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:
Kevin Clancy and Heather Miller. 2017. Monotonicity Types for Distributed Dataflow. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 10 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

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

$$s \leq s' \implies f(\ldots, s, \ldots) \leq f(\ldots, s', \ldots)$$

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 *inflative*; 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 $f : T_1 \times \ldots \times T_m \to T$ is an m-ary CRDT combinator then for all $i \in 1..m$ and $s_i, s_i' \in T_i$ we have that $s_i \leq_{T_i} s_i'$ implies $f(\ldots, s_i, \ldots) \leq_T f(\ldots, s_i', \ldots)$.

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 ($\lambda \mathbf{x}.t$), 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

*Northeastern University
†Northeastern University

Conference'17, Washington, DC, USA

DOI: 10.1145/nnnnnnn.nnnnnnn

monotonically when for any pair of closing substitutions γ and γ' of Γ with $\gamma(\mathbf{x}) \leq \gamma'(\mathbf{x})$ we have $\gamma t \leq \gamma' t$.

It's not clear that programmers can easily determine which positions in a term are monotone. A sound type system is not effective if the programmer does not intuitively understand the rules which govern it, because writing a program then entails a grueling process of trial and error.

Another arguable weakness of Datafun is that monotonicity is "hard coded" into Datafun's type system. The typing context is partitioned into two parts: one for variables which may only be used monotonically and the other for variables without usage restrictions. There are separate application rules for regular and monotone functions. For general purpose programming, it may be preferable to use a type system in which monotonicity is just one of the many tracked properties (including inflativeness) which propagate systematically across function composition.

Finally, Datafun requires the programmer to consider partial orderings over function types. This may be less intuitive than our approach, which avoids the need for partially ordering function types by providing primitive multi-argument functions.

3 TOWARD A CRDT TYPE SYSTEM

3.1 Notational preliminaries

We use colored overlines to represent vectors of syntactic objects. When such a vector is indexed by a set, the index set is indicated by a superscript. When vectors of the same color are juxtaposed, we assume they have the same length or index set. When vectors of different colors are juxtaposed, we assume they have differing lengths or index sets. When omitting a superscript, the index set should be clear from context. The superscript $i \in 1..n$ accompanying an overline indicates that the index set consists of all integers from 1 to some implicitly introduced natural number n. As an example, $\overline{x_i}^{i \in 1..n}$ denotes the vector of variables $x_1, x_2, \ldots, x_n$.

3.2 Type Syntax

The catalog of state-based CRDTs described in [4] suggests that a CRDT's semilattice order can be derived in a straightforward manner from the CRDT's type structure. As a starting point, we need base types such as Natural, with the standard ordering on natural numbers, 0 as a bottom element, and maximum as the join operator. We would also include semilattice base types for sets (parameterized by element type) and booleans. A type such as String, whose values do not inherently form a bounded join-semilattice, could be lifted to one by ordering its values discretely and adding top and bottom elements. Compound types such as records, arrays, and maps would be ordered coordinatewise by default. For example, we would define the order $\leq_T$ of the record type $T = \{\overline{l : T_l}\}$ as follows: $\{\overline{l = v_l}\} \leq_T \{\overline{l = u_l}\} \iff \overline{v_l \leq_{T_l} u_l}$. The state based Last-Writer-Wins Register would require a special pair type that is ordered lexicographically rather than coordinatewise.

3.3 Example

Consider implementing an intersection combinator for the 2P-Set CRDT described in [4]. A potential implementation of such a combinator is given in *Figure 1*. In this program the CRDT type *2PSet* is defined as a record, ordered coordinatewise, with labeled components A (for added elements) and R (for removed elements) each belonging to a standard IntSet datatype. The *2PIntersect* function's return type annotation is augmented with the *qualifiers* $\uparrow$ a and $\uparrow$ *b*, meaning that it is monotone separately in the formal arguments named a and b. In addition to the $\uparrow$ 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 $\sim$ 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*.

```
type 2Pset = {A : IntSet, R : IntSet}

fun 2PIntersect(a : 2Pset, b : 2Pset) : 2PSet[↑a, ↑b] =
  { A = intersect(a.A, b.A), R = union(a.R, b.R) }
```

Figure 1: An intersection combinator for the 2P-Set

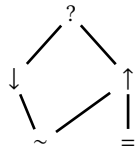

Figure 2: A Hasse diagram of the partial order on qualifiers

We prove *2PIntersect*'s monotonicity with respect to its formal arguments *a* and *b* in a bottom-up manner, by qualifying each of the subphrases of its body with respect to *a* and *b*. We will demonstrate this process in an informal manner. As a starting point, *a* has type $2Pset[= a, \sim b]$ and *b* has type $2Pset[\sim a, = b]$. *a.A* and *a.R* then have type $2Pset[\uparrow a, \sim b]$, while *b.A* and *b.R* have type $2Pset[\sim a, \uparrow b]$ due to the coordinatwise ordering of record types.

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

Computing these qualifiers is a two-step process. We start by computing qualifiers for the function

$$\hat{h}(a_x, b_x, a_y, b_y) = g(f_x(a_x, b_x), f_y(a_y, b_y))$$

We compute the qualifier for a_x by combining f_x'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 $\hat{h}$'s qualifiers for a_x ($\uparrow$) and a_y ($\sim$) using a *qualifier*

$\circ$	?	$\uparrow$	$\downarrow$	=	$\sim$
?	?	?	?	?	$\sim$
$\uparrow$	?	$\uparrow$	$\downarrow$	$\uparrow$	$\sim$
$\downarrow$	?	$\downarrow$	$\uparrow$	$\downarrow$	$\sim$
=	?	$\uparrow$	$\downarrow$	=	$\sim$
$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$

Figure 3: Qualifier composition $\circ$

+	?	$\uparrow$	$\downarrow$	=	$\sim$
?	?	?	?	=	?
$\uparrow$	?	$\uparrow$	?	=	$\uparrow$
$\downarrow$	?	?	$\downarrow$	=	$\downarrow$
=	=	=	=	=	=
$\sim$	?	$\uparrow$	$\downarrow$	=	$\sim$

Figure 4: Qualifier contraction +

contraction operator +, described in *Figure 4*, yielding $\uparrow$. h's qualifier for b is obtained similarly.

4 FORMALIZATION

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.

4.1 Introduction

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 $intersect(a.A, b.A)$ is not a *IntSet* but a term which reduces to one. Recall that we computed the type $IntSet[\uparrow a, \uparrow b]$ for $intersect(a.A, b.A)$. This is supposed to mean not only that the expression $intersect(a.A, b.A)$ reduces to an *IntSet* when a and b are substituted with $2PSets$, 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 $(\tilde{\lambda}(\overline{x : B}).\ \dot{t})$ 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 $\dot{t}$ 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

x, y, z		*Variables*
Type Syntax		
q ::=	$\uparrow \mid \downarrow \mid \sim \mid = \mid ?$	*Qualifiers*
Ξ ::=	$\emptyset \mid \Xi, q\ x$	*Qualifier maps*
A, B ::=	$Int \mid Bool$	*Base types*
$\dot{A}, \dot{B}$ ::=	$B \mid B[\Xi]$	Lifted base types
S, T, U ::=	$B \mid S \to T \mid (\overline{x_i : B_i}^{i \in 1..n}) \Rightarrow A[\Xi]$	*Types*
$\dot{S}, \dot{T}, \dot{U}$ ::=	$S \mid \dot{B} \mid \dot{S} \to \dot{T}$	*Lifted types*
Environments		
Γ ::=	$\emptyset \mid \Gamma, x : T$	*Type environments*
Ω, Φ ::=	$\emptyset \mid \Omega, x : B$	*Ambient environments*
Δ ::=	$\emptyset \mid \Delta, x : \dot{T}$	*Lifted type environments*
$\gamma, \omega, \phi, \delta$		*Environment valuations*
Term Syntax		
c ::=	$true \mid false \mid 0 \mid -1 \mid 1 \mid -2 \mid 2 \ldots$	*Base values*
$\dot{c}$ ::=	$c \mid <\overline{\omega \mapsto c}>$	*Lifted base values*
v, w ::=	$c \mid (\lambda x : S.t) \mid (\tilde{\lambda}(\overline{x : B}).\ \dot{t})$	*Values*
$\dot{v}, \dot{w}$ ::=	$v \mid \dot{c} \mid (\lambda x : \dot{S}.\dot{t})$	*Lifted values*
s, t, u ::=	$v \mid x \mid t\ t \mid t\ [\ \overline{t}\ ]$	*Terms*
$\dot{s}, \dot{t}, \dot{u}$ ::=	$t \mid \dot{v} \mid \dot{t}\ \dot{t} \mid \dot{t}\ [\ \overline{\dot{t}}\ ]$	*Lifted terms*
Evaluation contexts		
E ::=	$[] \mid E\ t \mid v\ E \mid E\ [\ \overline{t}\ ] \mid v\ [\ \overline{v}\ E\ \overline{t}\ ]$	*Evaluation contexts*
$\dot{E}$ ::=	$[] \mid \dot{E}\ \dot{t} \mid \dot{v}\ \dot{E} \mid \dot{E}\ [\ \overline{\dot{t}}\ ] \mid \dot{v}\ [\ \overline{\dot{v}}\ \dot{E}\ \overline{\dot{t}}\ ]$	*Lifted evaluation contexts*

Figure 5: Syntax and environments

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 reduciton.

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 $<\overline{\omega \to c}>$ 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 $\dot{c}$.

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 $Int[\uparrow a, \uparrow b]$ 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 $Int[\sim a, = b] \leq Int[\uparrow a, \uparrow b]$ because $\sim \leq \uparrow$ and $= \leq \uparrow$.

Whereas the terminal reduction relation models a series of function applications, lifted reduction interleaves function application with a special form of function composition. In this form of function 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 $h(a, b) = g(f_x(a, b), f_y(a, b))$ 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 composition 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 $\dot{s}$, $\dot{t}$, and $\dot{u}$ for lifted terms.

We've already introduced the syntactic form $\ll\overline{\omega \mapsto c}\gg$ for ambient maps and $(\tilde{\lambda}(\overline{x : B}).\ \dot{t})$ for sfuns. Also note that the lifted language provides a new abstraction form $(\lambda x : \dot{S}.\dot{t})$ 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 $\overline{t}$ of arguments is written $s\ [\ \overline{t}\ ]$.

4.3 Type syntax

We use the metavariables S, T, and U for terminal types and $\dot{S}$, $\dot{T}$, and $\dot{U}$ 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* Ξ, which associates each ambient variable in scope with a qualifier q. We'll use the abbreviation $B[= x]$ for $B[\Xi]$ when Ξ maps x to $=$ and all other ambient variables in context to $\sim$.

Terminal types include base types B and function types $S \to T$. They also include sfun types $(\overline{x_i : B_i}^{i \in 1..n}) \Rightarrow A[\Xi]$ where $\overline{x_i : B_i}$ is a vector of typed formal arguments and $A[\Xi]$ is a lifted base type which qualifies each of the formal arguments $\overline{x_i}$. The lifted language does not provide an additional constructor 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 Γ is a mapping from variables to terminal types. When $x \in dom(\Gamma)$ we write $\Gamma(x)$ to denote the type to which Γ maps x. A lifted type environment Δ is like a type environment, but maps variables to lifted types. An ambient environment Ω, which represents the formals of an sfun, is a mapping from variables to terminal base types. The notations $\Omega(x)$ and $\Delta(x)$ are analogous to $\Gamma(x)$.

A *valuation* γ of a type environment Γ maps each variable $x \in dom(\Gamma)$ to a value of type $\Gamma(x)$. We write $\gamma(x)$ for the value to which γ maps x. The symbols ω and ϕ are used for valuations of ambient environments and δ for valuations of lifted type environments. The notations $\omega(x)$ and $\delta(x)$ are analogous to $\gamma(x)$. Substitution of a valuation γ into a term t is written γt. While substitutions of the form γt and $\delta \dot{t}$ are defined in a standard manner (and thus elided), we will not typically substitute ambient valuations ω this way. This is because the ambient variables of Ω live in a namespace distinct from that of the variables in Γ and Δ. 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*.

$$\textsc{Red-Context}\ \frac{t \longrightarrow t'}{E[t] \longrightarrow E[t']} \qquad \textsc{Red-App}\ \frac{}{(\lambda x : S.t)\ v \longrightarrow [x/v]t}$$

$$\textsc{Red-SFun-App}\ \frac{\omega = [\overline{x \mapsto v}] \qquad \delta \in \mathcal{G}_{\overline{x:B}}[\![\overline{x : B[= x]}]\!]}{(\tilde{\lambda}(\overline{x : B}).\ \dot{t})\ [\ \overline{v}\ ] \longrightarrow \|\omega\|\delta\dot{t}}$$

Figure 6: Terminal reduction

$$\begin{aligned}
&\mathcal{C}[\![Bool]\!] \doteq \{true, false\}\\
&\mathcal{C}[\![Int]\!] \doteq \{0, 1, -1, 2, -2, \ldots\}\\
&\\
&\mathcal{V}[\![(\overline{x : B}) \Rightarrow A[\Xi]]\!] \doteq \{v \mid \overline{\forall c \in \mathcal{V}[\![B]\!]}.\ v\ [\ \overline{c}\ ] \in \mathcal{T}[\![A]\!]\}\\
&\mathcal{V}[\![x : S \to T]\!] \doteq \{v \mid \forall w \in \mathcal{V}[\![S]\!].\ v\ w \in \mathcal{T}[\![T]\!]\}\\
&\mathcal{V}[\![B]\!] \doteq \{c \mid c \in \mathcal{C}[\![B]\!]\}\\
&\\
&\mathcal{T}[\![T]\!] \doteq \{t \mid t \Downarrow v \in \mathcal{V}[\![T]\!]\}\\
&\\
&\mathcal{G}[\![\Gamma, x : T]\!] \doteq \{\gamma[x \mapsto v] \mid \gamma \in \mathcal{G}[\![\Gamma]\!] \wedge v \in \mathcal{V}[\![T]\!]\}\\
&\mathcal{G}[\![\emptyset]\!] \doteq \{\emptyset\}\\
&\\
&\mathcal{O}[\![\Omega, x : B]\!] \doteq \{\omega[x \mapsto c] \mid \omega \in \mathcal{O}[\![\Omega]\!] \text{ and } c \in \mathcal{V}[\![B]\!]\}\\
&\mathcal{O}[\![\emptyset]\!] \doteq \{\emptyset\}
\end{aligned}$$

Figure 7: Terminal logical relations

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 Ξ from type ascriptions. This is accomplished using an ambient substitution.

We will write $t \Uparrow$ to indicate that a term t diverges under terminal reduction, and write $t \Downarrow v$ 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

$$
\begin{array}{ll}
\|\omega\|\ t & = t \\
\|\omega\|\ \lessdot \cdots \omega \mapsto c_\omega \cdots \gtrdot & = c_\omega \\
\|\omega\|\ (\lambda x : \dot{S}.\dot{t}) & = (\lambda x : \|\omega\|\dot{S}.\|\omega\|\dot{t}) \\
\|\omega\|\ (\dot{t}\ \dot{t}) & = (\|\omega\|\dot{t})\ (\|\omega\|\dot{t}) \\
\|\omega\|\ \dot{t}\ [\ \overline{\dot{t}}\ ] & = \|\omega\|\dot{t}\ [\ \overline{\|\omega\|\dot{t}}\ ] \\
 & \\
\|\omega\|\ S & = S \\
\|\omega\|\ \dot{S} \to \dot{T} & = \|\omega\|\dot{S} \to \|\omega\|\dot{T} \\
\|\omega\|\ B[\Xi] & = B
\end{array}
$$

Figure 8: Ambient substitution

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 $\gamma \in \mathcal{G}[\![\Gamma]\!]$ we have $\gamma t \in \mathcal{T}[\![T]\!]$. As a consequence, all well-typed 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 Ω be an ambient environment. An ambient map $\lessdot \overline{\phi \mapsto c_\phi}^{\phi \in O[\![\Omega]\!]} \gtrdot$ 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* $\|\omega\|\ \lessdot \overline{\phi \mapsto c_\phi} \gtrdot$ performs such a resolution, yielding c_ω. More generally, a lifted term $\dot{t}$ 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 $\|\omega\|\dot{t}$ 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 an 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 $(\tilde{\lambda}(\overline{x : B}).\ \dot{t})\ [\ \overline{\|\omega\|\dot{c}}\ ] \Downarrow c_\omega$.

Under an ambient environment Ω, a term of the form $(\tilde{\lambda}(\overline{x : B}).\ \dot{t})\ [\ \overline{\dot{c}}\ ]$ can be thought of as a composition of the same sort as the example

$$h(a, b) = g(f_x(a, b), f_y(a, b))$$

described in *Section 3.3*, given the following identifications:

(1) The ambient environment Ω corresponds to the variables a and b.
(2) The base values $\overline{\dot{c}}$ correspond to the functions $f_x(a, b)$ and $f_y(a, b)$.
(3) The sfun $(\tilde{\lambda}(\overline{x : B}).\ \dot{t})$ corresponds to the function g.

$$\textsc{LRed-Context}\ \frac{\Omega \vdash \dot{t} \to \dot{t}'}{\Omega \vdash \dot{E}[\dot{t}] \to \dot{E}[\dot{t}']} \qquad \textsc{LRed-App}\ \frac{}{\Omega \vdash (\lambda x : \dot{S}.\dot{t})\ \dot{v} \to [x/\dot{v}]\dot{t}}$$

$$\textsc{LRed-SFun-App}\ \frac{\overline{(\tilde{\lambda}(\overline{x : B}).\ \dot{t})\ [\ \overline{\|\omega\|\dot{c}}\ ] \Downarrow c_\omega}^{\omega \in O[\![\Omega]\!]}}{\Omega \vdash (\tilde{\lambda}(\overline{x : B}).\ \dot{t})\ [\ \overline{\dot{c}}\ ] \to \lessdot \overline{\omega \mapsto c_\omega} \gtrdot}$$

Figure 9: Lifted reduction

We write $\Omega \vdash \dot{t} \Uparrow$ to indicate divergence and $\Omega \vdash \dot{t} \Downarrow \dot{v}$ to indicate convergence to a lifted value $\dot{v}$ under lifted reduction. *Theorem 4.2* tells us that lifted reduction properly characterizes terminal reduction.

Lemma 4.1. *If $\Omega \vdash \dot{t} \to \dot{t}'$ then for all $\omega \in O[\![\Omega]\!]$, $\|\omega\|\dot{t} \to^* \|\omega\|\dot{t}'$*

Proof. By induction on derivations of $\Omega \vdash \dot{t} \to \dot{t}'$. □

Theorem 4.2. *If $\Omega \vdash \dot{t} \Downarrow \dot{v}$ then for all $\omega \in O[\![\Omega]\!]$, $\|\omega\|\dot{t} \Downarrow \|\omega\|\dot{v}$.*

Proof. A simple corollary of *Lemma 4.1.* □

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 $\mathcal{K}$ of logical relations provides semantic interpretations in terms of ambient substitution for the various qualifiers. $\mathcal{X}[\![\Omega]\!]$ denotes the set of qualifier maps on ambient environment Ω. Because lifted reduction is performed under some ambient environment Ω, we parameterize our relations by ambient environments, which appear as the subscripts beneath $\mathcal{K}$, $\mathcal{V}$, $\mathcal{T}$, and $\mathcal{G}$.

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 Ω, a terminal base type B is interpreted equivalently to $B[\Xi]$ where Ξ 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 $S <: T$ 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 defers to lifted subtyping to ensure that the qualifiers of the result types respect the qualifier ordering.

Defined in *Figure 12*, a lifted subtyping judgment $\Omega \vdash \dot{S} <: \dot{T}$ means that under ambient environment Ω, the set of lifted values of type $\dot{S}$ is contained in the set of lifted values of type $\dot{T}$. 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

$$
\begin{array}{lcl}
\mathcal{X}[\![\Omega, x : B]\!] & \doteq & \{\Xi, q\ x \mid \Xi \in \mathcal{X}[\![\Omega]\!]\} \\
\mathcal{X}[\![\emptyset]\!] & \doteq & \{\emptyset\} \\
\\
\mathcal{K}_\Omega[\![\uparrow x]\!] & \doteq & \{\dot{c} \mid \forall \omega, \omega' \in O[\![\Omega]\!].(\ \omega(x) \leq \omega'(x) \wedge \forall y \in dom(\Omega) - \{x\}.\ \omega(y) = \omega'(y)\ ) \implies \|\omega\|\dot{c} \leq \|\omega'\|\dot{c}\} \\
\mathcal{K}_\Omega[\![\downarrow x]\!] & \doteq & \{\dot{c} \mid \forall \omega, \omega' \in O[\![\Omega]\!].(\ \omega(x) \leq \omega'(x) \wedge \forall y \in dom(\Omega) - \{x\}.\ \omega(y) = \omega'(y)\ ) \implies \|\omega\|\dot{c} \geq \|\omega'\|\dot{c}\} \\
\mathcal{K}_\Omega[\![\sim x]\!] & \doteq & \{\dot{c} \mid \forall \omega, \omega' \in O[\![\Omega]\!].(\forall y \in dom(\Omega) - \{x\}.\ \omega(y) = \omega'(y)) \implies \|\omega\|\dot{c} = \|\omega'\|\dot{c}\} \\
\mathcal{K}_\Omega[\![= x]\!] & \doteq & \{\dot{c} \mid \forall \omega \in O[\![\Omega]\!].\ \|\omega\|\dot{c} = \omega(x)\} \\
\mathcal{K}_\Omega[\![? x]\!] & \doteq & \{\dot{c} \mid true\} \\
\\
\mathcal{V}_\Omega[\![(\overline{x_i : B_i}^{i \in 1..n}) \Rightarrow A[\Xi]]\!] & \doteq & \{\dot{v} \mid \overline{\forall \Xi_i \in \mathcal{X}[\![\Omega]\!].\ \forall \dot{c}_i \in \mathcal{V}_\Omega[\![B_i[\Xi]]\!].\ \dot{v}\ [\ \overline{\dot{c}_i}\ ] \in \mathcal{T}_\Omega[\![A[(\overline{\textstyle\sum_{i=1}^{n} \Xi_i(z) \circ \Xi(x_i))\ z}^{z \in dom(\Omega)}]]\!]}\} \\
\mathcal{V}_\Omega[\![\dot{S} \to \dot{T}]\!] & \doteq & \{\dot{v} \mid \forall \dot{w} \in \mathcal{V}_\Omega[\![\dot{S}]\!].\ \dot{v}\ \dot{w} \in \mathcal{T}_\Omega[\![\dot{T}]\!]\} \\
\mathcal{V}_\Omega[\![B[\Xi]]\!] & \doteq & \{\dot{c} \mid (\forall \omega \in O[\![\Omega]\!].\ \|\omega\|\dot{c} \in \mathcal{V}[\![B]\!]) \wedge (\dot{c} \in \textstyle\bigcap_{x \in dom(\Omega)} \mathcal{K}_\Omega[\![\Xi(x)\ x]\!])\} \\
\mathcal{V}_\Omega[\![B]\!] & \doteq & \{\dot{c} \mid (\forall \omega \in O[\![\Omega]\!].\ \|\omega\|\dot{c} \in \mathcal{V}[\![B]\!]) \wedge (\dot{c} \in \textstyle\bigcap_{x \in dom(\Omega)} \mathcal{K}_\Omega[\![\sim x]\!])\} \\
\\
\mathcal{T}_\Omega[\![\dot{T}]\!] & \doteq & \{\dot{t} \mid \Omega \vdash \dot{t} \Downarrow \dot{v} \in \mathcal{V}_\Omega[\![\dot{T}]\!]\} \\
\\
\mathcal{G}_\Omega[\![\Delta, x : \dot{T}]\!] & \doteq & \{\delta[x \mapsto \dot{v}] \mid \delta \in \mathcal{G}_\Omega[\![\Delta]\!] \wedge \dot{v} \in \mathcal{V}_\Omega[\![\dot{T}]\!]\} \\
\mathcal{G}_\Omega[\![\emptyset]\!] & \doteq & \{\emptyset\}
\end{array}
$$

Figure 10: Lifted logical relations

$$\text{S-SFun}\ \frac{\overline{x : B} \vdash A_1[\Xi_1] <: A_2[\Xi_2]}{(\overline{x : B}) \Rightarrow A_1[\Xi_1] <: (\overline{x : B}) \Rightarrow A_2[\Xi_2]} \qquad \text{S-Fun}\ \frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2}$$

$$\text{S-Base}\ \frac{}{B <: B}$$

Figure 11: Terminal subtyping

$$\text{LS-SFun}\ \frac{\overline{x : B} \vdash \dot{A}_1[\Xi_1] <: \dot{A}_2[\Xi_2]}{\Omega \vdash (\overline{x : B}) \Rightarrow A_1[\Xi_1] <: (\overline{x : B}) \Rightarrow A_2[\Xi_2]}$$

$$\text{LS-Fun}\ \frac{\Omega \vdash \dot{T}_1 <: \dot{S}_1 \qquad \dot{S}_2 <: \dot{T}_2}{\Omega \vdash \dot{S}_1 \to \dot{S}_2 <: \dot{T}_1 \to \dot{T}_2} \qquad \text{LS-Base-TT}\ \frac{}{\Omega \vdash B <: B}$$

$$\text{LS-Base-TL}\ \frac{\overline{\sim \leq q_x}}{\Omega \vdash B <: B[\overline{q_x\ x}^{x \in dom(\Omega)}]} \qquad \text{LS-Base-LT}\ \frac{\overline{q_x \leq \sim}}{\Omega \vdash B[\overline{q_x\ x}^{x \in dom(\Omega)}] <: B}$$

$$\text{LS-Base-LL}\ \frac{\overline{p_x \leq q_x}}{\Omega \vdash B[\overline{p_x\ x}^{x \in dom(\Omega)}] <: B[\overline{q_x\ x}]}$$

Figure 12: Lifted subtyping

Ξ qualifies every ambient variable as $\sim$, B and $B[\Xi]$ are isomorphic with respect to subtyping; i.e., $\Omega \vdash B <: B[\Xi]$ and $\Omega \vdash B[\Xi] <: B$.

$$
\begin{array}{ll}
ty(0), ty(-1), ty(1), \ldots & = Int \\
ty(true), ty(false) & = Bool
\end{array}
$$

Figure 13: Constant types metafunction

$$\text{T-Const}\ \frac{}{\Gamma \vdash c : ty(c)} \qquad \text{T-Var}\ \frac{x \in dom(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \qquad \text{T-App}\ \frac{\Gamma \vdash t : S \to T \qquad \Gamma \vdash s : S}{\Gamma \vdash t\ s : T}$$

$$\text{T-SFun-App}\ \frac{\Gamma \vdash u : (\overline{x_i : A_i}) \Rightarrow B[\Xi] \qquad \overline{\Gamma \vdash s_i : A_i}}{\Gamma \vdash u\ [\ \overline{s_i}^{i \in 1..n}\ ] : B} \qquad \text{T-Fun}\ \frac{\Gamma, x : S \vdash t : T}{\Gamma \vdash (\lambda x : S.t) : S \to T}$$

$$\text{T-SFun}\ \frac{\Gamma; \overline{x_i : B_i}; \overline{x_i : B_i[= x_i]} \vdash \dot{u} : A[\Xi]}{\Gamma \vdash (\tilde{\lambda}(\overline{x_i : B_i}^{i \in 1..n}).\ \dot{u}) : (\overline{x_i : B_i}) \Rightarrow A[\Xi]} \qquad \text{T-Sub}\ \frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T}$$

Figure 14: Terminal typing

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 $\Omega \vdash \dot{S} <: \dot{T}$ then $\mathcal{V}_\Omega[\![\dot{S}]\!] \subseteq \mathcal{V}_\Omega[\![\dot{T}]\!]$.*

Proof. By induction on the derivation of $\Omega \vdash \dot{S} <: \dot{T}$. □

$$\text{LT-Const}\ \frac{}{\Gamma;\Omega;\Delta \vdash c : ty(c)} \qquad \text{LT-Var-Gamma}\ \frac{x \in dom(\Gamma)}{\Gamma;\Omega;\Delta \vdash x : \Gamma(x)} \qquad \text{LT-Var-Delta}\ \frac{x \in dom(\Delta)}{\Gamma;\Omega;\Delta \vdash x : \Delta(x)}$$

$$\text{LT-App}\ \frac{\Gamma;\Omega;\Delta \vdash \dot{t} : \dot{S} \to \dot{T} \qquad \Gamma;\Omega;\Delta \vdash \dot{s} : \dot{S}}{\Gamma;\Omega;\Delta \vdash \dot{t}\ \dot{s} : \dot{T}}$$

$$\text{LT-SFun-App}\ \frac{\Gamma;\Omega;\Delta \vdash \dot{u} : (\overline{x_i : B_i}) \Rightarrow A[\overline{p_i\ x_i}] \qquad \overline{\Gamma;\Omega;\Delta \vdash \dot{s}_i : B_i[\overline{q_{iz}\ z}]}}{\Gamma;\Omega;\Delta \vdash \dot{u}\ [\ \overline{\dot{s}_i}^{\,i\in 1..n}\ ] : A[\overline{(\Sigma_{i=1}^{n} q_{iz} \circ p_i)\ z}^{\,z\in dom(\Omega)}]}$$

$$\text{LT-Fun}\ \frac{\Gamma;\Omega;\Delta, x : \dot{S} \vdash \dot{t} : \dot{T} \qquad \Omega \vdash \dot{S}}{\Gamma;\Omega;\Delta \vdash (\lambda x : \dot{S}.\dot{t}) : \dot{S} \to \dot{T}} \qquad \text{T-Sub}\ \frac{\Gamma;\Omega;\Delta \vdash \dot{t} : \dot{S} \qquad \Omega \vdash \dot{S} <: \dot{T}}{\Gamma;\Omega;\Delta \vdash \dot{t} : \dot{T}}$$

Figure 15: Lifted typing

4.11 Typing

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 Γ and a lifted type environment Δ. Because every terminal value is also a lifted value, substituting terminal values for the variables of Γ in $\dot{t}$ 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 LT-SFun-App, for typing an sfun application $\dot{u}\ [\ \overline{\dot{s}_i}^{\,i\in 1..n}\ ]$. The first premise requires $\dot{u}$ to have an n-ary sfun type $(\overline{x_i : A_i}) \Rightarrow B[\overline{p_i\ x_i}]$. The other n premises require that each argument $\dot{s}_i$ has a lifted base type of the form $A_i[\overline{q_{iz}\ z}^{\,z\in dom(\Omega)}]$, where $\overline{q_{iz}\ z}$ is a qualifier map synthesized from type checking $\dot{s}_i$. The sfun application $\dot{u}\ [\ \dot{s}_i\ ]$ then has the type $B[\overline{\sum_{i=1}^{n} q_{iz} \circ p_i}]$, where $\circ$ is the operator defined in *Figure 3* and the "summation" $\Sigma_{i=1}^{n} \cdots$ uses the qualifier contraction operator $+$ defined in *Figure 4*.

LT-Fun requires a lifted type well-formedness derivation of $\Omega \vdash \dot{S}$, which ensures that all qualifiers in $\dot{S}$ are contained in the ambient environment Ω. 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.

Theorem 4.5 (Fundamental terminal typing theorem). *For all ambient environments Ω, if $\Gamma \vdash t : T$ then for all $\gamma \in \mathcal{G}[\![\Gamma]\!] \cap \mathcal{G}_\Omega[\![\Gamma]\!]$, $\gamma t \in \mathcal{T}[\![T]\!] \cap \mathcal{T}_\Omega[\![T]\!]$.*

Proof. By induction on the derivation of $\Gamma \vdash t : T$. □

Theorem 4.6 (Fundamental lifted typing theorem). *If $\Gamma;\Omega;\Delta \vdash \dot{t} : \dot{T}$ then for all $\gamma \in \mathcal{G}_\Omega[\![\Gamma]\!]$ and all $\delta \in \mathcal{G}_\Omega[\![\Delta]\!]$, $\gamma\delta\dot{t} \in \mathcal{T}_\Omega[\![\dot{T}]\!]$.*

Proof. By induction on the derivation of $\Gamma;\Omega;\Delta \vdash \dot{t} : \dot{T}$. □

The intersections in *Theorem 4.5* are necessary to allow the substitution of terminal values through sfun abstractions.

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 $(\tilde{\lambda}(\overline{x_i : B_i}^{\,i\in 1..n}).\ \dot{t})\ [\ \overline{\dot{c}_i}\ ]$. 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 $(\tilde{\lambda}(\overline{x_i : B_i}).\ \dot{t})$ extensionally. Because $(\tilde{\lambda}(\overline{x_i : B_i}).\ \dot{t})$ is well-typed we might invoke an inversion lemma to get $\emptyset;\Omega;\emptyset \vdash (\tilde{\lambda}(\overline{x_i : B_i}).\ \dot{t}) : (\overline{x_i : B_i}) \Rightarrow A[\Xi]$ and $\emptyset;\overline{x_i : B_i};\overline{x_i : B_i[=\ x_i]} \vdash \dot{t} : \dot{A}$. A substitution lemma may tell us that for $\delta \in \mathcal{G}_{\overline{x_i:B_i}}[\![\overline{x_i : B_i[=\ x_i]}]\!]$ we have $\emptyset;\overline{x_i : B_i};\emptyset \vdash \delta\dot{t} : \dot{A}$. 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 $\overline{x_i : B_i} \vdash \delta\dot{t} \to \dot{t}'$ for some $\dot{t}'$ where $\emptyset;\overline{x_i : B_i};\emptyset \vdash \dot{t}' : \dot{A}$, 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

Sfuns 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

$$(x : Int, y : Int) \Rightarrow Int[\uparrow x, \uparrow y]$$

and a subtraction operator with type

$$(x : Int, y : Int) \Rightarrow Int[\uparrow x, \downarrow y]$$

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

$$\mathcal{T}[\![T]\!] \doteq \{t \mid t \Uparrow \text{ or } t \Downarrow v \text{ where } v \in \mathcal{V}[\![T]\!]\}$$

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 aformentioned 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 *Bool* or *Int*, and t is a term of type *Bool*. Such a type describes the set of constants c of base type B such that $[c/v]t \Downarrow true$. Standard function types give way to dependent function types, which have the form $x : S \to T$. The variable x, which is scoped over type T, refers to the value to which a function of type $x : S \to T$ is applied.

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

$$\textsc{Sub-Base}\quad \frac{\forall \gamma \in \mathcal{G}[\![\Gamma, v : B]\!].\gamma s \Downarrow true \implies \gamma t \Downarrow true}{\Gamma \vdash \{v : B \mid s\} <: \{v : B \mid t\}}$$

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:

$$\mathcal{G}[\![\Gamma, x : T]\!] \doteq \{\gamma[x \mapsto v] \mid \gamma \in \mathcal{G}[\![\Gamma]\!] \text{ and } v \in \mathcal{V}[\![\gamma T]\!]\}$$

Our fundamental terminal subtyping and typing theorems would also change:

Theorem 5.1 (Dependent terminal subtyping theorem). *If* $\Gamma \vdash S <: T$ *then for all* $\gamma \in \mathcal{G}[\![\Gamma]\!]$, $\mathcal{V}[\![\gamma S]\!] \subseteq \mathcal{V}[\![\gamma T]\!]$.

Theorem 5.2 (Dependent terminal typing theorem). *If* $\Gamma \vdash t : T$ *then for all ambient environments* Ω *with* $FV(\Omega) = \emptyset$ *and all* $\gamma \in \mathcal{G}[\![\Gamma]\!] \cap \mathcal{G}_\Omega[\![\Gamma]\!]$, *we have* $\gamma t \Downarrow v$ *and* $v \in \mathcal{V}[\![\gamma T]\!] \cap \mathcal{V}_\Omega[\![\gamma T]\!]$.

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.

```
fun f(metric : Nat, n : Int,
      i : {v : Int | v ≤ n ∧ (metric = n-i)}) =
  if i = n then
    i
  else
    let metric' = n - (i+1) in
    i + f metric' n (i+1)
```

Figure 16: Decreasing, well founded termination metric

The fundamental lifted subtyping and typing theorems would be revised:

Theorem 5.3 (Dependent lifted subtyping theorem). *If* $\Gamma; \Omega; \Delta \vdash \dot{S} <: \dot{T}$ *then for all* $\gamma \in \mathcal{G}[\![\Gamma]\!] \cap \mathcal{G}_{\gamma\Omega}[\![\Gamma]\!]$ *and* $\delta \in \mathcal{G}_{\gamma\Omega}[\![\gamma\Delta]\!]$ *we have* $\mathcal{V}_{\gamma\Omega}[\![\gamma\delta\dot{S}]\!] \subseteq \mathcal{V}_{\gamma\Omega}[\![\gamma\delta\dot{T}]\!]$.

Theorem 5.4 (Dependent lifted typing theorem). *If* $\Gamma; \Omega; \Delta \vdash \dot{t} : \dot{T}$ *then for all* $\gamma \in \mathcal{G}[\![\Gamma]\!] \cap \mathcal{G}_{\gamma\Omega}[\![\Gamma]\!]$ *and* $\delta \in \mathcal{G}_{\gamma\Omega}[\![\gamma\Delta]\!]$ *we have* $\gamma\delta\dot{t} \in \mathcal{T}_{\gamma\Omega}[\![\gamma\delta\dot{T}]\!]$.

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 Nat as a shorthand for $\{v : Int \mid 0 \le v\}$, *Figure 16* demonstrates a function with such a termination metric. The function f, which given integers n and i with $i \le n$ 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:

$$m' : \{v : Nat \mid v < metric\}$$
$$\to n : Int \to i : \{v : Int \mid (v \le n) \ \wedge \ (m' = n - i)\}$$

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

LiquidHaskell treats the terminating fixpoint combinator $tfix$ 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 $\langle x < t \rangle\ T$ with the following introduction form

$$(\textbf{tfix}\ t\ x\ T\ y\ s)$$

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 $u\ \langle s \rangle$ 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 *IntArray*, an array of integers ordered componentwise. The sfun *getAt*, for getting the element at index i of array a, is monotone with respect to a because *IntArray* is ordered

RED-TFIX

$$(\textbf{tfix}\ v\ x\ T\ y\ s)\ \langle v' \rangle \longrightarrow [v'/x][(\textbf{tfix}\ v'\ x\ T\ y\ s)/y]s$$

T-TFIX-ABS

$$\frac{\Gamma \vdash t : Nat \qquad \Gamma, x : \{v : Nat \mid v < t\} \vdash T \qquad \Gamma, x : \{v : Nat \mid v < t\}, y : \langle x' < x \rangle\ [x'/x]T \vdash s : T}{\Gamma \vdash (\textbf{tfix}\ t\ x\ T\ y\ s) : \langle x < t \rangle\ T}$$

T-TFIX-APP

$$\frac{\Gamma \vdash u : \langle x < t \rangle\ U \qquad \Gamma \vdash s : \{v : Nat \mid v < t\}}{\Gamma \vdash u\ \langle s \rangle : [s/x]U}$$

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 *IntArray* are related. The sfun *setAt* sets element i of *IntArray* a to the integer v. It is monotone with respect to a and v due to *IntArray*'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 *len* function gets the length of an *IntArray* and the *makeArray* function creates a new array with all elements initialized to 0.

The nested function *sumCell* uses terminating recursion to iterate through the indices of the two arrays.

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

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

the type of the *IntArray* returned from makeArray on line 19 is subsumed by

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

Assuming that each call to *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

[1] Michael Arntzenius and Neelakantan R Krishnaswami. Datafun: a functional datalog. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, pages 214–227. ACM, 2016.

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

[3] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

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

[5] Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for haskell. In *ACM SIGPLAN Notices*, volume 49, pages 269–282. ACM, 2014.

[6] Noam Zeilberger. Principles of type refinement. http://noamz.org/oplss16/refinements-notes.pdf [Online; accesed 29-April-2017], 2016.