha_1 \,\}}\;[\text{Invoke}]$$

Fig. 11. Selected typing rules for LangPoly, extended with constraint collection.

and each ρ in $S(\alpha)$, we create one monomorphic variant. We do so by first substituting ρ for α in the parameter type τ and the function body t, and then monomorphizing the result.

The translation of terms is only defined on terms that do not contain free type variables, a precondition that will be enforced separately. In the translation of calls, constructors, and invocations, the type argument must be a ground type ρ. We translate them to using the appropriate names f_ρ, c_ρ, and m_ρ respectively. Furthermore, in accordance with the translation of signatures, we create clauses in matches and methods in objects for each ground type ρ in the solution set $S(\alpha_1)$. For brevity we only display a single clause and method in the source term. Each of them maps to a set of clauses and methods in the target term. Here, the type variable α_1 originates from the corresponding constructor or method signature. This ensures that the set of term-level clauses and methods matches the monomorphically declared type. We translate the right-hand sides of matches and methods *after* substituting ρ for α, which is important to ensure that the term under translation continues to have no free type variables. It also establishes the right lexical scoping for nested definitions. In all other cases, we simply recurse into subterms or have reached a base case.

While the given procedure specifies *some* monomorphization, we have not yet shown whether the translation is total, produces well-typed programs, and whether the target programs behave the same as the source. We demonstrate these properties in the following section.

3.4 Theorems

Monomorphization translates programs from our polymorphic source language LangPoly to our monomorphic target language LangMono. In the remainder of this section, we present theorems about both languages and the translation between them. The full proofs can be found in the appendix, which we submit as supplementary material.

3.4.1 Type Preservation. Our monomorphization procedure preserves types, which we show by following the three steps explained above. The proof has the following high-level structure:

(1) We relate constraint collection and solutions in the type system of LangPoly (Theorem 3.2).

Constraint Satisfaction $\boxed{S \vDash \tau \sqsubseteq \alpha}$

$$\frac{S^*(\tau) \subseteq S(\alpha)}{S \vDash \tau \sqsubseteq \alpha}\ [\text{Sub}]$$

$$\begin{array}{lcl} S^*(\alpha) & = & S(\alpha) \\ S^*(\text{Bool}) & = & \{ \text{Bool} \} \\ S^*(T[\tau]) & = & \{ T[\rho] \mid \rho \in S^*(\tau) \} \end{array}$$

Solution Typing $\boxed{\Sigma \mid \Delta \mid \Gamma \vdash t : \tau \Leftarrow S}$

$$\frac{\Sigma \mid \Delta \mid \Gamma \vdash t_0 : \tau_0 \Leftarrow S \qquad \Sigma \mid \Delta \mid \Gamma,\ x_0 : \tau_0 \vdash t : \tau \Leftarrow S}{\Sigma \mid \Delta \mid \Gamma \vdash \textbf{let}\ x_0 = t_0;\ t : \tau \Leftarrow S}\ [\text{Let}]$$

$$\frac{\textbf{enum}\ T[\alpha_0]\ \{ c[\alpha_1](\tau_1),\ ...\ \} \in \Sigma \qquad \Gamma(x_1) = \tau_1[\alpha_0 \mapsto \tau_0,\ \alpha_1 \mapsto \tau] \qquad S \vDash \tau \sqsubseteq \alpha_1}{\Sigma \mid \Delta \mid \Gamma \vdash c[\tau](x_1) : T[\tau_0] \Leftarrow S}\ [\text{Construct}]$$

$$\frac{\begin{array}{l}\textbf{enum}\ T[\alpha_0]\ \{ c[\alpha_1](\tau_1),\ ...\ \} \in \Sigma \qquad \Gamma(x_0) = T[\tau_0] \\ \Sigma \mid \Delta \mid \Gamma,\ \alpha,\ x : \tau_1[\alpha_0 \mapsto \tau_0,\ \alpha_1 \mapsto \alpha] \vdash t : \tau \Leftarrow S \qquad S \vDash \alpha_1 \sqsubseteq \alpha \quad ...\end{array}}{\Sigma \mid \Delta \mid \Gamma \vdash x_0\ \textbf{match}\ T[\tau_0]\ \{ c[\alpha](x) \Rightarrow t,\ ...\ \} : \tau \Leftarrow S}\ [\text{Match}]$$

$$\frac{\begin{array}{l}\textbf{trait}\ T[\alpha_0]\ \{ m[\alpha_1](\tau_1) : \tau_2,\ ...\ \} \in \Sigma \\ \Sigma \mid \Delta \mid \Gamma,\ \alpha,\ x : \tau_1[\alpha_0 \mapsto \tau_0,\ \alpha_1 \mapsto \alpha] \vdash t : \tau_2[\alpha_0 \mapsto \tau_0,\ \alpha_1 \mapsto \alpha] \Leftarrow S \qquad S \vDash \alpha_1 \sqsubseteq \alpha \quad ...\end{array}}{\Sigma \mid \Delta \mid \Gamma \vdash \textbf{new}\ T[\tau_0]\ \{ m[\alpha](x) \Rightarrow t,\ ...\ \} : T[\tau_0] \Leftarrow S}\ [\text{New}]$$

$$\frac{\begin{array}{l}\textbf{trait}\ T[\alpha_0]\ \{ m[\alpha_1](\tau_1) : \tau_2,\ ...\ \} \in \Sigma \\ \Gamma(x_0) = T[\tau_0] \qquad \Gamma(x_1) = \tau_1[\alpha_0 \mapsto \tau_0,\ \alpha_1 \mapsto \tau] \qquad S \vDash \tau \sqsubseteq \alpha_1\end{array}}{\Sigma \mid \Delta \mid \Gamma \vdash x_0.m[\tau](x_1) : \tau_2[\alpha_0 \mapsto \tau_0,\ \alpha_1 \mapsto \tau] \Leftarrow S}\ [\text{Invoke}]$$

Fig. 12. Selected typing rules in solution for LangPoly.

(2) We relate solutions in LangPoly to the specialization in LangMono (Theorem 3.4).
As a corollary of Theorems 3.2 and 3.4, we obtain type preservation:

Corollary 3.1 (Type Preservation).
When $\Sigma \mid \Delta \vdash P : \Delta \Rightarrow C$ *and* $S \vDash C$, *then* $[\![\Sigma]\!]_S \mid [\![\Delta]\!]_S \vdash [\![P]\!]_S : [\![\Delta]\!]_S$.

Recall that in the first step of monomorphization, we gather C following the rules of Figure 11. If we find a solution that satisfies all constraints, expressed as $S \vDash C$, then we can annotate the program with proofs that the solution satisfies all individual constraints where they are emitted, following the rules of Figure 12.

Theorem 3.2. *When* $\Sigma \mid \Delta \vdash P : \Delta \Rightarrow C$ *and* $S \vDash C$, *then* $\Sigma \mid \Delta \vdash P : \Delta \Leftarrow S$.

Proof. By considering function definitions individually, realizing that when $S \vDash C_1 \cup C_2$ then $S \vDash C_1$ and $S \vDash C_2$, and using Lemma 3.3. □

To this end, we use the following lemma that goes over terms and annotates them.

Lemma 3.3. *When* $\Sigma \mid \Delta \mid \Gamma \vdash t : \tau \Rightarrow C$ *and* $S \vDash C$, *then* $\Sigma \mid \Delta \mid \Gamma \vdash t : \tau \Leftarrow S$.

Monomorphization of Function Signatures

$$\llbracket f \,:\, \forall\, \alpha\,.\,\tau_1 \to \tau_2 \rrbracket_S \quad = \quad f_\rho \,:\, (\llbracket \tau_1[\alpha \mapsto \rho] \rrbracket \to \llbracket \tau_2[\alpha \mapsto \rho] \rrbracket),\ \ldots \quad \text{for each } \rho \in S(\alpha)$$

Monomorphization of Type Declarations

$$\begin{array}{lcl}
\llbracket \textbf{enum}\ T[\alpha_0]\ \{\ c[\alpha_1](\tau_1)\ \} \rrbracket_S & = & \textbf{enum}\ T_\rho\ \{\ \llbracket c[\alpha_1](\tau_1[\alpha_0 \mapsto \rho]) \rrbracket_S\ \},\ \ldots \\
 & & \text{for each } \rho \in S(\alpha_0) \\
\llbracket c[\alpha_1](\tau_1) \rrbracket_S & = & c_\rho(\llbracket \tau_1[\alpha_1 \mapsto \rho] \rrbracket),\ \ldots \\
 & & \text{for each } \rho \in S(\alpha_1) \\
\llbracket \textbf{trait}\ T[\alpha_0]\ \{\ m[\alpha_1](\tau_1) : \tau_2\ \} \rrbracket_S & = & \textbf{trait}\ T_\rho\ \{\ \llbracket m[\alpha_1](\tau_1[\alpha_0 \mapsto \rho]) : \tau_2[\alpha_0 \mapsto \rho] \rrbracket_S\},\ \ldots \\
 & & \text{for each } \rho \in S(\alpha_0) \\
\llbracket m[\alpha_1](\tau_1) : \tau_2 \rrbracket_S & = & m_\rho(\llbracket \tau_1[\alpha_1 \mapsto \rho] \rrbracket) : \llbracket \tau_2[\alpha_1 \mapsto \rho] \rrbracket,\ \ldots \\
 & & \text{for each } \rho \in S(\alpha_1)
\end{array}$$

Monomorphization of Programs

$$\llbracket f \mapsto \{\ [\alpha](x : \tau) \Rightarrow t\ \} \rrbracket_S \quad = \quad f_\rho \mapsto \{\ (x : \llbracket \tau[\alpha \mapsto \rho] \rrbracket) \Rightarrow \llbracket t[\alpha \mapsto \rho] \rrbracket_S]\ \},\ \ldots \quad \text{for each } \rho \in S(\alpha)$$

Monomorphization of Terms

$$\begin{array}{lcl}
\llbracket \text{true} \rrbracket_S & = & \text{true} \\
\llbracket x \rrbracket_S & = & x \\
\llbracket \textbf{let}\ x_0 = t_0;\ t \rrbracket_S & = & \textbf{let}\ x_0 = \llbracket t_0 \rrbracket_S;\ \llbracket t \rrbracket_S \\
\llbracket f[\rho](x_1) \rrbracket_S & = & f_\rho(x_1) \\
\llbracket c[\rho](x_1) \rrbracket_S & = & c_\rho(x_1) \\
\llbracket x_0\ \textbf{match}\ T[\rho_0]\ \{\ c[\alpha](x) \Rightarrow t\ \} \rrbracket_S & = & x_0\ \textbf{match}\ T_{\rho_0}\ \{\ c_\rho(x) \Rightarrow \llbracket t[\alpha \mapsto \rho] \rrbracket_S,\ \ldots\} \\
 & & \text{for each } \rho \in S(\alpha_1) \text{ where } \textbf{enum}\ T[\alpha_0]\ \{\ c[\alpha_1](\tau_1)\ \} \in \Sigma \\
\llbracket \textbf{new}\ T[\rho_0]\ \{\ m[\alpha](x) \Rightarrow t\ \} \rrbracket_S & = & \textbf{new}\ T_{\rho_0}\ \{\ m_\rho(x) \Rightarrow \llbracket t[\alpha \mapsto \rho] \rrbracket_S,\ \ldots\ \} \\
 & & \text{for each } \rho \in S(\alpha_1) \text{ where } \textbf{trait}\ T[\alpha_0]\ \{\ m[\alpha_1](\tau_1) : \tau_2\ \} \in \Sigma \\
\llbracket x_0.m[\rho](x_1) \rrbracket_S & = & x_0.m_\rho(x_1)
\end{array}$$

Fig. 13. Monomorphization.

Proof. By induction over the structure of terms. □

Given that the LangPoly program is well-typed under a solution, we then proceed to elaborate it into a LangMono program. This step always succeeds and preserves types:

Theorem 3.4. *When* $\Sigma \mid \Delta \vdash P : \Delta \Leftarrow S$, *then* $\llbracket \Sigma \rrbracket_S \mid \llbracket \Delta \rrbracket_S \vdash \llbracket P \rrbracket_S : \llbracket \Delta \rrbracket_S$.

Proof. By considering each function definition separately and using Lemma 3.5. □

The key to this proof is the following lemma of well-typedness of translated terms. An important assumption is that the term does not contain free type variables, which follows from Γ not containing type variables and well-formedness of terms and types.

Lemma 3.5. *When* $\Sigma \mid \Delta \mid \Gamma \vdash t : \rho \Leftarrow S$ *and* $ftv(\Gamma) = \emptyset$, *then* $\llbracket \Sigma \rrbracket_S \mid \llbracket \Delta \rrbracket_S \mid \Gamma \vdash \llbracket t \rrbracket_S : \rho$.

Proof. By induction over the size of terms. □

Induction is over the size of terms and not over terms, because in our translation we substitute ground types for type variables before recursively translating the result. Substitution does not

change the size of the term. Moreover, we use the following lemma of well-typedness under a solution after substitution.

Lemma 3.6 (Substitution). *When* $\Sigma \mid \Delta \mid \Gamma,\ \alpha,\ \Gamma' \vdash\ t\ :\ \tau_0\ \ \Leftarrow S$ *and* $\rho\ \in S(\alpha)$, *then* $\Sigma \mid \Delta \mid \Gamma,\ \Gamma'[\alpha \mapsto \rho] \vdash\ t[\alpha \mapsto \rho]\ :\ \tau_0[\alpha \mapsto \rho]\ \ \Leftarrow S$.

Proof. By induction over the typing derivation. Let us consider case Construct to illustrate how this lemma ensures that the solution correctly captures the flow of types into type variables. Consider the case where a constructor is applied to the type variable we are substituting for (that is $\tau\ =\ \alpha$). Our goal is to show that $\ldots \vdash\ c[\alpha](x_1)[\alpha \mapsto \rho]\ :\ T[\alpha \mapsto \rho]\ \ \Leftarrow S$, which simplifies to $\ldots \vdash\ c[\rho](x_1)\ :\ T\ \ \Leftarrow S$. In order to invoke Construct, we need to prove $S\ \vDash\ \rho\ \sqsubseteq\ \alpha_1$. Inversion on the typing derivation, only gives us $S\ \vDash\ \alpha\ \sqsubseteq\ \alpha_1$. However, inversion on it in turn gives us $S(\alpha)\ \subseteq\ S(\alpha_1)$, which we can combine with the premise $\rho\ \in S(\alpha)$ to obtain $\rho\ \in S(\alpha_1)$, and by definition $S\ \vDash\ \rho\ \sqsubseteq\ \alpha_1$. □

3.4.2 Semantics Preservation. The semantics of both, our source language LangPoly and our target language LangMono are defined in terms of abstract machines (Figures 7 and 10). For both languages we have the usual theorems of progress and preservation. To state those, we extend typing to machine states, written M ok.

Theorem 3.7 (Progress). *When* M ok, *then either* M *is in a final state or* $M \longrightarrow M'$.

Proof. By case analysis of the typing judgment on the term in M. □

Theorem 3.8 (Preservation). *When* M ok *and* $M \longrightarrow M'$, *then* M' ok.

Proof. By case analysis of the stepping relation. □

The proofs for LangPoly and LangMono are very similar. The difference lies in the extra handling of substitutions of type variables for LangPoly. Moreover, our polymorphic source language LangPoly has a stronger property of preservation: we extend well-typedness under solutions to machine states, written $M\ \ \Leftarrow S$ ok and prove that it is preserved through steps. This theorem expresses that the solution completely captures the flow of types into type variables during execution. It crucially relies on Lemma 3.6, where we substitute ground types for type variables.

Theorem 3.9. *When* $M\ \ \Leftarrow S$ ok *and* $M \longrightarrow M'$, *then* $M'\ \ \Leftarrow S$ ok.

Proof. By case analysis of the typing judgment on the term in M. □

The proof is identical in structure to the proof of Theorem 3.8 for LangPoly.

Finally, we extend monomorphization to machine states and prove that monomorphization with regard to a solution S distributes over steps, given that the source machine was well-typed under S.

Theorem 3.10. *When* $M\ \ \Leftarrow S$ ok *and* $M \longrightarrow M'$, *then* $[\![\ M\ ]\!]_S \longrightarrow [\![\ M'\ ]\!]_S$.

Monomorphization preserves semantics step-for-step. From this it easily follows that when a source machine stops with a final result after a number of steps, then the translated machine stops after the same number of steps with the same final result.

Corollary 3.11 (Semantics Preservation).
When $M\ \ \Leftarrow S$ ok *and* $M \longrightarrow^n \langle\, v \mid \bullet \mid E \mid P\,\rangle$, *then* $[\![\ M\ ]\!]_S \longrightarrow^n \langle\, [\![\ v\ ]\!]_S \mid \bullet \mid [\![\ E\ ]\!]_S \mid [\![\ P\ ]\!]_S\,\rangle$.

In addition to these theoretical results, in the next section, we present our implementation.

4 Implementation

To assess the practical feasibility of our approach, we have implemented the monomorphization algorithm for a superset of LangPoly. We provide it in the supplementary material as a JavaScript application with an HTML interface. The core of the implementation generally follows our formalization: First, we gather constraints, following the rules in Figure 11. Then, we simplify these constraints and check for monomorphizability via the procedure outlined in Section 3.3.2. If monomorphizable, we compute their solution as their transitive closure. Finally, we perform specialization using the simplified constraints following Figure 13. We now highlight some features that our implementation has but the formalization in Section 3 does not.

4.1 Syntactic Extensions

In our implementation, functions, constructors, and methods are multi-arity and thus can take zero or more arguments. Each of those arguments can be an arbitrary expression and does not have to be a variable. We allow for local function definitions that close over variables in scope, instead of only top-level function definitions. All of these are trivial extensions that we omit from the formalization for clarity of presentation and to simplify our proofs.

4.2 Multiple Type Parameters

The most involved extension in our implementation is that type declarations, functions, constructors, and methods can bind multiple type parameters. We could naïvely handle this by tracking each type variable independently and monomorphizing each structure over the Cartesian product of the sets of types flowing into its type parameters. However, doing so potentially creates a large number of unused copies, whose combination of type arguments is never realized in the program. For example, if a function `def foo[A, B]()` in our polymorphic source language is only called as `foo[Int, String]()` and `foo[Bool, Float]()`, the specializations `def foo_Int_Float()` and `def foo_Bool_String()` in our monomorphic target language clearly are unused. To overcome this problem, we instead associate whole vectors of type parameters, in this example `[A, B]`, with a single monomorphization variable. We then track the flow of vectors, vector components, and monomorphization variables. This prevents the creation of unused copies in those cases.

5 Related Work

While monomorphization is an important implementation technique, few formalizations of it exist.

Griesemer et al. [2020] present a design for generics for the real-world language Go, where monomorphization to Featherweight Go (FG) is the implementation technique they propose for their core calculus Featherweight Generic Go (FGG). Like us, they have formalized monomorphization, and proven that it preserves typing and semantics. Their core calculus FGG, being based on Go, covers a different design space than our language LangPoly. FGG features structural subtyping, dynamic type assertions, and top-level instances, while LangPoly features data and interface types, existentials, and local anonymous instances. Algorithmically, our technique is different from theirs. We formulate our algorithm as a *total* constraint generation phase that tracks the flow of types through type variables, a *partial* constraint solving phase, and a *total* monomorphization phase. In contrast, they perform a single fixpoint computation as part of their type and instance collection phase. The instance collection phase in itself is *partial* and non-monomorphizability is detected with a mutually defined nomono predicate. In our view, these differences in goal, language, and algorithm make our technique much easier to understand, implement, and prove correct.

5.1 Implementation of Monomorphization

Monomorphization is a common technique in the compilation of ML-family languages. Benton et al. [1998] describe a compilation of SML to Java Bytecode, involving the “radical” decision to monomorphize functions. The MLton[4] compiler monomorphizes in a compilation to machine code. Tolmach and Oliva [1998] use monomorphization when compiling ML to Ada. In these cases, monomorphization is made simpler by SML’s lack of support for polymorphic recursion, higher-rank types, and existential types. In a richer language, Tanaka et al. [2018] use monomorphization in a plugin for generating C code from Coq. However, this implementation also does not handle first-class polymorphism or polymorphic recursion. Languages such as Rust[5] and C++ [Stroustrup 2013] make use of monomorphization, but fall back to boxing and dynamic dispatch in the case of existential types. An alternative to monomorphization is type erasure, which is used in compiling Java [Bracha et al. 1998]. Rather than specializing generic functions and structures, an erasure transformation removes all polymorphic information from them, resulting in monomorphic code. While type erasure has the advantage of avoiding code duplication, it suffers from performance penalties. Of course, erasure side-steps the issues of first-class polymorphism and polymorphic recursion, as there is no need to determine the set of types at which a function may be applied. JIT compilation allows for a middle-of-the-road approach, where monomorphization can be performed as needed at compile-time for primitive types, while code is shared for reference types. This technique, described by Kennedy and Syme [2001] for the .NET CLR, reduces the duplication of full compile-time monomorphization, while incurring some runtime cost for runtime monomorphization.

5.2 Algebraic Subtyping

Our approach is loosely inspired by recent work on type inference for algebraic subtyping [Dolan 2017; Dolan and Mycroft 2017; Parreaux 2020]. Unlike unification-based traditional Hindley-Milner type inference [Hindley 1969; Milner 1978] which gathers *equality constraints*, algebraic subtyping collects inequalities between types. Intuitively, these inequalities can be read as a flow of types through unification variables. This notion of flow cannot only give rise to very precise structural types [Binder et al. 2022], but also be used to improve type-error messages [Bhanuka et al. 2023]. In the present paper, our type-based flow analysis uses a similar intuition. However, while algebraic subtyping is interested in the flow through unification variables, we are interested in a more coarse-grained flow through type abstractions.

6 Conclusion

Monomorphization is a common technique for implementing parametric polymorphism, yet little research literature covers it. We have presented a distillation of monomorphization to its simple essence, the flow of types into type variables. Our technique supports higher-rank types and existential types—features that intuitively seem impossible to monomorphize. Our presentation has allowed us to identify four kinds of polymorphic recursion, where monomorphization is not possible. One of these, polymorphic packing, is a previously unidentified challenge. We have formalized our technique for transforming programs from a polymorphic nominal type system to a monomorphic one, and proven that our transformation preserves types and semantics, and we have implemented it in a prototype with several extensions. We believe that our work provides an elegant model for monomorphization, a step toward understanding the true limits of monomorphization. In the future, it would be interesting to study a translation of SystemF to our polymorphic source language, as well as a translation of our monomorphic target language to STLC with records.

[4]http://mlton.org/Monomorphise

[5]https://blog.rust-lang.org/2015/05/11/traits.html

7 Data-Availability Statement

We have implemented a prototype of our monomorphization technique, as described in Section 4. The implementation is a JavaScript application with an HTML interface. It includes the example programs for reproduction of results and allows custom input for reuse [Lutze et al. 2025].

Acknowledgments

The work on this project was supported by the Deutsche Forschungsgemeinschaft (DFG – German Research Foundation) – project number DFG-448316946.

References

Nick Benton, Andrew Kennedy, and George Russell. 1998. Compiling standard ML to Java bytecodes. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '98)*. Association for Computing Machinery, New York, NY, USA, 129–140. https://doi.org/10.1145/289423.289435

Ishan Bhanuka, Lionel Parreaux, David Binder, and Jonathan Immanuel Brachthäuser. 2023. Getting into the Flow: Towards Better Type Error Messages for Constraint-Based Type Inference. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 237 (oct 2023), 29 pages. https://doi.org/10.1145/3622812

David Binder, Ingo Skupin, David Läwen, and Klaus Ostermann. 2022. Structural refinement types. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Type-Driven Development* (Ljubljana, Slovenia) *(TyDe 2022)*. Association for Computing Machinery, New York, NY, USA, 15–27. https://doi.org/10.1145/3546196.3550163

Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. 1998. Making the future safe for the past: adding genericity to the Java programming language. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Vancouver, British Columbia, Canada) *(OOPSLA '98)*. Association for Computing Machinery, New York, NY, USA, 183–200. https://doi.org/10.1145/286936.286957

Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. 2000. Flow-directed closure conversion for typed languages. In *Programming Languages and Systems: 9th European Symposium on Programming, ESOP 2000 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2000 Berlin, Germany, March 25–April 2, 2000 Proceedings 9*. Springer, 56–71.

Stephen Dolan. 2017. *Algebraic Subtyping: Distinguished Dissertation 2017.* BCS, Swindon, GBR.

Stephen Dolan and Alan Mycroft. 2017. Polymorphism, Subtyping, and Type Inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 60–72. https://doi.org/10.1145/3009837.3009882

Richard A. Eisenberg and Simon Peyton Jones. 2017. Levity polymorphism. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 525–539. https://doi.org/10.1145/3062341.3062357

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

J.R. Hindley. 1969. The principal type scheme of an object in combinatory logic. *Trans. of the American Mathematical Society* 146 (Dec. 1969), 29–60. https://doi.org/10.2307/1995158

Anders Kiel Hovgaard, Troels Henriksen, and Martin Elsman. 2018. High-Performance Defunctionalisation in Futhark. In *International Symposium on Trends in Functional Programming*. Springer, 136–156.

Andrew Kennedy and Don Syme. 2001. Design and implementation of generics for the .NET Common language runtime. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, USA) *(PLDI '01)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/378795.378797

Paul Blain Levy. 1999. Call-by-Push-Value: A Subsuming Paradigm. In *Typed Lambda Calculi and Applications*, Jean-Yves Girard (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 228–243.

Matthew Lutze, Philipp Schuster, and Jonathan Immanuel Brachthäuser. 2025. *The Simple Essence of Monomorphization (Artifact)*. https://doi.org/10.5281/zenodo.14591555

Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17 (1978), 248–375. https://doi.org/10.1016/0022-0000(78)90014-4

Alan Mycroft. 1984. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*. Springer, 217–228.

Lionel Parreaux. 2020. The Simple Essence of Algebraic Subtyping: Principal Type Inference with Subtyping Made Easy (Functional Pearl). *Proc. ACM Program. Lang.* 4, ICFP, Article 124 (Aug. 2020), 28 pages. https://doi.org/10.1145/3409006

Bjarne Stroustrup. 2013. *The C++ programming language.* Pearson Education.

Akira Tanaka, Reynald Affeldt, and Jacques Garrigue. 2018. Safe Low-level Code Generation in Coq Using Monomorphization and Monadification. *Journal of Information Processing* 26 (2018), 54–72. https://doi.org/10.2197/ipsjjip.26.54

Andrew Tolmach and Dino P Oliva. 1998. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming* 8, 4 (1998), 367–412.

Stephen Weeks. 2006. Whole-program compilation in MLton. *ML* 6 (2006), 1–1.