Dependently Typed Haskell in Industry (Experience Report)

DAVID THRANE CHRISTIANSEN, Galois, Inc., USA
IAVOR S. DIATCHKI, Galois, Inc., USA
ROBERT DOCKINS, Galois, Inc., USA
JOE HENDRIX, Galois, Inc., USA
TRISTAN RAVITCH, Galois, Inc., USA

Recent versions of the Haskell compiler GHC have a number of advanced features that allow many idioms from dependently typed programming to be encoded. We describe our experiences using this "dependently typed Haskell" to construct a performance-critical library that is a key component in a number of verification tools. We have discovered that it can be done, and it brings significant value, but also at a high cost. In this experience report, we describe the ways in which programming at the edge of what is expressible in Haskell's type system has brought us value, the difficulties that it has imposed, and some of the ways we coped with the difficulties.

CCS Concepts: • **Software and its engineering → Functional languages**; **Software development techniques**; *Data types and structures.*

Additional Key Words and Phrases: Haskell, dependent types, performance

ACM Reference Format:
David Thrane Christiansen, Iavor S. Diatchki, Robert Dockins, Joe Hendrix, and Tristan Ravitch. 2019. Dependently Typed Haskell in Industry (Experience Report). *Proc. ACM Program. Lang.* 3, ICFP, Article 100 (August 2019), 16 pages. https://doi.org/10.1145/3341704

1 INTRODUCTION

Galois, Inc. specializes in solving hard problems in computer science, with an emphasis on security, often using programming languages and verification. Symbolic simulation is a key technique that we use to reason about low-level, imperative programs. Galois has developed a general framework for writing symbolic simulators called Crucible. The implementation of Crucible makes heavy use of what is often referred to as "dependently typed Haskell," with many techniques that are reminiscent of those used in languages such as Coq, Agda, Idris, or Lean. These techniques statically ensure that a number of well-formedness properties are never violated, e.g. that all expressions are well-typed, that register references in control-flow graphs (CFGs) are well-scoped, and that evaluation order is explicit.

Based on our experiences, developing a large, performance-critical code base with these advanced Haskell features can be done in industry, and it brings significant value. However, this value comes with a high cost: this style of programming can require additional run-time checks to ensure type safety, and there is an especially high barrier to entry for new developers. We describe our experiences and detail the compromises that have been necessary to achieve acceptable performance without giving up key correctness checks.

Section 2 describes the context and background for Crucible's development. Section 3 provides an overview of how we have used some of the features of GHC Haskell to arrange for static

2475-1421/2019/8-ART100
https://doi.org/10.1145/3341704

checking of many key invariants. In order to successfully implement Crucible, Galois needed to develop a "standard library" for our style of mixed type- and value-level programming, called `parameterized-utils`, which is described in Section 4. We were not always able to achieve sufficient performance by using only the safe features of Haskell, and Section 5 describes our use of unsafe features. In Section 6, we reflect on the practicalities of using advanced type system features. In Section 7, we describe our reasoning for not pursuing particular related technologies, and in Section 8 we speculate about future developments in functional language design and implementation that would be beneficial to use cases like ours.

While Crucible is implemented in Haskell, and we have used Haskell's type-level programming features to implement it, we expect that many of the same challenges that we have experienced would arise when using other languages with comparable features. In particular, we expect that our experiences will provide useful input to designers and implementors of dependently typed languages that aim to be usable for large systems in which good performance is a requirement.

2 CONTEXT

One important technique that Galois uses for mathematically reasoning about programs written in low-level, imperative languages is symbolic simulation, where imperative programs whose termination conditions are clear can be converted into functional models [Dockins et al. 2016]. These extracted models can be reasoned about directly, compared against hand-written specifications for correctness, or be further compiled, such as in the RAMPARTS project[1] in which Julia programs are compiled into fully-homomorphic-encryption circuits.

Writing a symbolic simulator is a demanding task. Because we rely on the output of the simulation for security arguments, it must be *correct*. Additionally, because the simulator must explore every case when the program branches on a symbolic value, the simulation must also be *performant*.

Crucible is Galois's third symbolic simulator, following the JVM Symbolic Simulator (JSS) and the LLVM Symbolic Simulator (LSS). While JSS simulated JVM bytecode directly, LSS first translated LLVM to an intermediate representation more amenable to simulation. Crucible began as a simulator for MATLAB®, as part of a project called Grackle.[2] We realized that we would likely write more symbolic simulators and decided to implement a general-purpose framework for writing simulators in the intermediate-representation style. Crucible has been continually updated with more and more of the features from Grackle. Today, Crucible is used to implement robust simulators for Matlab, LLVM IR, x86_64 and PowerPC machine code, and JVM bytecode, as well as experimental simulators for Julia, Rust, and Go.

The run-time values of Crucible programs are formulas written in an abstraction layer called What4, named in honor of Bobot et al.'s Why3 [2011]. What4's features are a superset of those provided by a number of solvers, so a particular program may or may not work with a particular solver, depending on which features it uses. In addition to acting as a *lingua franca* between different solvers, What4 is also designed to be a convenient value representation for Crucible by simplifying symbolic formulas into concrete formulas where possible, which can decrease the search space.

Crucible was developed after two related tools: Cryptol [Lewis and Martin 2003] (a language for specifying cryptographic algorithms with support for statically-tracked bitvector widths) and SAWCore [Dockins et al. 2016] (a verification tool based on dependent type theory). These prior experiences gave us familiarity with dependently-typed intermediate representations, which informed the design of the Crucible IR. From the beginning, Crucible made heavy use of GADTs [Augustsson and Petersson 1994; Peyton Jones et al. 2006], type families [Chakravarty et al. 2005; Schrijvers

[1] https://galois.com/project/ramparts/
[2] https://grackle.galois.com/

et al. 2008], datakinds [Yorgey et al. 2012], and kind-indexed GADTs [Weirich et al. 2013]. Early in Crucible's development in 2014, we realized that large parts of the code could be used in a number of projects, so we extracted a library of commonly-used typeclasses and datatypes for working at higher kinds, called `parameterized-utils`, which has been developed together with Crucible.

The `parameterized-utils` library is more than just a collection of tools for relating types to their run-time witnesses. It also contains generalized versions of commonly-used Haskell typeclasses, such as `Eq`, `Show`, and `Traversable` [Gibbons and Oliveira 2009; McBride and Paterson 2008], that have been generalized to types of higher kind, as well as Template Haskell [Sheard and Peyton Jones 2002] metaprograms to make up for the lack of instance derivation for our new classes.

3 TECHNIQUES USED IN CRUCIBLE

While developing Crucible, our persistent aim has been to use Haskell's type system to enforce static invariants of our embedded languages. This includes type-correctness of both the run-time values and the static program representation of Crucible, as well as variable scoping and SSA invariants in our representation of control-flow graphs. While we were frequently able to base our work on existing techniques, they sometimes required extensions or adaptations to our needs.

3.1 Run-Time Representatives

Eisenberg and Weirich [2012] describe a technique for writing dependently typed programs in Haskell in which types are assigned run-time representatives (called *singletons*, because each has but a single non-$\bot$ inhabitant) that contain witnesses of type equality from which the represented type can be recovered. `parameterized-utils` doesn't use Eisenberg and Weirich's library, nor the exact same technique, because some of our needs require type witnesses that contain structure beyond that imposed by singletons. For instance, some parts of Crucible require that bitvectors have a size greater than one, but we are not able to effectively lift this constraint to the type level. By requiring a representative for the size that enforces the invariant, however, we can maintain the static safety that we desire. Additionally, manually constructing value representatives for types allows their names to be carefully chosen and their documentation to be written and made public, while code generation often leads to names and documentation that are not what a human would write. We find that making the singleton type definitions explicit makes them easier for developers to understand and engage with than auto-generated definitions would be. We judge this advantage to be worth the additional boilerplate code that must be written and maintained manually.

We have not yet explored the techniques described by Lindley and McBride [2013] for using the typeclass constraint solver to dispatch proof obligations. This is potentially an interesting area for future work.

3.2 Typed Abstract Syntax

The data constructors in Figure 1 are a selection of Crucible's expression formers, slightly simplified for presentation. The `App` datatype represents *app*lications of syntactic formers of the expression language. Note that the datatype is written in an "open recursion" style, where subterm occurrences are abstracted by the higher-kinded `f` parameter. We heavily use this style because it allows the expression datatype's pattern functor to be re-used in other contexts. For instance, `App` can be used in a traditional syntax tree by taking a fixed point, in an explicit-sharing DAG structure by applying it to the DAG nodes, and in A-normal form [Sabry and Felleisen 1993] by applying it to variable references.

The expression formers in Figure 1 exhibit a number of features of Crucible's type system. The most obvious is that subterms and the type of the resulting term are expressly given by the `CrucibleType` arguments to `f` and `App`. The type system includes (among others) types for booleans,

```
import GHC.TypeLits (Nat)

data Ctx k = EmptyCtx | Ctx k ::> k

data CrucibleType =
  BoolType | IntegerType | BVType Nat |
  FunctionHandleType (Ctx CrucibleType) CrucibleType
  ...

data NatRepr (n :: Nat) where ...

data App (f :: CrucibleType -> Type) (tp :: CrucibleType) :: Type where
  BoolLit :: Bool -> App f BoolType
  And :: f BoolType -> f BoolType -> App f BoolType
  IntLe :: f IntegerType -> f IntegerType -> App f BoolType
  BVLit :: (1 <= w) => NatRepr w -> Integer -> App f (BVType w)
  BVAdd :: (1 <= w) => NatRepr w -> f (BVType w) -> f (BVType w) -> App f (BVType w)
  ...
```

Fig. 1. Crucible expression datatypes

integers, real numbers, bitvectors of arbitrary non-zero width, and functions. The `Nat` parameter of the `BVType` constructor is a type-level natural number index that gives the number of bits in the classified bitvectors. The `NatRepr w` arguments of some of bitvector operations give a run-time representation of this type-level data; it is a singleton type for `Nat`, to use the terminology from Eisenberg and Weirich [2012].

The `Ctx` datakind is part of `parameterized-utils`. `Ctx k` represents a list of types of kind `k` that grows on the right. Here, `Ctx` is used here to represent the argument list for function types.

This strong typing discipline ensures that we cannot e.g. accidentally construct expressions for adding together integers and bitvectors, or even adding bitvectors of different widths. This type discipline is applied to the run-time representations as well; this ensures that the evaluator is able to rely on the input arguments being of the expected run-time type. As a result, the expression evaluation code has very few run-time error conditions that can occur; those that can are related to properties that are not tracked statically, such as index-in-bounds conditions for vectors of values.

Another example of rich type system features occurs in the static single assignment (SSA) form of control-flow graphs, which is Crucible's main program representation. A CFG consists of a collection of basic blocks, and each basic block consists of a sequence of straight-line statements followed by a block terminator. Each non-terminal statement may exist purely for its side effects, or it may assign a value to a fresh temporary variable. We maintain the SSA invariant by tracking, at each program point, a context of all temporary variables in scope so far in the given block. Statements may only extend this context by adding new temporaries, and the expressions appearing in statements may only rely on temporaries already in scope. Basic blocks have annotations that describe which temporaries are expected to be in scope at their start, so that control-flow transfer terminator statements are guaranteed to provide the expected number and types of values.

Figure 2 demonstrates a sample of the Crucible statement language, simplified for presentation. `Ctx CrucibleType` is used to represent the shape of the current local scope. The data type `Assignment f ctx` represents a collection of values whose types are computed using the type operator `f` for each element in the `ctx`. The type `Assignment f (EmptyCtx ::> a ::> b ::> c)`, of three-element assignments, is isomorphic to the tuple type `(f a, f b, f c)`.

```
data Assignment (f :: k -> Type) (ctx :: Ctx k) where ...
data Index (ctx :: Ctx k) (a :: k) where ...

lookup :: Assignment f ctx -> Index ctx tp -> f tp

data TypeRepr (tp :: CrucibleType) where
  BoolRepr    :: TypeRepr BoolType
  IntegerRepr :: TypeRepr Integer
  BVRepr      :: (1 <= w) => NatRepr n -> TypeRepr (BVType n)
  ...

newtype Expr (ctx :: Ctx CrucibleType) (tp :: CrucibleType)
   = App { expr :: App (Index ctx) tp }

data Stmt (ctx :: Ctx CrucibleType) (ctx' :: Ctx CrucibleType) where
  SetReg :: TypeRepr tp -> Expr ctx tp -> Stmt ext ctx (ctx ::> tp)
  CallHandle ::
    TypeRepr ret -> Assignment TypeRepr args ->
    Index ctx (FunctionHandleType args ret) ->
    Assignment (Index ctx) args -> Stmt ctx (ctx ::> ret)
  Assert :: Index ctx BoolType -> Index ctx StringType -> Stmt ctx ctx
  ...
```

Fig. 2. Some Crucible statements

The type `Index ctx tp` represents an index into `ctx` that points at the type `tp`; here, we use these to represent temporary variables of a basic block. An `Index ctx tp` can be used to project a value of type `f tp` from an `Assignment f ctx`. By construction, values of `Index ctx tp` are valid indices into the context, so context lookup is a typed, total operation.

The `Expr` datatype in Figure 2 is constructed from the `App` type seen above, and enforces an A-normal form invariant by instantiating the `f` parameter as `Index ctx`, so subterms may only be variable references. The `Stmt` datatype takes two context arguments: the context prior to executing the statement and the context after. Statements that introduce a new temporary (like `SetReg`) extend the context with new types, and ones that do not (like `Assert`) pass it through unchanged.

4 SOME PARAMETERIZED UTILITIES

When working with the sorts of Haskell features discussed in this paper, one runs almost immediately into a number of practical issues. One of these is that the usual Haskell support for automatic instance deriving fails on GADT datatypes for `Eq`, `Ord`, `Show` and similar instances. Another difficulty is that a variety of standard programming idioms, such as `Functor` and `Applicative` oriented programming, simply fail to apply and need to be generalized. In addition, wholly new idioms are required for working with run-time representatives of datakinds.

The extensive use of values indexed by datakinds imparts a need for a significant amount of support infrastructure, which is provided by `parameterized-utils`. This infrastructure includes:

- comparison classes that accommodate datakind-indexed values (`OrdF`, which complements `Data.Type.Equality.TestEquality`),
- canonical versions of common utilies, such as existential packages,
- type-level lists and their run-time counterparts (the `Context` and `Assignment` mentioned in Section 3.2),

```
data a :~: b where
  Refl :: a :~: a

class TestEquality (f :: k -> Type) where
  testEquality :: f a -> f b -> Maybe (a :~: b)

data OrderingF x y where
  LTF :: OrderingF x y
  EQF :: OrderingF x x
  GTF :: OrderingF x y

class TestEquality f => OrdF (f :: k -> Type) where
  comparef :: f a -> f b -> OrderingF a b

MapF :: (tp -> Type) -> (tp -> Type) -> Type
insert :: OrdF k => k tp -> v tp -> MapF k v -> MapF k v
lookup :: OrdF k => k tp -> MapF k v -> Maybe (v tp)
```

Fig. 3. Example type signatures from `parameterized-utils`

- a type-indexed map variant, called `MapF`, where keys and values share the same type index (see Figure 3), and
- variants of `Functor`, `Foldable`, and `Traversable` lifted to higher kinds.

The `TestEquality`[3] and `OrdF` classes in Figure 3 are generalizations of `Eq` and `Ord`, respectively. However, they additionally provide type-equality evidence in the case that the two items being compared are actually equal. This is essential when using datakind representatives, as it provides an important connection between the value level representatives and the type level datakinds.

Like the un-indexed `Ord`, the `OrdF` class has a means of comparing two values. Unlike `Ord`, `OrdF` does not require that the values being compared have the same index *a priori*. Additionally, should the two values are in fact be equal, `compareF` will produce evidence that their indices are also equal.

To insert a key and value into a `MapF`, their indices must coincide. Similarly, looking up a key in a `MapF` is guaranteed to produce a value whose index matches that of the key.

`FunctorF`, `FoldableF`, and `TraversableF` (see Figure 4), which are generalizations of their `-F`-less counterparts, demonstrate that an indexed type is respectively functorial, foldable, or traversable at each possible index. Some of the typeclasses in `parameterized-utils` are higher-kinded generalizations of classes in the Haskell prelude that only assert a uniform operation at all indices, such as `EqF`, `ShowF`, and `HashableF`; the recent introduction of quantified constraints to Haskell [Bottu et al. 2017] could obviate the need for these classes. Other typeclasses, such as `TestEquality` and `OrdF`, allow comparisons of terms with different argument types and produce evidence of type equality, so they will still be necessary.

The flexibility afforded by quantified constraints is not sufficient to capture the generalizations of `Functor`, `Foldable`, and `Traversable`. This is because both the `-F`-suffixed and the `-FC`-suffixed class methods take rank-2 polymorphic functions that operate universally over the type indices of some contained higher-kinded type. The `-F` versions are for types that do not themselves have an index, whereas the `-FC` versions are for those that do.

The `TraversableFC` class in particular is useful with `App`, the open-recursive AST representation described above. It precisely abstracts over the process of applying an operation uniformly to the

[3] `TestEquality` and the type-equality proof type `:~:` are part of the base library module `Data.Type.Equality`

```
class FunctorF (t :: (k -> Type) -> Type) where
  fmapF :: (forall x. f x -> g x) -> t f -> t g
class FoldableF (t :: (k -> Type) -> Type) where
  foldrF :: (forall x. e x -> b -> b) -> b -> t e -> b
class (FunctorF t, FoldableF t) => TraversableF t where
  traverseF :: Applicative m => (forall x. e x -> m (f x)) -> t e -> m (t f)

class FunctorFC (t :: (k -> Type) -> l -> Type) where
  fmapFC :: (forall x. f x -> g x) -> (forall x. t f x -> t g x)
class FoldableFC (t :: (k -> Type) -> l -> Type) where
  foldrFC :: (forall x. f x -> b -> b) -> (forall x. b -> t f x -> b)
class (FunctorFC t, FoldableFC t) =>
    TraversableFC (t :: (k -> Type) -> l -> Type) where
  traverseFC :: Applicative m =>
                (forall x. f x -> m (g x)) -> (forall x. t f x -> m (t g x))
```

Fig. 4. `parameterized-utils` classes

subterms of syntax formers. For instance, using `traverseFC` to pass through the syntax formers allows the conversion of syntax expression trees into A-normal form to be described irrespective of the actual encoding of ASTs. It is also used to traverse all the elements of an `Assignment`, e.g. to evaluate the actual arguments of a function call. Similarly, `foldrFC` is handy for computing the set of variables that occur in an expression as part of def/use analysis.

However, actually writing instances for these classes is quite tedious and error-prone for the datatypes in question. For example, `App` has over 150 data constructors! To ease this burden, `parameterized-utils` provides Template Haskell code that generates operations for the `TestEquality`, `OrdF`, `TraverseFC`, `ShowF`, and `HashableF` classes. Much like GHC's standard "deriving" mechanism, these templates examine the declaration of GADTs and do the expected thing in the straightforward cases. However, some cases require additional guidance, which is provided via a collection of type patterns and splices. When the template code encounters a datatype argument whose type matches one of the given patterns, it pastes in the given splice instead of performing its default behavior, which is to fall back on non-parameterized instances for `Eq`, `Ord`, etc. This allows developers exert as much control as is necessary, and no more, over code generation.

5 UNSAFE FEATURES

In some cases, we have found that our strict typing discipline creates unnecessary work at run time because data structures that straightforwardly witness the necessary type equalities are inefficient. In other cases, GHC's type system is not sufficiently expressive to allow us to prove e.g. that `x + x = 2 * x`, so we must instead test run-time representatives.

In some instances, these overheads can be quite significant: we recently discovered they were that 48% of overall run time on a particular benchmark. To work around these inefficiencies, we sometimes step outside the bounds of the type system and assert to the typechecker that we know better than it does by using `unsafeCoerce`. This warrants extreme care, and we only consider doing so when there are significant performance issues to be solved. We explain a few examples below.

5.1 Assignments

Crucible relies heavily on the `Assignment` data structure to represent function argument lists, local variable environments, collection of basic blocks in a CFG, etc. Accessing elements in an `Assignment`

is guaranteed to be type safe and total. `parameterized-utils` provides two implementations of `Assignment`: a safe variant using GADTs that essentially implements a snoc list with unary-encoded natural number indices, and a more efficient version that uses balanced trees and machine integers, but requires `unsafeCoerce`. The experience of profiling Crucible showed that linear access to an `Assignment` imposed an unacceptable overhead on the simulator.

We maintain both implementations so that we can isolate bugs that arise from unsafe code by testing against the safe, slow implementations. This requires that the safe implementation can be a drop-in replacement for the unsafe implementation; however, the language and tools do not provide support for automatically checking that the two versions of the API remain synchronized.

To ensure that the two APIs remain compatible, we have employed a manual process where the selected API is controlled by a flag in the build system, which is occasionally toggled for testing purposes. Beyond simple API drift, substantive differences have arisen. For example, GHC inferred different type roles [Weirich et al. 2011] for type parameters in the two versions, which allowed `Data.Coerce` [Breitner et al. 2014] to work with one but not the other; this was fixed by adding explicit role annotations after the discrepancy was discovered. Ideally, an ML-style module system could allow us to inform the compiler that we expect both variants to implement the same API. We have not yet explored using Backpack [Yang 2017] to ascribe a common signature to the two implementations due to lack of support in build tools.

5.2 Extending Indexed Maps

Because the type-indexed SSA presentation of control-flow graphs can be difficult to construct programmatically, Crucible includes a separate registerized presentation with fewer type constraints, and the library contains a facility for constructing a CFG in SSA form from one written in registerized form. This conversion is another example of a use of unsafe Haskell features that we have found to be necessary in order to achieve sufficient performance. As with the `Assignment` type mentioned above, the relevant operations have both safe and unsafe implementations.

As discussed in Section 3, the type of each statement constructor includes context parameters that indicate the local variable scope at that program point. The types of value references are arranged such that the type checker rejects references to values that are not in scope. SSA conversion maintains a map from expressions to the SSA registers to which they are bound. Note that this map includes a reference to the `ctx` parameter. Extending this map with additional entries during SSA construction changes the type of the map when a new temporary is bound, as seen in Figure 5. The safe implementation of this map extension operation reconstructs the map at each step of the SSA construction, which is immensely expensive for an operation that does not actually change the value of the map at all. In practice, we use the unsafe version of the operation.

5.3 Nonces

Another pervasive technique is to avoid expensive equality tests between run-time type representatives by instead comparing integer proxy values we call *nonces* and then asserting (with `unsafeCoerce`) that when two nonces are equal, their corresponding type indices are equal. They can be seen as a reification of pointer equality tests, or equivalently as a form of Lisp's `gensym`.

The type of nonces is `Nonce (s :: Type) (tp :: k)`, where `s` is a state thread parameter analogous to the parameter in `ST` [Launchbury and Peyton Jones 1994] and `tp` is a datakind index. The intended use is to associate nonces with typed data values so that comparisons of nonces recover type equalities. Nonces are implemented as 64-bit unsigned integers for efficiency, where the constructor for `Nonce` is hidden and generated based on a single source. In this way, as long as the counter is managed properly (which is enforced by implementation hiding), and as long as the counter does not overflow, comparison of integers suffices to preserve type equality.

```
#ifdef UNSAFE_OPS
type AppRegMap ctx = MapF (C.App (C.Reg ctx)) (C.Reg ctx)

appRegMap_extend :: AppRegMap ctx -> AppRegMap (ctx ::> tp)
appRegMap_extend = unsafeCoerce
#else
type AppRegMap ctx = Map (Some (C.App (C.Reg ctx))) (SomeReg ctx)

appRegMap_extend :: AppRegMap ctx -> AppRegMap (ctx ::> tp)
appRegMap_extend = Map.fromList . fmap f . Map.toList
  where f (Some app, SomeReg tp reg) =
        (Some (C.mapApp C.extendReg app), SomeReg tp (C.extendReg reg))
#endif
```

Fig. 5. Unsafe operations

Because the counter is only incremented, a program would need to run for at least one hundred years before overflow is a worry [Clochard et al. 2016]. The improvements from using nonces are often noticeable and substantial: replacing structural type equality tests with nonce comparisons reduced the running time of one large Crucible workload from 175 seconds to 91 seconds, which is approximately a 48% speedup.

6 COSTS AND BENEFITS OF VERY EXPRESSIVE TYPES

The Crucible source repository, which includes the LLVM and JVM simulators as well as What4, but not `parameterized-utils` or the simulators for machine code, contains more that 80,000 lines of Haskell. This code has been continuously maintained and developed for more than four years, serving a variety of research projects. There have been around thirty committers. Given this context, was using dependently typed Haskell an appropriate technical choice? Short of doubling our staff and implementing it a second time, it is impossible to precisely measure the change in productivity, prevalence of bugs, or altered user experience compared to the counterfactual situation where fancy type system features were instead avoided. Nonetheless, we believe that these techniques have been a net benefit in our particular context. In this section, we describe our concrete positive and negative experiences so that others can make their own assessments.

6.1 Correctness

Our own anecdotal evidence suggests that our use of advanced Haskell type system features has been very helpful for avoiding shallow bugs, for easing refactoring, for ensuring that sufficient care is taken with fiddly details, and for reducing time spent on code review by experts. Additionally, forcing developers to think in an upfront manner about the types of the expressions, values, and programs they are manipulating has led to more general designs. Forcing bitvector operations to track their widths has been extremely helpful, for example, to ensure that appropriate size extension and truncation operations are applied in the implementations of C library function primitives, a fiddly detail that is otherwise easily overlooked.

The type discipline was also extremely helpful when implementing SSA conversion because it forced us to insert sufficient type-checking operations on the Crucible embedded language during the conversion process to convince GHC itself that the data structures we were building were type-correct. As a result, we essentially never have type related issues with the resulting CFG representations, even though the SSA conversion process itself is rather sophisticated. This has, on

multiple occasions, revealed bugs in earlier program translation phases that would otherwise have manifested at simulation time, where they would likely have been much more difficult to diagnose.

We have managed to encode many useful well-formedness properties in the Haskell type system. For instance, we encode that some operations require equal bitvector widths and that no bitvector contains zero bits. The type system ensures well-formedness of control flow graphs, including that function calls are passed the right number and types of arguments. In SSA form, the type system ensures that all CFGs are well-scoped with respect to the local variables available after each statement, that we don't confuse one variable with another, and that the choice of bound variable names truly does not matter. When we specify the semantics of machine code, the type system helps us in maintaining the plethora of complicated well-formedness properties, including the widths of each field.

6.2 Refactoring

Aside from helping us write correct code, our extensive static well-formedness checks are helpful during maintenance and refactoring, especially large-scale refactorings. Because Crucible is such a large project that is used and updated by so many other projects at Galois, it is not practical for one person to have the whole system in their head at once.

While updating Crucible, we arrange for the compiler to emit type errors in places that have not yet been updated. This is a very effective means of ensuring that refactorings occur in a coordinated manner, and our use of a well-typed and well-scoped representation of syntax allows us to glean this benefit also for the embedded Crucible language itself.

6.3 Difficulties

These benefits come with a cost. It has been difficult to staff projects relying on Crucible, because most Haskell developers are not intimately familiar with dependent types or their encoding in Haskell. There are few introductory resources that bridge the gap between idiomatic Haskell and the style used in Crucible. The interactive tooling for Haskell was not designed for this style, and pushing the limits of the tools can uncover unique bugs. Finally, any type system rules out good programs too, and features like polymorphic function types are proving to be difficult to encode.

6.3.1 Training. It is not always easy to recruit developers who are skilled in Haskell, let alone dependently typed programming. Dependently typed programming in Haskell often seems to carry a higher cognitive overhead than in languages originally designed for dependent types, and some developers describe writing Idris or Lean code in their heads, and hand-compiling it to Haskell. Due to our close ties with the academic research community, Galois is at an advantage: we are often able to recruit engineers who already have experience in both dependently typed programming and in advanced Haskell features. As a result, less training is necessary than might be the case elsewhere. However, not everyone knows all these things. Engineering staff who are interested in contributing to Crucible and `parameterized-utils` typically ask for the necessary help from colleagues in the process of negotiating their involvement in the project. We have occasional short formal training sessions, but not often, and it's not nearly sufficient to make someone productive. We typically rely on one-on-one mentoring to bridge the gap.

6.3.2 Tool Support. Haskell's developer tooling is a product of its time. The state of the art is a batch mode compiler (`ghc`) and a REPL (`ghci`), rather than an incremental system designed to support type-aware interactive editing, and tools such as Haddock and profilers either lag behind or fail to function correctly with new language features.

The tooling for Haskell often lags behind the changes and additions to the GHC implementation of the language. For instance, Haddock does not always render kind-polymorphic type constructors as one might expect. Until a recent update, the equality type from Figure 3 was rendered:

```
data (k :~: (a :: k)) (b :: k) :: forall k. k -> k -> * where
```

Interactive environments for languages like Idris and Agda are able to use these languages' rich type systems to assist in writing programs. For instance, they can generate a covering case split, taking type indices into account, rather than requiring a programmer to manually enter all cases, only emitting warnings or errors after the fact. For highly-indexed datatypes like those we use in Crucible, this would be very convenient. Likewise, most Haskell environments provide no mechanism for looking up the documentation for a name in its scope, relying on string searching.

Upgrading GHC versions has often been fraught, as code constructs that work in one version of GHC sometimes require extra language extensions to work in later versions. Sometimes, the type checker is able to identify the new extension, but not always. Between GHC 8.0 and 8.6, this has manifested as more modules requiring `-XTypeInType` as other type checker bugs are fixed, and more code comes under the purview of that extension. We attempt to maintain compatibility with the most recent few releases of GHC at all times. When a new version is released, we build Crucible in it as soon as the dependencies have been updated. This practice helps us report bugs in a timely manner, it brings us the very real benefits of new GHC releases, and it keeps the work required to update from becoming overwhelming.

We encountered bugs in GHC at a higher rate than other projects. Furthermore, many of these bugs occur nondeterministically, and are only triggered on very large programs, so they are difficult to reduce for a bug report. For instance, GHC has generated `.hi` interface files that it could not parse again, which we have typically been able to solve by adding explicit kind signatures. As the `TypeInType` extension has been implemented, we have seen fewer bugs related to the implicit lifting of kind parameters. Perhaps the most serious bug we encountered was that we could not compile with profiling support in GHC 8.4 [GHC Issue #15186 2018]. Maintaining compatibility with a range of versions allows us to work around compiler bugs by using multiple versions, and it makes it easier to distribute programs that depend on Crucible.

Crucible's use of a large number of type families causes very long compile times. This is especially noticeable in encodings of machine code semantics, where there are both a large number of terms in the language (individual machine code instructions) and a large number of type families involved in specifying their semantics. Partial languages need to evaluate proofs strictly, so techniques like proof by reflection (see e.g. Bertot and Castéran [2004]) aren't applicable to speed up type checking.

6.3.3 Type System Limitations. Working at the limit of the Haskell type system, we sometimes encounter programs that are either very difficult or impossible to write. In particular, we have not yet succeeded in adding polymorphic functions to Crucible, which would be useful for simulating higher-level languages. Similarly, while `Ctx` and `Assignment` are useful for modeling non-dependent contexts, we have not yet found a suitable means of modeling telescopes [de Bruijn 1991].

There was a large fixed cost in the early development of Crucible, as we discovered how to use these advanced Haskell features. However, we expect that future projects will be able to benefit from the idioms and libraries in `parameterized-utils`, so we see this cost as an investment.

Even though using advanced type system features has a reputation for inducing fearsomely useless error messages, we have generally found GHC's error messages to be quite useful input into the process matching our programs to their specifications. The most difficult to interpret are those about missing arithmetic constraints, as the error message consists largely of the collection of facts in scope, which can be daunting in size.

```
addLemma :: (1 <= x, x + 1 <= y) => NatRepr x -> q y -> LeqProof 1 y
addLemma x y =
  leqProof one x `leqTrans` leqAdd (leqRefl x) one `leqTrans` leqProof (addNat x one) y
  where
    one :: NatRepr 1
    one = knownNat
```

Fig. 6. A low-level proof

6.3.4 Proofs. Working with length-indexed types in Haskell typically imposes a manual proof burden on developers, as GHC does not include a built-in decision procedure for type-level natural numbers. In the case of bitvector truncation, the expression constructor includes the constraint `(1 <= r, r + 1 <= w)`, stating that the truncation point `r` is between 1 and the width of the vector being truncated.

When constructing `BVTrunc` terms, programmers must manually arrange for evidence that the constraint is satisfied. These proofs are low-level, as there is no built-in support for generating them; see Figure 6 for an example. Our current codebase has over 100 locations involving manual proofs of simple properties of natural numbers. Users familiar with dependently typed programming languages find this style of programming tedious, while programmers unfamiliar with dependently typed programming often struggle to see the motivation or benefit. Moreover, the error messages produced by unmet proof obligations (e.g., the `(1 <= x, x + 1 <= y)` constraint above) are a serious barrier to entry for both types of programmer.

6.3.5 Run-Time Representatives. In order to construct the bit-vector truncation example from Section 6.3.4, we were forced to introduce run-time type representatives that allow us to connect run-time values to types, (as in Singletons [Eisenberg and Weirich 2012]).

The most significant burden imposed by manual and explicit use of run-time type representatives lies in the cognitive overhead that they impose throughout a large codebase. These run-time representatives are a significant source of confusion for developers who are learning this style of programming. Beyond the cognitive and syntactic burden of managing these representative values, creating the necessary types and instances presents an additional burden. We use Template Haskell to generate most of the necessary instances, but keeping the definitions of type representatives synchronized with the actual types of interest is an infrequent but consistent maintenance issue.

6.3.6 Typed/Untyped Interfaces. While not unsafe in the sense of `unsafeCoerce`, there is an interface between strongly-typed (with datakind indexes) code and untyped code that has been a persistent problem. The parts of Crucible discussed so far have datakind-indexed types for each term. In addition, the LLVM simulator uses a data structure called a *memory model* to track reads and writes to memory. Crucially, the values written to and read from the memory model are untyped. This interface has consistently been a source of errors, as callers must carefully select the appropriate Crucible type to use to hold values read from memory. The untyped interface arose due to the underlying machine being modeled, where reads and writes are essentially untyped (i.e., values can be written into memory at one type and read from memory at another type).

To support other interfaces with untyped data sources and to work around limitations in the static type system (e.g., polymorphism, as mentioned above), the Crucible IR has a special `Any` type that acts as a dynamic "escape hatch." Historically, most uses of the `Any` type in Crucible terms have led to coercion failures at run-time, indicating human error. A more expressive type system in Crucible (and, by extension, Haskell) may have been able to obviate the need for the `Any` type, though it is not clear that the cost in additional complexity would have paid off.

Rewriting programs represented in the Crucible IR after they have been constructed has been a persistent pain point. The need to rewrite arises from various verification tasks, including cases where assertions must be inserted after program construction (e.g., because we only know where to place assertions after performing some analysis on the program). Crucible programs are inherently difficult to modify, as some modifications require breaking the SSA invariants temporarily, but those invariants are enforced by the type system. We work around this by preserving the pre-SSA form of every program that we might need to modify; when necessary, we apply transformations to the pre-SSA form and then re-construct the SSA from that.

7 ROADS NOT TAKEN

Given that our time and resources were limited, we could not explore all possible options when writing Crucible. In this section, we describe some of the approaches that we did not explore, and our reasoning for not exploring them. We hope that the developers of some of these tools can use this information as a starting point for further development if they are seeking industrial adoption.

7.1 Type Checker Plugins

Section 6.3.4 describes difficulties that we have experienced while using type-level natural number arithmetic. These difficulties occur because Haskell's type system does not automatically apply arithmetic identities such as $n + m = m + n$ or $1 + n + 1 = 2 + n$. Similarly to Idris, Agda, Lean, or Coq, it becomes necessary to provide evidence of these identities, leading to the difficulties of writing proofs in a language not originally designed for them.

An alternative approach would be to use GHC's type checker plugin mechanism to extend Haskell's notion of type equivalence to include more natural number equalities. For instance, Iavor Diatchki developed a plugin [Diatchki 2015] that uses an SMT solver to dispatch many constraints that GHC would not otherwise solve. While type checker plugins are a great way to experiment with type system extensions without needing to rebuild the entire compiler, there are a number of pragmatic reasons not to use type checker plugins in production code. First, type checker plugins extend the trusted code base of the type checker, and bugs that undermine type soundness can be very subtle.[4] Second, plugins that rely on specific versions of SMT solvers make it even more difficult to build the system in a variety of contexts. Finally, depending on a specific type checker plugin makes the entire toolchain more difficult to use, as every tool that processes source code must be configured to use the plugin.

7.2 Full Dependent Types

Given the difficulties involved in being an early adopter of dependently typed Haskell, it would be natural to consider using a language that was designed from the start with dependently typed programming in mind. In particular, these languages usually feature interactive environments and tooling that is far superior to Haskell's, at least with respect to dependently typed programming. However, we chose not to pursue these languages for a number of reasons.

In a performance-sensitive code base like Crucible, it is absolutely essential to have a profiler to determine where time is being spent. When we embarked on this project, no dependently typed language had a profiler. While Agda has excellent support for the kind of indexed structures that we are working with, and compiles via Haskell, the increased distance between the profiler report and the source code makes it more difficult to interpret the information.

Additionally, Crucible is a part of a large suite of Haskell libraries that Galois uses in a variety of projects. Furthermore, we rely on the large number of high-quality libraries available for Haskell.

[4] For an example of the subtle bugs that can result, please see https://github.com/yav/type-nat-solver/issues/7.

Using a different language, even one that compiles via Haskell, would likely introduce a number of difficulties at the boundaries between languages. We were also afraid of becoming compiler maintainers in addition to tool implementors.

Finally, we believe that it is easier to teach a skilled Haskell programmer how to use our techniques than it is to teach them Agda, and the skills gained are more portable to other projects at Galois.

8 FUTURE PERSPECTIVES

Much of this report is concerned with describing the techniques that were necessary to achieve a program that both runs fast and uses advanced GHC type system features for safety, and we have described some of the costs that adopters should be willing to pay if they want to achieve the benefits of these techniques. In addition to those considering adopting some of these features of GHC Haskell, we believe that our experiences are also relevant to the designers of other languages with rich type systems, especially dependently typed languages, who are aiming for industrial adoption. As GHC approaches the expressive power of these languages, prior paradigms of user interaction with the system begin to break down.

While we have done our best to reduce the risks inherent in the use of unsafe language features, such as providing equivalent safe implementations for testing purposes, key parts of our libraries essentially amount to language extensions or new compiler primitives. We hope that documenting them here will help the community of researchers find opportunities to better support large-scale production use of advanced type system features, whether in Haskell or elsewhere.

In particular, certain additions to GHC and/or the Haskell language could make this style of programming more convenient. A decision procedure for linear arithmetic over type-level natural numbers could greatly reduce the time spent with manual proofs. If the `StableName` mechanism in GHC were extended with kind polymorphism and a `TestEquality` instance, we could drop our explicit nonce feature, which would simultaneously simplify a great deal of code, because there would no longer be a need to propagate the `ST` phantom type throughout the code. Extensions to the deriving mechanism could reduce our reliance on Template Haskell to generate instances And we eagerly await the upcoming `pi` quantifier to reduce the need for manual run-time type representative creation, as well as the need to juggle the trade-offs of explicitly-passed representatives vs type class dictionaries. While we do not expect that GHC will change its representations of algebraic datatypes in ways that are incompatible with our uses of `unsafeCoerce`, we do use a trick similar to Edward Kmett's `reflection` library (based on a paper by Kiselyov and Shan [2004]) to coerce between `KnownNat` dictionaries and `NatReprs`, which does depend deeply on GHC's current representations.

Liquid Haskell [Vazou et al. 2014] is not particularly well-suited to representing things like well-typed ASTs because refinement types essentially restrict the elements of a type to the subset that satisfies a predicate. The Haskell code that implements an operation indexed by `CrucibleType` types can be completely different at each individual Crucible type. However, Liquid Haskell's SMT-backed logic is very convenient when reasoning about inequalities of natural numbers. It is worth exploring whether we can replace some of our `Nat`-indexed types by Liquid Haskell refinement types.

Based on our experience with a large, multi-year, multi-developer project, we believe that the benefits of low-stress refactoring and confidence in the system's correctness have been worth the considerable costs of employing dependent Haskell in Crucible.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their suggestions and questions. The final version has been greatly improved as a result of their input.

REFERENCES

Lennart Augustsson and Kent Petersson. 1994. Silly Type Families. (September 1994). Unpublished draft.

Yves Bertot and Pierre Castéran. 2004. *Proof by Reflection*. Springer Berlin Heidelberg, Berlin, Heidelberg, 433–448. https://doi.org/10.1007/978-3-662-07964-5_16

François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. 2011. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*. Wrocław, Poland, 53–64. https://hal.inria.fr/hal-00790310.

Gert-Jan Bottu, Georgios Karachalias, Tom Schrijvers, Bruno C. d. S. Oliveira, and Philip Wadler. 2017. Quantified class constraints. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Oxford, United Kingdom, September 7-8, 2017*. 148–161. https://doi.org/10.1145/3122955.3122967

Joachim Breitner, Richard Eisenberg, Simon Peyton Jones, and Stephanie Weirich. 2014. Safe, zero-cost coercions for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM.

Manuel Chakravarty, Gabriele Keller, and Simon Peyton Jones. 2005. Associated Type Synonyms. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP '05)*. ACM.

Martin Clochard, Jean-Christophe Filliâtre, and Andrei Paskevich. 2016. How to Avoid Proving the Absence of Integer Overflows. In *Verified Software: Theories, Tools, and Experiments*, Arie Gurfinkel and Sanjit A. Seshia (Eds.). Springer International Publishing, Cham, 94–109.

N. G. de Bruijn. 1991. Telescopic Mappings in Typed Lambda Calculus. *Inf. Comput.* 91, 2 (1991), 189–204. https://doi.org/10.1016/0890-5401(91)90066-B

Iavor S. Diatchki. 2015. Improving Haskell Types with SMT. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell '15)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/2804302.2804307

Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, and Aaron Tomb. 2016. Constructing Semantic Models of Programs with the Software Analysis Workbench. In *Verified Software. Theories, Tools, and Experiments - 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17-18, 2016, Revised Selected Papers*. 56–72. https://doi.org/10.1007/978-3-319-48869-1_5

Richard Eisenberg and Stephanie Weirich. 2012. Dependently Typed Programming with Singletons. In *Proceedings of the 2012 Haskell Syposium (Haskell '12)*. ACM, 117–130.

GHC Issue #15186 2018. ghc#15186 8.4.2 panic in profiling build. https://ghc.haskell.org/trac/ghc/ticket/15186

Jeremy Gibbons and Bruno C. d. S. Oliveira. 2009. The Essence of the Iterator Pattern. *Journal of Functional Programming* 19, 3–4 (July 2009), 377–402.

Oleg Kiselyov and Chung-chieh Shan. 2004. Functional pearl: implicit configurations-or, type classes reflect the values of types. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2004, Snowbird, UT, USA, September 22-22, 2004*, Henrik Nilsson (Ed.). ACM, 33–44. https://doi.org/10.1145/1017472.1017481

John Launchbury and Simon Peyton Jones. 1994. Lazy functional state threads. In *ACM Conference on Programming Languages Design and Implementation, Orlando (PLDI'94)* (acm conference on programming languages design and implementation, orlando (pldi'94) ed.). ACM Press, 24–35. https://www.microsoft.com/en-us/research/publication/lazy-functional-state-threads/

J. R. Lewis and B. Martin. 2003. Cryptol: high assurance, retargetable crypto development and validation. In *IEEE Military Communications Conference, 2003. MILCOM 2003.*, Vol. 2. 820–825.

Sam Lindley and Conor McBride. 2013. Hasochism: The Pleasure and Pain of Dependently Typed Haskell Programming. In *Proceedings of the 2013 Haskell Symposium (Haskell '13)*. ACM.

Conor McBride and Ross Paterson. 2008. Applicative Programming with Effects. *Journal of Functional Programming* 18, 1 (2008), 1–13.

Laura McKinney and Jef Bell. 2014. The collaborative web: Building a divergent organizational model for research and development, based on 21st century notions of employee engagement. In *Proceedings of PICMET '14 Conference: Portland International Center for Management of Engineering and Technology; Infrastructure and Service Integration*. 7–17.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple Unification-Based Type Inference for GADTs. In *Proceedings of the 2006 International Conference on Functional Programming (ICFP '06)*. ACM.

Amr Sabry and Matthias Felleisen. 1993. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation* 6, 3 (1993), 289–360. https://doi.org/10.1007/BF01019462

Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. 2008. Type Checking with Open Type Functions. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*. ACM, New York, NY, USA, 51–62. https://doi.org/10.1145/1411204.1411215

Tim Sheard and Simon Peyton Jones. 2002. Template meta-programming for Haskell. 1–16.

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton Jones. 2014. Refinement Types for Haskell *(ICFP '14)*. ACM, 269–282.

Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. 2013. System FC with Explicit Kind Equality. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM.

Stephanie Weirich, Dimitrios Vytiniotis, Simon Peyton Jones, and Steve Zdancewic. 2011. Generative type abstraction and type-level computation. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (proceedings of the 38th annual acm sigplan-sigact symposium on principles of programming languages ed.). ACM SIGPLAN. https://www.microsoft.com/en-us/research/publication/generative-type-abstraction-and-type-level-computation/

Edward Z. Yang. 2017. *Backpack: Towards Practical Mix-In Linking In Haskell.* Ph.D. Dissertation. Stanford University.

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '12)*. ACM, New York, NY, USA, 53–66. https://doi.org/10.1145/2103786.2103795