Parallel Type-checking with Saturating LVars

Peter Fogg   Ryan R. Newton   Sam Tobin-Hochstadt
Indiana University
{pfogg, rrnewton, samth}@indiana.edu

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 a solution to these problems based on the LVish approach to deterministic-by-default parallel programming. We extend LVish with a novel class of concurrent data structures: *Saturating LVars*, which are the first LVars to safely release memory during the object's lifetime. Our new design allows us to achieve a parallel speedup on both traditional Hindley-Milner inference, and on the *Typed-Racket type-checking algorithm*, which yields up to an 8.46× parallel speedup on type-checking examples drawn from the Racket repository.

1. Introduction

Recent programming language advances often rely on sophisticated type systems (Bocchino Jr et al. 2009; Tobin-Hochstadt and Felleisen 2010; Jhala 2014; Siek and Taha 2006b; Brady 2013), 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 (Jhala 2014). In other cases—occurrence typing, dependent typing, and gradual typing (Tobin-Hochstadt and Felleisen 2008)—using an external solver is infeasible. Gradually-typed languages, for example, may employ uncommonly expressive type systems to capture idioms from dynamically typed programming. For example, Typed Racket combines subtyping with flexible union types, and the `(+)` operation ends up with a type that combines several hundred distinct function signatures. As a consequence, the Typed Racket repository[1], contains individual files which take over five minutes to typecheck!

In the era of ubiquitous parallel hardware, it would seem important 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 (Zobel 1992) and flow analyses (Prabhu et al. 2011). Further, to the best of our knowledge, no prior work has parallelized type-checking algorithms specifically.

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 the Haskell-based implementation inside GHC), mutation is often used for constraining type variables, complicating parallelization further. Our goal is to parallelize despite these constraints, using linguistic abstraction to enforce disciplined, monotonic use of mutable state and minimize or eliminate unintended nondeterminism.

In this work we perform two experiments in parallel typechecking, and we use Haskell as our implementation language for writing parallel checkers. Unfortunately, the mechanisms used to mutate type variables in sequential type checkers (State and ST monads), do not generalize safely to the parallel case. Moreover, as described in §5.1, using immutable data structures with purely functional task parallelism (futures) is not a good fit for this problem.

LVars for Type Variables Fortunately, there are a class of *synchronization variables* that are safe to share between computations in a functional language. Single assignment variables, or *IVars* (Arvind et al. 1989), are one early example in this class, supported in Haskell via the *Par monad* (Marlow et al. 2011). More recently, Kuper and Newton (2013) introduced a generalization that enables deterministic, functional programs to synchronize on arbitrary *monotonic data structures*, called *LVars*. LVars 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

[1] `http://github.com/plt/racket`

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

monotonically. But application-specific LVars can be constructed as well; in particular, LVars 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 cannot be caught except in the `IO` monad.
2. While LVars are a good fit for *And-parallelism*—where threads join information concurrently—they do not help with the *Or-parallelism* found in some type systems, where speculative, alternative additions of information must be considered.

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

- We introduce Saturating LVars (§4), adding the capability for both trapped-failure, and memory reclamation—addressing a major limitation of previous LVar designs. We use Saturating LVars in the process of speeding up an implementation of Hindley-Milner type inference.
- We develop a modular formulation of Or-parallel constraint systems, parameterized by an algebra of substitution streams. We provide an efficient implementation of this algebra using *generators*. We explain this system in the simplified context of satisfiability problems §5.
- We then scale this architecture to the type system of a full blown language (Typed Racket, §6) with a modestly widespread user base, achieving both (sequential) speedups due to deforestation and parallel speedups.

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 (hereinafter *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 the argument to the `put`.

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 set of the lattice's elements, known as the *threshold set*. This set, semantically an implicit argument to `get`, allows 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 $\forall a, b \in Q$, $a \sqcup b = \top$.

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, with this ordering:

```
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 addition to thresholded `get` operations, changes to an LVar can be observed through *handlers* (callbacks). 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 may only occur after a global barrier to avoid races between `put` and `freeze`.

LVars, in practice In the implementation of the LVish library, parallel computations are exposed through a `Par` monad. These computations have type `Par e s a`, where: `a` is the return value of the monadic `Par` computation; the `s` parameter keeps LVars from being shared between different parallel regions (like the `ST` monad); and the `e` type parameter documents the *effect signature* of the computation. For example, a function from `Int` to `Int` that executes in the LVish monad and also 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`. Both of these `runPar` variants are used to return *pure* Haskell results in of type `a`. If we wish to return LVars, we 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), implemented by each LVar in the library, that "casts" the monotonic/mutable version of the LVar, to a pure/immutable sister type.

A note on notation Effect signatures are important to the LVish library; some effects are only *conditionally* deterministic—canceling read-only futures is fine, but canceling a call to `foo` above would introduce an observable data-race. Effect signatures are not central to the way we use LVars in this paper, so we will abbreviate `Par e s Int` as `Par Int`, and mention effect constraints in the prose only where they are relevant.

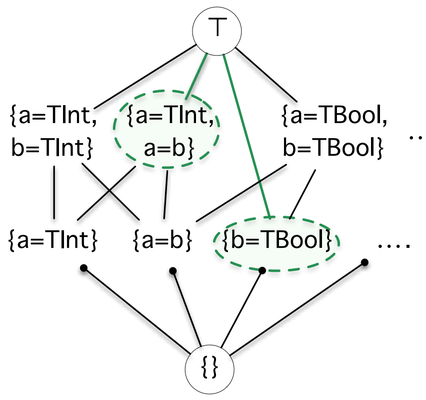


Figure 1. The partial order for the store containing all type variables used in a type-inference execution. The two highlighted nodes are *incompatible*—their lub is ⊤.

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

We now have what we need to parallelize a type checking algorithm. We begin with what is perhaps the most well-known unification-base type inference algorithm: the Hindley-Milner system (Damas and Milner 1982).

Sequential Hindley-Milner The Hindley-Milner algorithm operates by walking over an expression and computing constraints for each value. These constraints are then 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: `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. 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)
```

Where the type variable is represented directly by its pointer to the mutable location. The `STRef` (Launchbury and Peyton Jones 1994) 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 (p1, p2) of
   (TVar v ref, t) → do
     when (not (occurs v t)) (writeSTRef ref (Just t))
   (t, TVar v ref) → do
     when (not (occurs v t)) (writeSTRef ref (Just t))
   (TFun t1 t2, TFun t1' t2') → do unify t1 t1'
                                   unify t2 t2'
   (TInt, TInt) → return ()
   _ → 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
    fun ← infer env e1
    arg ← infer env e2
    res ← freshTVar ()
    unify fun (TFun arg res)
    return res
```

Alas, in exchange for increased performance in the single threaded case, it would appear that `STRefs` have destroyed our opportunity for deterministic parallelism!

Exploiting parallelism Fortunately, we can parallelize the algorithm by switching from `STRef`'s to LVars, and can even asynchronously share type constraints between threads. The collection of type-variable constraints accumulated during type-checking forms a partial order under unification (Figure 1). We can model this state either with a single `Map` LVar or with an LVar per type variable (`TyVar`)—we use the latter approach here.

A `TyVar` is an application-specific LVar that is either empty (unconstrained) or filled with a type, and when additional types, containing type variables, are joined into the `TyVar` via `put`, unification is invoked recursively as a callback. With `TyVars`, the previous definition of `Mono` changes to include `TVar TyVar`. Because `TyVar`'s are simply a mutable pointer to a piece of immutable data, the `TyVar` abstract datatype is easy to implement compared to LVars that require complicated lock-free algorithms.

Much of the type-checking code is fundamentally unchanged, beyond replacing `writeSTRef` with the analogous operation on `TyVars` and switching type signatures for their `Par` version. After that refactoring, we can inject parallelism in the application case of the `infer`, to infer the types of both subexpressions in parallel:

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

Here the `par2` combinator simply executes two actions, returning two values. We can also unify function types in parallel:

```
unify :: Mono → Mono → Par ()
unify (TFun t1 t2) (TFun t1' t2') = do
  fork (unify t1 t1')
  unify t2 t2'
```

Note that unification called for (monotonic) side effects only. This approach yields a speedup when used to implement a micro-ML calculus 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. For real, large programs with Hindley Milner inference, the challenge will be finding problems that take long enough

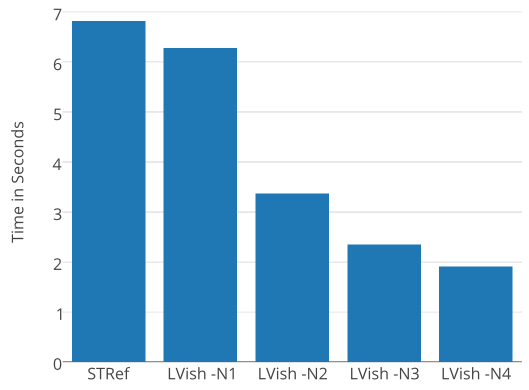


Figure 2. Hindley-Milner type inference on a generated term with roughly a million nodes. The sequential (STRef) version is compared against the LVish version. Note that even on one core, the LVish version is still *concurrent*, sharing constraint information between concurrent computations via LVars.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

in practice (relative to the amount of memory they read) that they are worth parallelizing.

4. Saturating LVars: Trapped Failure

There is one problem with the formulation of Hindley-Milner type inference in the previous section—what if two types do *not* unify? If type-checking fails we would like to simply return `False` or `Nothing`. But with implementation in the previous section this failure instead appears as incompatible puts, e.g. putting `TInt` and `TBool` to an LVar. Further, in LVar-based programs, adding contradictory information to an LVar always triggers the $\top$ state, which in previous implementations of LVish meant throwing a Haskell exception.

What is wrong with throwing an exception? Answering this requires a bit of background. Haskell is a purely functional but *partial* language, whose exception semantics (Peyton Jones et al. 1999) requires that exceptions be handled only in the `IO` monad to retain referential transparency. In this case, it is important that we keep the type checking phase *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`".

In fact, there are significant benefits to strictly deterministic compilers, generally. The Nix package manager and NixOS operating system (Dolstra and Löh 2008) have demonstrated the benefits that accrue from compilation being a mathematical function from bits to bits. Writing a compiler and type-checker in Haskell with no IO (outside of reading and writing files) is one way to achieve this goal.

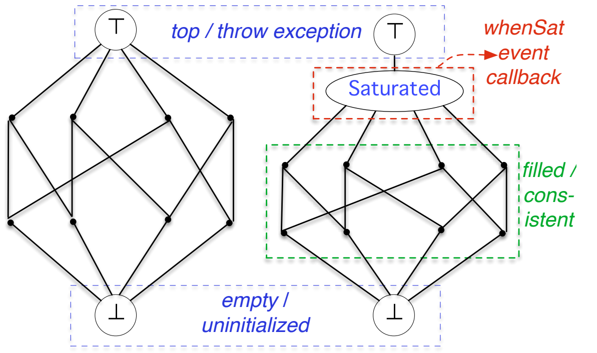


Figure 3. Any valid LVar lattices is turned into a Saturating LVar by adding an extra, penultimate state.

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. Such an LVar is thus given by a five-tuple $(D, \sqsubseteq, \bot, \top, Sat)$, where:

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

An example lattice extended with the Sat state is pictured in Figure 3. While this convention is simple, its ramifications are not:

1. The only usable (incompatible) threshold set for performing blocking read or adding a handler[3] is $\{Sat\}$.
2. A saturating LVar's state moves monotonically, but it does *not* monotonically gain information. Notably an LVar in the saturated state can be represented by as little as one bit. This means that saturating LVars are the first LVars **that can release memory** during their lifetime.
3. Computations whose *only* effect is to write to a Saturating LVar can be *cancelled* if that LVar saturates (§4.1).

Because of the first restriction, saturating LVars become effectively *write only*. Further, all Saturating LVars can provide the following operations:

```
class SatLVar lv where
  saturate   :: lv → Par ()   -- Force it to Sat state
  whenSat    :: lv → Par () → Par ()
  isSat      :: Frzn lv → Bool
```

[3] 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`.

This interface provides the ability to force LVars to saturate, respond to saturation, and test for saturation after a parallel computation is complete.[4]

Of course, every saturating LVar provides specific methods outside of the common interface—methods for constraining a type variable, inserting into a map or set, and so on. But even the common API is enough to consider some use cases.

Example use-cases Consider an application-specific LVar representing constraints upon a single variable. The Sat state would corresponding to conflicting constraints (i.e. *failure*). If we collect those individual LVars into a container, such as a `Map` LVar, it could represent a complete environment. Thus we would expect that the *entire environment fails* when any entry does. Indeed, this is possible with API we described above—before inserting each variable in the `Map`, we attach a `whenSat` handler which in turn calls `saturate` on the entire environment, propagating the failure, as in the following:

```
do env ← newEmptyMap
   ...
   cv ← newConstrainedVar
   whenSat cv (saturate env)
   insert vr cv env
```

In fact, there's more that a library can do to enable efficient compositions of LVars and saturating LVars. For example, 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. 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 LVars can be discarded at runtime, freeing memory and shrinking the set's physical size.

As we will see, we can use `FiltSet` to accumulate results from a search process when dealing with type systems that include disjunction. Or, as a simpler example, consider implementing a program analogous to the following query using a data structure of type `(FiltSet SatCounter)`:

```
SELECT MEDIAN(SIZE) FROM CLASSES WHERE SIZE < 30
```

We can maintain a set counters of that are updated with fetch-and-add, and are set to saturate upon hitting a tally of 30. Over-threshold counters would automatically be removed from the set, leaving only those which are under the threshold at the end of the `runParThenFreeze` call.

[4] The LVar can only be tested for saturation with `isSat` *after* a parallel region has ended and it is in a "frozen" state. Freezing LVars is covered in detail in (Kuper et al. 2014b).

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.

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 as memory as it goes.

In the Hindley-Milner type-checking case, we use saturating LVars for each individual `TyVar`. Saturation does not come into play in typing *well-typed* terms as in Figure 2. But when discovering that a term is ill-typed, saturation (and as we will see below, *cancellation*) are relevant. Further, in other type systems, 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 Figures 1 and 2 show their performance relative to more basic LVar counterpart data structures.

4.1 Saturation and Cancellation: a safe idiom

Above we saw how to handle a saturation event by attaching an action to the `whenSat` handler. But we can go further; because saturation often means that the rest of the computation is useless, it is valuable to be able to *cancel* remaining computations in response to saturation.

Cancellation is a feature supported by LVish and explored in previous work (Kuper et al. 2014a). In that work, however, cancellation was only safe for LVish computations with `get` but never any kind of `put` or `freeze` (or `saturate`). That restriction rules out canceling upon saturation, because to have saturation events in the first place, we must perform writes to an LVar. The insight here is that, based on the lattice structure of saturating LVars (Figure 3), after saturation, all *further* writes have no observable effect. Thus:

Conjecture 1 (Safe-Cancellation)**.** *Every LVish program with store S, such that all writable LVars l, $[l \rightarrow Sat] \in S$, is observationally equivalent to the program* `return` *() or $\bot$ (divergence).*

If we are amenable to converting some nonterminating outcomes to terminating ones[5], then it becomes safe to cancel these computations. We restrict to the case of writing to a single saturating LVar for simplicity, but how do we enforce that a given subcomputation can *only* write that LVar, while reading from arbitrarily many other LVars? To accomplish this, we can use LVish's effect-tracking capabilities to formulate a safe `withSatLVar` idiom, which runs a block of code in `ReadOnly` mode, but with an "escape hatch" for modifying *only* the a designated saturating LVar:

```
withSatLVar lv (λ modIt →
  do (k,v) ← ...readonly...
     modIt (insert k v) -- Send out a write to lv
     ...readonly...)
```

In this example, `lv` is a `SatMap`, and the `modIt` function allows executing code in a separate effect environment which allows writes, but which can operate *only* on `lv` and no other LVar. Like most programs with tracked effects, the type of `withSatLVar` is more complicated than the code that uses it. The type `withSatLVar` makes heavy use of the "`e s`" parameters which we have been eliding when writing `Par a` rather than `Par e s a`. Here we show that full type:

```
withSatLVar :: (SatLVar lv, ReadOnly e1, Det e0)
            ⇒ lv s1
            → ((∀ s0. lv s0 → Par e0 s0 ())
               → Par e1 s1 a)
            → Par e2 s1 (Maybe a)
```

Note that because of the cancellation possibility, a `Maybe` value is returned which may be `Nothing` if `lv` saturates and the computation is cancelled. In the case of Hindley-Milner type inference, `Nothing` corresponds to no valid unification returned from the type checking algorithm.

5. *Or-Parallelism*: Satisfiability solvers

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 use LVars as we did in the previous section (mapping each type variable to exactly one LVar and updating it destructively). Nevertheless, Or-parallelism is an core feature of Typed Racket, the type system that is our main target in this paper (§6). Yet rather than dive directly into Typed Racket in this section, we first introduce the implementation techniques using a smaller example: satisfiability (SAT).

[5] This respects the monotonicity principle in Haskell's denotational semantics, and has precedent in the fact that GHC will turn some provably infinite loops into exceptions.

```
generic :: ∀ a b . Eq a ⇒
           Ringlike (Vector (Var,a)) b → Tree a → b
generic Ringlike{mkNum,mkZer,add,mul} tr0 =
 case tr0 of Left  at → loop1 at
             Right ot → loop2 ot
 where
  loop1 :: AndTree a → b
  loop1 (Leaves vec) = mkNum vec
  loop1 (And vec) | null vec  = mkNum empty
                  | otherwise = mul (map loop2 vec)
  loop2 :: OrTree a → b
  loop2 (Or vec) | null vec  = mkZer
                 | otherwise = add (map loop1 vec)
```

Figure 4. The generic solution for satisfiability, parameterized by a `Ringlike` object.

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 (Costa et al. 1991; Gupta and Costa 1992) and parallel constraint solvers (Gent et al. 2011; Saraswat and Rinard 1989; Van Hentenryck et al. 1998). 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 (Hamadi et al. 2009, 2012).

Some type systems explicitly *parameterize* over the underlying constraint domain: for example, the HM(X) (Odersky et al. 1999) and OUTSIDEIN(X) (Vytiniotis et al. 2011) inference algorithms. Further, recent work in the GHC compiler has attempted to make this approach an implementation strategy as well as a formal reasoning tool.

Likewise, in the remainder of this paper, we abstract over the constraint domain. Thus, our goal in this section is not to implement an efficient SAT solver, but rather to study data-structure and parallel control-flow trade-offs when exhaustively searching a space of possibilities using techniques that apply to *any* constraint domain that can be formulated as an LVar. In the case of satisfiability, we have a constraint domain that consists of simple 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)
```

Represented in our Haskell implementation by the following data structure, which enforces a normal form where `And` and `Or` alternate, but both are N-ary rather than binary:

```
type Tree a = Either (AndTree a) (OrTree a)
```

```
data AndTree a = And (Vector (OrTree a))
               | Leaves (Vector (Var,a))
data OrTree  a = Or (Vector (AndTree a))
```

A simple, compositional solution must represent a *streams of substitutions*, each binding variables within the (sub)term. These solution streams can be combined by concatenation (`Or`) or by joining pairs of solutions drawn from the cartesian product of two streams (`And`). In fact, the streams form a ring-like structure, and we can formulate a simple generic solution abstracted over a set of methods matching the following signature:

```
data Ringlike leaf -- "leaves" of our computation
              elem -- elements of our ring
    = Ringlike
    { mkNum :: leaf → elem
    , mkZer :: elem
    , add :: Vector elem → elem
    , mul :: Vector elem → elem
    }
```

This provides a simple algebra for solution streams. The generic code that walks the `Tree` and calls the `Ringlike` methods is listed in Figure 4, and is used by all the implementations we discuss below.

5.1 The Simplest Stream Algebra

In a sequential Haskell implementation, the natural representation of solution streams is as a lazy list of variable assignments (`Map`s), parameterized over the type of values variables range over, "a":

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

And the algebra over these streams is implemented by:

```
listStrms :: Eq a ⇒ Ringlike (Vector (Var,a)) (Sol1 a)
listStrms = Ringlike
 { mkNum = λvec → maybeToList (foldlM insert empty vec)
 , mkZer = []
 , mul = foldl1 (λs1 s2 →
              catMaybes [ joinEnvs env1 env2
                        | env1 ← s1, env2 ← s2 ])
 , add = foldl' (++) []
 }
```

Here `Vector (Var,a)` is a block of variable assignments, and `insert` is a function that gives the block an interpretation in terms of `Sol1`s (inserting the variables into the `Map`, and failing if there is a conflict using the `Maybe` monad). 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 genuine parallel work is too entangled with book-keeping on lazy lists.

5.2 A Parallel Stream Algebra with Generators

Unfortunately, as we will see in §6, 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 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 `[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]. We can, for example, define our answer type to be a computation in the LVish `Par` monad, which gives us the following solution type for satisfiability problems:

```
type Cont a = Env a → Par ()
type Sol3 a = Cont a → Cont a

contBased :: (...) ⇒Ringlike (Vector (Var,a)) (Sol3 a)
contBased = Ringlike { ... }
```

A solution stream—the element type of our `Ringlike`—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`.

Further, note that each computation, `Par ()`, does *not* return a value. When using this formulation we extract a result by attaching a final continuation that inserts into an output `Set` or `FiltSet` LVar.

With this definition for `Sol3`, we can see that the *zero* for the algebra is `(λk w →return ())`, which drops the partial assignment `w` on the floor, *not* calling the continuation. Likewise the *one* value is `(λk w →k 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. Here we show the binary version of the operator, but the N-ary version in our implementation is a straightforward generalization which exposes the parallelism as a parallel loop rather than a single `fork`.) Further, we thus far assumed immutable environments, such that `w` does not need to be *copied* before being sent to different continuations. We will change that in a moment.

[6] First appearing in the language Alphard in the mid 1970s (Shaw et al. 1977), and later in CLU (Liskov 1993) and Icon (Griswold and Griswold 1996). A clear explanation of generators can be found in (Allison 1990).

[7] One example in the literature is the related concept of *push arrays* (Claessen et al. 2012) used in data-parallel programming.

AndPar: first technique First, 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)
```

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. Likewise `mkNum`, which handles the `Leaves` of the `AndTree`:

```
mkNum vec = λ k w → k (foldl constrain w vec)
```

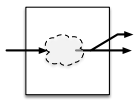
With the immutable definition of `Env` as `Map Var a`, we must fold the constraints into the environment *before* sending it along to the continuation. It is not possible to extract parallelism here at the level of the `Par` monad. (As in the previous section, it would be possible—but not profitable—to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

If we use LVars to represent the 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 that they receive. For example, `(k w)` below:

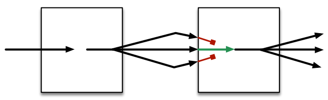
```
mkNum vec = λ k w → do mapM_ (constrain_ w) vec; k w
```

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

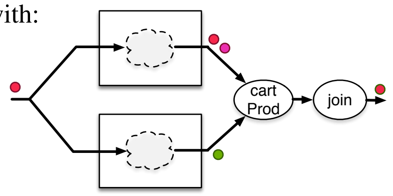
AndPar: second technique There is another option for parallelizing and, but it may result in repeatedly joining constraints that are already "settled". We introduce it visually first and then in code. Let us visualize each continuation transformer as a processing stage, pictured as a box. This is a *push-driven* 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, performing both *filtration* and *amplification* of the stream. A regular conjunction of disjunctions is accomplished by *chaining* these generators sequentially. Indeed, this is our sequential version of `mul`:



The green arrow 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 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 this as:

```
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 on 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 join 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 technique is quite effective in typing some Typed Racket programs.

6. Typed Racket Type Checking

Typed Racket (Tobin-Hochstadt and Felleisen 2008) is a typed version of Racket (Flatt and PLT 2010) that uses gradual typing (Siek and Taha 2006a; Tobin-Hochstadt and Felleisen 2006) 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 (St-Amour et al. 2012). 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 so-called *local type inference* (Pierce and Turner 2000) along with bidirectional type checking.

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`.

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 (St-Amour et al. 2012) and when inferring a type for large blocks of constant data. As a result of these and other issues, Typed Racket is known to have slow typechecking. In particular, some pathological cases can have typechecking times measured in minutes. As mentioned in the introduction, one case which we use as a benchmark was removed from the Typed Racket test suite since it took too long to run.

6.1 The core algorithm

The core of the inference algorithm is an extended form of the Pierce and Turner (2000) 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 algorithms 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, suceeding if one succeeds. For function overloading, which is modeled as intersection in Typed Racket, we simply consider the dual problem, with similar implications for parallelism.

Inference in Typed Racket also has the possibility for and-parallelism. If we wish to constraint (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

We first performed a direct port of the `infer` implementation in the Typed Racket source code. This function takes the names of type-variables to constrain, as well as the relevant types for type-checking a polymorphic function invocation:

`infer` *tvars actualTys formalTys resultTy expectedTy*

In order to perform parallelization experiments using LVish, we port the code and the grammar of types to Haskell modulo two changes: (1) we omit variable arity functions, and (2) we substitute Haskell datatypes for Racket ones.

Refactoring for parallelism Next we rewrite the algorithm in monadic style and abstract the core constraint-gen (`cg`) recursion so that it returns an element of our *solution stream algebra* (§5). The following is a subset of the cases in the heart of that algorithm, showing each possible behavior:

```
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)

  (Pair a b, Pair a' b') → andPar (cg a a') (cg b b')
  ...
  _ → blowup
```

Each implementation of the solution algebra provides the following functions, which we have given names more appropriate to the task at hand:

- `goodsofar` (one) – result indicating no conflicts observed at this point in the search
- `blowup` (zero) – result indicating a conflict found
- `orSplit` (add) – test N alternative (`S <: T`) subtyping constraints
- `constrain` (mkNum) – add an upper and lower bound on a type variable to all solutions
- `andPar` (mul) – join constraints with (optional) parallelism
- `extract` – run the computation to produce a valid type variable assignments

Solution strategies We implement this parallel algebra of solution streams while leaving knobs to toggle and-parallelism and or-parallelism independently at compile time. The `andPar` function precisely resembles the code for `mul` in §5.2. Or-parallelism becomes a standard parallel `forEach` from the LVish library:

```
orSplit :: (a → a → Solution) → a → [a] → Solution
orSplit msg 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 but can return immediately.

Testing While we do not run directly on Typed Racket source programs, we *traced* calls to Typed Racket subtype checking procedure, generating a log of over 50,000 test cases that we validate our Haskell implementations (sequential and parallel) against. Further, because we know the typing constraints should form a partial order, we randomly generate types with the QuickCheck library and test lattice properties—e.g., that everything is above bottom, or the lub of two constraints is above each of them.

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 that takes minutes to check. Second, we consider checking large constant data against a small type.

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], designed as an example of Typed Racket's *variable-arity polymorphism* (Strickland et al. 2009).

```
(: map-with-funcs
   (All (b a ...)
     ((a ... -> b) * -> (a ... -> (Listof b)))))
(define (map-with-funcs . fs)
  (lambda as
    (map (lambda: ([f : (a ... 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.

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 +. (St-Amour et al. 2012). 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. Type checking this program takes many minutes to complete.

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

[8] Taken from Typed Racket's online tests at `http://git.io/ve4PJw`

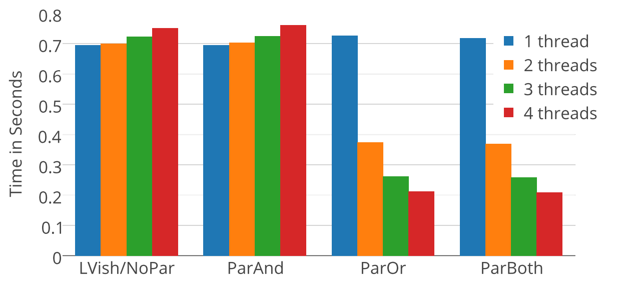


Table 3. Highly polymorphic inputs benchmark. Time, in seconds, to check `(map-with-funcs + - * /)`.

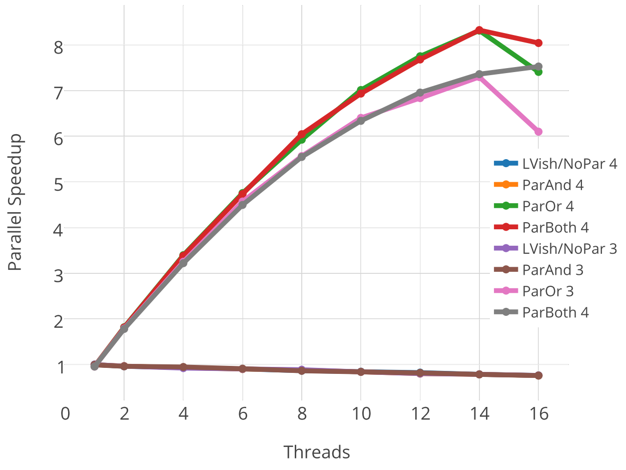


Figure 5. Parallel speedup on server platform, three- and four-argument version.

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 consisting of trees of symbols and strings:

```
(define-type (Tree A) (Rec X (U (Leaf A) (Pair X X))))
(: leftmost : (All (A) (Tree A) -> A))
(define (leftmost t)
  (if (pair? t) (leftmost (car t)) (leaf-val t)))
```

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.

Benchmark Results We evaluate on one desktop-class and one server-class system, with one Intel Xeon i5-3470, and two Xeon E5-2670 CPUs, respectively. We distinguish two different kinds of run, where we generate either *all* or substitutions, or just the first, which we explain further below.

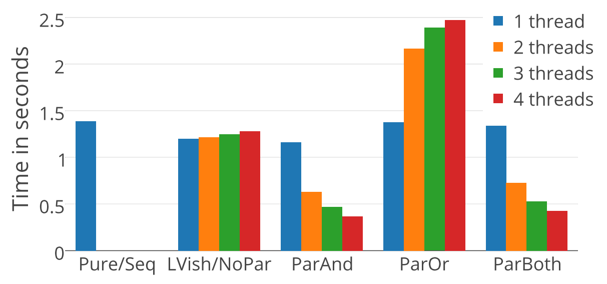


Figure 6. Large constant benchmark of size 2^{16}. Here Or-parallelism is not useful, because in each binary-Or, one branch is always an immediate dead-end.

Figure 6 shows the result of applying `leftmost` to constant of size 2^{16}. In this benchmark there is only one solution, so all vs. first is immaterial. The original Typed Racket inference algorithm takes 338s on our server platform for size 7, where the isomorphic Haskell port takes 1.39s for a size of 15. 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. For this reason the `ParOr` variant of the benchmark not only fails to speed up, but it 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.

Next, we evaluate three and four argument variants of `map-with-funcs`. First we consider the all-solutions variant. This benchmark is a straightforward exercise in Or-parallelism, and the LVish port of the program gets a large benefit over the purely functional Haskell version by not using lists for streams. Even before adding parallelism, LVish achieves a 9.37× speedup over the pure implementation, which takes 2.36s for `(map-with-funcs + - *)` on our server platform. Additionally, the parallelized version achieves up to 8.46× parallel speedup on the server platform, and 3.43× on the desktop. Parallel speedups for the three- and four-argument variants are shown in Figures 3 and 5.

Computing all solutions is wasteful in this case, with the purely function version taking only 0.18s as opposed to 2.36s to compute the first vs. all answers on the server platform. However, the Haskell port achieves a substantial speedup over the Typed Racket implementation on this input, with the original Typed Racket version taking 3.0s to compute the first solution to `(map-with-funcs + - *)`.

When scaling to the four-argument version, LVish is much faster than the purely functional versions even though it is computing all solutions. In fact, this example produces only a few hundred solutions, which is small compared to the size of the search space. Union types in Typed Racket programs represent or-parallelism with a very low *survival rate*—often only one of the variants in a union matches.

If we are willing to admit nondeterminism, it is extremely straightforward to have the parallel LVish implementation asynchronously report the first result it finds, and kill the rest of the `runPar` session. However, this is counter to our goal of deterministic parallelism. 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. (2011) presents parallel type inference as a motivating example, but no implementation was evaluated. In fact, any implementation would have been severely limited due to the restriction that IVars only be written once. As a result, a given type variable could be constrained a *single* time, which is not compatible with most type systems (including Hindley-Milner).

Work on parallel Prolog (Costa et al. 1991; Gupta and Costa 1992) solves similar issues of And- and Or-parallelism. Prolog's control model is distinct from our LVish implementation, though, 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 (1989) discuss `cc(`$\downarrow$`, `$\rightarrow$`)`, 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. However, it makes no determinism guarantees, and is intentionally limited to constraint programming contexts.

Deterministic Search Algorithms Herley et al. (2002) 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 (CPLEX 2009) 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 tackle this challenge, using the LVar framework to ensure determinism in addition to parallelism. We show 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 sharply in need of parallelization, with up to 7.68× speedup. We hope to extend our approach to accomodate choosing a single answer, and use our techniques in the Typed Racket implementation.

References

L. Allison. Continuations implement generators and streams. *The Computer Journal*, 33(5):460–465, 1990. doi: 10.1093/comjnl/33.5.460. URL `http://comjnl.oxfordjournals.org/content/33/5/460.abstract`.

Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11: 598–632, October 1989. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/69558.69562. URL `http://doi.acm.org/10.1145/69558.69562`.

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. *ACM Sigplan Notices*, 44(10):97–116, 2009.

E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(05):552–593, 2013.

K. Claessen, M. Sheeran, and B. J. Svensson. Expressive array constructs in an embedded gpu kernel programming language. In *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*, pages 21–30. ACM, 2012.

V. S. Costa, D. H. Warren, and R. Yang. *Andorra I: a parallel Prolog system that transparently exploits both And-and or-parallelism*, volume 26. ACM, 1991.

I. I. CPLEX. V12. 1: Users manual for cplex. *International Business Machines Corporation*, 46(53):157, 2009.

L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.

E. Dolstra and A. Löh. Nixos: A purely functional linux distribution. In *ACM Sigplan Notices*, volume 43, pages 367–378. ACM, 2008.

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

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 *Proceedings PMCS 2011 Workshop on Parallel Methods for Constraint Solving*. Citeseer, 2011.

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. ISBN 0-201-89502-1. doi: 10.1145/234286.1057830. URL `http://doi.acm.org/10.1145/234286.1057830`.

G. Gupta and V. S. Costa. And-or parallelism in full prolog with paged binding arrays. In *PARLE'92 Parallel Architectures and Languages Europe*, pages 617–632. Springer, 1992.

Y. Hamadi, S. Jabbour, and L. Sais. Manysat: a parallel sat solver. *JSAT*, 6(4):245–262, 2009.

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

K. T. Herley, A. Pietracaprina, and G. Pucci. Deterministic parallel backtrack search. *Theoretical Computer Science*, 270(1):309–324, 2002.

R. Jhala. Refinement types for haskell. In *Proceedings of the ACM SIGPLAN 2014 workshop on Programming languages meets program verification*, pages 27–28. ACM, 2014.

L. Kuper and R. R. Newton. Lvars: lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing (FHPC'13)*, pages 71–84. ACM, 2013.

L. Kuper, A. Todd, S. Tobin-Hochstadt, and R. R. Newton. Taming the parallel effect zoo: Extensible deterministic parallelism with lvish. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 2. ACM, 2014a.

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, 2014b.

J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. In *ACM SIGPLAN Notices*, volume 29, pages 24–35. ACM, 1994.

B. Liskov. A history of clu. *SIGPLAN Not.*, 28(3):133–147, Mar. 1993. ISSN 0362-1340. doi: 10.1145/155360.155367. URL `http://doi.acm.org/10.1145/155360.155367`.

S. Marlow, R. Newton, and S. Peyton Jones. A monad for deterministic parallelism. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 71–82, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0860-1. doi: http://doi.acm.org/10.1145/2034675.2034685. URL `http://doi.acm.org/10.1145/2034675.2034685`.

M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and practice of object systems*, 5 (LAMP-ARTICLE-1999-001):35, 1999.

S. Peyton Jones, A. Reid, F. Henderson, T. Hoare, and S. Marlow. A semantics for imprecise exceptions. In *ACM SIGPLAN Notices*, volume 34, pages 25–36. ACM, 1999.

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

T. Prabhu, S. Ramalingam, M. Might, and M. Hall. Eigencfa: accelerating flow analysis with gpus. In *ACM SIGPLAN Notices*, volume 46, pages 511–522. ACM, 2011.

V. A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245. ACM, 1989.

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. ISSN 0001-0782. doi: 10.1145/359763.359782. URL `http://doi.acm.org/10.1145/359763.359782`.

J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop, University of Chicago, Technical Report TR-2006-06*, pages 81–92, September 2006a.

J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006b.

V. St-Amour, S. Tobin-Hochstadt, M. Flatt, and M. Felleisen. Typing the numeric tower. pages 289–303, 2012.

T. S. Strickland, S. Tobin-Hochstadt, , and M. Felleisen. Practical variable-arity polymorphism. pages 32–46, Mar. 2009.

S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: from scripts to programs. pages 964–974. (Companion (Dynamic Languages Symposium), 2006.

S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. pages 395–406, 2008.

S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 117–128, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. doi: 10.1145/1863543.1863561. URL `http://doi.acm.org/10.1145/1863543.1863561`.

P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc (fd). *The Journal of Logic Programming*, 37(1-3):139–164, 1998.

D. Vytiniotis, S. Peyton Jones, T. Schrijvers, and M. Sulzmann. Outsidein (x) modular type inference with local assumptions. *Journal of Functional Programming*, 21(4-5):333–412, 2011.

A. Zobel. Program structure as basis for parallelizing global register allocation. In *Computer Languages, 1992., Proceedings of the 1992 International Conference on*, pages 262–271, Apr 1992. doi: 10.1109/ICCL.1992.185490.