Parallel Type-checking with Haskell using Saturating LVars and Stream Generators

Ryan R. Newton[1] Ömer S. Ağacan[1] Peter Fogg[2] Sam Tobin-Hochstadt[1]

Indiana University[1], edX[2]

{rrnewton, oagacan, samth}@indiana.edu, peter.p.fogg@gmail.com

Abstract

Given the sophistication of recent type systems, unification-based type-checking and inference can be a time-consuming phase of compilation—especially when union types are combined with subtyping. It is natural to consider improving performance through parallelism, but these algorithms are challenging to parallelize due to complicated control structure and difficulties representing data in a way that is both efficient and supports concurrency. We provide techniques that address these problems based on the LVish approach to deterministic-by-default parallel programming. We extend LVish with *Saturating LVars*, the first LVars implemented to release memory during the object's lifetime. Our design allows us to achieve a parallel speedup on worst-case (exponential) inputs of Hindley-Milner inference, and on the *Typed Racket type-checking algorithm*, which yields up an 8.46× parallel speedup on 14 cores for type-checking examples drawn from the Racket repository.

1. Introduction

Recent programming language advances often rely on sophisticated type systems [7, 8, 21, 33, 38], many of which incur a substantial computational expense at type-inference or type-checking time. In some cases, such as *Liquid Haskell*'s refinement types, it is possible to offload this work to an optimized (SMT) solver [21]. In other cases—occurrence typing, dependent typing, and gradual typing [37]—using an external solver is infeasible. Gradually-typed languages, for example, may employ uncommonly expressive type systems to capture idioms from dynamically typed programming.

Typed Racket, for example, combines subtyping with first-class polymorphism and flexible union types, resulting in difficult-to-compute type inference problems even when restricted to local inference. This difficulty is not unique to Typed Racket, it is shared by type systems for JavaScript (flow [3], Typescript [4]), Erlang (Dializer [2]), and Ceylon [1]. But in this paper we focus on Typed Racket as a case in point. In Typed Racket, even the arithmetic plus operation combines *hundreds* of distinct type signatures. As a result, a Typed Racket user recently posted a problematic program which took more than 40 seconds to compile, dominated by type inference for just a few function calls.[1]

In the era of ubiquitous parallel hardware, one idea is to *parallelize* these computationally expensive phases to reduce the compile-edit-debug latency and enhance the software development experience. Yet there has been little work on parallelizing compilation of code below the granularity of a file or module, with the exception of register allocation [40] and flow analyses [30]. Further, to the best of our knowledge, no prior work has parallelized type-checking algorithms for a full language like Typed Racket, nor shown results for parallel unification in type-checking.

Most type checkers involve a unification process that contains latent parallelism but exhibits poor locality. A simple example is Hindley-Milner type inference, in which distinct expressions might be processed in parallel, but where each individual *type variable* can gain information from distant parts of the program (and therefore from different threads). Indeed, even in functional-language implementations of type inference—such as in the Glasgow Haskell Compiler (GHC)—mutable references are often used for constraining type variables, complicating parallelization further.

Our goal is to parallelize despite these constraints, using language constructs that enforce disciplined, monotonic use of mutable state and eliminate unintended nondeterminism. In this work we perform two experiments in parallel typechecking, using Haskell as our implementation language for writing parallel checkers.

LVars for Type Variables? The use of mutable type variables requires introducing synchronization in any parallel type-checking implementation, to avoid data-races. In fact, there exists a class of synchronization variables that are safe to share between computations in a functional language while retaining data-race freedom and even *determinism* (both of which purely functional programs enjoy by construction). Single assignment variables, or *IVars* [6], are one early example in this class. More recently, we introduced a generalization that enables deterministic, functional programs to synchronize on arbitrary *monotonic data structures*, called *LVars* Kuper and Newton [22]. LVars preserves external determinism while internally scheduling tasks nondeterministically. LVars share similarities with Concurrent Constraint programming [31], and enable a general form of deterministic-by-default parallel programming, implemented in Haskell by the "LVish" library[2].

Previous work on LVars introduced general-purpose LVar data structures including: (1) lock-free collections with concurrent insertion (but not deletion), and (2) counters that increase monotonically. But application-specific LVars can be constructed as well; in particular, an LVar would *seem* to provide a promising way to deal with type variables shared between threads. A custom LVar could capture the partial order implied by type unification. However, two problems arise:

1. All published examples of LVars respond to conflicting information by throwing an exception, which in general cannot be caught deterministically within the purely functional core of a language [28].
2. While LVars are a good fit for *and-parallelism*—where threads join information concurrently—they do not help with the *or-*

[1] Program due to Antonio Leitao, `https://groups.google.com/forum/#!msg/racket-dev/7LZWawMMg04/u3DOQVaEAQAJ`.

[2] `http://hackage.haskell.org/package/lvish`

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

PPoPP '16, March 12 - 16, 2016, Barcelona, Spain
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4092-2/16/03...$15.00
DOI: http://dx.doi.org/10.1145/2851141.2851142

parallelism found in some type systems, where speculative, alternative additions of information must be considered.

Contributions In this paper, we overcome these two problems and demonstrate the **first wall-clock parallel speedup on type inference with unification**. Specifically:

- We introduce Saturating LVars (§4), an application of the existing LVar theory that adds the capability for both trapped-failure and memory reclamation—addressing a major limitation of previously studied LVars. As a demonstration, we use Saturating LVars for speeding up an implementation of Hindley-Milner type inference on some inputs.
- We introduce *stream algebras* to describe modular formulations of or-parallel constraint systems parameterized by an algebra for manipulating streams of partial solutions. We provide an efficient implementation using *generators* (§5).
- We then scale this stream generator architecture to apply it to the Typed Racket type system (§6), achieving wall-clock speedups both due to deforestation and due to parallelization.

2. LVars & LVish: Background

LVars generalize the earlier IVar model by allowing multiple writes. Where IVars simply signal an error upon writing to an already-full location, LVars allow the states to be *joined* in a monotonically increasing fashion according to a partial order on the possible states of the data structure. The state space (hereafter *lattice*) contains two distinguished elements $\bot$ and $\top$ (representing uninitialized and error respectively) along with a partial ordering $\sqsubseteq$. One way to increase the state of an LVar is through a `put` operation that takes the *least upper bound* of its current state and its argument.

As a simple example, consider the lattice of natural numbers ordered by the relation $\leq$. In the following program, two threads race to write to an LVar lv:

```
do lv ← newMaxIntLVar
   fork (put lv 1)
   put lv 2
```

Regardless of the order in which the threads write to lv, the join operation ensures that the final state of lv is "2"—the lub of both writes. As a result, we can freely share LVars between threads, safe in the knowledge that we will deterministically receive a result (or an error, in the case of the $\top$ state), because `put` operations always *commute*. In this way, each LVar provides a parallel *reduction* with an associative operation (least upper bound).

LVars also allow a restricted form of read via the `get` operation. Generalizing the blocking reads of IVars, this operation will block until the LVar's state has reached one of a designated subset of the lattice elements, known as the *threshold set*. This threshold set determines what kind of `get` operation is performed—i.e., reading a specific slot in an array, or key in a map. Get operations allow us only to observe that the LVar is *above* some element of the threshold set, rather than its precise state. The threshold set Q is required to be *incompatible*, that is, $\forall a, b \in Q, a \neq b \rightarrow a \sqcup b = \top$.

As an example, an associative map LVar would have states such as $\{(k_1, v_1), (k_2, v_2), \dots\}$, and the threshold set corresponding to a blocking read at key k_1 would be $\{(k_1, v_i) | v_i \in Q\}$. Of course, this threshold set, being potentially infinite, is not represented at runtime.

In addition to thresholded `get` operations, changes to an LVar can be observed through *handlers* (callbacks). When we attach a handler to an LVar, it is called upon *each* change, and receives the new state or state delta. With a container LVar, such as the associative map, the handler would be called on every element added, e.g.:

```
addHandler mapLV (λ (key,val) → ... )
```

One interesting aspect of handlers is that `addHandler` must *commute with puts*. That is, upon adding the handler above, it fires for all existing elements as well as future additions to the set. Later in this paper we will see how the concept of handlers interacts with our proposed extension, saturating LVars.

Finally, LVars also offer the option of reading their full contents *exactly*, after they have been *frozen*. The `freeze` operation disallows further modifications, raising an exception if this occurs. For full determinism, freezing must occur only after a global barrier to avoid races between `put` and `freeze`.

Language agnostic LVars are a general parallel programming construct and could be implemented in any language. The benefits are clearest, however, in a *deterministic by default* language that is able to statically enforce determinism by controlling side effects. Deterministic Parallel Java [7], for instance, is a research prototype that meets this criteria. Haskell, on the other hand, is the only full featured language—continuously developed and with a sizable supporting community—that fits this description. Thus the LVish library is currently implemented for Haskell. Following that, the code examples in this paper are written in Haskell, and use LVish conventions and functions where applicable.

2.1 A quick Haskell primer

We already saw some small Haskell examples in the previous subsection, but here we give a quick introduction to the Haskell features used in this paper, for readers unfamiliar with the language. Conversely, those familiar are encouraged to skip to Section 2.2.

We assume a basic understanding of functional languages. Haskell can be read as a mathematical notation with programs consisting of equations defining functions. A caveat is that function application is written `f x` rather than `f(x)`, e.g.:

```
f x = 2 ∗ x + 1
```

Also, we show anonyous functions using standard λ notation, such as `(λ x → 2∗x+1)`.

Types The operator for type-annotation is also important, where `(e :: t)` is read *expression* `e` *has type* `t`, for example: `3::Int` or `(λx→x+1) :: Int →Int`. The grammar of types in Haskell includes the application of type constructors to arguments, such as `T Int Bool` (rather than `T<Int,Bool>`) which enables parametric polymorphism, aka generics. The syntax of types is complicated by the fact that some type constructors have special or infix syntax: e.g. `[T]` and `T1→T2` rather than "`List T`" or "`Fun T1 T2`".

An additional complication comes from Haskell's system of constrained types. For example, the literal `3` does not have type `Int` in Haskell, but rather `Num a ⇒ a`, which is read "any type `a` where `a` is in type class `Num`". Everything before ⇒ is part of the constraint, rather than the type proper. Standard classes include `Eq`, `Ord`, and `Num`, and we will see these constraints in LVish computations where they document or limit parallel communication effects.

New data types in Haskell are declared using the `data` keyword. Any data type may be an enumeration or *sum* type. For instance, a data type with three constructors can be defined like this:

```
data T = T1 Int | T2 Bool | T3
```

Unlike a simple enum, each variant can have one or more *fields* attached to it: `Int`, `Bool`, and nothing, respectively. A `case` expression can be used to handle each variant in turn:

```
f :: T → String
f v = case v of
        T1 i → "case T1: " ++ show i
        T2 b → "case T2: " ++ show b
        T3   → "case T3"
```

Any argument to a function or λ can also pattern match against such constructors to extract the fields.

Side effects Being a purely functional language, Haskell restricts side effects. That is, effects are permitted, but are tracked by the type system, and typically introduced using a notation similar to statements in an imperative language:

```
do statement1
   x ← statement2
   return (x + 1)
```

Here `statement1` is a *monadic* expression that performs an effect, and doesn't return anything useful. `statement2` is like `statement1`, except it returns a value in addition to performing an effect, and the return value is bound to variable `x`. The last line adds one to `x`, and puts the value in the monadic context with `return`.

A `do` block forms a single expression in Haskell, and it's type is `m a` where `m` is a monad and `a` is the type of return value. The choice of monad determines which effects may be performed. The most common monad is `IO`. For instance, the `do`-block above could have the type `IO Int`. In this paper we focus not on IO, but on a `Par` monad for parallelism, separate from any other effects.

2.2 LVars, in practice

In the LVish library, parallel computations are exposed through a `Par` monad. That is, `do`-blocks that use LVish constructs have type `Par e s a`, where `a` is the return value of the monadic `Par` computation; the `s` parameter associates LVars with a specific session and prevents them from escaping; and the `e` type parameter documents the *effect signature* of the computation. For example, a function over `Int`s that executes in LVish's Par monad and may `put` to an LVar has this type:

```
foo :: (HasPut e) ⇒ Int → Par e s Int
```

The `Par` monad is further equipped with various functions to launch parallel computations and extract their result:

```
 runPar        :: Det e ⇒ (∀ s. Par e s a) → a
 runParNonDet  ::          (∀ s. Par e s a) → IO a
```

These ensure that only deterministic (`Det`) combinations of effects are used from inside purely functional code, whereas nondeterministic combinations require `IO`. This is standard, as the `IO` type in Haskell encompasses everything that cannot be considered purely functional, including file IO and nondeterminism.

Irrespective of which `runPar` variant is used, the final return values of type 'a' are pure Haskell values, i.e., not LVars. The "∀ s" above gives `runPar` a *rank-2 type* and is a standard trick to ensure that LVars do not escape a `runPar` session, i.e. it is the same approach used by Haskell mutable references, aka `STRefs`, which we will encounter in the next section. If one wishes to *return* LVars without copying, they instead use `runParThenFreeze`, which uses the implicit barrier at the end of a `runPar` parallel region to guarantee a race-free freeze of the result:

```
runParThenFreeze :: (Det e, DeepFrz a)
                 ⇒ Par e NonFrzn a → FrzType a
```

Freezing has no runtime cost. Rather, `FrzType` is a type-level function (type family) that "casts" the monotonic/mutable version of the LVar to a pure/immutable sister type. `FrzType` is associated with the `DeepFrz` class and implemented by each LVar in the library. (`NonFrzn` is a safety detail—placed in the `s` parameter to prevent mutating LVars that were frozen in other runPar sessions.)

A note effect signatures Effect signatures are important to the LVish library; some effects are only *conditionally* deterministic; for instance, canceling read-only futures is fine, but canceling a call to `foo` above would introduce an observable data-race. Yet effect signatures are not central to the way we use LVars in this paper, so `e` parameters may be largely ignored. Through the rest of the paper we will use LVars, handlers, and freezing. Also, in §4, we will extend the mechanisms already covered with *Saturating LVars*.

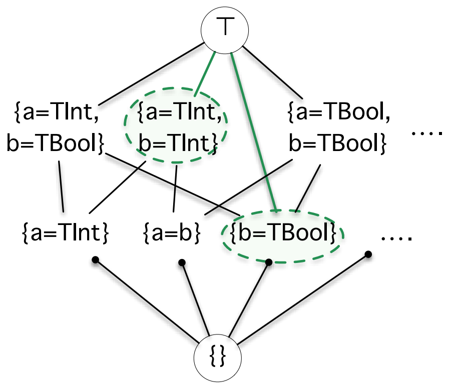


Figure 1. The partial order for the store containing all type variables used in a type-inference execution. The two highlighted nodes are *incompatible*—their lub is ⊤.

3. *And-Parallelism*: Hindley-Milner Typing

We now have what we need to parallelize a basic type checking algorithm. We begin with what is perhaps the most well-known unification-based type inference algorithm: the Hindley-Milner system [12]. Opportunities for effective parallelism in Hindley-Milner are limited, but we use it as a warm-up exercise before proceeding to the Typed Racket type system in §6.

Error handling In our exposition here we do not address error reporting. Effective type error reporting is an active area of study in its own right. Error reporting is often handled by a separate, "slow path" algorithm, whereas the fast path—compiling well-typed code—can use a type checking algorithm that merely returns true or false. (Indeed, this is already the case for Typed Racket's inference function.)

Sequential Hindley-Milner The Hindley-Milner algorithm operates by walking over an expression, generating constraints over type variables. These constraints are unified together to produce a final typing judgement for the term. An implementation that uses only immutable data would keep a store mapping type variables to types, e.g., `Map Var Type`. Recursive calls in the unifier produce partial maps that are *joined* together. This process is widely regarded as inefficient, and in practice even type checkers written in Haskell use a *mutable* representation of type variables to perform unification updates in-place. In this case, an explicit type variable store is unnecessary and monomorphic types can be defined as:

```
data Mono s = TVar Name (STRef s (Maybe (Mono s)))
            | TInt
            | TFun (Mono s) (Mono s)
```

Here the type variable is represented directly by its pointer to the mutable location. The `s` parameter plays the standard escape-prevention role discussed above and must be plumbed through `Mono` to reach the mutable `STRef`. The `STRef` [24] allows a type variable to be imperatively updated in the `unify` function, as shown below.

```
unify :: Mono s → Mono s → ST s ()
unify t1 t2 = do
  case (t1, t2) of
   (TInt, TInt)     → return ()
   (t, TVar v r)    → unify (TVar v r) t
   (TVar v ref, t) →
     if occurs v t
     then error "can't construct infinite type"
```

```
      else do contents ← readSTRef ref
              case contents of
                Nothing → writeSTRef ref (Just t)
                Just t' → unify t' t
  (TFun t1 t2, TFun t1' t2') →
    do unify t1 t1'
       unify t2 t2'
  _ → error "can't unify"
```

The infer function simply walks over the expression, unifying type variables as they are found. For brevity, only the application case is shown.

```
infer :: Env s → Term → ST s (Mono s)
infer env expr = case expr of
  ...
  App e1 e2 →
    do fnT ← infer env e1
       arg ← infer env e2
       (v, ref) ← freshTVar
       unify fnT (TFun arg (TVar v ref))
       return (TVar v ref)
```

Alas, in exchange for increased performance in the single threaded case, using full STRefs (instead of pure functions) effectively takes statically-guaranteed deterministic parallelism off the table.

Exploiting parallelism Fortunately, we can parallelize the algorithm by switching from STRefs to an LVar constraint store, and can even asynchronously share type constraints *between threads* (as in concurrent constraint programming [31]). The collection of type-variable constraints accumulated during type-checking forms a partial order under unification. We can model this state with a single Map-like LVar. Note that, semantically, we cannot make *each* type variable an LVar, because they lack a non-interference property—constraining one may affect another.

Thus a TyVarMap is an application-specific LVar with a lattice as shown in Figure 1; each type variable in the store is either empty (unconstrained) or filled with a type, and an LVar put operation corresponds to adding new information to a type variable. In this case, unification of that information is a built-in behavior of the LVar. Likewise, the LVar internally performs the occurs check, so that a put which would cause a cycle goes to $\top$ instead.

The TyVarMap is indexed by an opaque type TyVar, and maps each TyVar onto a Mono type. Mono remains the same as in our previous example, except with TyVar replacing the STRef:

```
data Mono s = TVar Name (TyVar s) | ...
```

Unification is modified to internally include parallelism:

```
unify :: (HasPut e) ⇒
         TyVarMap s → Mono s → Mono s → Par e s ()
unify m ta tb = case (ta, tb) of
  ...
  (TFun t1 t2, TFun t1' t2') → par2 (unify m t1 t1')
                                    (unify m t2 t2')
```

Here par2 is a fork/join combinator, executing two actions in parallel and returning two values.

Mono, together with operations on the TyVarMap become an ADT for monotonic unification. We expose unify in this API which subsumes a direct put operation on a specific TyVar. We also need a way to extend a TyVarMap by allocating a fresh type variable with a fresh name, TVar n k:

```
freshTVar :: TyVarMap s → Par e s (Mono s)
```

Finally, at the very end of type checking, when the constraint store is final (frozen), it becomes possible to read out individual entries:

```
getTyVar :: TyVarMap Frzn → TyVar Frzn → (Mono Frzn)
```

Optimizations The API above can be implemented using *safe* constructs that use memory deterministically (and monotonically) by construction, but this comes at a performance cost. For instance, if we use an existing set LVar we could accumulate constraints, but would read the set (and check for conflicts) until the end of a type-checking job. This is an extreme form of the disadvantages described earlier for purely functional (non-STRef) implementations of unification.

Thus, as with virtually all LVars, it makes sense to use raw, unsafe memory memory operations to implement a safe ADT. That is, we want to mutable update each type variable and perform downstream unification on the fly, leveraging the fact that unification is associative and commutative.

We can also go further. Because (1) the key type (TyVar) is opaque, (2) we permit only one TyVarMap per runPar, and (3) we expose no operation for iterating over the map, we can thus deforest the map altogether, and define TyVar itself as a pointer to a mutable location. Thus, even though we logically have a single LVar per type checking session, operationally, this brings us back to a runtime representation similar to the per-type-variable STRef, with each TyVar key a mutable pointer.

Parallel Inference Irrespective of the implementation of Mono and unification, inference is built on top of that ADT using only safe-by-construction parallelism constructs. For instance, in the application case of infer, we can recur on both subexpressions in parallel while sharing a constraint environment:

```
infer :: TyVarMap s → Env s → Term → Par e s (Mono s)
infer tm env expr =
 case expr of
   ...
   App e1 e2 →
     do (fnT, arg) ← par2 (infer env e1) (infer env e2)
        tv ← freshTVar
        unify fnT (TFun arg tv)
        return tv
```

These recursive calls use unify to add constraints to type variables. Then the result of infer can extracted with runParThenFreeze to yield a (Mono Frzn), which allows traversing the type variables, using getTyVar, to read out the full monotype.

This approach yields a speedup when used to implement a micro-ML calculus and run on a synthetic benchmark (see Figure 2). (For type-checking benchmarks drawn from actual code, see §6.) In this case, the benchmark is a large program with 1000 copies of a known-exponential nested-let expression. This is only a small proof of concept—for real, large programs with Hindley Milner inference, the challenge will be finding problems that take long enough in practice (relative to the amount of memory they read) that they are worth parallelizing. Nevertheless, with algorithms such as Hindley-Milner—which perform well in the average case but have dismal worst-case performance—it may be worth researching parallelism as an "insurance policy" to ameliorate these worst case outcomes.[3]

4. Saturating LVars: Trapped Failure

There is a problem with the formulation of Hindley-Milner type inference in the previous section. If type-checking fails we would like to simply return False or Nothing. But with the implementation in the previous section this failure instead appears as incompatible puts, e.g. putting TInt and TBool to the same TyVar. Further, in LVar-based programs, adding contradictory information to an LVar always triggers the $\top$ state, which in previous implementations of LVish meant throwing an exception.

[3] Of course, a linear parallel speedup can't match an exponential slowdown, but it needn't—even outliers, slow infer calls, in real programs are of fixed nested-binding depth, not *growing* in the dimension that incurs exponential complexity.

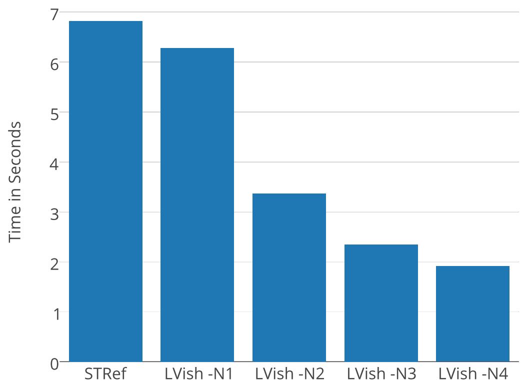


Figure 2. Hindley-Milner type inference on a generated term with roughly a million nodes. The sequential (STRef) version is compared against the LVish version. Note that even on one core, the LVish version is still *concurrent*, sharing constraint information between concurrent computations via LVars. Experiments were performed on a desktop-class Intel Xeon i5-3470, with GHC 7.8.3 and `+RTS -qa`.

What is wrong with throwing an exception? Answering this requires a bit of background. Haskell enables purely functional but *partial* programming, and Haskell's exception semantics [28] require that exceptions be handled only in the `IO` monad to retain referential transparency. In this case, it is important that we keep our type checker *out of* the IO monad. Because we aim for a deterministic type checker, we should either use only determinism-safe features (not `IO`) or reduce the amount of "trusted code" that uses unsafe features that may introduce nondeterminism—most especially avoiding Haskell's infamous "`unsafePerformIO`".

Keeping failures in Par In order to avoid the exception handling problem, we must capture and respond to type checking failures *within* the `Par` monad. That is, when type variables gain conflicting information, we want to simply return a value indicating no valid substitution exists.

To enable trapped failures, we introduce *Saturating LVars*, defined as an LVar whose lattice structure includes an additional state Sat. Following the semantics for LVars formalized in our previous work [23], such an LVar is given by a five-tuple $(D, \sqsubseteq, \bot, \top, Sat)$, extended to include the designated saturation state as well as the usual set of states (D), a partial order ($\sqsubseteq$), and designated bottom and top states. It should further hold that:

$$\bot, \top, Sat \in D, \bot \neq \top, \; Sat \neq \top$$
$$\forall d \in D, \; (\bot \sqsubseteq d \sqsubseteq \top) \; \wedge \; (d \sqsubseteq Sat \vee d = \top)$$

An example lattice extended with the Sat state is pictured in Figure 3. Alternatively, instead of adding a new state Sat, we could globally reinterpret $\top$ for all LVars, but we eschew this option so as to use Saturating LVars in programs that also have other LVars. Further, while the penultimate-Sat-state convention is simple, its ramifications are not:

1. The only usable (incompatible) threshold set for performing blocking read or adding a handler[4] are singleton threshold sets, of which the only useful one is $\{Sat\}$.
2. A saturating LVar's state moves monotonically up the lattice, but it does *not* monotonically gain information (bits). Notably an LVar in the saturated state can be represented by as little as one bit. This means that saturating LVars are the first practical example of LVars that can release memory during their lifetime. As saturating LVar's don't change the theory, this potential was always present—when a higher state in a lattice requires fewer bits to represent—but no extant LVars fell into this category.
3. Computations whose *only* effect is to write to a Saturating LVar can be *cancelled* if that LVar saturates.

[4] Actually, there is a safe way to add handlers that are notified of *every* `put` to the `SatLVar`, but it requires that the handlers be attached *at the point the LVar is created*, which prevents `addHandler` racing with `saturate`.

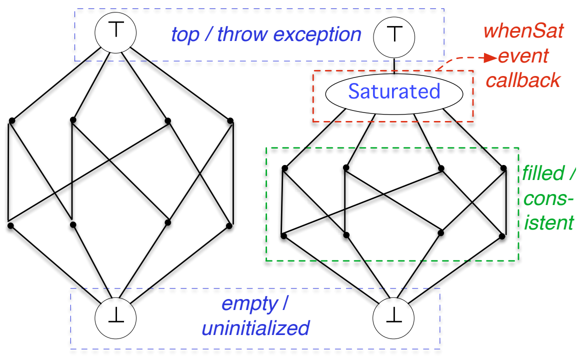


Figure 3. Any valid LVar lattice is turned into a Saturating LVar by adding an extra, penultimate state.

Because of the lack of useful threshold sets to underpin a family of `get` operations (beside $\{Sat\}$), saturating LVars become effectively *write only*. But all Saturating LVars can provide the following operations (in addition to LVar-specific `put` operations):

```
class DeepFrz lv ⇒ SatLVar lv where
  saturate  :: lv → Par e s () -- Force to Sat state
  whenSat   :: lv → Par e s () → Par e s ()
  isSat     :: FrzType lv → Bool
```

This interface provides the ability to force LVars to saturate, respond to saturation, and test for saturation after a parallel computation is complete.[5] Using this generic interface, we could, e.g., arrange for a saturatable set of LVars to become saturated whenever one of its element does, e.g.:

```
do set ← newEmptySet
   -- Register a callback on each element inserted:
   let addToSet sv =
         do whenSat sv (saturate set)
            insert sv set
   ...
```

We have produced a new, modified LVish library that supports saturating LVars and in this library we provide:

- a *`SatMap`* data structure, which is a *single* LVar that maps keys onto pure Haskell values that are instances of `PartialJoinSemiLattice`. That is, multiple puts are allowed on the same key, and are joined, but the `join` function may fail, saturating the entire `SatMap`.
- a *`FiltSet`* data structure that takes advantage of saturated LVars in a different way—it represents a dynamic collection of not-yet-failed saturated LVars. We observe that the type `Set (Maybe a)` is isomorphic to `(Bool, Set a)`, that is, there's no need to store *each* element which has saturated. If we are interested in collecting `SatLVar`'s that have *not* failed, then failed

[5] The LVar can be tested for saturation with `isSat` only *after* a parallel region has ended and it is in a "frozen" state. Freezing LVars is covered in detail in [23].

insert(10)	OLS Regression	R^2 goodness-of-fit
Map LVar	19.5ns	0.991
SatMap	18.4ns	0.993
Set LVar	18.5ns	0.989
FiltSet	91.1ns	0.990

Table 1. Microbenchmark: the cost of creating a new structure and inserting ten new `Int` elements. The cost of this (comparatively) cheap operation is measured by varying the number of iterations of benchmark, and computing a linear regression between iterations and cycles (above). All measurements are from the *desktop* platform described in §6.3.

insert/sat/insert	Set LVar	FiltSet
cycles	17838	15160
bytes alloc	14733	14128
bytes copied	759	115

Table 2. Microbenchmark: the cost of inserting 10 `Counter` elements in a set, saturating the previous 10, and repeating N times. The `LVar.Set` version must store the data as a set of nested LVars to enable saturation of the inner variables. The `FiltSet` directly supports multiple assignments to a key, so requires one LVar rather than $10N+1$. Above we regress N against cycles, and below we regress against bytes allocated and bytes copied during garbage collection. This verifies that the while the `FiltSet` benchmark allocates $O(N)$ memory, it releases memory as it goes.

LVars can be discarded at runtime, freeing memory and shrinking the set's physical size.

In type checking, the `SatMap` structure is useful for representing environments containing constraints, and a `FiltSet` can serve as the accumulator when searching for a valid environment. These two data structures are part of the parallel-type-checking toolkit we provide in our new library; the microbenchmarks in Tables 1 and 2 show their performance relative to more basic LVar counterpart data structures.

5. *Or-Parallelism* with Stream Algebras

With LVish plus the saturating LVar extension, we've acquired the first tools in our parallel type-checking toolbox, enabling us to handle parallel *conjunctions* over constraint-generating computations. Indeed, Hindley-Milner type inference required only conjunction, never disjunction. In this section we begin to address a broader—and more expensive in practice—class of type checking algorithms: those with *or-parallelism*. This is challenging, because we cannot directly employ LVars the way we did in the previous section, storing type variable bindings in an LVar and updating it destructively.

Nevertheless, it is important to exploit or-parallelism for effective speedups in Typed Racket. Yet rather than dive directly into Typed Racket in this section, we first introduce the implementation techniques using a simpler example problem that incorporates conjunctions and disjunctions: satisfiability (SAT).

Parallel Constraint Solving Type systems are a particular flavor of constraint problem. Indeed, if we view parallel type checking as a parallel constraint satisfaction problem, we can look for guidance from previous work on parallel logic programming [10, 17] and parallel constraint solvers [15, 31, 39]. Unfortunately, the data-structures and synchronization strategies employed in these works are extremely specialized to the constraint system being solved: for example, SAT solvers have developed a large body of specialized data structures and parallelization strategies [18, 19].

Here we explore parallelization strategies that apply to *any* constraint domain that can be formulated as an LVar. We will then apply the same techniques to Typed Racket in §6. To illustrate our approach we start with a simple constraint domain that consists of variable assignments, e.g., `x=4`, closed under conjunction and disjunction. Thus the input to our algorithm is a term such as:

```
((x = 3 ∧ y = 4) ∨ (y = 3)) ∧ (y = 3 ∧ z = 9)
```

We will solve these constraints using a recursive algorithm that traverses the term, returning a stream of possible solutions for each subterm. These solution streams can be combined by concatenation (or), as well as by joining partial of solutions drawn from the cartesian product of two streams (and). In fact, these stream operations form a semiring, and we can formulate a simple generic solution abstracted over a set of methods matching the following signature:

```
data Semiring t
  = Semiring { one :: t
             , zer :: t
             , add :: t → t → t
             , mul :: t → t → t
             , fromAssigns :: Map Var Int → t
             }
```

This provides a simple algebra for solution streams, plus `fromAssigns`, which is specific to the SAT problem and initializes a solution from an initial variable assignment. In fact, the above primitives are also used by our Typed Racket implementation to manipulate streams of type environments, changing only the type of `fromAssigns`.

5.1 The Simplest Stream Algebra

In a sequential Haskell implementation, the natural representation of solution streams for SAT is as a lazy list of variable assignments, each represented by a `Map`:

```
 type Sol1 a = [Env a]
 type Env a = Map Var a
```

For simple equality constraints, the algebra is implemented as:

```
listStrms :: Semiring (Sol1 Int)
listStrms = Semiring
 { one = [Map.empty]
 , zer = [ ]
 , mul = λs1 s2 → catMaybes [ joinEnvs env1 env2
                            | env1 ← s1, env2 ← s2 ]
 , add = λs1 s2 → s1 ++ s2
 , fromAssigns = λx→x
 }
```

The cartesian product operation above creates a list of many `join` computations, which could be evaluated in parallel. In fact, we attempted to parallelize in the standard Haskell way by adding `parList` or `parBuffer` annotations to this list, either before or after the `catMaybes` call. Unfortunately, this does not yield a parallel speedup (either for satisfiability or full Typed Racket), because the parallel work is too entangled with bookkeeping on lazy lists.

5.2 A Parallel Stream Algebra with Generators

List-based streams incur a lot of overhead. Intermediate lists are assembled and deconstructed repeatedly. Further, the aggressive fusion optimizations performed by GHC and its libraries **cannot eliminate operations like cartesian product**.

Fortunately, there are more efficient ways to represent streams, in particular as *generators*. Generators have a long history as a control mechanism in programming languages[6]. A generator takes a partial answer and a continuation; it modifies, tests, or bifurcates

[6] Generators first appeared in the language Alphard in the mid 1970s [32], and later in CLU [25] and Icon [16]. A clear explanation of generators can be found in [5].

the partial answer; and then passes one or more answers on to the continuation.

```
type Cont = PartialAns → [PartialAns]
type Generator = Cont → Cont
               = Cont → PartialAns → [PartialAns]
```

Generators can be composed without creating intermediate lists—only the final step allocates the list, `[PartialAns]`. Indeed, generators, formulated in terms of continuations, have been used for this deforestation benefit in many contexts. Yet there has been little work on their use in parallel programming[7]. For example, if our answer type is a computation in the LVish `Par` monad, we get the following solution type for satisfiability problems:

```
type Cont e s a = Env a → Par e s ()
type Sol2 e s a = Cont e s a → Cont e s a

contBased :: (...) ⇒Semiring (Sol2 e s a)
contBased = Semiring { zer = (λ_ _ → return ())
                       ... }
```

A solution stream—the element type of our `Semiring`—becomes a function of the form `(λ k w →action)`, where `action` constrains the variable assignment, `w`, in one or more ways and passes each variant on to the continuation `k`. Each computation, `Par e s ()`, does *not* return a value. Rather, we extract a result by attaching a final continuation that inserts into an output `Set` or `FiltSet` LVar.

We see above that the *zero* for the algebra drops the partial assignment `w` on the floor, *not* calling the continuation. The *one* value is `(λk w →k w)`, and `fromAssigns` adds its input constraints before passing on `w`. The disjunction, or add operation is the obvious place to add parallelism:

```
add s t = λ k w →
                  do fork (s k w)
                     t k w
```

This parallel-OR duplicates the continuation, passing the incomplete answer to alternative code paths that extend it in different ways. Note that because we thus far assumed immutable environments, `w` does not need to be *copied* before it is sent to different destinations `s` and `t`.

AndPar: first technique The sequential version of binary conjunction with immutable environments is:

```
mul s t = λ k w → s (λ x → t k x) w
        = λ k → s (t k)
        = s ∘ t
```

Indeed, there is no obvious opportunity to parallelize this as long as environments are immutable. The continuation transformers are composed, but `w` is threaded through linearly.

With the immutable definition of `Env` as `Map Var a`, we fold the constraints into the environment *before* sending it along to the continuation. It is not possible to extract parallelism here at the level of the `Par` monad. (It would be possible—but not profitable—to *spark* the `foldl` computation, attempting very fine-grained parallelism within the updates to a `Map`.)

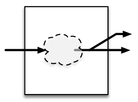
If we use an LVar to represent a (monotonically mutable) environment, we retain the generator design but and-parallelism becomes more feasible. We change the `Env` to an LVar, such as `SatMap a`, and each generator modifies this environment *as an effect*. Generators representing conjunctions only perform these effects and *always* pass the same environment pointer on to their continuation. For example, `(k w)` below:

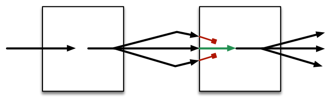
```
fromAssigns init = λ k w →
  do mapM_ (constrain_ w) init
     k w
```

[7] One example in the literature is the related concept of *push arrays* [9] used in data-parallel programming.

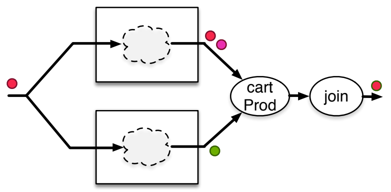
Depending on the size of `init` it is possible to `fork` the entire `(mapM_ ...)` expression or to turn it into a parallel loop.

AndPar: second technique There is another option for parallelizing and, but it may result in repeatedly traversing constraints that are already “settled”. We introduce it visually first and then in code. Let us visualize each continuation transformer as a processing stage, pictured as a box. This is a *push-driven* form of stream processing, where for every partial answer pushed to the input stream, the generator performs some processing and pushes zero or more partial answers on its output stream. A regular conjunction of disjunctions is accomplished by *chaining* these generators sequentially. Indeed, this is our sequential version of `mul`:





Passing rightward through a box corresponds to a partial answer “passing the test” and moving on to the next round. Our second technique for performing and-parallelism is to replace the above picture with:



Here we take each partial answer, *duplicate* it, and feed it through both machines in parallel. The top and bottom boxes may or may not contain disjunctions. All partial answers that make it through the gauntlet on the top are joined with all answers on the bottom, and, if the join succeeds, passed on. In code, we write:

```
mul s t = λ k w →
  do s1 ← newEmptySet; s2 ← newEmptySet
     s3 ← cartesianProd s1 s2
     forEach s3 (λ (a,b) → case joinMaybe a b of
                            Nothing → return ()
                            Just w2 → k w2)
     fork (s w (‘insert‘ s1))
     t w (‘insert‘ s2)
     return ()
```

This uses LVars to accumulate the solutions from each branch, and to take their cartesian product (a monotonic operation, and a standard one for container LVars). There is a delicate trade-off, however, in applying this technique. First, because the input answer `w`, is passed to both branches, all the joins we perform redundantly combine the (obviously compatible) information in `w` with itself. Second, we have now removed some of the deforestation benefit of generators by accumulating the partial answers in set LVars. Nevertheless, we in the next section we will see that this parallel-and technique is quite effective in typing some Typed Racket programs.

6. Typed Racket Type Checking

Typed Racket [37] is a typed version of Racket [13] that uses gradual typing [33, 36] to integrate with untyped Racket. In this paper, we consider only the type checking of typed programs.

Typed Racket’s type system includes a number of features which combine to make type inference difficult. First, Typed Racket supports subtyping, which is used widely in a variety of ways in Racket programs. Second, types include non-disjoint union types, so that $T <: (\cup\ S\ T)$. Third, overloading on function types is supported with ordered intersection on function types [34].

Fourth, Typed Racket supports arbitrary equi-recursive types, used for modeling even simple data structures such as lists.

As a result, Typed Racket, following many other recent languages, does not attempt complete whole-program type inference in the fashion of Standard ML. Instead, it employs *local type inference* and bidirectional type checking [29], similar to, e.g, Scala, TypeScript, and Rust.

In this setting, the central inference problem is choosing an instantiation of type variables when a polymorphic function is applied to concrete arguments. For example, when a Typed Racket programmer writes:

```
(map add1 (list 1 2 3 4))
```

we need to infer that `map` should be instantiated with the types `Integer` and `Integer` for its inputs and outputs, respectively.

As a result, the type inference problem is somewhat simpler than in a global type inference setting. In the above case, if `map` has the type $\forall \alpha\beta.(\alpha \rightarrow \beta) \times \mathsf{listof}(\alpha) \rightarrow \mathsf{listof}(\beta)$, the inference algorithm must find a substitution for α and β that makes $\mathsf{Integer} \rightarrow \mathsf{Integer}$ a subtype of $\alpha \rightarrow \beta$ and $\mathsf{listof}(\mathsf{Integer})$ a subtype of $\mathsf{listof}(\alpha)$.

However, the inference problem in Typed Racket is made more complex than this simple example by several factors. First, type constraints in inference can involve subtyping, not just equality. Second, Typed Racket produces very large types in several circumstances—when providing extremely precise specification of function behavior [34] and when inferring types for large blocks of constant data. As a result of these and other issues, Typed Racket is known to have slow typechecking. We conducted an opportunity analysis of several large Typed Racket libraries, analyzing how long every type inference call took. We discovered one file in the Typed Racket `math` library spends *multiple seconds* typechecking a single pair of function calls, both involving numeric vector operations, and some pathological cases can have typechecking times measured in minutes—one such case was removed from the Typed Racket test suite since it took too long to run. As described in the introduction, this is a practical problem for Typed Racket users.

6.1 The core algorithm

The core of the inference algorithm is an extended form of the Pierce and Turner [29] algorithm, which handles union types, recursive types, and function overloading. The fundamental idea is that we grow a set of constraints on the type variables to be inferred based on the actual types of the arguments provided—in the `map` above the actual arguments are $\mathsf{Integer} \rightarrow \mathsf{Integer}$ and $\mathsf{listof}(\mathsf{Integer})$.

Each constraint is a pair of types: an upper and a lower bound. Two constraints can be combined by joining the lower bounds (represented by the $\cup$ operation) and taking the meet of the upper bounds (meets cannot always be represented exactly in Typed Racket, requiring some approximation). Inference fails if the constraints cross. To solve $S <: (\mu X.T)$, the algorithm must *unroll* recursive types; to ensure termination, the recursive solver must also keep a *seen* set, so that if, while unrolling, the same $S <: T$ is encountered again, the algorithm terminates successfully.

The additional complexity, and source of or-parallelism, comes from handling union types and overloaded function types. To make a type A a subtype of $(\cup\, B\, C)$, it must merely be a subtype of *one of* B or C. Therefore, the standard sequential algorithm as currently implemented in Typed Racket simply tries to solve $A <: B$, and if that fails, tries $A <: C$. Or-parallelism then enters by trying both of these possibilities simultaneously, succeeding if one succeeds. Similarly, overloaded functions in Typed Racket combine multiple signatures; and when performing a subtyping test between them they introduce both and-parallelism and or-parallelism: every possibility in the super type must be the above some function signature in the subtype.

Inference in Typed Racket also has the possibility for and-parallelism. If we wish to constrain (A, B) to be a subtype of (C, D), this implies a pair of constraints, both of which must succeed for a full solution.

6.2 Implementing Typed Racket Inference

The heart of Typed Racket's type checker is `infer`, which takes the names of type-variables to constrain, as well as the relevant types for type-checking a polymorphic function invocation:

`infer` $\mathit{tvars}\ \mathit{actualTys}\ \mathit{formalTys}\ \mathit{resultTy}\ \mathit{expectedTy}$

In order to perform parallelization experiments using LVish, we port the code and the grammar of types to Haskell with one omission of functionality—we omit variable arity functions, which do not occur in our benchmarks. Also, while we keep the structure of the code the same in this conversion, we substitute some data structures with idiomatic Haskell counterparts (replacing lists with sets or maps in places). We refer to this as the "Pure/Seq" version of the program—purely functional, non-monadic, and sequential.

Refactoring for parallelism Next we rewrite the algorithm in monadic style and abstract the core constraint-gen (`cg`) recursion so that it returns a solution stream as in §5, and factors out the corresponding methods for conjunction and disjunction. Thus it is possible to use the same core algorithm with different *evaluation strategies*, including sequential or parallel versions. Each implementation of the type checking solution algebra provides the following functions, which we have given names more appropriate to the task at hand, instead of the generic semiring interface:

- `goodsofar` (one) – result indicating no conflicts observed at this point in the search
- `blowup` (zero) – result indicating a conflict found
- `constrain` (fromAssigns) – add an upper and lower bound constraint on a type variable, apply this to all partial solutions processed
- `orSplit` (add) – test N alternative (`S <: T`$_i$) subtyping constraints, where $i \in [0, N)$
- `andPar` (mul) – join constraints with (optional) parallelism

Then, using the above, the following is a subset of the cases in the heart of the subtype checking algorithm, showing each possible behavior:

```
-- Make s a subtype of t:
case (s,t) of
 (s, t) | (s, t) ∈ seen → goodsofar
 (s, t) | s == t        → goodsofar
 (s, t) | subtype s t   → goodsofar
 (_, Top)               → goodsofar
 (Var x, t) | x ∈ xs → constrain bot x (demote vs t)
 (s, Rec _ _) → cg s (unfold t)
 -- N-way or-parallelism:
 (s, Union ts) → orSplit cg s (elems ts)
 -- and-parallelism:
 (Pair a b, Pair a' b') → andPar (cg a a') (cg b b')
 ...
 _ → blowup  -- s cannot be a subtype of t
```

Solution strategies We implement this parallel algebra of solution streams while leaving knobs to toggle and-parallelism and or-parallelism independently at compile time. Or-parallelism becomes a standard parallel `forEach` from the LVish library:

```
orSplit :: (a → a → Solution) → a → [a] → Solution
orSplit doConstraints s ts = λ k varmap →
  parForEach ts (λt → doConstraints s t k varmap)
```

Note that the `parForEach` is an asynchronous operation–it forks work and returns immediately, without any join.

6.3 Typed Racket Evaluation

For our implementation of the core of the Typed Racket inference algorithm, our evaluation focuses on two different demanding inference problems. First, we consider the case mentioned in the introduction—a small function with slow inference. Second, we consider checking large constant data against a small type. These are both representative problems that have been identified as the most serious performance problems for Typed Racket.

"Bigcall": Higher order functions over extremely polymorphic inputs Typed Racket supports both polymorphism and overloading, and when combined, these can produce computationally-intensive inference problems. The most significant of these is the following[8], originally designed as an example of Typed Racket's *variable-arity polymorphism* [35].

```
(: map-with-funcs
   (All (b a ...)
     ((a ... -> b) * -> (a ... -> (Listof b)))))
(define (map-with-funcs . fs)
  (lambda as
    (map (lambda ([f : (a ... -> b)]) (apply f as))
         fs)))
((map-with-funcs + - * /) 1 2 3 4 5)
```

This function consumes a variable number of functions, bound to the list `fs` and then a variable number of arguments, bound to the list `as`. It then applies each function `f` from the list to *all* of the `as`. We then apply `map-with-funcs` to a few arithmetic functions, and apply the result to numbers. The result is a list containing the sum, difference, product, and division of all five numbers (Racket's numeric operations all support arbitrarily many arguments).

The sequence of arguments `fs` is described in Typed Racket using variable-arity polymorphism. Since we omit this portion of the algorithm, we instead consider versions of `map-with-funcs` that consume 1, 2, 3, or 4 arguments: i.e., `(map-with-funcs + - *)` is the three argument case. We name these $bigcall(1)$ through $bigcall(4)$ for brevity.

Solving this inference problem requires handling several type variables, each of which is jointly constrained by all the arguments, but more importantly, Typed Racket provides very large overloaded types to give precise specifications to numeric operations such as `(+)`, which has *hundreds* of possible types [34]. Since the type of each arithmetic operator is an intersection, any choice of a single overload for one can be combined with any choice of an overload for another input, resulting in a combinatorial explosion of possibilities. Thus type checking all of bigcall(4) takes more than 30 minutes to complete in Typed Racket. (But in this section we focus on inference for an individual call to `map-with-funcs`.)

"Treecall": Dealing with large constant data The second challenge we consider is that of large constant data. Typed Racket supports flexible and precise types for structured data in S-expression format. If a large constant is present in a program and no extra annotation is provided, it will therefore infer the *most precise type*, which can be the same size as the data itself. When a polymorphic function such as `map` is applied to this data structure, inference must process this large type.

To simulate this in a controlled fashion, we designed a benchmark which is the equivalent of applying the following function to progressively larger inputs of trees of symbols and strings.

```
(define-type (Tree A) (Rec X (U (Leaf A) (Pair X X))))
(: left : (All (A) (Tree A) -> A))
```

[8] Taken from Typed Racket's online tests at `http://git.io/vTwBd`

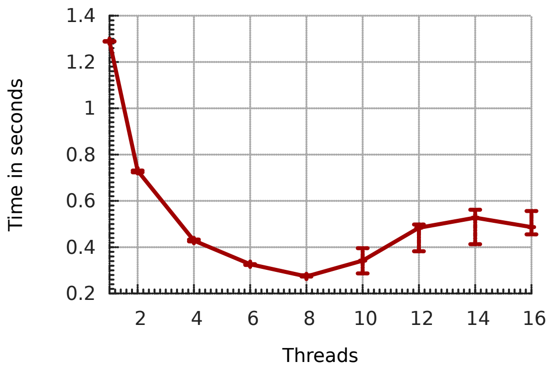


Figure 4. Treecall(16), ParAnd case: here scaling stops at eight cores. We plot time rather than parallel speedup and include minimum and maximum times across all trials in the error bars. In this way, we can see how runtimes become chaotic after scaling stops.

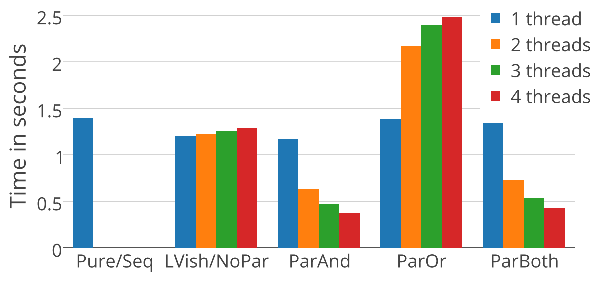


Figure 5. Treecall(16): Large-constant benchmark of size 2^{16}, desktop platform. Here or-parallelism is not useful, because in each binary-Or, one branch is always an immediate dead-end.

```
(define (left t) (if (pair? t) (left (car t)) (leaf-val t)))
```

Hence we call this "treecall", because it calls `left` on a constant tree datum. Treecall(N) corresponds to applying `left` to a tree of depth N, i.e. with 2^N leaves. In a language with a rich macro system like Racket's, large compile-time data is a reality, and is currently a Typed Racket performance problem.

6.3.1 Benchmark Results

We evaluate treecall and bigcall on one desktop-class and one server-class system, with one Intel Xeon i5-3470, and two Xeon E5-2670 CPUs, respectively (4 and 16 cores). All experiments were run using GHC 7.8.3 and all data points are the median of 5 trials. We distinguish two different kinds of run, where we generate either *all* substitutions, or just the first, which we explain further below.

Treecall results

Figures 4 and 5 show treecall(16)—the result of applying `left` to a constant of size 2^{16} on the desktop and server platforms, respectively. In this benchmark there is only one solution, so "all vs. first" solution is immaterial. Because of the union in the list type, `(Rec X (U (Leaf A) (Pair X X)))`, there are many *apparent* opportunities for or-parallelism in this benchmark. However, they are *unprofitable* opportunities, because only one of the two branches will succeed, and the other will fail quickly. The trick here is to not get derailed by or-parallelism, so that the actual and-parallelism present (across the large tree) can be exploited.

To start with, the original Typed Racket inference algorithm takes approximately 10× as long as the sequential Haskell implementation on treecall, due to differences in optimization and data structure choices. We evaluate several Haskell implementation variants. As described in §6, our idiomatic Haskell port is sequential, and structurally close to the original, whereas our parallel version is parameterized so as to take either a parallel or sequential implementation of conjunction and disjunction.

- Pure/Seq – original Racket-to-Haskell port.
- LVish/NoPar – refactored to use the LVish Par monad, and to use generators to represent solution streams. Still sequential.
- ParAnd – turn on parallel And.
- ParOr – turn on parallel Or.
- ParBoth – turn on parallel And and parallel Or.

Treecall provides unprofitable or-parallelism with very poor granularity. For this reason the `ParOr` variant of the benchmark not only fails to speed up, but gets a parallel *slowdown* due to useless tasks and threads bouncing between cores. Not only does the ParAnd variant work well, but ParBoth is fine as well. ParAnd can cure the problem with or-parallelism in this case, because the `andPar` function is essentially the "outer loop", and it is this level of tasks that are stolen most in the underlying work-stealing implementation of the `Par` monad.

In summary, Treecall(16) takes 1.29s in the pure (non-LVish) Haskell version on the server platform. Then, when switching to LVish, it gets up to a 7.68× parallel speedup on 8 threads on the server, and 3.17× speedup on 4 threads on the desktop platform.

Bigcall results

Next, we evaluate bigcall(3) and bigcall(4): three-argument and four-argument variants of `map-with-funcs`. The numbers we report here for LVish compute *all solutions*. For bigcall there are a modest number of solutions (dozens) for very large search spaces (millions). We also timed an LVish version that cancels threads after reporting the first solution, which is faster, but is not strictly deterministic. Thus we focus on the predictable all-solutions version here, which also corresponds to the case where the infer fails and the whole search space must be traversed.

There are two distinct opportunities in this benchmark:

- Deforestation: the Pure/Seq version uses lists to represent streams, and exploring the search space requires a lot of list bookkeeping.
- Or-parallelism: because of the combinatorial explosion of different possible types for `(+)`, `(*)` etc., there should be plenty of parallel work in exploring this search space in parallel.

Indeed, the LVish version of the program achieves a big speedup in both these categories. For example, the Pure/Seq Haskell implementation takes 2.36 seconds to compute all solutions for bigcall(3) (server platform). Just by deforesting intermediate lists using generators, LVish achieves a 9.37× speedup over this baseline. And, even though the remaining time is only a short 0.25s, LVish ParOr and ParBoth variants still achieve greater than 7× parallel speedup, resulting in a total speedups of 70× or higher over the original, idiomatic Haskell version ported from Racket. The speedup compared to original Racket version would be even greater, because the Racket version takes slightly longer, 3.0s, to compute the first solution to bigcall(3).

Note that in bigcall, computing all solutions is wasteful, with the purely functional Haskell version (Pure/Seq) taking only 0.18s as opposed to 2.36s on bigcall(3) to compute the first rather than all

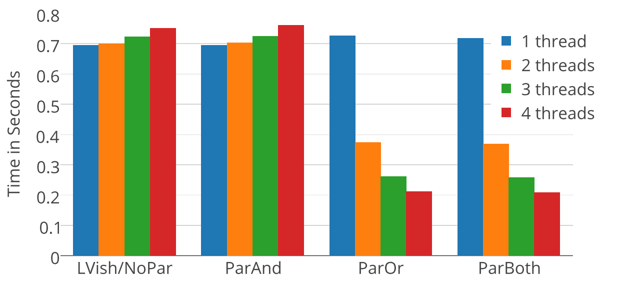


Figure 6. Bigcall(4): Highly polymorphic inputs benchmark on the desktop platform; shows time, in seconds, to type check `(map-with-funcs + - * /)`, computing all solutions. Note that the time for Pure/Seq is too much larger than the LVish version to show here (7.93s for even the first solution).

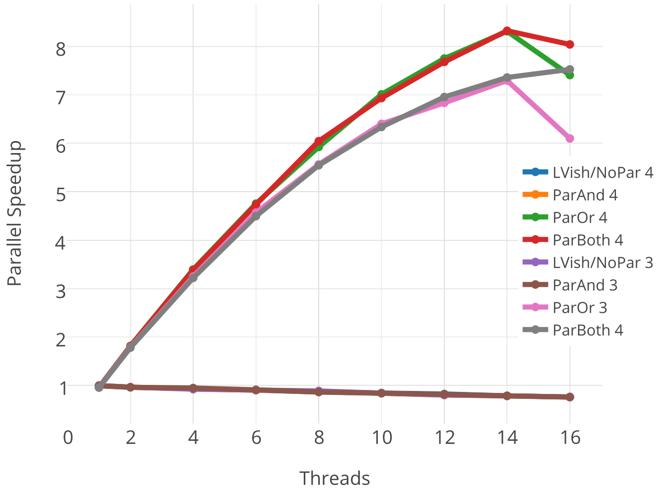


Figure 7. Bigcall(3) and bigcall(4): Parallel speedup on server platform, three- and four-argument version. Even bigcall(3) is time-consuming enough to get a good parallel speedup. This graph also demonstrates the "last core slowdown" which is still a problem with some parallel programs on GHC 7.8.3.

answers on the server platform. However, the deforestation benefit from switching to LVish makes up for the difference.

When scaling to the four-argument version, bigcall(4), LVish is much faster than the purely functional versions. Union types in Typed Racket programs contain or-parallelism opportunities with a very low *survival rate*—often only one of the variants in a union matches. For this larger case, the parallel speedups go to 8.46× (16-core server, using 14 cores) and 3.43× (4-core desktop), and total speedup of LVish over Pure/Seq approaches 80×.

Note that the "first found" version we mentioned earlier is faster still, but it requires that we admit "don't care" nondeterminism, whereas our goal here was strict, statically-enforced deterministic parallelism with LVish. In future work, we plan to study the issue of extracting a *deterministic* result, in parallel.

7. Related Work

The *monad-par* system predated LVish and provided IVars as the sole synchronization construct. Marlow et al. [27] presents parallel analysis of interdependent definitions as a motivating example, but no implementation was evaluated. Later, Marlow developed this into an example in his parallel programming tutorial (e.g. at CEFP'12 [26]). This example shows the principle of performing analysis of interdependent (but non-recursive, acyclic) definitions by representing the inter-definition dependence graph with IVars. Definitions only communicate their types after generalization in this example—there is no attempt to parallelize during the unification process. (Besides, top-level definitions in languages like Haskell and Typed Racket allow mutual recursion and so do not provide an acyclic dependence graph.)

Work on parallel Prolog [10, 17] addresses issues of and- and or-parallelism. Prolog's control model is significantly different from LVish, however, due to the presence of nondeterminism and the `cut` predicate. As a result, the data structures used are specific to logic programming, where ours are more general.

Saraswat and Rinard [31] discuss `cc(↓,→)`, a language for concurrent constraint logic programming. Their system also includes a notion of blocking reads analogous to the threshold reads of LVars, and additionally requires that concurrently written constraints be consistent with one another. It has nondeterministic operations, but also does identify a subset of operations that retain determinism. Modern concurrent constraint programming systems are available in software packages like Gecode [14]—a C++ library. The builtin constraint types and search strategies would not apply to, e.g., Typed Racket typechecking, lacking Herbrand constraints. But one could use such a system as an alternate starting point for this research: writing new C++ code to extend the system with new models, propagators, and branchers. However, the verification or testing burden to ensure this C++ code retains determinism is a much more difficult obligation than with LVish. In LVish, we must ensure *only* that a `TyVar` LVar's join function has the appropriate properties (commutative, associative, idempotence, and absorption). But join is just a pure function that can be tested for these properties with QuickCheck or other methods.

Deterministic Search Algorithms In [20], the authors give a deterministic parallel algorithm for backtracking search problems. COMMON-CRCW, their computational model, allows for arbitrary concurrent reads, and restricts concurrent writes by requiring that all threads write the same value. IBM's CPLEX system [11] offers a parallel solver for integer linear programs, with some extensions. Their solver is deterministic, except in the case where the user provides *control callbacks*, which allow observation and modification of the state of the parallel search.

8. Conclusion

Type checking in modern type systems is an expensive process, but not one that has previously been parallelized. We saw how the LVar framework is one possible way to address this challenge while also ensuring determinism in addition to gaining parallelism. We showed substantial parallel scaling and improvement in wall-clock time on two very different type systems: one very widely used, with 3.57× parallel speedup, and the other slow and sharply in need of parallelization, with up to 4.71× and 8.46× parallel-speedup on our two benchmarks, respectively.

Acknowledgements

This work was supported by NSF CISE awards CCF-1218375 and CCF-14535085. We also thank Vijay Saraswat and Tom Schrijvers, for their advice on this topic, which they have generously provided in person and in correspondence.

References

[1] . Ceylon language website. `http://ceylon-lang.org/`, .

[2] . Erlang dializer type checker. `http://www.erlang.org/doc/man/dialyzer.html`, .

[3] . Facebook flow project website. `https://code.facebook.com/projects/1524880081090726/`, .

[4] . Typescript project website. `http://www.typescriptlang.org/`, .

[5] L. Allison. Continuations implement generators and streams. *Comp. J*, 33(5):460–465, 1990. doi: 10.1093/comjnl/33.5.460.

[6] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *TOPLAS*, 11:598–632, 1989.

[7] R. L. Bocchino Jr, V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. In *OOPSLA*, pages 97–116, 2009.

[8] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *JFP*, 23(05):552–593, 2013.

[9] K. Claessen, M. Sheeran, and B. J. Svensson. Expressive array constructs in an embedded GPU kernel programming language. In *DAMP*, pages 21–30, 2012.

[10] V. S. Costa, D. H. D. Warren, and R. Yang. *Andorra I: A Parallel Prolog System That Transparently Exploits Both And-and Or-parallelism*, volume 26 of *PPOPP '91*. ACM, 1991.

[11] I. I. CPLEX. V12. 1: User's Manual for CPLEX. *International Business Machines Corporation*, 46(53):157, 2009.

[12] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212, 1982.

[13] M. Flatt and PLT. Reference: Racket. Technical report, PLT Design, Inc., 2010. `http://racket-lang.org/tr1/`.

[14] Gecode Team. Gecode: Generic constraint development environment, 2006. Available from `http://www.gecode.org`.

[15] I. P. Gent, C. Jefferson, I. Miguel, N. Moore, P. Nightingale, P. Prosser, and C. Unsworth. A preliminary review of literature on parallel constraint solving. In *PMCS*. Citeseer, 2011.

[16] R. E. Griswold and M. T. Griswold. History of Programming languages—II. chapter History of the Icon Programming Language, pages 599–624. ACM, New York, NY, USA, 1996.

[17] G. Gupta and V. S. Costa. And-or parallelism in full Prolog with paged Binding Arrays. In *PARLE'92*, pages 617–632. 1992.

[18] Y. Hamadi, S. Jabbour, and L. Sais. Manysat: A Parallel SAT Solver. *JSAT*, 6(4):245–262, 2009.

[19] Y. Hamadi, S. Jabbour, and J. Sais. Control-based clause sharing in parallel SAT solving. In *Autonomous Search*, pages 245–267. 2012.

[20] K. T. Herley, A. Pietracaprina, and G. Pucci. Deterministic parallel backtrack search. *TCS*, 270(1):309–324, 2002.

[21] R. Jhala. Refinement types for Haskell. In *PLPV*, pages 27–28, 2014.

[22] L. Kuper and R. R. Newton. LVars: Lattice-based data structures for deterministic parallelism. In *FHPC*, pages 71–84. ACM, 2013.

[23] L. Kuper, A. Turon, N. R. Krishnaswami, and R. R. Newton. Freeze after writing: Quasi-deterministic parallel programming with LVars. In *POPL*, pages 257–270, 2014.

[24] J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. In *PLDI*, pages 24–35, 1994.

[25] B. Liskov. A History of CLU. *SIGPLAN Not.*, 28(3):133–147, Mar. 1993. ISSN 0362-1340.

[26] S. Marlow. Parallel and concurrent programming in Haskell. In *Central European Functional Programming School*, pages 339–401. 2012.

[27] S. Marlow, R. Newton, and S. Peyton Jones. A monad for deterministic parallelism. In *Haskell '11*, pages 71–82. ACM, 2011.

[28] S. Peyton Jones, A. Reid, F. Henderson, T. Hoare, and S. Marlow. A semantics for imprecise exceptions. In *PLDI*, pages 25–36, 1999.

[29] B. C. Pierce and D. N. Turner. Local Type Inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

[30] T. Prabhu, S. Ramalingam, M. Might, and M. Hall. EigenCFA: Accelerating Flow Analysis with GPUs. POPL '11, pages 511–522, 2011.

[31] V. A. Saraswat and M. Rinard. Concurrent constraint programming. In *POPL*, pages 232–245. ACM, 1989.

[32] M. Shaw, W. A. Wulf, and R. L. London. Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators. *Commun. ACM*, 20(8):553–564, Aug. 1977.

[33] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and FP*, pages 81–92, 2006.

[34] V. St-Amour, S. Tobin-Hochstadt, M. Flatt, and M. Felleisen. Typing the numeric tower. In *PADL*, pages 289–303, 2012.

[35] T. S. Strickland, S. Tobin-Hochstadt, , and M. Felleisen. Practical variable-arity polymorphism. In *ESOP*, pages 32–46, 2009.

[36] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: From scripts to programs. In *OOPSLA*, pages 964–974, 2006.

[37] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *POPL*, pages 395–406, 2008.

[38] S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *ICFP*, pages 117–128, 2010.

[39] P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc (FD). *J. Logic Prog.*, 37(1-3):139–164, 1998.

[40] A. Zobel. Program structure as basis for parallelizing global register allocation. In *Computer Languages*, pages 262–271, Apr 1992. doi: 10.1109/ICCL.1992.185490.