Monotonicity Types for Distributed Dataflow

Kevin Clancy
Heather Miller

ABSTRACT
The Lasp [2] programming language provides combinator functions such as Union and Intersection for combining set CRDTs. When designing a CRDT combinator, care must be taken to ensure that the combinator is monotone separately in each of its arguments, so that applying it to a tuple of increasing input streams yields an increasing output stream. We consider designing a type system which can prove the monotonicity of CRDT combinators.

ACM Reference format:
DOI: 10.1145/nmnmnmnmnmnn

1 INTRODUCTION
The Lasp language provides a convenient framework for programming distributed systems in which edge computations are performed with minimal need to synchronize with servers. The Lasp programmer defines one or more increasing streams of state-based CRDT values. A stream of CRDT type T provides a monotonic read operation, which takes a threshold value of type T and yields control until the stream produces a value that is greater than or equal to the threshold with respect to a semilattice order defined over the values of T.

Another fundamental tool provided by Lasp is the ability to combine several CRDT streams. For example, from two increasing streams whose values belong to a set CRDT, we can create a new increasing stream whose values are equal to the unions of the values at the heads of the two input streams. Such a CRDT combinator is executed by a process acting as follows. The process first waits for a change to a value at the head of any of its input streams. When a change is detected, the process applies a function to the tuple of values taken from the heads of each input stream, writing the result of this application to an output stream. In the set union example, when one of the two input sets grows, the two sets at the heads of the input streams are used as arguments to the set union function.

The values of the output stream are joined into a third CRDT instance. Lest joining subsequent values of the output stream becomes idempotent, care must be taken so that they increase along the semilattice order. In order to guarantee this, we require that CRDTs be combined using monotonic m-ary functions. An m-ary function f mapping an m-tuple of CRDT values to a CRDT value is a monotonic m-ary function whenever

\[ s \leq s' \implies f(s_1, \ldots, s_m) \leq f(s'_1, \ldots, s'_m) \]

for each of the m argument positions of f. In words, a monotonic m-ary function is a function that is monotone in each of its arguments separately when all other arguments are held fixed. When applied to an m-tuple of increasing CRDT streams, a monotonic m-ary function produces an increasing output stream. This property is crucial for Lasp; an output stream should increase along its CRDT’s semilattice order, just as a stream generated from a CRDT replica does. This way we can perform monotonic reads on it and provide it as input to other CRDT combinators.

2 MONOTONICITY TYPING
Lasp provides a fixed API of CRDTs and CRDT combinators. Included is an Observed-Remove Set CRDT, which models arbitrary set addition and removal monotonically. It also provides combinators for producing the union, intersection, and cartesian product of two observed-remove sets. Programmers may want to construct their own CRDTs along with combinators acting upon them. In such cases, we believe it would be helpful to have a type system which ensures that the implementations of CRDTs and CRDT combinators respect CRDT semantics. In particular, letting \( \leq_T \) denote the semilattice order of CRDT T and letting \( s.u(a) \) denote the invocation of a method u on CRDT instance s with argument a, such a system would prove the following properties:

- A CRDT’s update methods are inflationary; that is, if s, is an instance of CRDT T then for all update methods u and arguments a, s \( \leq_T \) s.u(a).
- CRDT combinators are monotone in each argument separately; that is, if f : \( T_1 \times \cdots \times T_m \rightarrow T \) is an m-ary CRDT combinator then for all \( i \in \{1,\ldots,m\} \) and \( s_i, s'_i \in T_i \) we have that \( s_i \leq_T s'_i \) implies \( f(s_1, s'_2, \ldots) \leq_T f(s'_1, s'_2, \ldots) \).

Properties such as inflativeness and monotonicity are not typically tracked by type systems. But because these properties propagate systematically across function composition, we expect that a simple, compositional, deductive system similar to a type system can be employed.

In the context of logic programming, Datafun[1] provides a concrete example of a type system which tracks monotonicity. Datafun imposes a top-down style of monotonicity reasoning which may prove counterintuitive. To create a single-argument monotone function, the programmer gives the function abstraction a special syntax \( (\mathbf{x} . f) \), writing the bound variable \( \mathbf{x} \) in bold to indicate that it is monotone. Within the abstraction body t, \( \mathbf{x} \) may only occur monotonically. For a term-in-context \( \Gamma \vdash t \), the variable \( \mathbf{x} \) occurs
When such a vector is indexed by a set, the index set is indicated by \( \{ \text{parameterized by element type} \} \) and booleans. A type such as String, The catalog of state-based CRDTs described in [4] suggests that

3.3 Example

Consider implementing an intersection combinator for the 2P-Set

\[
\begin{align*}
\text{Figure 1: An intersection combinator for the 2-P-Set} \\
\text{Figure 2: A Hasse diagram of the partial order on qualifiers}
\end{align*}
\]

is defined as a record, ordered coordinatewise, with labeled components \( \text{A} \) (for added elements) and \( \text{R} \) (for removed elements) each belonging to a standard IntSet datatype. The \( \text{2PIntersect} \) function's return type annotation is augmented with the qualifiers \( \uparrow \text{a} \) and \( \uparrow \text{b} \), meaning that it is monotone separately in the formal arguments named \( \text{a} \) and \( \text{b} \). In addition to the \( \uparrow \text{a} \) qualifier, which denotes monotonicity, we also include the qualifier \( \downarrow \) for antitonicity, = for arguments which are equal to their function's result, \( ? \) for arguments whose relation to their function's result is unknown, and \( ~ \) for superfluous arguments which do not affect their function's result. In the spirit of refinement types [6], these qualifiers are organized into a preordered set (in this case, a poset) which induces a subtyping relation. The partial order among qualifiers is shown in the Hasse diagram of Figure 2.

We prove \( \text{2PIntersect} \)’s monotonicity with respect to its formal arguments \( \text{a} \) and \( \text{b} \) in a bottom-up manner, by qualifying each of the subphrases of its body with respect to \( \text{a} \) and \( \text{b} \). We will demonstrate this process in an informal manner. As a starting point, \( \text{a} \) has type \( \text{2PSet} \{ \leq \text{a}, \sim \text{b} \} \) and \( \text{b} \) has type \( \text{2PSet} \{ \sim \text{a}, \leq \text{b} \} \). \( \text{a.A} \) and \( \text{a.R} \) then have type \( \text{2PSet} \{ \leq \text{a}, \sim \text{b} \} \), while \( \text{b.A} \) and \( \text{b.R} \) have type \( \text{2PSet} \{ \sim \text{a}, \uparrow \text{b} \} \) due to the coordinatwise ordering of record types.

We compute the type of \( \text{intersect}(\text{a.A}, \text{b.A}) \) by viewing it as a function composition. \text{intersect}’s functional characteristics are given by the type \( (x : \text{IntSet}, y : \text{IntSet}) \Rightarrow \text{IntSet}[\uparrow x, \sim y] \), provided by a built-in APL. The terms \( \text{a.A} \) and \( \text{b.A} \) defining \text{intersect}’s actual arguments are both functions of \( \text{2PIntersect} \)’s formal arguments \( \text{a} \) and \( \text{b} \), and so their types are qualified with respect to these formal arguments. Denoting \( \text{a.A} \) and \( \text{b.A} \) with functional notations \( f_a(x, b) \) and \( f_b(a, b) \) respectively, and denoting intersect with the functional notation \( g(x, y) \), our goal is then to compute qualifiers for the function \( h(a, b) = g(f_a(x, b), f_b(a, b)) \).

Computing these qualifiers is a two-step process. We start by computing qualifiers for the function \( h(a, b) \) by combining \( f_a \)’s qualifier for \( a \) (\( \uparrow \)) with \( g \)’s qualifier for \( x \) (\( \uparrow \)) using a qualifier composition operator \( \circ \) described in Figure 3, yielding the qualifier \( \uparrow \). Qualifiers for \( b_x \), \( a_y \), and \( b_y \) are obtained similarly. Finally, to obtain \( h \)'s qualifier for \( a \), we combine \( h \)'s qualifiers for \( a_x (\uparrow) \) and \( a_y (\sim) \) using a qualifier
We present a simple calculus containing novel features which could be used for tracking monotonicity. Along with this, we also present an outline of a soundness proof. We are currently in the process of proving the stated theorems.

4 FORMALIZATION

The computed type of a program term conveys the form of the value to which it reduces. For example, $2 + 3$ is not an $Int$ but instead a term, consisting of two integers and an operator, which reduces to the Int 5. Likewise $intersect(a, b, a)$ is not a IntSet but a term which reduces to one. Recall that we computed the type $IntSet(a, b)$. This is supposed to mean not only that the expression $intersect(a, b, a)$ reduces to an IntSet when $a$ and $b$ are substituted with 2PSet, but also that substituting $a$ with a larger $2PSet$ will result in an expression which reduces to a larger IntSet (and likewise for $b$).

Since monotonicity is about the values resulting from multiple instantiations of the variables of a term rather than just a single instantiation, it falls outside the standard static typing paradigm.

To make sense of this we define a language in Figure 5 which is similar to a standard lambda calculus but with an additional function form called the $sfun$. An $sfun \lambda(x:B)$ is a first-order, multi-argument function intended for defining CRDT combinators such as Section 3.3’s $2PIntersect$. While a term $t$ which falls outside the scope of any $sfun$ abstraction is said to belong to the terminal language, the body $i$ of an $sfun$ belongs to a superset of the terminal language called the lifted language, and is typed under a separate lifted typing relation.

We define a standard reduction relation, referred to as the terminal reduction relation, on our terminal language. Terminal reduction is statically characterized by a terminal typing relation. The lifted language’s reduction relation is for normalizing the body of an $sfun$ to a set-of-pairs representation that is by Theorem 4.2 extensionally equivalent to the $sfun$ itself; while lifted reduction is not computationally tractable, it is statically characterized by a lifted typing relation which can prove this set of pairs monotone, indirectly proving that the $sfun$ is monotone under terminal reduction.

The lifted language generalizes the terminal language’s base type constants to functions of the enclosing $sfun$’s formal parameters. In the lifted language, a terminal language constant such as the integer 1 is reinterpreted as a constant-valued function which maps any valuation of the enclosing $sfun$’s formal parameters to the value 1. The lifted language includes another form of base type value called the ambient map; an ambient map $\langle \omega \rightarrow c \rangle$ is a function—represented extensionally as a set of pairs rather than syntactically as a lambda abstraction—from the enclosing $sfun$’s domain to values of a specific terminal base type. Collectively, constants and ambient maps are referred to as base values, a class of syntactic objects defined in Figure 5 as productions of the metavariable $c$.

To account for this enrichment of base values at the type level, we extend each base type into a poset of refinements. Specifically, this poset is a product of the qualifier set of Figure 2, associating one product component to each formal argument of the enclosing $sfun$, ordered componentwise. For example, in addition to type $Int$ (which is the type of integer-valued constants and ambient maps which are constant with respect to the formals of the enclosing $sfun$) our lifted language’s type syntax also includes the type $Int[\top a, \top b]$ of integer-valued constants and ambient maps which are monotone.
with respect to formal arguments $a$ and $b$. As an example of the
preorder on these types, consider that $\text{Int}[a \leq b] \leq \text{Int}[a, a, 1, \top, \top]$
because $\leq \top$ and $\leq \top$.

Whereas the terminal reduction relation models a series of func-
tion applications, lifted reduction interleaves function application
with a special form of function composition. In this form of func-
tion composition, the outputs of $n$ ambient maps are forwarded as
inputs to an $n$-ary sfun abstraction. In the example of Section 3.3,
the expression $h(a, b) = \text{gt}(f_x(a, b), f_y(a, b))$ represents one such
composition, where $f_x$ and $f_y$ are the ambient maps and $g$ is the
binary sfun into which their results are forwarded. Such a com-
position results in an ambient map, which we’ll discuss in more detail
in Section 4.8.

From now on, we will often refer to the formals of an sfun as
ambient variables, because each base value is implicitly a function
of these variables.

### 4.2 Term syntax

We use the metavariables $s, t$, and $u$ for terms, and use $s, t,$ and $u$
for lifted terms.

We’ve already introduced the syntactic form $\langle \alpha \mapsto c \rangle$ for am-

bient maps and $(\lambda (x : B). t)$ for sfuns. Also note that the lifted

language provides a new abstraction form $(\lambda (x : S, t))$ for functions

whose formal argument type and body are lifted.

A function application is written as the juxtaposition of two

terms, whereas the application of an sfun $s$ to a vector $\vec{t}$ of arguments

is written $s \; \vec{t}$.

### 4.3 Type syntax

We use the metavariables $S, T$, and $U$ for terminal types and $s, t,$

and $u$ for lifted types. As with terms, the lifted type syntax is a

superset of the terminal type syntax.

A lifted base type $B[\Xi]$ refines a terminal base type $B$ with a

qualifier map $\Xi$, which associates each ambient variable in scope

with a qualifier $q$. We’ll use the abbreviation $B[\Xi]$ for $B[\Xi]$ when

$\Xi$ maps $x$ to $\Xi$ and all other ambient variables in context to $\top$.

Terminal types include base types $B$ and function types $S \rightarrow T$.

They also include sfun types $(\bar{x_i} : B_i) \rightarrow A[\Xi]$ where $\bar{x_i} : B_i$
is a vector of typed formal arguments and $A[\Xi]$ is a lifted base
type which qualifies each of the formal arguments $\bar{x_i}$. The lifted

language does not provide an additional construct for sfun types

because we consider an sfun abstraction as encapsulated from the

ambient variables of its context.

### 4.4 Type environments and valuations

A type environment $\Gamma$ is a mapping from variables to terminal types.

When $x \in \text{dom}(\Gamma)$ we write $\Gamma(x)$ to denote the type to which $\Gamma$

maps $x$. A lifted type environment $\Delta$ is like a type environment, but

maps variables to lifted types. An ambient environment $\Omega$, which

represents the formals of an sfun, is a mapping from variables to

terminal base types. The notations $\Omega(x)$ and $\Lambda(x)$ are analogous to

$\Gamma(x)$.

A valuation $\gamma$ of a type environment $\Gamma$ maps each variable $x \in$

$\text{dom}(\Gamma)$ to a value of type $\Gamma(x)$. We write $\gamma(x)$ for the value to which

$\gamma$ maps $x$. The symbols $\omega$ and $\sigma$ are used for valuations of ambient

environments and $\delta$ for valuations of lifted type environments. The

notations $\omega(x)$ and $\delta(x)$ are analogous to $\gamma(x)$. Substitution of a

dvaluation $\gamma$ into a term $t$ is written $t \gamma$. While substitutions of the form $yt$ and $\delta t$

are defined in a standard manner (and thus elided), we will not typically substitute ambient valuations $\omega$ this way. This

is because the ambient variables of $\Omega$ live in a namespace distinct

from that of the variables in $\Gamma$ and $\Delta$. The former can occur only in

the domains of ambient maps and qualified base types, whereas the

latter occur only as terms. Accordingly, we will use a special form

of substitution for ambient valuations, described in Section 4.7 and

Figure 8.

### 4.5 Terminal reduction

Under terminal reduction, described in Figure 6, an application is

reduced by substituting the value on the right-hand side into the

body of the abstraction on the left-hand side. Sfun application must

not only substitute the supplied arguments into the body of the

sfun, but also convert the body of the sfun from a lifted term to a

terminal one by dropping all qualifier maps $\Xi$ from type ascriptions.
This is accomplished using an ambient substitution.

We will write $t \parallel u$ to indicate that a term $t$ diverges under terminal

reduction, and write $t \parallel v$ to indicate that it normalizes to a value $v$.

### 4.6 Terminal logical relations

In Figure 7 we define a collection of predicates on closed terms which is
indexed by the set of terminal types. We also define a collection

environment-indexed predicates on substitutions. These definitions follow the logical relations technique (an accessible introduction to
which can be found in chapter 12 of [3]) for reasoning about typed languages.

A typing judgment for a terminal term conveys information about the behavior of that term under terminal reduction; the terminal logical relations capture this behavior. More concretely, we will prove that if \( \Gamma \vdash t : T \) then for all \( y \in G[T] \) we have \( yT \in T[T] \), as a consequence. All well-formed terminal terms normalize under terminal reduction. The toy language which we present lacks recursion, and so normalization is not hard to achieve. We will discuss a technique in Section 5.4 for extending our language with a controlled form of recursion that precludes the possibility of divergence.

### 4.7 Ambient substitution

Let \( \Omega \) be an ambient environment. An ambient map \( \langle \phi \mapsto c_{\phi} \rangle_{\omega} \) of domain \( O[\Omega] \) represents a constant whose value is undetermined but can be resolved given some valuation \( \omega \in O[\Omega] \). The ambient substitution \( \|\omega\| < \phi \mapsto c_{\phi} > \) performs such a resolution, yielding \( c_{\omega} \). More generally, a lifted term \( t \) whose ambient maps share the domain \( O[\Omega] \) represents a terminal term with several undetermined constants in various locations. These constants can be collectively resolved by a single \( \omega \in O[\Omega] \). Such a resolution is written \( \|\omega\|t \) and defined inductively in Figure 8.

### 4.8 Lifted reduction

While ambient substitution resolves the undetermined constants of a lifted term, the lifted reduction relation of Figure 9 allows us to reduce a lifted term before performing any such resolution. The LRed-SFUN-APP rule reduces a sfun application by brute force, performing each possible ambient substitution, normalizing under terminal reduction, and finally collecting the results into an ambient map. For each \( \omega \in O[\Omega] \), it has one premise of the form \((\lambda(x:B). t)[\bar{\omega}]\).

Under an ambient environment \( \Omega \), a term of the form \( (\lambda(x:B). t)[\bar{\omega}] \) can be thought of as a composition of the same sort as the example

\[ h(a, b) = g(f_2(a, b), f_3(a, b)) \]

as described in Section 3.3, given the following identifications:

1. The ambient environment \( \Omega \) corresponds to the variables \( a \) and \( b \).
2. The base values \( \bar{\omega} \) correspond to the functions \( f_2(a, b) \) and \( f_3(a, b) \).
3. The sfun \((\lambda(x:B). t)\) corresponds to the function \( g \).

---

### 4.9 Lifted logical relations

The lifted logical relations in Figure 10 are analogous to the terminal ones, except that types are interpreted through the lens of lifted reduction and ambient substitution. A new class \( K \) of logical relations provides semantic interpretations in terms of ambient substitution for the various qualiﬁers. \( X[\Omega] \) denotes the set of qualiﬁer maps on ambient environment \( \Omega \). Because lifted reduction is performed under some ambient environment \( \Omega \), we parameterize our relations by ambient environments, which appear as the subscripts beneath \( K, V, T, \) and \( G \).

Because the lifted types are a superset of the terminal types, we must take care to deﬁne lifted logical relations for terminal base types as well as qualiﬁed ones. Under an ambient environment \( \Omega \), a terminal base type \( B \) is interpreted equivalently to \( B[\Xi] \) where \( \Xi \) assigns every ambient variable in \( dom(\Omega) \) to the qualiﬁer \( \sim \). This reﬂects that inside an sfun body a value of terminal type typically originates from outside the sfun and therefore should not depend on the arguments to which the sfun is applied.

### 4.10 Subtyping

Defined in Figure 11, a terminal subtyping judgment \( S \ll T \) means that the set of values of type \( S \) is contained in the set of values of type \( T \). Terminal subtyping judgments are derived using standard rules for base and function types. The S-SFUN rule for sfun subtyping requires matching argument types and deﬁers to lifted subtyping to ensure that the qualiﬁers of the result types respect the qualiﬁer ordering.

Defined in Figure 12, a lifted subtyping judgment \( \Omega \vdash S \ll T \) means that under ambient environment \( \Omega \), the set of lifted values of type \( S \) is contained in the set of lifted values of type \( T \). Four rules are required to handle base types, since both the left and right side can be either qualiﬁed or terminal. Because the lifted logical relations interpret terminal base types \( B \) equivalently to \( B[\Xi] \) where

---

**Figure 8: Ambient substitution**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( |\omega| t ) = t</td>
<td></td>
</tr>
<tr>
<td>( |\omega| \langle \phi \mapsto c_{\phi} \rangle_{\omega} )</td>
<td>undetermined</td>
</tr>
<tr>
<td>( |\omega| (\lambda x : S.t) ) = ( (\lambda x : |\omega|S) (|\omega|t) )</td>
<td></td>
</tr>
<tr>
<td>( |\omega| ((t \ i) \ ] \ t) ) = ( (|\omega|t)ogeneity ( |\omega| ) |\omega| [ ] )</td>
<td></td>
</tr>
<tr>
<td>( |\omega| S ) = S</td>
<td></td>
</tr>
<tr>
<td>( |\omega| S \rightarrow T ) = ( |\omega|S \rightarrow |\omega|T )</td>
<td></td>
</tr>
<tr>
<td>( |\omega| B[\Xi] ) = B</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 9: Lifted reduction**

We write \( \Omega \vdash i \uparrow \) to indicate divergence and \( \Omega \vdash i \downarrow \bar{\omega} \) to indicate convergence to a lifted value \( \bar{\omega} \) under lifted reduction. Theorem 4.2 tells us that lifted reduction properly characterizes terminal reduction.

**Lemma 4.1.** If \( \Omega \vdash i \rightarrow i' \) then for all \( \omega \in O[\Omega] \), \( \|\omega\|i \rightarrow^* \|\omega\|i' \)

**Proof.** By induction on derivations of \( \Omega \vdash i \rightarrow i' \).

**Theorem 4.2.** If \( \Omega \vdash i \downarrow \bar{\omega} \) then for all \( \omega \in O[\Omega] \), \( \|\omega\|i \downarrow \|\omega\|\bar{\omega} \).

**Proof.** A simple corollary of Lemma 4.1.
∀x : B \implies \{ \Xi, q \mid \Xi \in X[\Omega] \} \\
S-FUN \quad x : B \implies A_1[\Xi_1] : A_2[\Xi_2] \\
S-FUN \quad T_1 : S_1 \implies S_2 : T_2 \\
S-BASE \quad B < B

\begin{align*}
\text{Figure 11: Terminal subtyping} & \\
\text{LS-FUN} & \quad x : B \implies A_1[\Xi_1] : A_2[\Xi_2] \\
\text{LS-BASE} & \quad \Omega + x : B \implies A_1[\Xi_1] < : (x : B) \implies A_2[\Xi_2] \\
\text{LS-BASE-TT} & \quad \Omega + T_1 < : S_1 \implies S_2 < : T_1 \implies T_2 \\
\text{LS-BASE-LT} & \quad \Omega + B < : B \\
\text{LS-BASE-LL} & \quad \Omega + B < : B[\{ \Xi, q \mid \Xi \in X[\Omega] \}] \\
\end{align*}

\begin{align*}
\Xi \text{ qualifies every ambient variable as } \sim, & \quad B & \text{ and } B[\Xi] \text{ are isomorphic} \\
& \text{ with respect to subtyping; i.e., } \Omega + B < : B[\Xi] & \text{ and } \Omega + B[\Xi] < : B.
\end{align*}

The following theorems ensure that every subtyping derivation implies a containment among logical relations.

**Theorem 4.3 (Fundamental terminal subtyping theorem).** 
If $S < : T$ then $\mathcal{V}[S] \subseteq \mathcal{V}[T]$.

**Proof.** By induction on the derivation of $S < : T$. 

**Theorem 4.4 (Fundamental lifted subtyping theorem).** 
If $\Omega + \hat{S} < : \hat{T}$ then $\mathcal{V}_\Omega[\hat{S}] \subseteq \mathcal{V}_\Omega[\hat{T}]$.

**Proof.** By induction on the derivation of $\Omega + \hat{S} < : \hat{T}$. 

The terminal typing relation is described in Figure 14. The only interesting rule is T-SFUN for sfun abstractions, which types the body of the sfun under the lifted typing relation with new ambient and lifted environments formed from the sfun’s formal arguments. Recall from Section 4.4 that ambient variables live in a namespace separate from those of the terminal and lifted type environments; T-SFUN leverages this by adding identically-named variables to both ambient and lifted environments.

The lifted typing relation is described in Figure 15. Its context includes both a terminal type environment $\Gamma$ and a lifted type environment $\Delta$. Because every terminal value is also a lifted value, substituting terminal values for the variables of $\Gamma$ in $T$ does not pose a problem. This typing relation does not include a rule for sfun abstractions; we consider programs with nested sfun abstractions ill-typed. However, the body of an sfun abstraction may still refer to other sfun abstractions through variables. As a consequence, a well-typed term may step to an ill-typed term when an sfun abstraction is substituted through the boundary of another sfun abstraction. Lack of type preservation should not pose a problem to our soundness proof outline, because it uses logical relations.

The only novel lifted typing rule is T-SFUN-APP, for typing an sfun application $\hat{u} [\, \tilde{s}_i, \ldots, \tilde{s}_n \,]$. The first premise requires $\hat{u}$ to have an $n$-ary sfun type $(\tilde{x}_1 : A_1) \Rightarrow B[\hat{s}_1, \hat{s}_2]$ The other $n$ premises require that each argument $\tilde{s}_i$ has a lifted base type of the form $A_i[q \in \Sigma \in \mathcal{dom}(\Omega)]$, where $q \in \Sigma$ is a qualifier map synthesized from type checking $\tilde{s}_i$. The sfun application $\hat{u} [\, \tilde{s}_i \,]$ then has the type $B[q \in \Sigma \in \mathcal{dom}(\Omega)]$, where $\circ$ is the operator defined in Figure 3 and the "summand" $\Sigma_{i=1}^n \ldots$ uses the qualifier contraction operator $\oplus$ defined in Figure 4.

LT-FUN requires a lifted type well-formedness derivation of $\Omega \vdash S$, which ensures that all qualifiers in $S$ are contained in the ambient environment $\Omega$. The lifted type well-formedness rules have been elided because they are not interesting.

The following fundamental typing theorems imply that our language is sound.

4.12 Why logical relations?

Why reason about the soundness of our language using logical relations rather than the more popular approach of progress and preservation? Under lifted reduction, type preservation across sfun applications would prove problematic. Explained briefly, we care about type preservation across normalization rather than single-step reduction.

More specifically, imagine trying to prove type preservation for a well-typed term of the form $(\tilde{\lambda}(\tilde{x}_1 : B_1) \cdot i)(\tilde{\varepsilon})$. By LRED-SFUN-APP, such a term reduces in a single step to an ambient map, but deducing a lifted base type for this ambient map requires characterizing $(\tilde{\lambda}(\tilde{x}_1 : B_1), i)$ extensionally. Because $(\tilde{\lambda}(\tilde{x}_1 : B_1), i)$ is well-typed we might invoke an inversion lemma to get $\emptyset; \Omega; \emptyset \vdash (\tilde{\lambda}(\tilde{x}_1 : B_1), i) : (\tilde{x}_1 : B_1) \Rightarrow A[\varepsilon]$ and $\emptyset; \tilde{x}_1 : B_1; \tilde{x}_1 = \tilde{x}_1 \vdash i : A$. A substitution lemma may tell us that for $\delta \in \mathcal{G}_{\mathcal{O}[\delta]}[\tilde{x}_1 = \tilde{x}_1]$, we have $\emptyset; \tilde{x}_1 : B_1; \emptyset \vdash \delta i : A$. Not knowing the derivation height of this latter judgment, we could not apply the inductive hypothesis to it. Even if we could, we would only conclude that $\tilde{x}_1 : B_1 \vdash \delta i \Rightarrow i'$ for some $i'$ where $\emptyset; \tilde{x}_1 \vdash B_1; \emptyset \vdash i' : A$, whereas we actually need type preservation across normalization.

5 ADDITIONAL FEATURES

The language of Section 4 has been kept simple to avoid distracting from its novel feature: the sfun. Calling it a toy language would be charitable however, as it lacks recursion, conditionals, and the other features discussed below.

5.1 Sfun primitives

Sfun abstractions only become useful when we have primitive sfuns with which to define them. When working with integers, for example, we’d like an addition operator with type

$$(x : \text{Int}, y : \text{Int}) \Rightarrow \text{Int}(\uparrow x, \uparrow y)$$

and a subtraction operator with type

$$(x : \text{Int}, y : \text{Int}) \Rightarrow \text{Int}(\downarrow x, \downarrow y)$$

Likewise, for working with sets we’d like sfun primitives including union, intersection, and set difference.
5.2 Divergence dilemma

The standard approach to recursion using fixpoint combinators would introduce the possibility that well typed terms diverge. Logical relations tend to accommodate recursion by defining

\[ T[T] = \{ t \mid t \uparrow \text{ or } t \downarrow u \text{ where } u \in \mathcal{V}[T] \} \]

Such definitions leverage the idea that if \( t \) fails to normalize then we can soundly claim it belongs to any type \( T \), since the set of values it might normalize to is vacuously contained in \( \mathcal{V}[T] \). However, conditionals would introduce the possibility that some applications of an sfun normalize while other applications of the same sfun diverge. In this case, given that lifted reduction simulates the terminal reduction of all possible applications simultaneously, it would be reasonable for the sfun body to diverge under lifted reduction. This is unsound, as it allows typing derivations associating any qualifier map to the sfun’s result type, even ones which contradict its set of normalizing applications.

5.3 Dependent refinement types

We would like to solve the aforementioned dilemma using dependent refinement types, drawing inspiration from LiquidHaskell [5]. In this approach a refined base type has the form \( \{ v : B \mid t \} \), where \( B \) is an unrefined base type such as \( \text{Bool} \) or \( \text{Int} \), and \( t \) is a term of type \( \text{Bool} \). Such a type describes the set of constants \( c \) of base type \( B \) such that \( c/v \) is \( t \). Standard function types give way to dependent function types, which have the form \( x : S \rightarrow T \). The variable \( x \), which is scoped over type \( T \), refers to the value to which a function of type \( x : S \rightarrow T \) is applied.

Because types may contain types, subtyping becomes parameterized over a typing context, which is significant to the following subtyping rule for refined base types.

\[
\text{SUB-BASE: } \forall \gamma \in \mathcal{G}[\Gamma, v : B], \gamma S \downarrow \text{true} \implies \gamma T \downarrow \text{true} \]

Γ ⊢ \{ v : B \mid t \} \downarrow \text{true}

While this rule’s premise is undecidable, it can be conservatively approximated by an SMT solver, which is the approach taken by LiquidHaskell.

In a dependent refinement type system, the type assigned to a variable by a type environment may depend on the variables which precede it, so \( \mathcal{G}[-] \) would be redefined with:

\[ \mathcal{G}[\Gamma, x : T] = \{ y[x \mapsto v] \mid y \in \mathcal{G}[\Gamma] \text{ and } v \in \mathcal{V}[yT] \} \]

Our fundamental terminal subtyping and typing theorems would also change:

**Theorem 5.1 (Dependent terminal subtyping theorem).** If

\[ \Gamma \vdash S \Downarrow T \text{ then for all } \gamma \in \mathcal{G}[\Gamma], \gamma V[yS] \subseteq \gamma V[yT] \]

**Theorem 5.2 (Dependent terminal typing theorem).** If

\[ \Gamma \vdash t : T \text{ then for all ambient environments } \Omega \text{ with } FV(\Omega) = \emptyset \text{ and all } y \in \mathcal{G}[\Gamma] \cap \mathcal{G}[\Omega], \text{we have } \forall \gamma \Downarrow \gamma V[yT] \cap \gamma V_\Omega[yT] \]

Our lifted typing contexts would have dependencies as well. An entry of the ambient environment would depend on entries of the terminal environment. An entry of the lifted environment would depend both on entries of the terminal environment and preceding entries of the lifted environment.

5.4 Terminating fixpoints

A dependent refinement type system can be extended with a terminating fixpoint construct, which allows recursion given a decreasing, well founded termination metric. Using \( \text{Nat} \) as a shorthand for \( \{ v : \text{Int} \mid 0 \leq v \} \), Figure 16 demonstrates a function with such a termination metric. The function \( f \), which gives integers \( n \) and \( i \) with \( i \leq n \) computes the sum of integers from \( i \) to \( n \), has a termination metric as its leading argument. A typechecker can then prove this function terminating by checking its body under a weakened context in which \( f \) has the type:

\[ m' : \{ v : \text{Nat} \mid v < \text{metric} \} \rightarrow n : \text{Int} \rightarrow i : \{ v : \text{Int} \mid (v \leq n) \land (m' = n - i) \} \]

Using this technique LiquidHaskell is able to prove 96% of all recursive functions terminating [5].

LiquidHaskell treats the terminating fixpoint combinator \( \text{fix} \) as a built-in constant, but to address its subtleties we treat it as a first class component of the language, which we summarize in Figure 17. We add a type constructor \( \langle x \times T \rangle \) with the following introduction form:

\[ (\text{fix} t x T y s) \]

This introduction form represents a suspended recursive computation where \( t \) is the current termination metric, \( x \) is a variable for the subsequent termination metric, \( T \) is the type of the recursive term (which should depend on \( x \), \( y \) is a recursive self-reference, and \( s \) is a term (which should include occurrences of \( x \) and \( y \)) representing the computation itself. The elimination form \( u(s) \) resumes the recursive computation \( u \) using \( s \) as the subsequent value of its termination metric.

5.5 Another example

As a final example, Figure 18 describes a CRDT combinator for adding two GCounter CRDTs. We define the GCounter type in terms of a language primitive called \( \text{IntArray} \), an array of integers ordered componentwise. The sfun \( \text{getAt} \), for getting the element at index \( i \) of array \( a \), is monotone with respect to \( a \) because \( \text{IntArray} \) is ordered
\[ \text{Red-TFix} \]
\[ (tfix \; v \; x \; T \; y \; s) \langle v' \rangle \rightarrow [v'/x][(tfix \; v' \; x \; T \; y \; s)/y]s \]

\[ \text{T-Fix-Abs} \]
\[ \Gamma \vdash t : \text{Nat} \quad \Gamma, x : \{ v : \text{Nat} \mid v < t \} \vdash T \]
\[ \Gamma, x : \{ v : \text{Nat} \mid v < t \}, y : (x' < x) [x'/x]T \vdash s : T \]
\[ \Gamma \vdash (tfix \; t \; x \; T \; y \; s) : (x < t) T \]

\[ \text{T-Fix-App} \]
\[ \Gamma \vdash u : (x < t) U \quad \Gamma \vdash s : \{ v : \text{Nat} \mid v < t \} \]
\[ \Gamma \vdash u \langle s \rangle : [s/x]U \]

**Figure 17: Terminating fixpoint**

componentwise. Its relation to \( i \) is unknown (thus qualified with \( ? \)) since we make no assumptions about the way distinct elements of an \( \text{IntArra} \) are related. The sfun \( \text{setAt} \) sets element \( i \) of \( \text{IntArra} \) \( a \) to the integer \( v \). It is monotone with respect to \( a \) and \( v \) due to \( \text{IntArra} \)'s componentwise ordering, and unknown with respect to \( i \) due to the lack of information about how distinct elements of the array are related. The \( \text{len} \) function gets the length of an \( \text{IntArra} \) and the \( \text{makeArray} \) function creates a new array with all elements initialized to 0.

The nested function \( \text{sumCell} \) uses terminating recursion to iterate through the indices of the two arrays.

Because the type \( \{ v : \text{IntArra} \mid \text{len}(v) = n \} \) is isomorphic to

\[ \{ v : \text{IntArra} \mid \text{len}(v) = n \}[\sim x, \sim y] \]

the type of the \( \text{IntArra} \) returned from \( \text{makeArray} \) on line 19 is subsumed by

\[ \{ v : \text{IntArra} \mid \text{len}(v) = n \}[\uparrow x, \uparrow y] \]

Assuming that each call to \( \text{setAt} \) does not make a new copy of the entire array but instead returns a reference to the existing one, updating arrays in this manner is unsound. An array update invalidates the types of existing aliases to that array. With such a need for strong updates, it may be worth investigating the addition of linear or affine types.

6 CONCLUSION

While the proposed approach may have practical benefits, we are still attempting to provide it with a suitable operational model and soundness proof. Nonetheless, the daunting number of possibilities for combining CRDTs in a Lasp-style language provides strong motivation for the type checking technique that we have outlined.
getAt :: (a : IntArray, i : {v: Nat | v < len(a)}) ⇒ Int[↑ a, ? i]

setAt :: (a : IntArray, i : {v: Nat | v < len(a)}, v : Int) ⇒ IntArray[↑ a, ? i, ↑ v]

len :: IntArray → Int

makeArray :: n: Nat → { v : IntArray | len(v) = n }

+ :: (x:Int , y: Int ) ⇒ Int[↑ x, ↑ y]

type GCounter = { array : IntArray }

sfun sumCounters (x : GCounter, y : {v: GCounter | len(v) = len(x)}) : GCounter[↑ x, ↑ y] =

let xArray : IntArray[↑ x, ~ y] = x. array

let yArray : IntArray[~ x, ↑ y] = y. array

// termination metric : ( len acc ) - ind

fun sumCell (ind : Nat , acc : IntArray[↑ x, ↑ y]) : IntArray[↑ x, ↑ y] =

if ind = (len acc) then

acc

else

let acc' = setAt acc ind ((getAt xArray ind) + (getAt yArray ind))

sumCell (ind+1) acc'

let newArray : IntArray[↑ x, ↑ y] = makeArray (len x)

GCounter { array = sumCell 0 newArray }

Figure 18: A CRDT combinator computing the sum of two GCounters

REFERENCES

datalog. In Proceedings of the 21st ACM SIGPLAN International Conference on

[2] Christopher Meiklejohn and Peter Van Roy. Lasp: A language for distributed,
coordination-free programming. In Proceedings of the 17th International Sympo-
sium on Principles and Practice of Declarative Programming, pages 184–195. ACM,
2015.


[4] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A compre-
hensive study of convergent and commutative replicated data types. PhD thesis,
Inria–Centre Paris-Rocquencourt; INRIA, 2011.
