Peridot: A Programming Language for Extensible Compilation

Eashan Hatti

Abstract

We present Peridot, a two-level language for high-performance, high-level programming. Peridot is split into two fragments, with one meant for metaprogramming and one meant for ordinary programming. Using metaprogramming, all program optimizations are implemented in the language itself. This allows programmers to easily implement performance-critical optimizations for their code, rather than relying on built-in general-purpose optimizations. First, we provide examples of optimizations implemented in Peridot. Second, we discuss the language in detail, elaborating upon the major aspects of its design. Third, we discuss the shortcomings of the language. Finally, we compare Peridot to related work, which includes multistage programming.

1 Introduction

High-level programming and program performance are at odds. High-level languages enable complex, pervasive abstractions, whereas high performance demands these abstractions be reduced to a minimum. Thus, an optimizing compiler is an essential part of a high-level language that seeks to accomplish both goals. However, even the most sophisticated optimizer can fall short when presented with abstraction it was not built to deal with. As programmers develop new abstractions, a choice must be made between the following options:

1. Augmenting the optimizer to deal with these new abstractions

2. Abandoning performance in exchange for high-level programming

3. Abandoning high-level programming in exchange for performance

Option 1 is the most attractive, as it would allow our programs to be both high-level and high-performance. Options 2 and 3 are not attractive since we must abandon one of the two. However, Option 1 has shortcomings too! As a language's library ecosystem grows, so does the number of abstractions that programmers will use. If these abstractions are to perform well, the optimizer must build-in new optimizations to target them, or these abstractions must rely on existing general-purposes optimizations. However, neither of these options are ideal. Relying on general-purpose optimizations is often not as effective as needed. Because the optimizations are general, they often will fail to fire on code for difficult to diagnose reasons. Coaxing them to fire often involves construing code into a less readable and less maintainable form, which is a loss of high-level programming. Building-in new optimizations is even less appealing for the following reasons:

1. It presents too significant of an effort for the compiler developers. The library ecosystem of popular languages can be enormous. For instance, as of this writing the Rust programming language's package index has 92654 entries. Implementing optimizations for even just the most widely used ones poses a significant challenge.

2. Updates to libraries can break optimizations. For instance, an optimization which depended upon a library's API will have to be reimplemented or dropped entirely when that API changes. This again increases the workload for the compiler developers - it requires them to go back and update the compiler to facilitate these new optimizations. If they do not, performance suffers.

3. Compiler developers may not have the domain-specific knowledge to implement optimizations for a library. A library and its optimizations may be complex. Requiring compiler developers to be familiar with the complexity of many libraries again increases the workload to an intolerable amount.

Since continuously building-in optimizations for each new abstraction is impractical, general-purpose optimizations must be used. Thus, the language becomes more high-level, yet the optimizer – and in turn program performance – falls behind. Programmers are again confronted with a choice between high-level programming and high-performance programs. Taking all of this into account, it makes sense why, despite their shortcomings, options 2 and 3 are often chosen. Option 1 would be extremely valuable, but it is also costly to implement.

To investigate a potential solution to this problem, we developed Peridot, a novel programming language. Peridot's main contribution is that it makes option 1 viable; it enables library developers to easily extend the language with new optimizations themselves. Peridot's key idea is that optimizations are implemented as *metaprograms*. This allows the optimization workload to be taken off the compiler developers' hands and given to the library developers: a much larger set of people who can specialize in each problem domain. This allows optimizations to be much more effective while also being much more numerous.

To facilitate this, Peridot uses logic programming in a novel way. The language's basis lies in λProlog and two-level type theory (2LTT). It is split into two fragments or "levels", one level supporting logic programming (the "meta level"), the other supporting high-level functional programming[1] (the "object level"). Optimizations are implemented in the former fragment as metaprograms. The language's basis in 2LTT allows it to cleanly separate these two levels using *types* - it is essentially two languages that look and feel like one language. For implementing optimizations, the meta level's basis in λProlog allows metaprograms to be written declaratively, avoid issues such as the *phase-ordering problem*, deal with variable binding easily, automatically expose equalities between programs, be easily extensible, and more. Peridot reconciles high-level programming and high-performance by giving users easy and complete control over the optimization and compilation process.

1.1 Implementation

A demo implementation is available here. It is important to note that as of this preprint, some features have not been implemented. These include `let` insertion, syntactic sugar for `variant` and `metavariant`, other minor syntactic sugar, and the committed choice operator.

1.2 Overview

- In Section 2 we show the basic syntax and semantics of the language.
- In Section 3 we provide several examples of backend components implemented using metaprogramming.
- In Section 4 we discuss Peridot in more detail, elaborating upon the major parts of the language's design.
- In Section 5 we address Peridot's shortcomings and consider potential changes to the language that would remedy these shortcomings.
- In Section 6 we compare Peridot to related work.

1.3 Acknowledgements

András Kovács's Elaboration Zoo was immensely helpful during the implementation of Peridot. We strongly recommend it to anyone learning to implement dependent type systems.

1. This choice is not central to the design. A functional language was picked because it has many interesting optimization problems that can serve as examples

2 Basic Syntax

In this section we will show the basic syntax and semantics of the language. As stated before, Peridot is a language split into two fragments. The functional fragments is called the "object level", the logic fragment is called the "meta level". The object level supports lazy, dependently typed, functional programming. The meta level supports typed logic programming.

2.1 Object Level

Values are assigned names with `def`.

```
def true2: Bool = true
```

The type annotation can be omitted, in which case the type will be inferred.

```
def true2 = true
```

Functions are introduced with `fun(X) => BODY`. Function types are introduced with `Fun(TY1) -> TY2`. Function calls use parenthesis.

```
def not: Fun(Bool) -> Bool =
  fun(b) =>
    if b {
      false
    } else {
      true
    }

def false2 = not(true)
```

There is also sugar for the above sort of declaration.

```
fun not(b: Bool) -> Bool =
  if b {
    false
  } else {
    true
  }
```

The language also has `let` bindings. `let` bindings can bind any sort of top-level declaration - `variant`s, `fun`s, etc.

```
let {
  def true2 = true
} in {
  not(true2)
}
```

Algebraic datatypes are introduced with `variant`. Type parameters are listed in parenthesis next to the name.

```
variant LinkedList(A) {
  nil
  cons(A, LinkedList(A))
}
```

Pattern matching is introduced with `match`. The match must be exhaustive. Type parameters are applied to generic types with parenthesis, just like function calls.

```
fun not_list(list: List(Bool)) -> List(Bool) =
  match list {
    nil => nil
    cons(b, rest) => cons(not(b), not_list(rest))
  }
```

Type variables are introduced with backticks

```
fun map(f: Fun(`A) -> `B, list: List(`A)) -> List(`B) =
  match list {
    nil => nil
    cons(x, rest) => cons(f(x), map(f, rest))
  }
```

2.2 Meta Level

Datatypes are introduced with `metavariant`. Although syntactically it appears identical to `variant`, they are desugared to very different constructs. This will be elaborated upon in Section 4.

```
metavariant Nat {
  zero
  plus_one(Nat)
}
```

Predicates are declared with `pred`. This is unlike Prolog where we do not have to declare predicates at all. However, allowing this complicates the implementation somewhat, so we require declaration-before-use.

```
pred Add(Nat, Nat, Nat)
```

We introduce rules for a predicate with `axiom`. `axiom`s use "`:-`" syntax when the rule has a body.

```
axiom NAdd(zero, `n, `n)

axiom
  NAdd(plus_one(`n), `m, plus_one(`j))
  :- NAdd(`n, `m, `j)
```

Backticked variables are the equivalent of uppercase variables in Prolog. For example, `` `m `` translates to `M`, and `` `n `` translates to `N`.

Definitions are introduced with `metadef`.

```
metadef one = plus_one(zero)
```

We introduce variables that may have solutions assigned to them ("metavariables") with `metavar`. We can then `query` a predicate to assign those solutions.

```
metavar n: Nat
query Add(one, one, n)

Solutions:
  n = plus_one(plus_one(zero))
```

Again, `metavar`s usually do not have to be declared in logic languages but requiring it simplifies the implementation.

3 Implementing a Backend

We will now provide practical examples of transformations and optimizations implemented with Peridot's metaprogramming. These examples show off the various features of its metaprogramming that make Peridot uniquely suited for implementing such backend components. Although the focus is not on the object level, we will show a few samples of object-level programs as well.

First, this section assumes familiarity with λProlog and `let` insertion, and basic familiarity with staging constructs (`Code`, quoting and splicing). The details that are glossed over in this section will be revisited in detail in Section 4. We encourage reading that section first if the following examples are not clear. Second, the implementations of these transformations will not necessarily be the most efficient or effective. Their primary purpose is to introduce the language.

3.1 Constant Folding

Constant folding is an optimization that executes small, terminating computations at compile time. For instance, constant folding would take the following program

```
def foo = add(2, 3)
def bar = mul(foo, 6)
```

and transform it into

```
def foo = 5
def bar = 30
```

We will now implement this as a metaprogram. We start by declaring our predicate.

```
pred ConstantFold(Code(‘A), Code(‘A))
```

`Code(A)` is the type of object programs with type `A` that have been *quoted*. "Quoting" refers to the same concept in languages like Scheme and Clojure - it takes object-level code and turns it into *data* that metaprograms can manipulate. Likewise, "splicing" takes this data and turns it back into an object-level program. We will see both shortly. Quoting is written `<e>` and splicing is written `~e`, where `e` is an expression.

```
axiom
  ConstantFold(‘head<(fun(x) => ~‘body(<x>))(~‘arg)>, ‘body(‘arg))
  :- Terminating(‘head, ‘arg)

axiom
  ConstantFold(<~‘head(~‘arg)>, ‘e)
  :- ConstantFold(‘head, ‘head2)
  ,  ConstantFold(<~‘head2(~‘arg)>, ‘e)

pred Terminating(Code(‘A), Code(‘B))
/* Rules not shown, refer to below */
```

These are the central two rules in our implementation of constant folding. They perform compile-time reduction on function application. The first rule attempts to evaluate the application head as a function. It then extracts the body of this function into `body` and substitutes the argument (this will be discussed further in Section 4.2.2). If that rule fails, the second rule will attempt to constant fold the application head and then constant fold the new application. It is important to note that this predicate does *not* automatically handle termination, thus we need to check termination ourselves with the `Terminating` predicate. Termination analysis is beyond the scope of this paper, so we will not define it here. This limitation will be discussed in Section 5.

The rest of the rules simply apply the predicate to the subterms of compound terms (`if` expressions, `let` bindings, etc), so we will omit them. The predicate is extremely simple, but can constant fold *any* function application.

3.2 Specialization

3.2.1 Definition

Specialization is an optimization that specializes generic functions to certain types. This exposes more concrete type information to the backend, which aids additional optimizations. Specialization will work as follows:

The object program will be traversed and at every `let` binding a `letlocus` will be inserted. `letlocus`es serve as points for `let` insertion. The binding's name will then be added to a mapping sending locuses to variables.

```
pred Bound(Code(‘A), Locus)

pred Specialize(Code(‘A), Code(‘A))

axiom Specialize(
  <letdef ‘e ‘x in ~‘body>
  letlocus ‘loc in ‘body2)
  :- Bound(‘x, ‘loc) => Specialize(‘body, ‘body2)
```

The `P => Q` operator is *implication*. It adds `P` to a local stack of facts and then queries `Q`.

When we encounter a function application, we will first check if the application head is a `let` bound variable. If it is, we will retrieve the locus mapped to that variable and proceed. We will then check if the application is terminating. Again, termination analyses are out of this paper's scope, so the rules of this predicate will not be shown.

If the application is terminating, we will proceed. There is a mapping in scope from applications to variables assigned to specialized definitions. If the application is present in this mapping, we retrieve the `let`-bound variable containing the specialized definition and simply return it, in which case we are done. If it is not present, we will need to perform the specialization. Generating the definition requires some circular thinking: We want to add the specialized definition to the mapping and *then* specialize the definition. If we do not do this, specialization would loop on recursive functions.

To accomplish this, we will simply assign the `let` binding containing the specialized definition to a metavariable. This binding will then be added to the mapping and the definition specialized. When the same application is encountered inside the recursive definition, the predicate will find the application in the mapping and return the `let` bound variable. The specialized definition will be assigned to the metavariable once the predicate completes. Specialization will have generated a recursive binding.

This rule also uses the committed choice operator (`P | Q`). This way it only choses one of the two options (using an earlier specialization or generating a new one) instead of nondeterministically choosing both.

```
pred Defined(Code(‘A), Code(‘B), Code(‘C))

axiom
  Specialize(<~‘head@(fun(x) => ~‘body(<x>))(~‘arg)>, ‘e)
  :- Bound(‘head, ‘loc)
  ,  Terminating(‘head, ‘arg)
  ,  ( Defined(‘head, ‘arg, ‘e)
```

```
    | Defined(‘head, ‘arg, ‘def) =>
      ( GenLet(‘loc, ‘speced, ‘def)
      , Specialize(‘body(‘arg), ‘speced)
      )
    )
```

Finally, we need to handle specialization on multi-argument functions. The predicate will specialize the application head. If this succeeds, it will return a `let`-bound variable, which can be applied to the argument. Then the application itself will be specialized.

```
axiom
  Specialize(<~‘head(~‘arg)>, ‘e)
  :- Specialize(‘head, ‘head2)
  ,  Specialize(<~‘head2(~‘arg)>, ‘e)
```

As with constant folding, this predicate requires some additional rules that simply traverses non-application terms and applies the predicate to their constituents. The rules all follow this form, so the rest will be omitted.

```
axiom
  Specialize(
    <if ~cond {
       ~true_body
     } else {
       ~false_body
     }>,
    <if ~cond2 {
       ~true_body2
     } else {
       ~false_body2
     }>)
  :- Specialize(cond, cond2)
  ,  Specialize(true_body, true_body2)
  ,  Specialize, false_body2)
```

3.2.2 Results

We will now use the predicate. Recall that backticked variables are transformed into implicit parameters. For example, `map's` signature `Fun(Fun(‘A) -> ‘B, List(‘A)) -> List(‘B)` is equivalent to `Fun(inf A: Type, inf B: Type, Fun(‘A) -> ‘B, List(‘A)) -> List(‘B)`.

```
fun not(b: Bool) -> Bool =
  if b {
    false
  } else {
    true
  }

fun map(fn: Fun(‘A) -> ‘B, list: List(‘A)) -> List(‘B) =
  match list {
    nil => nil
    cons(x, rest) => cons(fn(x), map(fn, rest))
  }

metavar spec_term
query Specialize(map(not, cons(true, cons(false, nil))), spec_term)
```

The above query results in the following program

```
fun not(b: Bool) -> Bool =
  if b {
    false
  } else {
    true
  }

fun map(fn: Fun(‘A) -> ‘B, list: List(‘A)) -> List(‘B) =
  match list {
    nil => nil
    cons(x, rest) => cons(fn(x), map(fn, rest))
  }

fun gen6948(list: List(Bool)) -> List(Bool) =
  match list {
    nil => nil
    cons(x, rest) => cons(not(x), gen6948(rest))
  }

Solutions:
  spec_term = gen6948(cons(true, cons(false, nil)))
```

3.3 List Fusion

Fusion is an optimization that removes intermediate data structures. For example, `(map f ∘ map g) list` is semantically equivalent to `map (f ∘ g) list`, but the former performs two traversals and creates one extra list. The latter performs one traversal and does not produce an extra list.

The fusion system we will implement here is called `foldr/build` fusion. The idea behind is to have a small set of core functions that all others are defined in terms of. We then define fusion for this core set and get fusion for the derived functions for free. It is simple to implement in Peridot while simultaneously being very beneficial for idiomatic functional programs.

Let us first define `foldr` and `build`.

```
variant List(A) {
  cons(A, List(A)),
  nil
}

fun foldr(f: Fun(‘A, ‘B) -> ‘B, init: ‘B, list: List(‘A)) -> ‘B =
  match list {
    nil => init
    cons(x, rest) => f(x, foldr(f, init, rest))
  }

fun build(g: Fun(‘A, ‘B) -> ‘B) -> ‘B =
  g(cons, nil)
```

We will now define the optimizer itself. This rule implements the fusion rule `foldr(f, z, build(g)) = g f z`.

```
pred Fuse(Code(‘A), Code(‘A))

axiom Fuse(<foldr(~‘f, ~‘init, build(~‘g))>, <~‘g(~‘f, ~‘init)>)
```

We are already done! The rest of the rules simply traverses compound terms and applies the predicate to their subterms, like in the previous section.

4 Peridot in Detail

In the previous section we discussed Peridot's features informally for the sake of explanation. We will now transition to explaining the major aspects of the language in more detail.

4.1 Two-Level Type Theory

4.1.1 Splitting the Language

Peridot's design is centered around two-level type theory (2LTT). In 2LTT, the language is split into two *levels*, or "fragments" of the language. This is realized using two universes: `Type` for object-level types and `MetaType` for meta-level types. For instance, recalling some types from the previous section, we have `Bool : Type` and `Locus : MetaType`.

A key aspect of 2LTT is that all types must stay on their own level. For instance, types `Fun(Locus) -> Locus` and `MetaFun(Bool) -> Bool` are invalid because `Fun` is object-level while `Locus` is meta-level, and `MetaFun` is meta-level while `Bool` is object-level. To illustrate, the formation rule for `MetaFun` is

$$\frac{\Gamma \vdash A : \text{MetaType} \quad \Gamma, x : A \vdash B : \text{MetaType}}{\text{MetaFun}(x : A) \to B}$$

This rule makes explicit that `A : MetaType` and `B : MetaType`! Meta-level and object-level values cannot intermingle. The separation of the levels is done purely using types, as opposed to ad-hoc syntactic restrictions or other such methods.

Declarations are also separated in this manner:

- `def NAME: TY = VAL` requires that `TY : Type`
- `metadef NAME: TY = VAL` requires that `TY : MetaType`
- `axiom TY` requires that `TY : MetaType`
- `metavar NAME: TY` requires that `TY : MetaType`

4.1.2 Moving Between Levels

So far, the system is extremely limited since values cannot cross the levels at all. As seen in Section 3, there are three operations which enable the levels to interact:

- Lifting, or `Code(A)`. Given `A : Type`, we have `Code(A) : MetaType`. `Code(A)` is is the type of quoted object-level code
- Quoting, or `<v>`. Given `v : A` and `A : Type`, we have `<v> : Code(A)`. Quoted code is meta-level data representing an object-level program
- Splicing, or `~x`. Given `x : Code(A)`, we have `~x : A`. Splicing takes quoted code and inserts it back into an object-level program

Note that none of these constructs can be given function types because they take values from one level to another. Say we attempted to type lifting as

```
Code: MetaFun(Type) -> MetaType
```

It would be invalid because a meta-level type (`MetaFun`) contains an object-level type (`Type`).

4.1.3 Significance

With these simple rules, 2LTT gives us a clean type-based way to stratify languages. We essentially have *two languages* that look and feel like one language. This gives us a large amount of design flexibility: Since we're essentially designing two languages they can cater to entirely different domains. We can have entirely different sorts of types and features on each level.

In Peridot's case, the object level is designed for general purpose programming while the meta level is designed specifically for metaprogramming. Without 2LTT, it would be much harder to express this two-part design in a clean and user-friendly way. In the next sections we will explain the object language's and meta language's designs in detail.

Note 1. We will use "object level" and "object language" interchangeably, as well as "meta level" and "meta language" interchangeably

4.2 Meta Level

The object language is one half of Peridot, the other half is the *meta language*. Logic programming with hereditary Harrop formulas, pattern unification, and higher-order abstract syntax (HOAS) is especially effective for expressing *program transformations*, which are what a compiler backend is composed of.

Note 2. This section assumes familiarity with logic programming using *Horn clauses*, i.e Prolog

4.2.1 Basic Logic Programming

Peridot's logic programming allows program transformations to be extremely *declarative* - the language handles many aspects of these transformations automatically. Basic logic programming handles two of these:

1. *Object program equivalences*. Logic programs execute via unification, which tries to find where terms are equal. In Peridot, this includes applying object-level computation rules! We will assume an `Int` type and integers with the obvious computation rules. For instance, say we have the following definitions

   ```
   pred OptimizeInt(Code(Int), Code(Int))
   axiom OptimizeInt(<(‘x * ‘y) / ‘y>, ‘x)

   def foo: Int = 3 * 2
   ```

   Querying this predicate with the following code will still deduce that `x = <3>`!

   ```
   metavar x: Code(Int)
   query Optimize(<foo / 2>, x)

   Solutions:
     x = <3>
   ```

   `foo * 2` is not syntactically equivalent to `(‘x * ‘y) / ‘y`, so how did this compute? The answer is that query automatically evaluated `foo` when attempting to unify `foo / 2` with `(‘x * ‘y) / ‘y`; it automatically reduced `foo / 2` to `(3 * 2) / 2`, which then easily produces the results `‘x = <3>`. This scales to larger cases such as inlining entire functions.

Unification will apply the regular computation rules of the object language when performing search. These computation rules will change depending on the features of the object language. For instance, in an impure language evaluating `foo` would not be valid, since `foo` may cause side effects.

Note 3. There are helpful equivalences which cannot be realized just with built in computation rules. This will be discussed further in Section 5.1.

Peridot also automatically handles simpler syntactic equivalences. For instance, `2 + 2` is considered equivalent to `(+) 2 2`. Unification does *not* operate on raw syntax. Rather, it operates on a "core language", which all terms are translated to. This translation eliminates simple syntactic differences between programs, which is a common source of frustration in other metaprogramming frameworks. In the core language, `2 + 2` and `(+) 2 2` have exactly the same representation.

2. *The phase-ordering problem*. One classic problem is that applying some optimizations may block other, more beneficial optimizations. For instance, inlining may eliminate a function call, but may block common subexpression elimination (CSE). Nondeterminism presents a solution to this by simply producing *all* possible optimizations - all optimizations can be applied in every order, the results collected, and then a cost predicate applied to find the best result. This will be elaborated upon in Section 4.2.4.

```
pred Optimize(Code(‘A), Code(‘A))
axiom Optimize(‘x, ‘y) :- CommonSubexprElim(‘x, ‘y)
axiom Optimize(‘x, ‘y) :- Inline(‘x, ‘y)
...
metavar optimized_prog
query Optimize(prog, optimized_prog)
```

This query will produce multiple solutions for `optimized_prog`, in some of which where inlining was performed and others where CSE was performed. This is as opposed to only one, possibly suboptimal, path being taken.

Note 4. Peridot's implementation of nondeterminism suffers from high space consumption, see Section 5.2.

4.2.2 Pattern Unification

Basic logic programming automatically handles two aspects of program transformations, but there is a third which it cannot: *variable binding*. We would like to handle variable binding declaratively: without having to reimplement substitution. Peridot accounts for this by extending basic logic programming with *pattern unification*. Consider this predicate which inlines a function call

```
pred Inline(Code(‘A), Code(‘A))
axiom Inline(<(fun(x) => ~(‘f(<x>)))(~‘arg)>, ‘f(‘arg))
```

The key term is `~(‘f(<x>))`. This sort of term is dealt with as a special case by pattern unification. When unifying `~(‘f(<x>))` with a `fun`ction's body, it solves `‘f` as follows:

1. Check that `‘f` is applied to a series of distinct quoted or non-quoted variables
2. Check that the body only contains these applied variables
3. Wrap the body in a `metalam` for each of these variables.

4. Wrap each bound variable in the body in a splice

5. Wrap the whole body in a quote

For instance, querying

```
def not: Fun(Bool) -> Bool =
  fun(x) =>
   if x {
     false
   } else {
     true
   }
metavar inlined_not: Code(Bool)
query Inline(<not(true)>, inlined_not)
```

Would produce this solution for `‘f` while executing (for clarity, the type of ‘f is also written)

```
Solutions:
  ‘f: MetaFun(Code(Bool)) -> Code(Bool)
    =
      metafun(x) =>
        <if ~x {
          false
        } else {
          true
        }>
```

Thus, the second argument of the `Inline` rule - `‘f(‘arg)` - substitutes `‘arg` inside the body to get

```
<if ~<true> {
  false
} else {
  true
}>
```

Splices and quotes cancel out, and we get our final, inlined function:

```
Solutions:
  inlined_not =
    <if true {
      false
    } else {
      true
    }>
```

In short, pattern unification allows us to extract the bodies of binders to `metafun`ctions which perform substitution.

4.2.3 Hereditary Harrop Formulas

Another traditionally difficult aspect of program transformations is handling fresh variable generation. Say we were writing a predicate that replaces occurrences of a function's variable with `true`.

```
pred Replace(Code(‘A), Code(‘A))

axiom
```

```
  Replace(<fun(x) => `body>, <fun(x) => `body2)
  :- Replace(`body, `body2)

axiom
  Replace(`x, <true>)
  :- ???
```

How can we track what variables were bound? What we want is the ability to generate our own variables which we can track and substitute inside the body. We cannot generate variables using a side effect because the meta language is pure. `VarState` contains a source of fresh variables, which we increment every time we generate a new one.

```
pred Replace(Vars, VarState, VarState, Code(`A), Code(`A))

axiom
  Replace(
    `vars,
    `varstate1,
    `varstate3,
    <fun(x) => ~`body(<x>)>,
    <fun(x) => ~`body2(<x>))
  :- FreshVar(`varstate1, `varstate2, `var)
  ,  Replace(
       insert(`var, `vars),
       `varstate2,
       `varstate3,
       `body(`var),
       `body2(`var))

axiom
  Replace(`vars, `varstate, `varstate, `x, <true>)
  :- Contains(`vars, `x)
```

This is quite ugly. First, we must manually pass the new `VarState` around to every predicate that uses it, meaning there are several variables which exist only to pass states around. This state passing approach is error-prone for that reason. Furthermore, statically typing this program is complicated. We would like a *declarative* approach to variable generation, one rooted in a logic. There is an extension to Horn clauses which does exactly this: *Hereditary Harrop formulas*.

```
pred IsBound(Code(`A))

pred Replace(Code(`A), Code(`A))

axiom
  Replace(<fun(x) => ~`body(<x>)>, <fun(x) => ~`body2(<x>)>
  :- forall(x) -> IsBound(x) => Replace(`body(x), `body2(x))

axiom
  Replace(`x, <true>)
  :- IsBound(`x)
```

The two extensions are universal quantifiers in predicate bodies (`forall`) and implication `=>`.

Note 5. The implication syntax and function syntax both use `=>`. The two should not be confused.

How do these new constructs execute? `forall(x) => P` can be understood as generating a fresh variable which can be used in `P`. `P => Q` can be understood as adding `P` to the list of facts in scope *locally* when querying `Q`. This list grows and shrinks like a stack. Observe the trace of a query:

```
metavar tm
query Replace(<fun(x) => x>, tm)
```

We attempt to prove the first rule.

```
Stack:
Goal: Replace(<fun(x) => ~‘body(<x>)>, <fun(x) => ~‘body2(<x>)>)
  ‘body = metafun(x) => <~x>
  tm = <fun(x) => ~‘body2(<x>)>
```

This succeeds, so we move on to the body of the predicate

```
Stack:
Goal: forall(x) -> IsBound(x) => Replace(‘body(x), ‘body2(x))
```

We are attempting to prove a `forall`. A fresh variable is generated and bound to `x`. We will call this variable `#0`. We move on to the body of the `forall`.

```
Stack:
Goal: IsBound(#0) => Replace(‘body(#0), ‘body2(#0))
```

Now we must prove an implication. This will add its left hand side to the stack of facts and attempt to prove the right hand side.

```
Stack: IsBound(#0)
Goal: Replace(‘body(#0), ‘body2(#0))
```

This goal matches the second rule, so we attempt to prove its body.

```
Stack: IsBound(#0)
Goal: IsBound(<~#0>)
```

Quotes and splices cancel out, so this rule matches and we are done. Te predicate has solved `tm = <fun(x) => ~<true>>`. The extension of Horn clauses to hereditary Harrop formulas gives us the tools to handle variable generation and binding *declaratively*. Plumbing of fresh variable state and keeping track of scope is unnecessary.

4.2.4 Pruning Undesired Solutions

As discussed in Section 4.2.1, nondeterminism allows us to produce all possible results of a combination of optimizations, thus obviating the phase-ordering problem. However, eventually we have to determine a single best solution. Peridot allows the solutions of a `metavar` to be iterated and pruned. For instance, say we have the following query:

```
pred Foo(Code(Bool))
axiom Foo(<true>)
axiom Foo(<false>)

metavar b: Code(Bool)
query Foo(b)

Solutions:
  b = <true>
  b = <false>
```

This query has produced multiple solutions for `b`. We can amend our query to prune all the solutions which are not equal to `<true>`

```
query
  with Foo(b)
  for b as x {
    Equal(b, <true>)
  }

Solutions:
  b = <true>
```

The `with P for V as X in { Q }` executes with the following steps:

1. Query `P`
2. Gather all the solutions for `V`. Iterate over the solutions, assigning each one to the variable `X` in turn
3. Query `Q`. If the query fails, prune that solution for `V`. It it succeeds, keep that solution

When there is no `P` predicate to be queried, `with P` may be ommitted.

Returning to optimization, the following query optimizes some program `main`, then prunes the results to find the best. The cost of a program is summarized as a `Nat` and determined by the `Cost` predicate. `CostsLess` determines if the first program costs less than the second program.

```
pred Cost(Code(Int), Nat)
/* ‘Cost‘ will remain abstract */
pred Less(Nat, Nat)
/* ‘Less‘ will remain abstract */

pred CostsLess(Code(Int), Code(Int))
axiom
  CostsLess(‘prog1, ‘prog2)
  :- Cost(‘prog1, ‘cost1)
  ,  Cost(‘prog2, ‘cost2)
  ,  Less(‘cost1, ‘cost2)

metavar optimized_main: Code(Int)
query
  with Optimize(main, optimized_main)
  for optimized_main as prog1 {
    for optimized_main as prog2 {
      CostsLess(prog1, prog2)
    }
  }
```

This query will loop over all possible results and prune all except the one which costs the least.

4.2.5 `let` Insertion

As demonstrated in Section 3, it is useful to be able to generate `let` bindings nonlocally. `let` insertion in Peridot works via two constructs: `letlocus LOC in BODY`, and the `GenLet` predicate. `letlocus` marks a point where `let` bindings may be inserted, and `GenLet` inserts a `let` binding at a given locus with a given definition, returning the `let`-bound variable. `GenLet` also uses *memoization*: If a `let` bound definition is already present in a locus, `GenLet` will return that `let`-bound variable instead of generating a new definition. For instance, the following query

```
pred Foo(Code(‘A), Code(‘A))

axiom
  Foo(
    letlocus ‘loc in ‘_,
    letlocus ‘loc in <struct { a = ~‘var1, b = ~‘var2 }>)
  :- GenLet(‘loc, <true>, ‘var1)
  ,  GenLet(‘loc, <true>, ‘var2)

metavar term
query Foo(letlocus ‘loc in ‘_, term)
```

will produce this solution

```
Solutions:
  term =
    let {
      def gen6246 = true
    } in {
      struct { a = gen6246, b = gen6246 }
    }
```

instead of this solution

```
Solutions:
  term =
    let {
      def gen6246 = true
      def gen7273 = true
    } in {
      struct { a = gen6246, b = gen7273 }
    }
```

The latter is without memoization: an additional binding was generated even though they bind identical terms. This is especially useful in cases like specialization (Section 3.2) where we want to avoid code size explosion.

`letlocus` is not strictly needed. We could infer where to place `let` bindings based on the variables present in the definition being bound, but we did not implement this in order to simplify the implementation. Inferring placements also makes memoization less reliable.

4.2.6 Algebraic Datatypes

Desugaring for the meta level's ADTs is much simpler than the object level's (Section 4.3.3). The type constructor and data constructors are each translated to an `axiom`. For instance, the following program

```
metavariant List(A) {
  nil
  cons(A, List(A))
}
```

is lowered to

```
axiom List: MetaFun(MetaType) -> MetaType
axiom nil: List(‘A)
```

```
axiom cons: MetaFun(‘A, List(‘A)) -> List(‘A)
```

4.2.7 Adding New Primitives

We can also add new primitives to the object language as `axiom`s. For instance, primitive integers could be added as

```
axiom IntC: Code(Type)
axiom oneC: Code(~Int)
axiom zeroC: Code(~Int)
axiom addC: Code(Fun(~Int, ~Int) -> ~Int)
axiom multiplyC: Code(Fun(~Int, ~Int) -> ~Int)
```

We could then define spliced versions for convenience

```
def Int = ~IntC
def one = ~oneC
def zero = ~zeroC
def add = ~addC
def multiply = ~multiplyC
```

The backend can then be extended with rules to compile these new primitives.

Of course, these new primitives do not have any computational behavior. For instance, `add(one, zero)` does *not* reduce to `one`. This will be discussed further in Section 5.1.

4.3 Object Level

The object language supports lazy, pure (except for nontermination), dependently typed, functional programming. It is a fairly standard functional language where functions are defined by recursion and pattern matching. Dependent records are the only atypical feature of the object language, so we will discuss them at length.

It is important to note that there is no hard, formal reason why this design of the object level was chosen. Peridot's two-level design with logic programming on top would work with any sort of language, for instance a low-level imperative language. A functional object level was chosen because the author is most familiar with implementing functional languages. The following sections are not central to the language's design goals.

Note 6. We assume familiarity with dependent types

4.3.1 Type System

The object level's type system is composed of only three[2] primitives: dependent function types (`Fun(x: A) -> B`), dependent record types (`Struct { a: A, b: B }`), and a `Bool` type. The object language is *structurally typed*, meaning all complex types are composed of these primitives.

Function types are standard, and arguments may be marked `inf` for implicit. For instance, the identity function may be written and called as

```
def id: Fun(inf A: Type, A) -> A =
  fun(x) => x
```

2. Although new primitives can be added using the meta language

```
def example = id(true)
```

Dependent record types are introduced with `Struct`, the records themselves introduced with `struct`, and field selection uses a dot (`.`). The following is a trivial example to demonstrate the syntax

```
def Ty: Type = Struct { A: Type, x: A }

def val: Ty = struct { A = Bool, x = true }

def Bool2: Type = val.A
def true2: Bool = val.x
```

Dependent records are very interesting, as they can be used to implement algebraic datatypes and ML-style modules. These will be covered in the following sections.

4.3.2 Purity

The object language is pure - it does not implicitly allow side effects. This does have a significant effect on the language as a whole, in that it increases the number of program equivalences that can be automatically deduced. As covered in Section 4.2.1, the language will attempt to equate programs by applying object-level computation rules. If the object level were impure, δ-equivalence (reduction of `let`s to their bindings substituted in their bodies) and β-equivalence (reduction of function application) would not be trivially valid. In both cases side effects may be duplicated or discarded, in the latter case due to lazy evaluation.

However as referenced before, this is does not mean making the object level impure is an invalid idea. For instance, a language similar to Peridot but with a low-level imperative language on the object-level would be very interesting to see.

4.3.3 Algebraic Datatypes

Dependent records can be used to implement ADTs as typesafe tagged unions. The `tag` field determines the variant, and the `data` field carries the payload of the union.

```
def Unit: Type = Struct {}
def unit: Unit = struct {}

def Nat: Type =
  Struct {
    tag: Bool,
    data:
      if tag {
        Unit
      } else {
        Nat
      }
  }

def zero: Nat =
  struct {
    tag = true,
    data = unit
  }
```

```
def plus_one: Fun(Nat) -> Nat =
  fun(nat) => struct {
    tag = false,
    data = nat
  }
```

In fact, the ADT syntax from Section 2.1 is just syntactic sugar for the above sort of declarations. The sample below is directly translated to the sample above.

```
variant Nat {
  zero
  plus_one(Nat)
}
```

Because the object level is structurally typed, any `variant` declarations with the same "shape" will be considered equal - they are translated to the same `Struct` type.

Finally, `match` syntax is directly translated to use of `if` and field selection. Continuing our `Nat` example, the following sample

```
match plus_one(zero) {
  zero => foo
  plus_one(n) => bar(n)
}
```

is translated to the following. We `let`-bind `val` in order to avoid duplicating it in the branches.

```
let {
  def val = plus_one(zero)
} in {
  if val.tag {
    foo
  } else {
    bar(val.data)
  }
}
```

4.3.4 ML-Style Modules

Dependent records also allow us to express ML-style modules: Signatures are record types, structures are records, and functors are ordinary functions. For instance, here are modules for ordering and sets

```
variant Ordering {
  lt
  eq
  gt
}

def ORD: Type = Struct {
  T: Type,
  compare: Fun(T, T) -> Ordering
}

def BoolOrd: ORD = struct {
  T = Bool,
  compare = fun(b1, b2) =>
```

show examples and the motivation for this

```
      if b1 {
        if b2 {
          eq /* true == true */
        } else {
          gt /* true > false */
        }
      } else {
        if b2 {
          lt /* false < true */
        } else {
          eq /* false == false */
        }
      }
  }

  def SET: Type = Struct {
    ElemT: Type,
    SetT: Fun(Type) -> Type,
    empty: SetT(ElemT),
    insert: Fun(SetT(ElemT), ElemT) -> SetT(ElemT),
    remove: Fun(SetT(ElemT), ElemT) -> SetT(ElemT),
    member: Fun(SetT(ElemT), ElemT) -> Bool
  }

  variant List(A: Type) {
    nil
    cons(A, List(A))
  }

  def ListSet: Fun(ORD) -> SET =
    fun(ord) => struct {
      ElemT = ord.T,
      SetT = List,
      empty = nil,
      ... /* Other fields ommitted for space */
    }
```

Additionally, we can modify signatures through *record patching*, which functions similarly to SML's `where type` and Rust's `TRAIT<Item = T>`. For instance, we can modify the `SET` signature to only hold `Bool` values with `patch SET { ElemT = Bool }`. The typechecker translates this to use of *singleton types*, types that only hold a single value. Thus, the typechecker can simply look at the signature of a patched module to determine the values of the patched fields.

4.3.5 Lazy Evaluation

The object language also uses *lazy evaluation*. This is the reason why recursive types do not cause the typechecker to loop - the recursive occurrences are only evaluated when demanded by unification. This works for many recursive types, but it is worth noting that some instances still cause loops.

```
  def Foo: Type = Bar
  def Bar: Type = Foo

  def foo: Foo = bar
```

```
def bar: Bar = foo
```

Typechecking `foo` and `bar` causes nontermination. Everything else about lazy evaluation is standard, so we will not discuss it further.

5 Shortcomings & Possible Extensions

Peridot's design has shortcomings that must be addressed. These concerns regard both expressivity and performance.

5.1 User-Defined Computation Rules

As discussed in Section 4.2.1, use of computation rules during unification is very useful for program transformations, as they allow for equivalences between programs to be exploited. However, there are many useful equivalences that cannot be realized with the built-in rules alone. For instance, map fusion transforms `map f ∘ map g` into `map (f ∘ g)`, which is much more efficient, but this rule does not hold automatically. Thus, it would be beneficial to allow extending the object language with new computation rules. The best way to implement this is left for future work.

5.2 Space Consumption

Nondeterministic transformations require space exponential in the amount of rules they have. This is an obvious problem, as it means that using nondeterminism to avoid the phase-ordering problem is impractical for large programs. It is possible that equality saturation could be used internally to mitigate this.

5.3 Verification

There is no way to formally verify the correctness of program transformations in Peridot. Of course, most compilers are not verified, and the meta level's declarativeness makes it easier to manually verify correctness, but this is still a considerable issue. The approach of Abella could be adopted - a meta-level reasoning logic added on top of the current meta-level specification logic.

5.4 Termination

The meta level does not automatically check termination of object-level programs; thus termination analysis must be defined and used manually. For instance, the constant folding predicate in Section 3 would send the compiler into a loop without the call to `Terminating`. Although it would not be a perfect solution, termination analysis could be applied behind the scenes to determine which object-level computations are safe to execute.

6 Related Work

6.1 GHC Rewrite Rules

GHC allows programmers to specify rewrite rules which transform programs. For instance, they can be used to implement specialization as well as certain kinds of fusion. However, rewrite rules have many limitations, such as

- They can only be used to implement *local* transformations. All the examples from Section 3 cannot be implemented. Specialization can almost be implemented, but the specialized implementations need to be manually written, which is a huge limitation

- They suffer from the phase-ordering problem. Phase numbers need to be manually specified, and in the case of multiple matching rules a random rule is chosen

- They do not exploit complex program equalities. GHC uses an algorithm that matches expressions syntactically modulo α-equivalence and η-equivalence. The limitation is that is does not use β-equivalence[3], which makes many interesting equalities invisible to rewrite rules

Peridot's metaprogramming subsumes the functionality of GHC's rewrite rules and does not suffer from the above limitations.

6.2 Staged Programming

The goals of staging and Peridot align: providing both high-level programming and high-performance programming in a single language. The literature on staging is rich, including languages such as MetaOCaml, MetaML, and Scala LMS. Staging can be thought of as user-directed partial evaluation - programmers insert "staging annotations" which direct specialization.

However, Peridot's approach is distinct from staging. Annotations are not used to direct a single transformation. Rather, metaprogramming is used to *implement* transformations. Peridot's approach allows performance concerns to be separated from regular programming - users do not have to think about where to place performance-improving annotations while writing programs, performance concerns are alleviated after the fact by writing metaprograms.

Of course, it is possible to infer uses of staging constructs, which lowers the mental overhead of using them. Scala LMS does this - quotes and splices need not be written by users. Still, Peridot's approach is much more flexible, allowing all sorts of transformations to be specified rather than using partial evaluation for everything.

7 Conclusion

We have presented Peridot, a language designed to reconcile high-performance and high-level programming. The language uses logic programming in a novel way to facilitate user-defined program transformations. We view Peridot as a starting point for investigation into languages with user-extensible compilation. There are several directions in which the designs of such languages could proceed, and Peridot is only one point in that space. Some of these directions are:

- *Infinite levels*. Peridot uses a system with two dissimilar levels, which allows the meta-language and object language to be designed separately. An alternative system could use infinite *identical* levels. Although this system would lose the above property, it would allow for meta-meta-level compilers: A compiler for object-level programs could be written. This compiler could then compile *itself*, since the meta language and object language would be identical. In contrast, compilation for metaprograms in Peridot would need to be built-in.

- *Alternative object languages*. Peridot's object language is dependently typed and functional. However, this is not an inherent requirement of the two-level design. The choice of object level affects what object-level equalities the system can automatically use. For instance, β equality and δ equality would not be trivially valid with an impure object language.

We encourage those enticed by Peridot to explore the design space.

3. Technically, GHC has β-equivalence in the form of *inlining*. However, relying on inlining to aid rewrite rules can be finicky. Allowing matching to directly use β-equivalence yields more reliable results.

8 References

1. Annenkov, Danil, Paolo, Capriotti, Nicolai, Kraus, and Christian, Sattler. “Two-Level Type Theory and Applications.” (2017).

2. Kovács, Andras. “Staged Compilation with Two-Level Type Theory".*Proc. ACM Program. Lang.* 6, no.ICFP (2022).

3. Taha, Walid, and Tim, Sheard. “Multi-Stage Programming with Explicit Annotations.” . In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (pp. 203–217). Association for Computing Machinery, 1997.

4. Taha, Walid, and Tim, Sheard. “Multi-Stage Programming with Explicit Annotations".*SIGPLAN Not.* 32, no.12 (1997): 203–217.

5. Kiselyov, Oleg, and Jeremy, Yallop. “let (rec) insertion without Effects, Lights or Magic.” (2022).

6. Yallop, Jeremy, and Oleg, Kiselyov. “Generating Mutually Recursive Definitions.” . In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (pp. 75–81). Association for Computing Machinery, 2019.

7. Jang, Junyoung, Samuel, Gélineau, Stefan, Monnier, and Brigitte, Pientka. “Moebius: Metaprogramming using Contextual Types – The stage where System F can pattern match on itself (Long Version).” (2021).

8. Nanevski, Aleksandar, Frank, Pfenning, and Brigitte, Pientka. “Contextual Modal Type Theory".*ACM Trans. Comput. Logic* 9, no.3 (2008).

9. Wang, Yuting, and Gopalan, Nadathur. “A Higher-Order Abstract Syntax Approach to Verified Transformations on Functional Programs.” (2015).

10. Amy Felty. “A Tutorial on Lambda Prolog and its Applications to Theorem Proving.” (1997).

11. Tiark Rompf. ‘‘Lightweight Modular Staging and Embedded Compilers: Abstraction without Regret for High-Level High-Performance Programming.” (2012).

12. Oleg Kiselyov. “The Design and Implementation of BER MetaOCaml - System Description.” . In *FLOPS*.2014.

13. Peyton Jones, Simon, and Simon, Marlow. “Secrets of the Glasgow Haskell Compiler Inliner".*J. Funct. Program.* 12, no.5 (2002): 393–434.

14. Olivier Savary-Belanger, Mathieu Boespflug, Stefan Monnier, and Brigitte Pientka. “Programming type safe transformations in Beluga.” (2013).

15. The GHC Team. “7.14. Rewrite rules”. https://downloads.haskell.org/~ghc/7.0.1/docs/html/users_guide/rewrite-rules.html