Semantic Subtyping with an SMT Solver

Gavin M. Bierman Andrew D. Gordon Cătălin Hriţcu

David Langworthy

Revision of December 2011

Technical Report
MSR–TR–2010–99

Microsoft Research
Roger Needham Building
7 J.J. Thomson Avenue
Cambridge, CB3 0FB
United Kingdom

Publication History

An abridged version of this paper appears in the proceedings of the ACM SIGPLAN International Conference on Functional Programming, held in Baltimore, Maryland, USA, on September 27–29, 2010.

Contents

Semantic Subtyping with an SMT Solver

Gavin M. Bierman
Andrew D. Gordon
Microsoft Research

Cătălin Hriţcu
Saarland University and
University of Pennsylvania

David Langworthy
Microsoft Corporation

Abstract

We study a first-order functional language with the novel combination of the ideas of refinement type (the subset of a type to satisfy a Boolean expression) and type-test (a Boolean expression testing whether a value belongs to a type). Our core calculus can express a rich variety of typing idioms; for example, intersection, union, negation, singleton, nullable, variant, and algebraic types are all derivable. We formulate a semantics in which expressions denote terms, and types are interpreted as first-order logic formulas. Subtyping is defined as valid implication between the semantics of types. The formulas are interpreted in a specific model that we axiomatize using standard first-order theories. On this basis, we present a novel type-checking algorithm able to eliminate many dynamic tests and to detect many errors statically. The key idea is to rely on an SMT solver to compute subtyping efficiently. Moreover, using an SMT solver allows us to show the uniqueness of normal forms for non-deterministic expressions, to provide precise counterexamples when type-checking fails, to detect empty types, and to compute instances of types statically and at run-time.

1. Introduction

This paper studies first-order functional programming in the presence of both refinement types (types qualified by Boolean expressions) and type-tests (Boolean expressions testing whether a value belongs to a type). The novel combination of type-test and refinement types appears in a recent commercial functional language, code-named M [69], whose types correspond to relational schemas, and whose expressions compile to SQL queries. Refinement types are used to express SQL table constraints within a type system, and type-tests are useful for processing relational data, for example, by discriminating dynamically between different forms of union types. Still, although useful and extremely expressive, the combination of type-test and refinement is hard to type-check using conventional syntax-driven subtyping rules. The preliminary implementation of M uses such subtyping rules and has difficulty with certain sound idioms (such as uses of singleton and union types). Hence, type safety is enforced by dynamic checks, or not at all.

This paper studies the problem of type-checking code that uses type-tests and refinements via a core calculus, named Dminor, whose syntax is a small subset of M, and which is expressive enough to encode all the essential features of the full M language. In the remainder of this section, we elaborate on the difficulties of type-checking Dminor (and hence M), and outline our solution, which is to use semantic subtyping rather than syntactic rules.

1.1 Programming with Type-Test and Refinement

The core types of Dminor are structural types for scalars, unordered collections, and records. (Following the database orientation of M, we refer to records as *entities*.) We write $S <: T$ for the subtype relation, which means that every value of type S is also of type T.

Two central primitives of Dminor are the following:

- A *refinement type*, $(x : T$ **where** $e)$, consists of the values x of T satisfying the Boolean expression e.
- A *type-test expression*, e **in** T, returns **true** or **false** depending on whether or not the value of e belongs to type T.

As we shall see, many types are derivable from these primitive constructs and their combination. For example, the singleton type $[v]$, which contains just the value v, is derived as the refinement type $(x :$ Any **where** $x == v)$, where Any is the type of all values. The union type $T \mid U$, which contains the values of T together with the values of U, is derived as $(x :$ Any **where** $(x$ **in** $T) \mid\mid (x$ **in** $U))$.

Here is a snippet from a typical Dminor (and M) program for processing a DSL, a language of while-programs. The type is a union of different sorts of statements, each of which is an entity with a kind field of singleton type. (The snippet relies on an omitted—but similar—recursive type of arithmetic expressions.)

```
type Statement =
    {kind:["assignment"]; var: Text; rhs: Expression;} |
    {kind:["while"]; test:Expression; body:Statement;} |
    {kind:["if"]; test:Expression; tt:Statement; ff:Statement;} |
    {kind:["seq"]; s1:Statement; s2:Statement;} |
    {kind:["skip"];};
```

In languages influenced by HOPE [24], such as ML and Haskell, we would use the built-in notion of algebraic type to represent such statements. But like many data formats, including relational databases, semi-structured data, S-expressions, and JavaScript Object Notation (JSON) [30], the data structures of M and Dminor do not take as primitive the idea of data tagged with data constructors. Instead, we need to follow an idiom such as shown above, of taking the union of entity types that include explicit tags that are given distinct singleton types.

If y has type Statement, we may process such data as follows:

$$((y.\mathsf{kind} == \texttt{"assignment"})?\ y.\mathsf{var} : \texttt{"NotAssign"})$$

Intuitively, this code is type-safe because it checks the kind field before accessing the var field, which is only present for assignment statements. More precisely, to type-check the then-branch y.var at type Text, we have y : Statement (that is, a union type encoded using refinements and type-test, which after expansion has the form $(x :$ Any **where** $\ldots \mid\mid \ldots))$, know that y.kind $==$ "assignment", and need to decide $[y] <: \{$var : Text;$\}$. Subtyping should succeed, but clearly requires relatively sophisticated symbolic computation, including case analysis and propagation of equations. This is a typical example where syntax-driven rules for refinements and type-test are inadequate (if one ignores the refinement in the definition of Statement, then Any is not a subtype of $\{$var : Text;$\}$), and indeed this simple example cannot be checked statically by the preliminary release of M. Our proposal is to delegate the hard work to an external prover.

1.2 An Opportunity: SMT as a Platform

Over the past few years, there has been tremendous progress in the field of Satisfiability Modulo Theories (SMT), that is, for (fragments of) first-order logic plus various standard theories such as equality, real and integer (linear) arithmetic, bit vectors, and (extensional) arrays. Some of the leading systems include CVC3 [11], Yices [38], and Z3 [33]. There are common input formats such as Simplify's [35] unsorted S-expression syntax and the SMT-LIB standard [75] for sorted logic. Hence, first-order logic with standard theories is emerging as a computing platform. Software written to generate problems in a standard format can rely on a wide range of back-end solvers, which get better over time due in part to healthy competition,[1] and which may even be run in parallel when sufficient cores are available. There are limitations, of course, as first-order validity is undecidable even without any theories, so solvers may fail to terminate within a reasonable time, but recent progress has been remarkable.

1.3 Semantic Subtyping with an SMT Solver

The central idea in this paper is a type-checking algorithm for Dminor that checks subtyping by invoking an external SMT solver. To check whether S is a subtype of T, we construct first-order formulas $\mathbf{F}[\![S]\!](x)$ and $\mathbf{F}[\![T]\!](x)$, which hold when x belongs to the type S and the type T, respectively, and ask the solver whether the formula $\mathbf{F}[\![S]\!](x) \implies \mathbf{F}[\![T]\!](x)$ is valid, given any additional constraints known from the typing environment. This technique is known as *semantic subtyping* [2, 43], as opposed to the more common alternative, *syntactic subtyping*, which is to define syntax-driven rules for checking subtyping [72].

The idea of using an external solver for type-checking with refinement types is not new. Several recent type-checkers for functional languages, such as SAGE [41, 57, 58], F7 [12], Fine [82], and Dsolve [77], rely on various SMT solvers. However, these systems all rely on syntactic subtyping, with the solver being used as a subroutine to check constraints during subtyping.

To the best of our knowledge, our proposal to implement semantic subtyping by calling an external SMT solver is new. Semantic subtyping nicely exploits the solver's ability to handle logical connectives efficiently; for example, we represent union and intersection types as logical disjunctions and conjunctions. Hence, we avoid the implementation effort of explicit propagation of constraints, and of syntax-driven rules for union and intersection types [71, 37, 36]. Moreover, we exploit the theories of equality, integer arithmetic, extensional arrays [34], and algebraic datatypes.

1.4 Contributions of the Paper

(1) Investigation of semantic subtyping for a core functional language with both refinement types and type-test expressions (a novel combination, as far as we know). We are surprised that so many typing constructs are derivable from this combination.

(2) Development of the theory, including both a declarative type assignment relation, and algorithmic rules in the bidirectional style. Our correctness results cover the core type assignment relation, the bidirectional rules, the algorithmic purity check, and some logical optimizations.

(3) An implementation based on checking semantic subtyping by constructing proof obligations for an external SMT solver. The proof obligations are interpreted in a model that is formalized in Coq and axiomatized using standard first-order theories (equality, integers, datatypes and extensional arrays).

(4) Devising a systematic way to use the SMT solver in order to show the uniqueness of normal forms for non-deterministic expressions, to provide precise counterexamples when type-checking fails, to detect empty types, and to compute instances of types. The latter enables a new form of declarative constraint programming, where constraints arise from the interpretation of a type as a formula.

[1] Most important is the SMT-COMP [10] competition held each year in conjunction with CAV and in which more than a dozen SMT solvers contend.

1.5 Structure of the Paper

§2 describes the formal syntax of Dminor together with a small-step operational semantics, $e \rightarrow e'$, where e and e' are expressions. We encode a series of type idioms to illustrate the expressiveness of the language and its type system.

§3 presents a logical semantics of pure expressions (those without side-effects, including non-termination) and Dminor types. We require that expressions used as refinements be pure, so that they have a direct interpretation as predicates. Each pure expression e is interpreted as a term $\mathbf{R}[\![e]\!]$ and each type T is interpreted as a first-order logic formula $\mathbf{F}[\![T]\!](t)$, where t is a FOL term. The formulas are interpreted in a specific model that we have formalized in Coq. Theorem 1 is a full abstraction result: two pure expressions have the same logical semantics just when they are operationally equivalent.

§4 introduces a tractable property, *algorithmic purity*, for use in our typing rules. Algorithmic purity is defined using a syntactic termination restriction together with a confluence check that relies on the logical semantics. Theorem 2 shows that our algorithmic purity check is indeed a sufficient condition for purity.

§5 presents the declarative type system for Dminor. The type assignment relation has the form $E \vdash e : T$, meaning that expression e has type T given typing environment E. Theorem 3 concerns logical soundness of type assignment; if e is assigned type T then formula $\mathbf{F}[\![T]\!](\mathbf{R}[\![e]\!])$ holds. Progress and preservation results (Theorems 4 and 5) relate type assignment to the operational semantics, entailing that well-typed expressions cannot go wrong.

§6 develops additional theory to justify our implementation techniques. First, we present simpler variations of the translations $\mathbf{R}[\![e]\!]$ and $\mathbf{F}[\![T]\!](t)$, optimized by the observation that during type-checking we only interpret well-typed expressions, and so we need not track error values. Theorem 6 shows the soundness and completeness of this optimization. Second, since the declarative rules of §5 are not directly algorithmic, we propose type checking and synthesis algorithms, presented as bidirectional rules. Theorem 7 shows these are sound with respect to type assignment.

§7 shows how to use the models produced by the SMT solver to provide very precise counterexamples when type-checking fails and to find inhabitants of types statically or dynamically. §8 reports some details of our implementation. We survey related work in §9, before concluding in §10.

The appendixes describe our intended logical model of Dminor and its formalization in Coq (Appendix A); report on the axiomatization of the model passed to the SMT solver during type-checking (Appendix B); provide detailed proofs (Appendix C); and describe how our type-checker may be used to check for systems configuration errors (Appendix D).

Our implementation, as well as sample code and listings of Dminor runs, the Coq formalization of our model, and also a screencast comparing the effectiveness of Dminor with the standard M type-checker, are all available at `http://research.microsoft.com/dminor/`. A preliminary abridged version of this work appears in a conference proceedings [16]. Although we have formalized our logical model, and some other definitions, in Coq, our proofs are not in general mechanized in Coq. We list the theorems proved in Coq in §8.

Finally, we report that the future of the M language, the inspiration for Dminor, is rather uncertain at present. In a September 2010 blog posting [19], Microsoft announced that prototype software based on M would not be brought to market. Whatever the future of the M language itself, our hope is that the concepts we have developed in Dminor will be valuable in other settings.

2. Syntax and Operational Semantics

Dminor is a strict first-order functional language whose data includes scalars, entities, and collections; it has no mutable state, and its only side-effects are non-termination and non-determinism. This section describes: (1) the syntax of expressions, types, and global function definitions; (2) the operational semantics; (3) the definition of pure expressions (those without side-effects); and (4) some encodings to justify our expressiveness claims.

The following example introduces the basic syntax of Dminor. An accumulate expression is a fold over an unordered collection; to evaluate **from** x **in** e_1 **let** $y = e_2$ **accumulate** e_3, we first evaluate e_1 to a collection v, evaluate e_2 to an initial value u_0, and then compute a series of values u_i for $i \in 1..n$, by setting u_i to the value of $e_3\{v_i/x\}\{u_{i-1}/y\}$, and eventually return u_n, where $v_1, \ldots, v_n$ are the items in the collection v, in some arbitrary order.

```
NullableInt ≜ Integer | [null]

removeNulls(xs : NullableInt*) : Integer*
 { from x in xs let a = ({}:Integer*) accumulate (x!=null) ? (x :: a) : a }
```

The type NullableInt is defined as the union of Integer with the singleton type containing only the value **null**. The type Integer$*$ denotes a collection of values of type Integer. We then define a function removeNulls that iterates over its input collection and removes all null elements. As expected, executing removeNulls($\{1,$ **null**, $42,$ **null**$\}$) produces $\{1, 42\}$ (which denotes the same collection as $\{42, 1\}$).

Given that the collection xs contains elements of type NullableInt (xs : NullableInt$*$), that x is an element of xs, and the check that x != **null**, our type-checking algorithm infers that on the if branch x : Integer, and therefore the result of the comprehension is Integer$*$, as declared by the function. If we remove the check that x != **null**, and copy all elements with x :: a then type-checking fails, as expected.

2.1 Expressions and Types

We observe the following syntactic conventions. We identify all phrases of syntax (such as types and expressions) up to consistent renaming of bound variables. For any phrase of syntax ϕ we write $\phi\{e/x\}$ for the outcome of a capture-avoiding substitution of e for each free occurrence of x in ϕ. We write $fv(\phi)$ for the set of variables occurring free in ϕ.

We assume some base types for integers, strings, and logical values, together with constants for each of these types, as well as a **null** value. We also assume an assortment of primitive operators; they are all binary apart from negation !, which is unary.

Scalar Types, Constants, and Operators:

$G ::=$ Integer $\mid$ Text $\mid$ Logical	scalar type
$K(\text{Integer}) = \{\underline{i} \mid \text{integer } i\}$	
$K(\text{Text}) = \{\underline{s} \mid \text{string } s\}$	
$K(\text{Logical}) = \{\textbf{true}, \textbf{false}\}$	
$c \in K(\text{Integer}) \cup K(\text{Text}) \cup K(\text{Logical}) \cup \{\textbf{null}\}$	scalar constants
$\oplus \in \{+, -, \times, <, >, ==, !, \&\&, \|\|\}$	primitive operators

A *value* may be a *simple value* (an integer, string, boolean, or **null**), a *collection* (a finite multiset of values), or an *entity* (a finite set of fields, each consisting of a value with a distinct label). (We follow M terminology, but entities would usually be called records.)

Syntax of Values:

$v ::=$	value
c	scalar (or simple value)
$\{v_1, \ldots, v_n\}$	collection (multiset; unordered)
$\{\ell_i \Rightarrow v_i \; ^{i \in 1..n}\}$	entity (ℓ_i distinct)

We identify values u and v, and write $u = v$, when they are identical up to reordering the items within collections or entities. While collections are unordered, ordered lists can be encoded using nested entities (see §2.4).

Syntax of Types:

$S, T, U ::=$	type
Any	the top type
G	scalar type
$T*$	collection type
$\{\ell\colon T\}$	(single) entity type
$(x : T$ **where** $e)$	refinement type (scope of x is e)

All values have type Any, the top type. The values of a scalar type G are the scalars in the set $K(G)$ defined above. The values of type $T*$ are collections of values of type T. The values of type $\{\ell\colon T\}$ are entities with (at least) a field ℓ holding values of type T. (We show in §2.4 how to define multi-field entity types as a form of intersection type.) Finally, the values of a *refinement type* $(x : T$ **where** $e)$ are the values v of type T such that the boolean expression $e\{v/x\}$ returns **true**. As a convenient shorthand, we write T **where** e for the refinement type (value : T **where** e), where the omitted variable defaults to value. For example, Integer **where** value > 0 is the type of positive numbers.

Syntax of Expressions:

$e ::=$	expression
x	variable
c	scalar constant
$\oplus(e_1, \ldots, e_n)$	operator application
$e_1 ? e_2 : e_3$	conditional
let $x = e_1$ **in** e_2	let-expression (scope of x is e_2)
e **in** T	type-test
$\{\ell_i \Rightarrow e_i \; ^{i \in 1..n}\}$	entity (ℓ_i distinct)
$e.\ell$	field selection
$\{v_1, \ldots, v_n\}$	collection (multiset)
$e_1 :: e_2$	adding element e_1 to collection e_2
from x **in** e_1	iteration over collection
let $y = e_2$ **accumulate** e_3	(scope of x and y is e_3)
$f(e_1, \ldots, e_n)$	function application

Variables, constants, operators, conditionals, and let-expressions are standard. When $\oplus$ is binary, we often write $e_1 \oplus e_2$ instead of $\oplus(e_1, e_2)$. A *type-test*, e **in** T, returns a boolean to indicate whether or not the value of e inhabits the type T.

The accumulate primitive can encode all the usual operations on collections: counting the number of elements or the occurrences of a certain element, checking membership, removing duplicates and elements, multiset union and difference, as well as comprehensions in the style of the nested relational calculus [21]. (The form **bind** $x \leftarrow e_1$ **in** e_2 is the monadic bind for the multiset monad.)

Derived Collection Expressions:

$\{e_1, \ldots, e_n\} \triangleq e_1 :: \ldots :: e_n :: \{\}$

e.**Count** $\triangleq$ **from** x **in** e **let** $y = 0$ **accumulate** $y + 1$

e.**Count**$(e_2) \triangleq$
  let $z = e_2$ **in** (**from** x **in** e **let** $y = 0$ **accumulate** $(x == z) ? y + 1 : y)$

$e_1 \in e_2 \triangleq (e_2.\textbf{Count}(e_1) > 0)$

$e.\textbf{Distinct} \triangleq \textbf{from } x \textbf{ in } e \textbf{ let } y = \{\} \textbf{ accumulate } (x \in y)?y : (x :: y)$

$e.\textbf{Remove}(e_2) \triangleq \textbf{let } z = e_2 \textbf{ in}$
$\quad(\textbf{from } x \textbf{ in } e \textbf{ let } y = \{\mathsf{found} = \textbf{false}, \mathsf{res} = \{\}\}$
$\qquad\textbf{accumulate } (x == z \;\&\&\; !y.\mathsf{found})?\{\mathsf{found} = \textbf{true}, \mathsf{res} = y.\mathsf{res}\}$
$\qquad\qquad : \{\mathsf{found} = y.\mathsf{found}, \mathsf{res} = x :: y.\mathsf{res}\}$
$\quad).\mathsf{res}$

$e_1 \cup e_2 \triangleq \textbf{from } x \textbf{ in } e_1 \textbf{ let } y = e_2 \textbf{ accumulate } x :: y$

$e_1 \setminus e_2 \triangleq \textbf{from } x \textbf{ in } e_2 \textbf{ let } y = e_1 \textbf{ accumulate } y.\textbf{Remove}(x)$

$\textbf{bind } x \leftarrow e_1 \textbf{ in } e_2 \triangleq \textbf{from } x \textbf{ in } e_1 \textbf{ let } y = \{\} \textbf{ accumulate } e_2 \cup y$

In example code, we often rely on the following derived syntax for from-where-select expressions in LINQ style [66]. The expression **from** x **in** e_1 **where** e_2 **select** e_3 computes the collection e_1, and returns the collection of items e_3, for each member x of e_1 to satisfy the predicate e_2.

Derived LINQ Queries:

$\textbf{from } x \textbf{ in } e_1 \textbf{ where } e_2 \textbf{ select } e_3 \triangleq$
$\qquad\textbf{from } x \textbf{ in } e_1 \textbf{ let } y = \{\} \textbf{ accumulate } e_2?(e_3 :: y) : y$

To complete the syntax of Dminor, we interpret types and expressions in the context of a fixed collection of first-order, dependently-typed, potentially recursive function definitions. We assume for each expression $f(e_1,\dots,e_n)$ in a source program that there is a corresponding function definition for f with arity n.

Function Definitions: $f(x_1 : T_1,\dots,x_n : T_n) : U\{e\}$

We assume a finite, global set of *function definitions*, each of which associates a function name f with a dependent signature $x_1 : T_1,\dots,x_n : T_n \to U$, formal parameters $x_1,\dots,x_n$, and a body e, such that $fv(e) \subseteq \{x_1,\dots,x_n\}$ and $fv(U) \subseteq \{x_1,\dots,x_n\}$.

2.2 Operational Semantics

We define a nondeterministic, potentially divergent, small-step reduction relation $e \to e'$, together with a standard notion of expressions going wrong, to be prevented by typing.

Each primitive operator is a partial function represented by a set of mappings of the form $\oplus(v_1,\dots,v_n) \mapsto v_0$ where each v_i is a value. The $==$ operator implements syntactic equality, which for collections and entities is up to reordering of elements. Apart from $==$, the other operators only act on scalar values. For example, the equations for $+$ are $(\underline{i}+\underline{j}) \mapsto \underline{i+j}$. The other operators are defined by similar equations, and we omit the details.

Reduction Contexts:

$\mathscr{R} ::= \qquad$ reduction context
$\quad\oplus(v_1,\dots,v_{j-1},\bullet,e_{i+1},\dots,e_n)$
$\quad\bullet?e_2 : e_3 \mid \textbf{let } x = \bullet \textbf{ in } e_2 \mid \bullet \textbf{ in } T$
$\quad\{\ell_i \Rightarrow v_i\;{}^{i\in 1..j-1}, \ell_j \Rightarrow \bullet, \ell_i \Rightarrow e_i\;{}^{i\in j+1..n}\}$
$\quad\bullet.\ell \mid \bullet :: e \mid v :: \bullet \mid \textbf{from } x \textbf{ in } \bullet \textbf{ let } y = e_2 \textbf{ accumulate } e_3$
$\quad f(v_1,\dots,v_{j-1},\bullet,e_{i+1},\dots,e_n)$

Reduction Rules for Standard Constructs:

$e \to e' \implies \mathscr{R}[e] \to \mathscr{R}[e']$
$\oplus(v_1,\dots,v_n) \to v \quad$ if $\oplus(v_1,\dots,v_n) \mapsto v$ defined
$\textbf{true}?e_2 : e_3 \to e_2$
$\textbf{false}?e_2 : e_3 \to e_3$
$\textbf{let } x = v \textbf{ in } e_2 \to e_2\{v/x\}$
$\{\ell_i \Rightarrow v_i\;{}^{i\in 1..n}\}.\ell_j \to v_j \qquad$ where $j \in 1..n$
$v :: \{v_1,\dots,v_n\} \to \{v_1,\dots,v_n,v\}$
$\textbf{from } x \textbf{ in } \{v_1,\dots,v_n\} \textbf{ let } y = e_2 \textbf{ accumulate } e_3$
$\quad\to \textbf{let } y = e_2 \textbf{ in let } y = e_3\{v_1/x\} \textbf{ in} \dots \textbf{let } y = e_3\{v_n/x\} \textbf{ in } y$
$f(v_1,\dots,v_n) \to e\{v_1/x_1\}\dots\{v_n/x_n\}$
$\quad$given function definition $f(x_1 : T_1,\dots,x_n : T_n) : U\{e\}$

Reduction Rules for Type-Test:

$v \textbf{ in } \mathsf{Any} \to \textbf{true}$

$$v \textbf{ in } G \to \begin{cases} \textbf{true} & \text{if } v \in K(G) \\ \textbf{false} & \text{otherwise} \end{cases}$$

$$v \textbf{ in } \{\ell_j : T_j\} \to \begin{cases} v_j \textbf{ in } T_j & \text{if } v = \{\ell_i \Rightarrow v_i\;{}^{i\in 1..n}\} \wedge j \in 1..n \\ \textbf{false} & \text{otherwise} \end{cases}$$

$$v \textbf{ in } T* \to \begin{cases} v_1 \textbf{ in } T \;\&\&\dots\&\&\; v_n \textbf{ in } T & \text{if } v = \{v_1,\dots,v_n\} \\ \textbf{false} & \text{otherwise} \end{cases}$$

$v \textbf{ in } (x : T \textbf{ where } e) \to v \textbf{ in } T \;\&\&\; e\{v/x\}$

The reduction rules for type-test expressions, e **in** U, first reduce e to a value v and then proceed by case analysis on the structure of the type U. In case U is a refinement type $(x : T$ **where** $e)$ then v is a value of U if and only if v is a value of type T and $e\{v/x\}$ reduces to the value **true**.

The reduction relation would be deterministic were it not for the reduction rule for accumulate expressions. (If the primitive syntax for collections was not $\{v_1,\dots,v_n\}$ but instead was $\{e_1,\dots,e_n\}$ where the e_i are not necessarily values, nondeterminism would also arise from the reduction of collections to values.) Since collections are unordered, the rule applies for any permutation of $\{v_1,\dots,v_n\}$. For example, consider the expression $\mathsf{pick}\ v_1\ v_2 \triangleq \textbf{from } x \textbf{ in } \{v_1,v_2\} \textbf{ let } y = \textbf{null accumulate } x$; we have both $\mathsf{pick}\ \textbf{true false} \to^* \textbf{true}$ and $\mathsf{pick}\ \textbf{true false} \to^* \textbf{false}$.

Next, we use reduction to define an evaluation relation, which relates a closed expression to its return values, or to **Error**, in case reduction gets stuck before reaching a value.

Stuckness, Results, and Evaluation: $e \Downarrow r$ **for closed** e

Let e be *stuck* if and only if e is not a value and $\neg\exists e'.e \to e'$.
$r ::= \textbf{Error} \mid \textbf{Return}(v) \qquad$ results of evaluation
$e \Downarrow \textbf{Return}(v)$ if and only if $e \to^* v$
$e \Downarrow \textbf{Error}$ if and only if there is e' such that $e \to^* e'$ and e' is stuck.

Let closed expression e *go wrong* if and only if $e \Downarrow \textbf{Error}$. For example, we have that $\mathsf{stuck} \Downarrow \textbf{Error}$, where $\mathsf{stuck} \triangleq \{\}.\ell$ for some label ℓ. In the presence of type-test and refinement types, expressions can go wrong in unusual ways. For example, given the refinement type $T = (x : \mathsf{Any} \textbf{ where } \mathsf{stuck})$, any type-test v **in** T goes wrong. The main goal of our type system is to ensure that no closed well-typed expression goes wrong.

Encoding Type-Assertion:

$\textbf{assert}(e : T) \triangleq \textbf{let } x = e \textbf{ in } ((x \textbf{ in } T)?x : \mathsf{stuck})$

Using type-tests we can easily encode *type-assertions*. The expression $\textbf{assert}(e : T)$ enforces that the result of the expression e is a value of type T. Operationally, $\textbf{assert}(e : T)$ returns the value of e if this is an element of T, and goes wrong otherwise. Since our type system ensures that well-typed expressions do not go wrong, it also ensures statically that type-assertions always succeed.

Calling a function with arguments that do not have their declared types does *not* necessarily go wrong. Similarly, the operational semantics does not force functions to return a result that matches the declared type. One can, however, use explicit type-assertions to enforce that the declared types are respected by rewriting any function definition $f(x_1 : T_1,\dots,x_n : T_n) : U\{e\}$ into $f(x_1 : T_1,\dots,x_n : T_n) : U\{(x_1 \textbf{ in } T_1 \&\&\dots\&\& x_n \textbf{ in } T_n)?\textbf{assert}(e : U) : \mathsf{stuck}\}$. Our type system enforces in any case that declared types are respected. This enables us to express pre- and post-conditions of functions using refinement types.

2.3 Pure Expressions and Refinement Types

A problem in languages with refinement types $(x : T$ **where** $e)$ is that the refinement expression e, even though well-typed, may have effects, such as non-termination or non-determinism, and so makes no sense as a boolean condition. In Dminor calls to recursive functions can cause divergence, and since collections are unordered, iterating over them with accumulate may be nondeterministic, as above.

To address this problem, we define the set of *pure* expressions, the ones that may be used as refinements. The details, below, are a little technical, but the gist is that pure expressions must be terminating, have a unique result (which may be **Error**), and must only call functions whose bodies are pure. The typing rule (Type Refine) in §5 requires that for $(x : T$ **where** $e)$ to be well-formed, the expression e must be pure and of type Logical (which guarantees that e yields **true** or **false** without getting stuck). Checking for purity is undecidable, but we present sufficient conditions for checking purity algorithmically, in §4.

We assume that a subset of the function definitions are *labeled-pure*; we intend that only these functions may be called from pure expressions. Let an expression e be *terminating* if and only if there exists no unbounded sequence $e \to e_1 \to e_2 \to \ldots$. Let a closed expression e be *pure* if and only if (1) e is terminating, (2) there exists a unique result r such that $e \Downarrow r$, (3) for every subexpression $f(e_1, \ldots, e_n)$ of e, the function f is labeled-pure, and (4) all subexpressions of e are pure. Let an arbitrary expression e be *pure* if and only if $e\sigma$ is pure for all closing substitutions σ that assign a value to each free variable in e. Finally, we require that the body of every labeled-pure function is a pure expression.

2.4 Derived Types

We end this section by exploring the expressiveness of the primitive types introduced above, and in particular of the combination of refinement types and dynamic type-test. We show that the range of derivable types is rather wide. We begin with some basic examples.

Encoding of Empty and Singleton Types:

$\mathsf{Empty} \triangleq (x : \mathsf{Any}\ \textbf{where false})$
$[e] \triangleq (x : \mathsf{Any}\ \textbf{where}\ x == e) \quad (e \text{ pure}, x \notin fv(e))$

The type Empty has no elements; it is a subtype of all other types. The *singleton type*, $[e]$, contains only the value of pure expression e (for example, type [**null**] consists just of the **null** value).

Our calculus includes the operators of propositional logic on boolean values. We lift these operators to act on types as follows.

Encoding of Union, Intersection, and Negation Types:

$T \mid U \triangleq (x : \mathsf{Any}\ \textbf{where}\ (x\ \textbf{in}\ T) \mid\mid (x\ \textbf{in}\ U)) \qquad (x \notin fv(T,U))$
$T\ \&\ U \triangleq (x : \mathsf{Any}\ \textbf{where}\ (x\ \textbf{in}\ T)\ \&\&\ (x\ \textbf{in}\ U))$
$!T \triangleq (x : \mathsf{Any}\ \textbf{where}\ !(x\ \textbf{in}\ T))$

A value of the *union type*, $T \mid U$, is a value of T or of U. A value of the *intersection type*, $T\ \&\ U$, is a value of both T and U. A value of the *negation type*, $!T$, is a value that is not a value of T. We omit the details, but we could go in the other direction too: Boolean operators are derivable from union, intersection, and complement types.

Next, we define the types of simple values, collections, and entities. We rely on the primitive types Integer, Text, and Logical, the primitive type constructor $T*$ for collections, and the fact that every proper value is either a scalar, a collection, or an entity: so the type of entities is the complement of the union type General | Collection.

Encoding of Supertypes:

$\mathsf{General} \triangleq \mathsf{Integer} \mid \mathsf{Text} \mid \mathsf{Logical} \mid [\textbf{null}]$
$\mathsf{Collection} \triangleq \mathsf{Any}*$
$\mathsf{Entity} \triangleq !(\mathsf{General} \mid \mathsf{Collection})$

(This encoding illustrates the power of types based on propositional logic, but is fragile; if we were to extend the language with other primitive types, it would be better to take Entity to be primitive too, rather than defining it as a complement.)

The primitive type of entities is unary: the type $\{\ell : T\}$ is the set of entities with a field ℓ whose value belongs to T (and possibly other fields). As in Forsythe [76], we derive *multiple-field entity types* as an intersection type. One advantage of this approach is that it immediately entails width and depth subtyping for entities.

Encoding of Multiple-Field Entity Types:

$\{\ell_i : T_i;\ ^{i\in 1..n}\} \triangleq \{\ell_1 : T_1\}\ \&\ \ldots\ \&\ \{\ell_n : T_n\} \quad (\ell_i \text{ distinct}, n > 0)$

We can also derive *closed entity types*, which only contain entities with a fixed set of labels, and therefore allow depth but not width subtyping. To do so we constrain the multiple-field entity types above to additionally satisfy an eta law.

Encoding of Closed Entity Types:

$\textbf{closed}\{\ell_i : T_i;\ ^{i\in 1..n}\} \triangleq$
$\quad (x : \{\ell_i : T_i;\ ^{i\in 1..n}\}\ \textbf{where}\ x == \{\ell_i \Rightarrow x.\ell_i\ ^{i\in 1..n}\})$

Pair types are just a special case of closed entity types. Given pair types, refinement types, and type-test, we can also encode *dependent pair types* $\Sigma x : T.\, U$ where x is bound in U.

Encoding of Pair Types and Dependent Pair Types:

$T * U \triangleq \textbf{closed}\{\mathsf{fst} : T; \mathsf{snd} : U;\}$
$(\Sigma x : T.\, U) \triangleq (p : T * \mathsf{Any}\ \textbf{where let}\ x = p.\mathsf{fst}\ \textbf{in}\ (p.\mathsf{snd}\ \textbf{in}\ U))$

Sum types are obtained from union types by adding an additional Boolean tag; *variant types* are a generalization.

Encoding of Sum and Variant Types:

$T + U \triangleq ([\textbf{true}] * T) \mid ([\textbf{false}] * U)$
$\langle \ell_1 : T_1; \ldots; \ell_n : T_n \rangle \triangleq \textbf{closed}\{\ell_1 : T_1\} \mid \ldots \mid \textbf{closed}\{\ell_n : T_n\}$

Recursive types can be encoded as boolean recursive functions that dynamically test whether a given value has the required type.

Encoding Recursive Types

$\mu X.T \triangleq (x : \mathsf{Any}\ \textbf{where}\ f_{\mu X.T}(x))$, where $fv(T) = \varnothing$
and $f_{\mu X.T}(x)$ is a new labeled-pure function defined by
$f_{\mu X.T}(x : \mathsf{Any}) : \mathsf{Logical}\ \{x\ \textbf{in}\ T\{(x : \mathsf{Any}\ \textbf{where}\ f_{\mu X.T}(x))/X\}\}$

The usual contractivity condition is replaced by the requirement that $f_{\mu X.T}$ is labeled-pure.

Using recursive, sum, and pair types we can encode any *algebraic datatype*. For instance the type of lists of elements of type T can be encoded as follows.

Encoding List Types

$\mathsf{List}_T \triangleq \mu X.\ (T * X) + [\textbf{null}]$

In this encoding of lists, the cons-cell $v :: u$ is represented by a couple of nested entities $\{\mathsf{fst} \Rightarrow \textbf{true}, \mathsf{snd} \Rightarrow \{\mathsf{fst} \Rightarrow v, \mathsf{snd} \Rightarrow u\}\}$. More efficient representations can be easily supported, for instance $\mu X.\ \textbf{closed}\{\mathsf{hd} : T, \mathsf{tl} : X\} \mid [\textbf{null}]$, for which a cons-cell is a single entity: $\{\mathsf{hd} \Rightarrow v, \mathsf{tl} \Rightarrow u\}$.

Lists can be used to encode XML and JSON. Hence, Dminor can be viewed as a richly typed functional notation for manipulating data in XML format. And while, DTDs can be encoded as Dminor types, XML data can be loaded into Dminor even if there is no prior schema. We map an XML element to an entity, with a field to represent the name of the element, additional fields for any attributes on the element, and a final field holding a list of all the items in the body of the element.

Next, we show how to derive entity types for the common situation where the type of one field depends on the value of another. A *dependent intersection type* $(x : T \,\&\, U)$ [60] is essentially the intersection of T and U, except that the variable x is bound to the underlying value, with scope U. The type T cannot mention x, but we can rely on $x : T$ when checking well-formedness of U.

Encoding of Dependent Intersection Types:

$(x : T \,\&\, U) \triangleq (x : T \textbf{ where } x \textbf{ in } U)$

With this construct, we can define entity types where the type of one field depends on the value of another. For example, $(p : \{X : \mathsf{Integer}\} \,\&\, \{Y : (y : \mathsf{Integer} \textbf{ where } y < p.X)\})$ is the type of points below the diagonal.

M allows the field names of previously defined fields to be used within the types of subsequent fields. We can encode M's dependent entity types as follows.

Encoding Dependent Entities:

$\{\ell : T;\}U \triangleq (x : \{\ell : T\} \,\&\, U\{x.\ell/\ell\}) \qquad \text{where } x \notin fv(T,U)$
$\{\ell_1 : T_1; \dots; \ell_n : T_n;\} \triangleq \{\ell_1 : T_1;\} \dots \{\ell_n : T_n;\}\mathsf{Any}$

Our example type of points below the diagonal is written in M as follows (where the field name X appears directly as an expression in the type of the field Y).

$$\{\mathsf{X} : \mathsf{Integer}; \mathsf{Y} : \mathsf{Integer} \textbf{ where } \mathsf{Y} > \mathsf{X};\}$$

Our encoding turns this M syntax into the following type, which is equivalent to the more direct encoding given above.

$$(x_1 : \{\mathsf{X} : \mathsf{Integer}\} \,\&\, (x_2 : \{\mathsf{Y} : \mathsf{Integer} \textbf{ where } \mathsf{Y} > x_1.\mathsf{X}\} \,\&\, \mathsf{Any}))$$

Kopylov [60] explains in detail the relationship between dependent intersection and encodings of dependent entities (records).

To further illustrate the power of collection types combined with refinements, we give types below that express universal and existential quantifications over the items in a collection. Collection $\{v_1, \dots, v_n\} : T*$ has type $\mathsf{all}(x : T)e$ if $e\{v_i/x\}$ for all $i \in 1..n$, and, dually, it has type $\mathsf{exists}(x : T)e$ if $e\{v_i/x\}$ for some $i \in 1..n$.

Quantifying Over Collections:

$\mathsf{all}(x : T)e \triangleq (x : T \textbf{ where } e)* \qquad (e \text{ pure})$
$\mathsf{exists}(x : T)e \triangleq T* \,\&\, !(\mathsf{all}(x : T)!e)$

Curiously, a boolean test for whether a value is a member of a collection need not be primitive in the calculus; we can make use of the type $\mathsf{exists}(x : \mathsf{Any})(x == e_i)$ of collections that contain the item e_i, as follows.

Collection Membership as a Type-Test:

$\mathsf{Mem}(e_i, e_c) \triangleq (e_c \textbf{ in } \mathsf{exists}(x : \mathsf{Any})(x == e_i)) \quad (e_i \text{ pure})$

The boolean expression $\mathsf{Mem}(e_i, e_c)$ holds just when the value of e_i is a member of the collection denoted by e_c. (This example is to illustrate the expressiveness of the type system; collection membership is definable more directly by using an **accumulate** expression, as shown in §2.1.)

The following dependent entity type consists of a collection of song titles Songs, together with a default. The type includes the constraint that the default song is a member of Songs.

{Songs: Text∗; Default: Text **where** Mem(value,Songs);}

3. Logical Semantics

In this section we give a set-theoretic semantics for types and pure expressions. Pure expressions are interpreted as first-order terms, while types are interpreted as formulas in many-sorted first-order logic (FOL). These formulas are interpreted in a fixed model, which we formalize in Coq. We represent a Dminor subtyping problem as a logical implication, supply our SMT solver with a set of axioms that are true in our intended model, and ask the solver to prove the validity of the implication. We use Coq to state our model and to reason about the soundness of the axioms given to the SMT solver, but semantic subtyping calls only the SMT solver, not Coq.

To represent the intended logical model formally sets are encoded as Coq types, and functions are encoded as Coq functions. We start by encoding scalars, values and results, which were given as grammars in §2, as Coq types General, Value and Result.

Model: Scalars, Values and Results:

```
Inductive General : Type :=
  | G_Integer : Z → General
  | G_Text : string → General
  | G_Logical : bool → General
  | G_Null : General.
Inductive RawValue : Type :=
  | G : General → RawValue
  | E : list (string ∗ RawValue) → RawValue
  | C : list RawValue → RawValue.
Definition Value := {x : RawValue | Normal x}.
Inductive Result : Type :=
  | Error : Result
  | Return : Value → Result.
```

Scalars and results are represented directly as Coq inductive types, while for values additional care is needed to prevent duplicate labels in entities and to ensure that the representation is canonical. Our Coq representation of values is explained in detail in Appendix A (for example we define Normal in Appendix A), but the precise details are not relevant at this point. For the sake of readability in this section we continue to use the intuitive notation for values and results introduced in §2.

We define a predicate Proper that is true for results that are not **Error**, and a function out_V that returns the value inside if the result passed as argument is proper and **null** otherwise (the functions in the model are total, so in cases like this we return an arbitrary value).

Model: Proper Results:

```
Definition Proper (res : Result) :=
  match res with | Return v ⇒ true | Error ⇒ false end.
Definition out_V (res : Result) : Value :=
  match res with | Return v ⇒ v | Error ⇒ v_null end.
```

Our semantics uses many-sorted first-order logic, and each sort is interpreted by a Coq type of the same name. We write predicates as functions to sort bool (interpreted by type bool in Coq), with truth values **true** and **false** – we let the context disambiguate between the truth values in the model and the corresponding Dminor boolean values. We assume a collection of sorted function symbols whose interpretation in the intended model is given below. Let t range over FOL terms; we write $t : \sigma$ to mean that term t has sort σ; if we omit the sort of a bound variable, it may be assumed to be Value. Similarly, free variables have sort Value by default. If F is a formula, let $\models F$ mean that F is valid in our intended model.

Our semantics consists of three translations:

- For any expression e that only calls labeled-pure functions, we have the FOL term $\mathbf{R}[\![e]\!] : \mathsf{Result}$.
- For any Dminor type T and FOL term $t : \mathsf{Value}$, we have the FOL formula $\mathbf{F}[\![T]\!](t)$, which is valid in the intended model if and only if the value denoted by t is a member of the type T.

- For type T and FOL term $t : \mathsf{Value}$, we have the formula $\mathbf{W}\llbracket T \rrbracket(t)$, which holds if and only if a type-test goes wrong when checking whether the value denoted by t is a member of T. For instance, we have $\models \mathbf{W}\llbracket(x : \mathsf{Any}\ \mathbf{where}\ \mathsf{stuck})\rrbracket(\mathbf{null}) \Leftrightarrow \mathbf{true}$, but $\models \mathbf{W}\llbracket\mathsf{Any}\rrbracket(\mathbf{null}) \Leftrightarrow \mathbf{false}$.

The Relations Between Translations:

$$\begin{tikzcd}
\mathbf{F}\llbracket T \rrbracket(x) \arrow[ddr, bend right, "(x : T\ \mathbf{where}\ e)"'] & & \mathbf{W}\llbracket T \rrbracket(x) \arrow[ddl, bend left, "(x : T\ \mathbf{where}\ e)"] \\
 & x\ \mathbf{in}\ T \arrow[ul] \arrow[ur] \arrow[d] & \\
 & \mathbf{R}\llbracket e \rrbracket &
\end{tikzcd}$$

These three (mutually recursive) translations are defined below. We rely on notations for let-binding within terms ($\mathbf{let}\ x = t\ \mathbf{in}\ t'$), and terms conditional on formulas ($\mathbf{if}\ F\ \mathbf{then}\ t\ \mathbf{else}\ t'$). These notations are supported directly by most SMT solvers. They can be translated to pure first-order logic by introducing auxiliary definitions, but we omit the details. Given these we can define the monadic bind for propagating errors as a simple notation. Notice that $\models (\mathbf{Bind}\ x \Leftarrow \mathbf{Return}(v)\ \mathbf{in}\ t) = t\{v/x\}$ and $\models (\mathbf{Bind}\ x \Leftarrow \mathbf{Error}\ \mathbf{in}\ t) = \mathbf{Error}$.

Notation: Monadic Bind for Propagating Errors:

$\mathbf{Bind}\ x \Leftarrow t_1\ \mathbf{in}\ t_2 \stackrel{\triangle}{=}$
$\quad(\mathbf{if}\ \neg\mathsf{Proper}(t_1)\ \mathbf{then\ Error\ else\ let}\ x = \mathsf{out\_V}(t_1)\ \mathbf{in}\ t_2)$

We begin by describing the semantics of some core types and expressions. The semantics of refinement types $\mathbf{F}\llbracket(x : T\ \mathbf{where}\ e)\rrbracket(t)$ relies on the result of evaluating e with x bound to t. Remember however that operationally the type test $v\ \mathbf{in}\ (x : T\ \mathbf{where}\ e)$ evaluates to $\mathbf{Error}$ if $e\{v/x\}$ evaluates to $\mathbf{Error}$ or to a value that is not $\mathbf{true}$ or $\mathbf{false}$. We use $\mathbf{W}\llbracket(x : T\ \mathbf{where}\ e)\rrbracket(t)$ to record this fact, and we enforce that $\mathbf{R}\llbracket e\ \mathbf{in}\ T\rrbracket$ returns $\mathbf{Error}$ if $\mathbf{W}\llbracket T\rrbracket(t)$ holds. Tracking type-tests going wrong is crucial for our full-abstraction result.

Semantics: Core Types and Expressions:

$\mathbf{F}\llbracket\mathsf{Any}\rrbracket(t) = \mathbf{true}$
$\mathbf{W}\llbracket\mathsf{Any}\rrbracket(t) = \mathbf{false}$

$\mathbf{F}\llbracket(x : T\ \mathbf{where}\ e)\rrbracket(t) = \mathbf{F}\llbracket T\rrbracket(t) \wedge \mathbf{let}\ x = t\ \mathbf{in}\ (\mathbf{R}\llbracket e\rrbracket = \mathbf{Return}(\mathbf{true}))$
$\mathbf{W}\llbracket(x : T\ \mathbf{where}\ e)\rrbracket(t) = \mathbf{W}\llbracket T\rrbracket(t) \vee$
$\quad\mathbf{let}\ x = t\ \mathbf{in}\ (\neg(\mathbf{R}\llbracket e\rrbracket = \mathbf{Return}(\mathbf{false}) \vee \mathbf{R}\llbracket e\rrbracket = \mathbf{Return}(\mathbf{true})))$

$\mathbf{R}\llbracket x\rrbracket = \mathbf{Return}(x)$
$\mathbf{R}\llbracket e_1 ? e_2 : e_3\rrbracket = \mathbf{Bind}\ x \Leftarrow \mathbf{R}\llbracket e_1\rrbracket\ \mathbf{in}$
$\quad(\mathbf{if}\ x = \mathbf{true\ then}\ \mathbf{R}\llbracket e_2\rrbracket\ \mathbf{else}\ (\mathbf{if}\ x = \mathbf{false\ then}\ \mathbf{R}\llbracket e_3\rrbracket\ \mathbf{else\ Error}))$
$\mathbf{R}\llbracket\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\rrbracket = \mathbf{Bind}\ x \Leftarrow \mathbf{R}\llbracket e_1\rrbracket\ \mathbf{in}\ \mathbf{R}\llbracket e_2\rrbracket$
$\mathbf{R}\llbracket e\ \mathbf{in}\ T\rrbracket = \mathbf{Bind}\ x \Leftarrow \mathbf{R}\llbracket e\rrbracket\ \mathbf{in}\ (\mathbf{if}\ \mathbf{W}\llbracket T\rrbracket(x)\ \mathbf{then\ Error\ else}$
$\qquad(\mathbf{if}\ \mathbf{F}\llbracket T\rrbracket(x)\ \mathbf{then\ Return}(\mathbf{true})\ \mathbf{else\ Return}(\mathbf{false})))$

Next, we specify the semantics of scalar types and values. Function is_G in the model tests whether a value is a scalar or not, and if this is the case out_G returns this scalar. Similarly, functions is_G_Logical, is_G_Integer, and is_G_Text test whether a scalar has the corresponding scalar type.

Model: Testers for Simple Values:

```
Definition In_Logical v := (is_G v) && is_G_Logical (out_G v).
Definition In_Integer v := (is_G v) && is_G_Integer (out_G v).
Definition In_Text v := (is_G v) && is_G_Text (out_G v).
```

Semantics: Scalar Types, Simple Values and Operators:

$\mathbf{F}\llbracket\mathsf{Integer}\rrbracket(t) = \mathsf{In\_Integer}(t)$ $\qquad \mathbf{R}\llbracket c\rrbracket = \mathbf{Return}(c)$
$\mathbf{F}\llbracket\mathsf{Text}\rrbracket(t) = \mathsf{In\_Text}(t)$ $\qquad \mathbf{W}\llbracket G\rrbracket(t) = \mathbf{false}$
$\mathbf{F}\llbracket\mathsf{Logical}\rrbracket(t) = \mathsf{In\_Logical}(t)$

$\mathbf{R}\llbracket\oplus(e_1,\dots,e_n)\rrbracket = \mathbf{Bind}\ x_1 \Leftarrow \mathbf{R}\llbracket e_1\rrbracket\ \mathbf{in}\ \dots\mathbf{Bind}\ x_n \Leftarrow \mathbf{R}\llbracket e_n\rrbracket\ \mathbf{in}$
$\quad(\mathbf{if}\ \mathbf{F}\llbracket T_1\rrbracket(x_1) \wedge \dots \wedge \mathbf{F}\llbracket T_n\rrbracket(x_n)$
$\quad\ \mathbf{then\ Return}(\mathsf{O}_{\oplus}(x_1,\dots,x_n))\ \mathbf{else\ Error})$
where $\oplus : T_1,\dots,T_n \to T$

The semantics of primitive operators on simple values is defined uniformly. We state below the signature $\oplus : T_1,\dots,T_n \to T$ for each operator $\oplus$. We also name a Coq function $\mathsf{O}_{\oplus}$ to define the meaning of each operator. Then we define the semantics $\mathbf{R}\llbracket\oplus(e_1,\dots,e_n)\rrbracket$ of operator expressions. Each of the functions $\mathsf{O}_{\oplus}$ is defined in Appendix A.2.

Model: Signatures ($\oplus : T_1,\dots,T_n \to T$) and Semantics ($\mathsf{O}_{\oplus}$):

$+ : \mathsf{Integer},\mathsf{Integer} \to \mathsf{Integer}$	$\mathsf{O}_{+} = \mathsf{O\_Add}$
$- : \mathsf{Integer},\mathsf{Integer} \to \mathsf{Integer}$	$\mathsf{O}_{-} = \mathsf{O\_Minus}$
$\times : \mathsf{Integer},\mathsf{Integer} \to \mathsf{Integer}$	$\mathsf{O}_{\times} = \mathsf{O\_Mult}$
$< : \mathsf{Integer},\mathsf{Integer} \to \mathsf{Logical}$	$\mathsf{O}_{<} = \mathsf{O\_LT}$
$> : \mathsf{Integer},\mathsf{Integer} \to \mathsf{Logical}$	$\mathsf{O}_{>} = \mathsf{O\_GT}$
$== : \mathsf{Any},\mathsf{Any} \to \mathsf{Logical}$	$\mathsf{O}_{==} = \mathsf{O\_EQ}$
$! : \mathsf{Logical} \to \mathsf{Logical}$	$\mathsf{O}_{!} = \mathsf{O\_Not}$
$\&\& : \mathsf{Logical},\mathsf{Logical} \to \mathsf{Logical}$	$\mathsf{O}_{\&\&} = \mathsf{O\_And}$
$\vert\vert : \mathsf{Logical},\mathsf{Logical} \to \mathsf{Logical}$	$\mathsf{O}_{\vert\vert} = \mathsf{O\_Or}$

The semantics of an entity type $\{\ell : T\}$ is the set of all values (denoted by t) that are proper entities ($\mathsf{Good\_E}(t)$) having the field ℓ ($\mathsf{v\_has\_field}(\ell, t)$), which contains a value of type T ($\mathbf{F}\llbracket T\rrbracket(\mathsf{v\_dot}(t,\ell))$). The model functions v_has_field and v_dot use the Coq library function TheoryList.assoc to obtain the value associated with a given key in a list of pairs. Similarly, a type-test of the form $v\ \mathbf{in}\ \{\ell : T\}$ goes wrong only when v is an entity having a field ℓ that contains a value v_ℓ for which the type-test $v_\ell\ \mathbf{in}\ T$ goes wrong. If v is not an entity having field ℓ then the type-test $v\ \mathbf{in}\ \{\ell : T\}$ will simply return $\mathbf{false}$, and not go wrong. This is reflected in the definition of $\mathbf{W}\llbracket\{\ell : T\}\rrbracket$ below.

Model: Functions and Predicates on Entities:

```
Program Definition v_has_field (s : string) (v : Value) : bool :=
  match TheoryList.assoc eq_str_dec s (out_E v) with
  | Some v ⇒ true | None ⇒ false end.
Program Definition v_dot (s : string) (v : Value) : Value :=
  match TheoryList.assoc eq_str_dec s (out_E v) with
  | Some v ⇒ v | None ⇒ v_null end.
```

Semantics: Entity Types and Expressions:

$\mathbf{F}\llbracket\{\ell : T\}\rrbracket(t) = \mathsf{Good\_E}(t) \wedge \mathsf{v\_has\_field}(\ell,t) \wedge \mathbf{F}\llbracket T\rrbracket(\mathsf{v\_dot}(t,\ell))$
$\mathbf{W}\llbracket\{\ell : T\}\rrbracket(t) = \mathsf{Good\_E}(t) \wedge \mathsf{v\_has\_field}(\ell,t) \wedge \mathbf{W}\llbracket T\rrbracket(\mathsf{v\_dot}(t,\ell))$
$\mathbf{R}\llbracket\{\ell_i \Rightarrow e_i\ {}^{i \in 1..n}\}\rrbracket = \mathbf{Bind}\ x_1 \Leftarrow \mathbf{R}\llbracket e_1\rrbracket\ \mathbf{in}\ \dots\mathbf{Bind}\ x_n \Leftarrow \mathbf{R}\llbracket e_n\rrbracket\ \mathbf{in}$
$\quad\mathbf{Return}(\{\ell_i \Rightarrow x_i\ {}^{i \in 1..n}\})$
$\mathbf{R}\llbracket e.\ell\rrbracket = \mathbf{Bind}\ x \Leftarrow \mathbf{R}\llbracket e\rrbracket\ \mathbf{in}$
$\quad(\mathbf{if}\ \mathsf{is\_E}(x) \wedge \mathsf{v\_has\_field}(\ell,x)\ \mathbf{then\ Return}(\mathsf{v\_dot}(x,\ell))\ \mathbf{else\ Error})$

The semantics of $\mathbf{from}\ x\ \mathbf{in}\ e_1\ \mathbf{let}\ y = e_2\ \mathbf{accumulate}\ e_3$ relies on a function res_accumulate that folds over a collection by applying a function of sort ClosureRes2, and if no error occurs at any step it returns a value, otherwise it returns $\mathbf{Error}$. If the accumulate expression is pure it produces the same result no matter what order is used when folding. The model of the sort ClosureRes2 is the set of functions from Value to Value to Result. We write the lambda-abstraction $\mathbf{fun}\ x\ y \to \mathbf{R}\llbracket e_3\rrbracket$ for such a function. There are several standard techniques for representing lambda-abstractions in first-order logic [67]. Our implementation generates a fresh function symbol to represent each lambda-abstraction occurring in its input as a closure of sort ClosureRes2.

Model: Functions and Predicates on Collections:

```
Program Definition v_mem (v cv : Value) : bool :=
  TheoryList.mem eq_rval_dec v (out_C cv).
Program Definition v_add (v cv : Value) : Value :=
  (C (insert_in_sorted_vb v (out_C cv))).
Definition ClosureRes2 := Value → Value → Result.
Program Fixpoint res_acc_fold (f : ClosureRes2) (vb : VBag) (a :
     Result) {measure List.length vb} : Result :=
  match vb with
  | nil ⇒ a
  | v :: vb' ⇒ match a with Return va ⇒ res_acc_fold vb' (f va v)
                    | Error ⇒ Error end
  end.
Definition res_accumulate (f : ClosureRes2) (cv v : Value) : Result :=
  if is_C cv then res_acc_fold f (out_C cv) (Return v) else Error.
```

The semantics of the collection type $T*$ is the set of all values (denoted by t) that are proper collections ($\mathsf{Good\_C}(t)$) containing only elements of type T ($\forall x.\mathsf{v\_mem}(x,t) \Rightarrow \mathbf{F}[\![T]\!](x)$). On the other hand, a type-test goes wrong for a collection type $T*$, if the value being tested is a proper collection containing *some* value that causes evaluation to go wrong when testing whether it belongs to type T. The $\mathbf{F}[\![T*]\!]$ and $\mathbf{W}[\![T*]\!]$ cases are the only ones in our semantics that generate logical formulas containing first-order quantifiers.

Semantics: Collection Types and Expressions:

$$\mathbf{F}[\![T*]\!](t) = \mathsf{Good\_C}(t) \wedge (\forall x.\mathsf{v\_mem}(x,t) \Rightarrow \mathbf{F}[\![T]\!](x)) \quad x \notin fv(T,t)$$
$$\mathbf{W}[\![T*]\!](t) = \mathsf{Good\_C}(t) \wedge (\exists x.\mathsf{v\_mem}(x,t) \wedge \mathbf{W}[\![T]\!](x)) \quad x \notin fv(T,t)$$

$$\mathbf{R}[\![\{v_1,\ldots,v_n\}]\!] = \mathbf{Return}(\{v_1,\ldots,v_n\})$$
$$\mathbf{R}[\![e_1 :: e_2]\!] = \\ \quad \mathbf{Bind}\ x_1 \Leftarrow \mathbf{R}[\![e_1]\!]\ \mathbf{in}\ \mathbf{Bind}\ x_2 \Leftarrow \mathbf{R}[\![e_2]\!]\ \mathbf{in} \\ \quad (\mathbf{if}\ \mathsf{is\_C}(x_2)\ \mathbf{then}\ \mathbf{Return}(\mathsf{v\_add}(x_1,x_2))\ \mathbf{else}\ \mathbf{Error})$$
$$\mathbf{R}[\![\mathbf{from}\ x\ \mathbf{in}\ e_1\ \mathbf{let}\ y = e_2\ \mathbf{accumulate}\ e_3]\!] = \\ \quad \mathbf{Bind}\ x_1 \Leftarrow \mathbf{R}[\![e_1]\!]\ \mathbf{in}\ \mathbf{Bind}\ x_2 \Leftarrow \mathbf{R}[\![e_2]\!]\ \mathbf{in} \\ \quad \mathsf{res\_accumulate}((\mathbf{fun}\ x\ y \to \mathbf{R}[\![e_3]\!]),x_1,x_2)$$

In order to give a semantics to function applications we recall that we only consider expressions that only call labeled-pure functions, and that the body of a labeled-pure function is itself a pure expression. For each labeled-pure function definition $f(x_1 : T_1,\ldots,x_n : T_n) : U\{e\}$, the model of the symbol f is the total function $\underline{f} \in \mathsf{Value}^n \to \mathsf{Result}$ such that $\underline{f}(v_1,\ldots,v_n)$ is the result r such that $e\{v_1/x_1\}\ldots\{v_n/x_1\} \Downarrow r$. (We know that there is a unique r such that $e\{v_1/x_1\}\ldots\{v_n/x_1\} \Downarrow r$ because e is pure.) Hence, the following holds by definition:

LEMMA 1. *If $f(x_1 : T_1,\ldots,x_n : T_n) : U\{e\}$, and e is pure, and $e\{v_1/x_1\}\ldots\{v_n/x_n\} \Downarrow r$ then $\models \underline{f}(v_1,\ldots,v_n) = r$.*

Semantics: Function Application:

$$\mathbf{R}[\![f(e_1,\ldots,e_n)]\!] = \\ \quad \mathbf{Bind}\ x_1 \Leftarrow \mathbf{R}[\![e_1]\!]\ \mathbf{in}\ \ldots \mathbf{Bind}\ x_n \Leftarrow \mathbf{R}[\![e_n]\!]\ \mathbf{in}\ \underline{f}(x_1,\ldots,x_n)$$

The operational semantics preserves the logical meaning of closed pure expressions:

LEMMA 2. *For all closed pure expressions e and e', if $e \to e'$ then $\models \boldsymbol{R}[\![e]\!] = \boldsymbol{R}[\![e']\!]$.*

Moreover, we have a full abstraction result for this first-order language: the equalities induced by the operational and logical semantics of closed pure expressions coincide.

THEOREM 1 (Full Abstraction). *For all closed pure expressions e and e', $\models \boldsymbol{R}[\![e]\!] = \boldsymbol{R}[\![e']\!]$ if and only if, for all r, $e \Downarrow r \Leftrightarrow e' \Downarrow r$.*

The proofs of Lemma 2 and Theorem 1 are in Appendix C.1.
We calculate the semantics of some example types from §2.4.

Semantics of Derived Forms:

$$\models \mathbf{R}[\![e_1 == e_2]\!] = \mathbf{Bind}\ x_1 \Leftarrow \mathbf{R}[\![e_1]\!]\ \mathbf{in}\ \mathbf{Bind}\ x_2 \Leftarrow \mathbf{R}[\![e_2]\!]\ \mathbf{in} \\ \qquad \mathbf{Return}(\mathsf{v\_logical}(x_1 = x_2))$$
$$\models \mathbf{F}[\![\mathsf{Empty}]\!](t) \Leftrightarrow \mathbf{false}$$
$$\models \mathbf{F}[\![[e]]\!](t) \Leftrightarrow \mathbf{R}[\![e]\!] = \mathbf{Return}(t)$$
$$\models \neg\mathbf{W}[\![T]\!](t) \wedge \neg\mathbf{W}[\![U]\!](t) \implies \\ \quad (\mathbf{F}[\![T \mid U]\!](t) \Leftrightarrow (\mathbf{F}[\![T]\!](t) \vee \mathbf{F}[\![U]\!](t)))$$
$$\models \neg\mathbf{W}[\![T]\!](t) \wedge \neg\mathbf{W}[\![U]\!](t) \implies \\ \quad (\mathbf{F}[\![T\ \&\ U]\!](t) \Leftrightarrow (\mathbf{F}[\![T]\!](t) \wedge \mathbf{F}[\![U]\!](t)))$$
$$\models \neg\mathbf{W}[\![T]\!](t) \implies (\mathbf{F}[\![!T]\!](t) \Leftrightarrow \neg\mathbf{F}[\![T]\!](t))$$
$$\models \bigwedge_{i\in 1..n}\neg\mathbf{W}[\![T_i]\!](\mathsf{v\_dot}(t,\ell_i)) \implies (\mathbf{F}[\![\{\ell_i : T_i\ ^{i\in 1..n}\}]\!](t) \Leftrightarrow \\ \quad \mathsf{Good\_E}(t) \wedge \bigwedge_{i\in 1..n}(\mathsf{v\_has\_field}(\ell_i,t) \wedge \mathbf{F}[\![T_i]\!](\mathsf{v\_dot}(t,\ell_i))))$$
$$\models \neg\mathbf{W}[\![U]\!](t) \implies \\ \quad (\mathbf{F}[\![(s : T\ \&\ U)]\!](t) \Leftrightarrow \mathbf{F}[\![T]\!](t) \wedge \mathbf{let}\ s = t\ \mathbf{in}\ \mathbf{F}[\![U]\!](t))$$

4. Algorithmic Purity Check

Our definition of purity defined in §2.3 is undecidable, so in this section we introduce a tractable property, *algorithmic purity*, on which we rely instead of purity itself in the subsequent definitions of our type systems. Algorithmic purity is defined in terms of a syntactic termination condition on function applications to avoid divergence, and a restriction on accumulate expressions to avoid nondeterminism. We show, Theorem 2 below, that algorithmic purity implies purity.

We call an expression e *algorithmically pure* if and only if the following three conditions hold:

(1) if e is a function application $f(e_1,\ldots,e_n)$ then f is labeled-pure,

(2) if e is of the form $\mathbf{from}\ x\ \mathbf{in}\ e_1\ \mathbf{let}\ y = e_2\ \mathbf{accumulate}\ e_3$ then

$$\models \mathbf{R}[\![\mathbf{let}\ y = e_3\{x_1/x\}\{y_1/y\}\ \mathbf{in}\ e_3\{x_2/x\}]\!] = \\ \mathbf{R}[\![\mathbf{let}\ y = e_3\{x_2/x\}\{y_1/y\}\ \mathbf{in}\ e_3\{x_1/x\}]\!]$$

(where the variables x_1, x_2, and y_1 do not appear free in e_3);

(3) all the proper subexpressions of e are algorithmically pure (including the ones inside all refinement types contained by e).

Furthermore, we require that each labeled-pure function f has an algorithmically pure body that only calls f (directly or indirectly) on structurally smaller arguments; since termination-checking is not the focus of this paper, we omit the rather technical details, which may be found elsewhere [45].

Thus condition (1) enforces termination of algorithmically pure expressions: only labeled-pure functions can be called and if these functions are recursive then recursive calls can only be on syntactically smaller arguments. Condition (2) only allows accumulates in an algorithmically pure expression if the order in which the elements are processed is irrelevant for the final result. In general we call a (mathematical) function $f : X \times Y \to Y$ *order-irrelevant* if $f(x_1, f(x_2,y)) = f(x_2, f(x_1,y))$ for all x_1, x_2 and y. Enforcing that the semantics of the body of accumulate expressions is an order-irrelevant function is a sufficient condition for the uniqueness of evaluation results. We phrase this condition in terms of the logical semantics and check it using the SMT solver. Order-irrelevance is less restrictive than conditions found in the literature such as associativity and commutativity [29, 62]. If f is associative and commutative then f is also order-irrelevant, but the converse fails in general.[2] If f is order-irrelevant its two arguments need not even have

[2] The weaker order-irrelevance condition is sufficient in our setting because in an accumulate expression non-determinism only arises from the different orders in which the elements of a collection can be processed. On the other hand, the execution model is more complicated for user-defined aggregate functions in a database [29], since the database management system can

the same type. For instance none of the derived collection expressions from §2.1 is either associative or commutative, and in most of the cases the accumulator has a different type from the iterator.

Accumulate expressions are often useful inside refinements. For instance, as shown in §2.1 the number of elements of collections can be computed using an accumulate expression: **from** x **in** e **let** $y = 0$ **accumulate** $y+1$. Showing that this expression is algorithmically pure boils down to showing that $f(x,y) = y+1$ is order-irrelevant. More precisely, the (in this case trivial) proof obligation discharged by the SMT solver has the following form:

$$\models \textbf{Bind}\ y \Leftarrow \mathbf{R}[\![y_1+1]\!]\ \textbf{in}\ \mathbf{R}[\![y+1]\!] = \textbf{Bind}\ y \Leftarrow \mathbf{R}[\![y_1+1]\!]\ \textbf{in}\ \mathbf{R}[\![y+1]\!]$$

where $\mathbf{R}[\![x+y]\!] = (\textbf{if}\ \mathsf{In\_Integer}(x) \wedge \mathsf{In\_Integer}(y)\ \textbf{then Return}(\mathsf{O\_Add}(x,y))\ \textbf{else Error})$

If instead we count only the number of occurrences of a particular element z using **from** x **in** e **let** $y = 0$ **accumulate** $(x == z)?y+1 : y$ we obtain the following (more interesting) proof obligation:

$$\begin{aligned}\models\ & \textbf{Bind}\ y \Leftarrow (\textbf{if}\ \mathsf{O\_EQ}(x_1,z) = \textbf{true then}\ \mathbf{R}[\![y_1+1]\!]\ \textbf{else Return}(y_1))\\ & \quad \textbf{in}\ (\textbf{if}\ \mathsf{O\_EQ}(x_2,z) = \textbf{true then}\ \mathbf{R}[\![y+1]\!]\ \textbf{else Return}(y)) =\\ & \textbf{Bind}\ y \Leftarrow (\textbf{if}\ \mathsf{O\_EQ}(x_2,z) = \textbf{true then}\ \mathbf{R}[\![y_1+1]\!]\ \textbf{else Return}(y_1))\\ & \quad \textbf{in}\ (\textbf{if}\ \mathsf{O\_EQ}(x_1,z) = \textbf{true then}\ \mathbf{R}[\![y+1]\!]\ \textbf{else Return}(y))\end{aligned}$$

We show that the algorithmic purity check is a sufficient condition for purity.

THEOREM 2. *If e is algorithmically pure then e is pure.*

PROOF: The details are given in Appendix C.2. □

The logical semantics is defined using purity to handle the case of (labeled-pure) function applications. Given the logical semantics, we obtain algorithmic purity, a sufficient condition for purity. In the remainder of the paper we rely only on algorithmic purity.

5. Declarative Type System

In this section, we give a non-algorithmic type assignment relation, and prove preservation and progress properties relating it to the operational semantics. In the next section, we present algorithmic rules—the basis of our type-checker—for proving type assignment.

Each judgment of the type system is with respect to a typing *environment* E, of the form $x_1 : T_1, \ldots, x_n : T_n$, which assigns a type to each variable in scope. We write $\varnothing$ for the empty environment, $dom(E)$ to denote the set of variables defined by a typing environment E, and $\mathbf{F}[\![E]\!]$ for the logical interpretation of E.

Environments and their Logical Semantics:

$E ::= x_1 : T_1, \ldots, x_n : T_n$	type environments
$dom(x_1 : T_1, \ldots, x_n : T_n) = \{x_1, \ldots, x_n\}$	
$\mathbf{F}[\![x_1 : T_1, \ldots, x_n : T_n]\!] \triangleq \mathbf{F}[\![T_1]\!](x_1) \wedge \cdots \wedge \mathbf{F}[\![T_n]\!](x_n)$	

Judgments of the Declarative Type System:

$E \vdash \diamond$	environment E is well-formed
$E \vdash T$	in E, type T is well-formed
$E \vdash T <: T'$	in E, type T is a subtype of T'
$E \vdash e : T$	in E, expression e has type T

Global Assumptions:

For each function definition $f(x_1 \colon T_1, \ldots, x_n \colon T_n) : U\{e_f\}$ we assume that $x_1 \colon T_1, \ldots, x_n \colon T_n \vdash e_f : U$.

exploit true parallelism, to start multiple threads of computation, which later have to be merged and their results combined.

Rules of Well-Formed Environments and Types: $E \vdash \diamond$, $E \vdash T$

(Env Empty)
$$\frac{}{\varnothing \vdash \diamond}$$

(Env Var)
$$\frac{E \vdash T \quad x \notin dom(E)}{E, x : T \vdash \diamond}$$

(Type Any)
$$\frac{E \vdash \diamond}{E \vdash \mathsf{Any}}$$

(Type Scalar)
$$\frac{E \vdash \diamond}{E \vdash G}$$

(Type Collection)
$$\frac{E \vdash T}{E \vdash T*}$$

(Type Entity)
$$\frac{E \vdash T}{E \vdash \{\ell \colon T\}}$$

(Type Refine)
$$\frac{E, x : T \vdash e : \mathsf{Logical} \quad e \text{ alg. pure}}{E \vdash (x : T\ \textbf{where}\ e)}$$

The subtype relation is defined as logical implication between the logical semantics of well-formed types.

Rule of Semantic Subtyping:

(Subtype)
$$\frac{E \vdash T \quad E \vdash T' \quad x \notin dom(E) \quad \models (\mathbf{F}[\![E]\!] \wedge \mathbf{F}[\![T]\!](x)) \implies \mathbf{F}[\![T']\!](x)}{E \vdash T <: T'}$$

Rules of Type Assignment: $E \vdash e : T$

(Exp Singular Subsum)
$$\frac{E \vdash e : T \quad E \vdash [e : T] <: T'}{E \vdash e : T'}$$

(Exp Var)
$$\frac{E \vdash \diamond \quad (x : T) \in E}{E \vdash x : T}$$

(Exp Const)
$$\frac{E \vdash \diamond}{E \vdash c : \mathsf{Any}}$$

(Exp Eq)
$$\frac{E \vdash e_1 : T_1 \quad E \vdash e_2 : T_2 \quad T = \mathsf{Logical}}{E \vdash e_1 == e_2 : T}$$

(Exp Operator)
$$\frac{\oplus \neq (==) \quad \oplus : T_1, \ldots, T_n \to T \quad E \vdash e_i : T_i \quad \forall i \in 1..n}{E \vdash \oplus(e_1, \ldots, e_n) : T}$$

(Exp Cond)
$$\frac{E \vdash e_1 : \mathsf{Logical} \quad E, \_ : \mathsf{Ok}(e_1) \vdash e_2 : T \quad E, \_ : \mathsf{Ok}(!e_1) \vdash e_3 : T}{E \vdash (e_1 ? e_2 : e_3) : T}$$

(Exp Let)
$$\frac{E \vdash e_1 : T \quad E, x \colon T \vdash e_2 : U \quad x \notin fv(U)}{E \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : U}$$

(Exp Test)
$$\frac{E \vdash e : \mathsf{Any} \quad E \vdash T}{E \vdash e\ \textbf{in}\ T : \mathsf{Logical}}$$

(Exp Entity)
$$\frac{E \vdash e_i : T_i \quad \forall i \in 1..n \quad E \vdash \diamond}{E \vdash \{\ell_i \Rightarrow e_i\ {}^{i \in 1..n}\} : \{\ell_i \colon T_i\ {}^{i \in 1..n}\}}$$

(Exp Dot)
$$\frac{E \vdash e : \{\ell \colon T\}}{E \vdash e.\ell : T}$$

(Exp Coll)
$$\frac{E \vdash v_i : T \quad \forall i \in 1..n \quad E \vdash \diamond}{E \vdash \{v_1, \ldots, v_n\} : T*}$$

(Exp Add)
$$\frac{E \vdash e_1 : T \quad E \vdash e_2 : T*}{E \vdash (e_1 :: e_2) : T*}$$

(Exp Acc)
$$\frac{E \vdash e_1 : T* \quad E \vdash e_2 : U \quad E, x : T, y : U \vdash e_3 : U \quad x, y \notin fv(U)}{E \vdash \textbf{from}\ x\ \textbf{in}\ e_1\ \textbf{let}\ y = e_2\ \textbf{accumulate}\ e_3 : U}$$

(Exp App)
$$\frac{\begin{array}{l}\text{given } f(x_1 \colon T_1, \ldots, x_n \colon T_n) : U\{e_f\}\\ \{x_1, \ldots, x_n\} \cap dom(E) = \varnothing\\ \sigma_i = \{e_1/x_1\} \ldots \{e_i/x_i\} \quad \forall i \in 0..n\\ e_i \text{ alg. pure} \quad E \vdash e_i : T_i\sigma_{i-1} \quad \forall i \in 1..n\end{array}}{E \vdash f(e_1, \ldots e_n) : U\sigma_n}$$

For the sake of parsimony, the conclusion $E \vdash c : \mathsf{Any}$ of the rule (Exp Const) says only that a constant c is well-typed given that E is a well-formed environment. If $c \in K(G)$, the two alternative conclusions $E \vdash c : G$ and $E \vdash c : [c : G]$ are derivable using (Exp Singular Subsum). (The algorithmic synthesis rule (Synth Const) yields the latter.)

The rule (Exp Cond) records the appropriate test expression in the environment, when typing the branches. The actual value of a type $\mathsf{Ok}(e)$ is arbitrary, the point is simply to record that condition e holds [46], provided e is algorithmically pure. When e is not algorithmically pure, $\mathsf{Ok}(e)$ is equivalent to Any.

Typed Singleton Types and Ok Types:

$$[e : T] \triangleq \begin{cases} (x : T\ \textbf{where}\ x == e) \quad (x \notin fv(e)) & \text{if } e \text{ alg. pure}\\ T & \text{otherwise}\end{cases}$$

$$\mathsf{Ok}(e) \triangleq \begin{cases} (x : \mathsf{Any}\ \mathbf{where}\ e) \quad (x \notin fv(e)) & \text{if } e \text{ alg. pure} \\ \mathsf{Any} & \text{otherwise} \end{cases}$$

The rule (Exp Singular Subsum) can be seen as a combination of the following conventional rules of subsumption and singleton introduction [6].

$$\text{(Exp Subsum)}\quad \frac{E \vdash e : T \quad E \vdash T <: T'}{E \vdash e : T'} \qquad \text{(Exp Singleton)}\quad \frac{E \vdash e : T}{E \vdash e : [e : T]}$$

Both these rules are derivable from (Exp Singular Subsum). In fact, we can go in the other direction too so that the type assignment relation would be unchanged were we to replace (Exp Singular Subsum) with (Exp Subsum) and (Exp Singleton). Still, the given presentation is simpler to work with because (Exp Singular Subsum) is the only rule not determined by the structure of the expression being typed.

The rule (Exp Singular Subsum), depends on the relation $E \vdash [e : T] <: T'$, which we refer to as *singular subtyping*. We illustrate (Exp Singular Subsum) and singular subtyping with regard to (Exp Const). For example, to derive that $E \vdash [42 : \mathsf{Any}] <: \mathsf{Integer}$ note that $\models \mathbf{F}[\![[42 : \mathsf{Any}]]\!](x) \Leftrightarrow x = 42$ and hence that $\models \mathbf{F}[\![[42 : \mathsf{Any}]]\!](x) \implies \mathsf{In\_Integer}(x)$.

One might wonder why we have the separate rule (Exp Eq) for equality, rather than allowing (Exp Operator) to derive $E \vdash e_1 == e_2 : \mathsf{Logical}$, relying on the signature $==: \mathsf{Any}, \mathsf{Any} \to \mathsf{Logical}$. The reason we cannot typecheck $e_1 == e_2$ in this way is because to typecheck each e_i at the type Any in general requires us to use (Exp Singular Subsum), along with the fact that $E \vdash [e_i : T_i] <: \mathsf{Any}$. When e_i is alg. pure, the syntax $[e_i : T_i]$ is short for $(x : T_i\ \mathbf{where}\ x == e_i)$, which to be well-formed requires us to typecheck an equality $x == e_i$. To break this circularity, we provide the rule (Exp Eq) explicitly.

In the rule (Exp App), the well-formedness conditions on the argument types amount to requiring that the e_i in a dependent function application $f(e_1, \ldots e_n)$ are algorithmically pure. To form, say, $f(e)$ where e is impure, we can work around this restriction by writing $\mathbf{let}\ x = e\ \mathbf{in}\ f(x)$ instead.

The following soundness property relates type assignment to the logical semantics of types and expressions. Point (1) is that the logical value of a well-typed expression satisfies the interpretation of its type as a predicate. Point (2) is that evaluating a type-test for a well-formed type cannot go wrong.

THEOREM 3 (Logical Soundness).

(1) *If e is alg. pure and $E \vdash e : T$ then:*
  (a) $\models \mathbf{F}[\![E]\!] \implies \mathsf{Proper}(\mathbf{R}[\![e]\!])$
  (b) $\models \mathbf{F}[\![E]\!] \implies \mathbf{F}[\![T]\!](\mathsf{out\_V}(\mathbf{R}[\![e]\!]))$
(2) *If $E \vdash U$ then $\models \mathbf{F}[\![E]\!] \implies \forall y. \neg \mathbf{W}[\![U]\!](y)$, for $y \notin fv(U)$.*

PROOF: The details of this proof are given in Appendix C.3

We show the safety of our type system by proving the important preservation and progress theorems [89]. The details for both these proofs are given in Appendix C.4.

THEOREM 4 (Preservation).
If $\varnothing \vdash e : T$ and $e \to e'$ then $\varnothing \vdash e' : T$.

THEOREM 5 (Progress).
If $\varnothing \vdash e : T$ and e is not a value then $\exists e'.\ e \to e'$.

By a standard argument, we show that no well-typed closed expression e goes wrong. For a contradiction, suppose that $\varnothing \vdash e : T$ for some T and that e goes wrong, that is, $e \Downarrow \mathbf{Error}$. We have that $e \to^* e'$ and e' is stuck. By Theorem 4, $\varnothing \vdash e : T$ and $e \to^* e'$ imply $\varnothing \vdash e' : T$. By Theorem 5, this implies e' cannot be stuck, a contradiction.

We conclude this section by considering the typing of the removeNulls function from §2, whose definition is as follows.

```
NullableInt ≜ Integer | [null]
removeNulls(xs : NullableInt*) : Integer*
 { from x in xs let a = ({}:Integer*) accumulate (x!=null) ? (x :: a) : a }
```

The power of the Dminor type system is demonstrated in the typing of the conditional expression, i.e.

$$\mathsf{a} : \mathsf{Integer*}, \mathsf{x} : \mathsf{NullableInt} \vdash ((\mathsf{x}{!}{=}\mathbf{null})?\ (\mathsf{x}{::}\mathsf{a}) : \mathsf{a}) : \mathsf{Integer*}$$

The typing derivation is as follows (where E is the typing environment $\mathsf{a} : \mathsf{Integer*}, \mathsf{x} : \mathsf{NullableInt}$, and e_x is the expression $\mathsf{x}\ {!}{=}\mathbf{null}$).

$$\dfrac{\dfrac{\Pi_1}{E \vdash \mathsf{x}{!}{=}\mathbf{null} : \mathsf{Logical}} \quad \Pi_2 \quad \dfrac{\text{(Exp Var)}}{E, \_ : \mathsf{Ok}(!e_x) \vdash \mathsf{a} : \mathsf{Integer*}}}{E \vdash ((\mathsf{x}{!}{=}\mathbf{null})?\ (\mathsf{x}{::}\mathsf{a}) : \mathsf{a}) : \mathsf{Integer*}}\ \text{(Exp Cond)}$$

Derivation Π_1 is trivial; derivation Π_2 is as follows.

$$\dfrac{\dfrac{\Pi_3}{E, \_ : \mathsf{Ok}(e_x) \vdash \mathsf{x} : \mathsf{Integer}} \quad \dfrac{\text{(Exp Var)}}{E, \_ : \mathsf{Ok}(e_x) \vdash \mathsf{a} : \mathsf{Integer*}}}{E, \_ : \mathsf{Ok}(e_x) \vdash (\mathsf{x}{::}\mathsf{a}) : \mathsf{Integer*}}\ \text{(Exp Add)}$$

Derivation Π_3 is as follows (where the last applied rule is (Exp Singular Subsum)).

$$\dfrac{\dfrac{\text{(Exp Var)}}{E, \_ : \mathsf{Ok}(e_x) \vdash \mathsf{x} : \mathsf{NullableInt}} \quad \dfrac{\text{(Subtype)}}{E, \_ : \mathsf{Ok}(e_x) \vdash [\mathsf{x} : \mathsf{NullableInt}] <: \mathsf{Integer}}}{E, \_ : \mathsf{Ok}(e_x) \vdash \mathsf{x} : \mathsf{Integer}}$$

As one might expect, the hard work has been delegated to semantic subtyping, i.e. the verification of the following implication.

$$(\mathbf{F}[\![\mathsf{a} : \mathsf{Integer*}, \mathsf{x} : \mathsf{NullableInt}, \_ : \mathsf{Ok}(e_x)]\!] \wedge \mathbf{F}[\![[\mathsf{x} : \mathsf{NullableInt}]]\!](x)) \\ \implies \mathbf{F}[\![\mathsf{Integer}]\!](x)$$

The key step in verifying this implication involves the following fact about the semantics of the Ok type introduced into the typing environment by the (Exp Cond) rule (where e is any algorithmically pure expression).

$$\begin{aligned} &\mathbf{F}[\![\_ : \mathsf{Ok}(e)]\!] \\ &\quad \Leftrightarrow \quad \mathbf{F}[\![\mathsf{Ok}(e)]\!](\_) \\ &\quad \Leftrightarrow \quad \mathbf{F}[\![(z : \mathsf{Any}\ \mathbf{where}\ e)]\!](\_) \qquad z \notin fv(e) \\ &\quad \Leftrightarrow \quad \mathbf{F}[\![\mathsf{Any}]\!](\_) \wedge \mathbf{let}\ z = \_\ \mathbf{in}\ (\mathbf{R}[\![e]\!] = \mathbf{Return}(\mathbf{true})) \\ &\quad \Leftrightarrow \quad (\mathbf{R}[\![e]\!] = \mathbf{Return}(\mathbf{true})) \end{aligned}$$

In our example e is the expression $e_x = \mathsf{x}{!}{=}\mathbf{null}$ and by the definition of the translation $\mathbf{R}[\![-]\!]$ we obtain that $\mathsf{x} \neq \mathbf{null}$. By expanding the assumption that $\mathbf{F}[\![\mathsf{NullableInt}]\!](\mathsf{x})$ this allows us to infer that $\mathbf{F}[\![\mathsf{Integer}]\!](\mathsf{x})$. By expanding the encoding of typed singleton types in $\mathbf{F}[\![[\mathsf{x} : \mathsf{NullableInt}]]\!](x)$ we deduce that $\mathsf{x} = x$, which allows us to prove the original implication.

6. Algorithmic Aspects

6.1 Optimizing the Logical Semantics

We build our logical semantics in §3, independently of the type system, and then in define our type system in §5 in terms of the logical semantics. Now that we have our type system, we show how to optimize the logical semantics.

Our logical semantics propagates error values so as to match the stuck expressions of our operational semantics. Tracking errors is important, but observe that when we use our logical semantics during semantic subtyping, we only ever ask whether well-formed types are related. Every expression occurring in a well-formed type is itself well-typed, and so, by Theorem 3, its logical semantics is a proper value, not **Error**.

This suggests that when checking subtyping we can optimize the logical semantics given the assumption that the expressions occurring within the two types are well-typed. In particular, we can apply the following lemma to transform monadic error-checking binds into ordinary lets.

LEMMA 3. *If e alg. pure and $E \vdash e : T$ then $\models \mathbf{F}[\![E]\!] \implies (\mathbf{Bind}\ x \Leftarrow \mathbf{R}[\![e]\!]\ \mathbf{in}\ t) = (\mathbf{let}\ x = \mathsf{out\_V}(\mathbf{R}[\![e]\!])\ \mathbf{in}\ t)$.*

PROOF: By definition of notation, $\mathbf{Bind}\ x \Leftarrow \mathbf{R}[\![e]\!]\ \mathbf{in}\ t$ is the term $(\mathbf{if}\ \neg\mathsf{Proper}(\mathbf{R}[\![e]\!])\ \mathbf{then}\ \mathbf{Error}\ \mathbf{else}\ \mathbf{let}\ x = \mathsf{out\_V}(\mathbf{R}[\![e]\!])\ \mathbf{in}\ t)$. By Theorem 3, $\models \mathbf{F}[\![E]\!] \implies \mathsf{Proper}(\mathbf{R}[\![e]\!])$. Hence the result. □

The following tables present the optimized definitions used in our type-checker, and the following theorem states their soundness and completeness with respect to the error tracking semantics of §3.

Optimized Semantics of Types: $\mathbf{F}'[\![T]\!](t)$

$\mathbf{F}'[\![\mathsf{Any}]\!](t) = \mathbf{true}$
$\mathbf{F}'[\![\mathsf{Integer}]\!](t) = \mathsf{In\_Integer}(t)$
$\mathbf{F}'[\![\mathsf{Text}]\!](t) = \mathsf{In\_Text}(t)$
$\mathbf{F}'[\![\mathsf{Logical}]\!](t) = \mathsf{In\_Logical}(t)$
$\mathbf{F}'[\![\{\ell : T\}]\!](t) = \mathsf{Good\_E}(t) \wedge \mathsf{v\_has\_field}(\ell, t) \wedge \mathbf{F}'[\![T]\!](\mathsf{v\_dot}(t, \ell))$
$\mathbf{F}'[\![T*]\!](t) = \mathsf{Good\_C}(t) \wedge (\forall x. \mathsf{v\_mem}(x,t) \Rightarrow \mathbf{F}'[\![T]\!](x))\quad x \notin fv(T,t)$
$\mathbf{F}'[\![(x : T\ \mathbf{where}\ e)]\!](t) =$
$\quad \mathbf{F}'[\![T]\!](t) \wedge \mathbf{let}\ x = t\ \mathbf{in}\ \mathbf{V}[\![e]\!] = \mathbf{true}\quad x \notin fv(T,t)$

Optimized Semantics of Pure Typed Expressions: $\mathbf{V}[\![e]\!]$

$\mathbf{V}[\![x]\!] = x$
$\mathbf{V}[\![c]\!] = c$
$\mathbf{V}[\![\oplus(e_1,\ldots,e_n)]\!] = \mathsf{O}_\oplus(\mathbf{V}[\![e_1]\!],\ldots,\mathbf{V}[\![e_n]\!])$
$\mathbf{V}[\![e_1 ? e_2 : e_3]\!] = (\mathbf{if}\ \mathbf{V}[\![e_1]\!] = \mathbf{true}\ \mathbf{then}\ \mathbf{V}[\![e_2]\!]\ \mathbf{else}\ \mathbf{V}[\![e_3]\!])$
$\mathbf{V}[\![\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2]\!] = \mathbf{let}\ x = \mathbf{V}[\![e_1]\!]\ \mathbf{in}\ \mathbf{V}[\![e_2]\!]$
$\mathbf{V}[\![e\ \mathbf{in}\ T]\!] = \mathsf{v\_logical}(\mathbf{F}'[\![T]\!](\mathbf{V}[\![e]\!]))$
$\mathbf{V}[\![e : T]\!] = \mathbf{V}[\![e]\!]$
$\mathbf{V}[\![\{\ell_i \Rightarrow e_i\ ^{i\in 1..n}\}]\!] = \{\ell_i \Rightarrow \mathbf{V}[\![e_i]\!]\ ^{i\in 1..n}\}$
$\mathbf{V}[\![e.\ell]\!] = \mathsf{v\_dot}(\mathbf{V}[\![e]\!], \ell)$
$\mathbf{V}[\![\{v_1,\ldots,v_n\}]\!] = \{v_1,\ldots,v_n\}$
$\mathbf{V}[\![e_1 :: e_2]\!] = \mathsf{v\_add}(\mathbf{V}[\![e_1]\!], \mathbf{V}[\![e_2]\!])$
$\mathbf{V}[\![\mathbf{from}\ x\ \mathbf{in}\ e_1\ \mathbf{let}\ y = e_2\ \mathbf{accumulate}\ e_3]\!] =$
$\quad \mathsf{v\_accumulate}((\mathbf{fun}\ x\ y \rightarrow \mathbf{V}[\![e_3]\!]), \mathbf{V}[\![e_1]\!], \mathbf{V}[\![e_2]\!])$

Optimized Semantics of Environments:

$$\mathbf{F}'[\![x_1 : T_1, \ldots, x_n : T_n]\!] \triangleq \mathbf{F}'[\![T_1]\!](x_1) \wedge \cdots \wedge \mathbf{F}'[\![T_n]\!](x_n)$$

THEOREM 6 (Correctness of Optimized Semantics).

(1) *If $E \vdash \diamond$ then $\models \mathbf{F}[\![E]\!] \Leftrightarrow \mathbf{F}'[\![E]\!]$.*
(2) *If $E \vdash T$ and $x \notin dom(E)$ then:*
$\quad \models \mathbf{F}'[\![E]\!] \implies (\mathbf{F}[\![T]\!](x) \Leftrightarrow \mathbf{F}'[\![T]\!](x))$.
(3) *If $E \vdash e : T$ then:*
$\quad \models \mathbf{F}'[\![E]\!] \implies (\mathbf{R}[\![e]\!] = \mathbf{Return}(\mathbf{V}[\![e]\!]))$.

PROOF: The proof is by simultaneous induction on the derivations of $E \vdash \diamond$ and $E \vdash T$ and $E \vdash e : T$, with appeal to Theorem 3 and Lemma 3. □

6.2 Bidirectional Typing Rules

The Dminor type-checker is implemented as a *bidirectional* type system [73]. The key concept of bidirectional type systems is that there are two typing relations, one for type *checking*, and one for type *synthesis*. The chief characteristic of these relations is that they are local in the sense that type information is passed between adjacent nodes in the syntax tree. This is an important feature, not least because it makes type error reporting easy—a disadvantage of languages that use ML-style inference [63]. Moreover, bidirectional type systems are simple to implement, predictable for programmers, and expressive; for example, the type system for C$^\sharp$ can be defined as a bidirectional type system [17], and several dependently-typed languages have bidirectional type systems [64, 57].

Judgments of the Algorithmic Type System:

$E \vdash e \rightarrow T$	in E, expression e synthesizes type T
$E \vdash e \leftarrow T$	in E, expression e checks against type T
$E \rhd \diamond$	environment E is alg. well-formed
$E \rhd T$	in E, type T is alg. well-formed
$E \rhd S <: T$	in E, type S is alg. a subtype of type T

The reader will recall that the rules characterizing well-formed environments and types in §5 made use of the *declarative* typing relation in rule (Type Refine). We thus need to define algorithmic versions of these rules that make use of the *bidirectional* type system.

Rules of Algorithmic Well-Formedness: $E \rhd \diamond$, $E \rhd T$

(Alg. Env Empty)
$$\frac{}{\varnothing \rhd \diamond}$$

(Alg. Env Var)
$$\frac{E \rhd T \quad x \notin dom(E)}{E, x : T \rhd \diamond}$$

(Alg. Type Any)
$$\frac{E \rhd \diamond}{E \rhd \mathsf{Any}}$$

(Alg. Type Scalar)
$$\frac{E \rhd \diamond}{E \rhd G}$$

(Alg. Type Collection)
$$\frac{E \rhd T}{E \rhd T*}$$

(Alg. Type Entity)
$$\frac{E \rhd T}{E \rhd \{\ell : T\}}$$

(Alg. Type Refine)
$$\frac{E, x : T \vdash e \leftarrow \mathsf{Logical} \quad e \text{ alg. pure}}{E \rhd (x : T\ \mathbf{where}\ e)}$$

We also make use of the optimized semantics from §6.1 to define the algorithmic semantic subtyping rule.

Rule of Algorithmic Semantic Subtyping:

(Alg. Subtype)
$$\frac{E \rhd T \quad x \notin dom(E) \qquad \models (\mathbf{F}'[\![E]\!] \wedge \mathbf{F}'[\![T]\!](x)) \implies \mathbf{F}'[\![T']\!](x)}{E \rhd T <: T'}$$

Rules of Type Synthesis: $E \vdash e \rightarrow T$

(Synth Var)
$$\frac{E \rhd \diamond \quad (x : T) \in E}{E \vdash x \rightarrow [x : T]}$$

(Synth Const)
$$\frac{E \rhd \diamond}{E \vdash c \rightarrow [c : typeof(c)]}$$

(Synth Operator)
$$\frac{E \vdash e_i \leftarrow T_i \quad \forall i \in 1..n \quad \oplus : T_1, \ldots, T_n \rightarrow T}{E \vdash \oplus(e_1,\ldots,e_n) \rightarrow [\oplus(e_1,\ldots,e_n) : T]}$$

(Synth Cond)
$$\frac{E \vdash e_1 \leftarrow \mathsf{Logical} \qquad E, \_ : \mathsf{Ok}(e_1) \vdash e_2 \rightarrow T_2 \quad E, \_ : \mathsf{Ok}(!e_1) \vdash e_3 \rightarrow T_3}{E \vdash (e_1 ? e_2 : e_3) \rightarrow (\mathbf{if}\ e_1\ \mathbf{then}\ T_2\ \mathbf{else}\ T_3)}$$

(Synth Let)
$$\frac{E \vdash e_1 \rightarrow T_1 \quad E, x : T_1 \vdash e_2 \rightarrow T_2 \quad E \vdash T_2\{e_1/x\}}{E \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \rightarrow T_2\{e_1/x\}}$$

(Synth Test)
$$\frac{E \vdash e \leftarrow \mathsf{Any} \quad E \rhd T}{E \vdash e \ \mathbf{in}\ T \to [e \ \mathbf{in}\ T : \mathsf{Logical}]}$$

(Synth Ascribe)
$$\frac{E \vdash e \leftarrow T}{E \vdash (e^T) \to T}$$

(Synth Entity)
$$\frac{E \vdash e_1 \to T_1 \quad \cdots \quad E \vdash e_n \to T_n \quad E \rhd \diamond}{E \vdash \{\ell_i \Rightarrow e_i\ ^{i \in 1..n}\} \to \{\ell_1 : T_1\} \ \&\ \cdots \ \&\ \{\ell_n : T_n\}}$$

(Synth Dot)
$$\frac{E \vdash e \to T \quad norm(T) = D \quad D.\ell \rightsquigarrow U}{E \vdash e.\ell \to [e.\ell : U]}$$

(Synth Coll)
$$\frac{E \vdash v_i \to T_i \quad \forall i \in 1..n \quad E \rhd \diamond}{E \vdash \{v_1, \ldots, v_n\} \to (T_1 \mid \ldots \mid T_n)*}$$

(Synth Add)
$$\frac{E \vdash e_1 \to T_1 \quad E \vdash e_2 \to T_2 \quad norm(T_2) = D_2 \quad D_2.\mathsf{Items} \rightsquigarrow U_2}{E \vdash e_1 :: e_2 \to ([e_1 : T_1] \mid U_2)*}$$

(Synth Acc)
$$\frac{\begin{array}{l} E \vdash e_1 \to T_1 \quad norm(T_1) = D_1 \quad D_1.\mathsf{Items} \rightsquigarrow U_1 \\ E \vdash e_2 \to T_2 \quad E, x : U_1, y : T_2 \vdash e_3 \leftarrow T_2 \end{array}}{E \vdash \mathbf{from}\ x\ \mathbf{in}\ e_1\ \mathbf{let}\ y = e_2\ \mathbf{accumulate}\ e_3 \to T_2}$$

(Synth App)
$$\frac{\begin{array}{l} \text{given } f(x_1 : T_1, \ldots, x_n : T_n) : U\{e_f\} \\ \sigma_i = \{e_1/x_1\} \ldots \{e_i/x_i\} \quad \forall i \in 0..n \\ e_i \text{ is alg. pure} \quad E \vdash e_i \leftarrow (T_i \sigma_{i-1}) \quad \forall i \in 1..n \end{array}}{E \vdash f(e_1, \ldots e_n) \to U \sigma_n}$$

The rules (Synth Var), and (Synth Const) follow the work of Aspinall [6] and yield singleton types for all variables and constants, where the function *typeof* returns the type of a given constant; let $typeof(c) = G$ if and only if $c \in K(G)$. Rule (Synth Entity) uses intersection types to encode multiple-field entity types (see §2.4).

The (Synth Cond) rule synthesizes a type **if** e_1 **then** T_2 **else** T_3, defined below, for the conditional expression $e_1?e_2 : e_3$. The synthesized type is the union of the two types synthesized for the branches, where we additionally record the test expression in the type (if it is algorithmically pure), which allows for more precise typing.

Encoding of Conditional Types:

$$\mathbf{if}\ e\ \mathbf{then}\ T\ \mathbf{else}\ U \triangleq \begin{cases} (\_ : T\ \mathbf{where}\ e) \mid (\_ : U\ \mathbf{where}\ !e) & \text{if } e \text{ alg. pure} \\ T \mid U & \text{otherwise} \end{cases}$$

The rule (Synth Let) faces the problem that the bound variable x should not escape into the result type T_2, and does so by substituting e_1 for x in T_2. In case $x \in fv(T_2)$ and e_1 is not pure, the rule does not apply, as the result type is not well-defined. In this case, the programmer needs to insert a type-ascription to remove the bound variable explicitly. Similarly, the rule (Synth App) for an application $f(e_1, \ldots, e_n)$ returns a type possibly containing the expressions $e_1, \ldots, e_n$. The rule is not applicable if these expressions are impure and occur in the result type. The programmer can work around this limitation by using a let-expression to compute the value of any impure expression before making the call to f. In practice we found that conversion to A-normal form can further improve the precision of our algorithmic type system, since purity checking is then done at a finer granularity and thus more singleton types are synthesized.

The (Synth Ascribe) rule allows the user to provide hints to the type-checker in the form of type annotations (e^T). Such type annotations are not part of the core language and have no operational significance, and are necessary in case the type-checker cannot infer the loop invariants of accumulate expressions. Although in the current presentation monadic bind expressions [21] are encoded using **accumulate** (see §2.1), the Dminor type-checker infers loop invariants for **bind** (and also for LINQ queries [66]) using an additional (Synth Bind) rule, which exploits the encoding of collection quantifiers as refinement types (see 2.4).

Inferring Loop Invariants for Bind:

(Synth Bind)
$$\frac{\begin{array}{l} E \vdash e_1 \to T_1 \quad norm(T_1) = D_1 \quad D_1.\mathsf{Items} \rightsquigarrow U_1 \\ E, x : U_1 \vdash e_2 \to T_2 \quad norm(T_2) = D_2 \quad D_2.\mathsf{Items} \rightsquigarrow U_2 \\ \text{if } E \rhd U_2 \text{ then } T = U_2, \\ \text{otherwise } T = (y : \mathsf{Any}\ \mathbf{where}\ (e_1\ \mathbf{in}\ \mathsf{exists}(x : U_1)(y\ \mathbf{in}\ U_2))) \end{array}}{E \vdash (\mathbf{bind}\ x \leftarrow e_1\ \mathbf{in}\ e_2) \to T*}$$

As explained above, type annotations are also sometimes necessary for let expressions and function applications.

In several of the type synthesis rules we need to inspect components of intermediate types. In simple type systems this is straightforward as one can rely on the syntactic structure of types, but for rich type systems such as the one of Dminor this is not possible. In other dependently-typed languages, either the programmer is required to insert casts to force the type into the appropriate syntactic shape [90], or types are first executed until a normal form is reached [7]. Unfortunately, neither approach is acceptable in Dminor: the former forces too many casts on the programmer, and the latter is not feasible because refinements often refer to rather large data sets. One pragmatic possibility is to attempt type normalization but place some ad hoc bound on evaluation [57]. As an alternative, we define a disjunctive normal form (DNF) for types, along with a normalization function, *norm*, for translating types into DNF, and procedures for extracting type information from DNF types. In practice, this approach works well.[3]

Normal Types (DNF) and Normalization:

$D ::= R_1 \mid \ldots \mid R_n$	normal disjunction (Empty if $n = 0$)
$R ::= x : C\ \mathbf{where}\ e$	normal refined conjunction
$C ::= A_1\ \&\ \ldots\ \&\ A_n$	normal conjunction (Any if $n = 0$)
$A ::= G \mid T* \mid \{\ell : T\}$	atomic type

$$\begin{array}{l}
norm(\mathsf{Any}) \triangleq x : \mathsf{Any}\ \mathbf{where\ true} \\
norm(G) \triangleq x : G\ \mathbf{where\ true} \\
norm(T*) \triangleq x : T*\ \mathbf{where\ true} \\
norm(\{\ell : T\}) \triangleq x : \{\ell : T\}\ \mathbf{where\ true} \\
norm(x : T\ \mathbf{where}\ e) \triangleq \\
\quad |_{i=1}^{n}\ Conj_{DD}(x_i : C_i\ \mathbf{where}\ e_i, norm_r(x : C_i\ \mathbf{where}\ e)) \\
\qquad \text{where } |_{i=1}^{n}\ (x_i : C_i\ \mathbf{where}\ e_i) = norm(T)
\end{array}$$

$$\begin{array}{l}
norm_r(x : C\ \mathbf{where}\ x\ \mathbf{in}\ T) \triangleq norm(C\ \&\ T) \quad \text{where } x \notin fv(T) \\
norm_r(x : C\ \mathbf{where}\ e_1 \mid\mid e_2) \triangleq \\
\quad norm_r(x : C\ \mathbf{where}\ e_1) \mid norm_r(x : C\ \mathbf{where}\ e_2) \\
norm_r(x : C\ \mathbf{where}\ e_1\ \&\&\ e_2) \triangleq \\
\quad Conj_{DD}(norm_r(x : C\ \mathbf{where}\ e_1), norm_r(x : C\ \mathbf{where}\ e_2)) \\
norm_r(x : C\ \mathbf{where}\ e) \triangleq (x : C\ \mathbf{where}\ e) \qquad \text{otherwise}
\end{array}$$

$$\begin{array}{l}
Conj_{DD}((R_1 \mid \ldots \mid R_n), D) \triangleq Conj_{RD}(R_1, D) \mid \ldots \mid Conj_{RD}(R_n, D) \\
Conj_{RD}(R, (R_1 \mid \ldots \mid R_n)) \triangleq Conj_{RR}(R, R_1) \mid \ldots \mid Conj_{RR}(R, R_n) \\
Conj_{RR}(x_1 : C_1\ \mathbf{where}\ e_1, x_2 : C_2\ \mathbf{where}\ e_3) \triangleq \\
\quad y : C_1\ \&\ C_2\ \mathbf{where}\ e_1\{y/x_1\}\ \&\&\ e_2\{y/x_2\} \\
\qquad\qquad\qquad\qquad \text{where } y \notin fv(C_1, C_2, e_1, e_2)
\end{array}$$

Normalization is defined using two functions: *norm* which normalizes a type, and $norm_r$ which normalizes a refinement type based on the structure of the refinement expression. We make use of helper

[3] A further alternative would be to embed the normalization process into subtyping [28]. We leave this for future work.

functions to build DNF types, principally the function, $Conj_{DD}$, that returns in DNF the conjunction of two disjunction types.

We define *partial* functions to extract field and item types from normalized entity and collection types, respectively.

Extraction of Field Type: $D.\ell \rightsquigarrow U$

(Field Disj)
$$\frac{R_i.\ell \rightsquigarrow U_i \quad \forall i \in 1..n}{(R_1 \mid \ldots \mid R_n).\ell \rightsquigarrow (U_1 \mid \ldots \mid U_n)}$$

(Field Refine)
$$\frac{C.\ell \rightsquigarrow U}{(x : C \textbf{ where } e).\ell \rightsquigarrow U}$$

(Field Conj)
$$\frac{S = \{U_i \mid A_i.\ell \rightsquigarrow U_i\} \neq \varnothing}{(A_1 \,\&\, \ldots \,\&\, A_n).\ell \rightsquigarrow (\& S)}$$

(Field Atom)
$$\frac{}{\{\ell : T\}.\ell \rightsquigarrow T}$$

The (Field Disj) rule requires that for every disjunct R_i there is a U_i such that $R_i.\ell \rightsquigarrow U_i$. In contrast the (Field Conj) rule requires only that there is at least one conjunct A_i for which there is a U_i such that $A_i.\ell \rightsquigarrow U_i$.

Extraction of Item Type: $D.\mathsf{Items} \rightsquigarrow U$

(Items Disj)
$$\frac{R_i.\mathsf{Items} \rightsquigarrow U_i \quad \forall i \in 1..n}{(R_1 \mid \ldots \mid R_n).\mathsf{Items} \rightsquigarrow (U_1 \mid \ldots \mid U_n)}$$

(Items Refine)
$$\frac{C.\mathsf{Items} \rightsquigarrow U}{(x : C \textbf{ where } e).\mathsf{Items} \rightsquigarrow U}$$

(Items Conj)
$$\frac{S = \{U_i \mid A_i.\mathsf{Items} \rightsquigarrow U_i\} \neq \varnothing}{(A_1 \,\&\, \ldots \,\&\, A_n).\mathsf{Items} \rightsquigarrow (\& S)}$$

(Items Atom)
$$\frac{}{(T*).\mathsf{Items} \rightsquigarrow T}$$

Rules of Type Checking: $E \vdash e \leftarrow T$

(Swap)
$$\frac{E \vdash e \rightarrow T \quad E \rhd [e : T] <: T'}{E \vdash e \leftarrow T'}$$

(Check Cond)
$$\frac{\begin{array}{l} E \vdash e_1 \leftarrow \mathsf{Logical} \\ E, \_ : \mathsf{Ok}(e_1) \vdash e_2 \leftarrow T \\ E, \_ : \mathsf{Ok}(!e_1) \vdash e_3 \leftarrow T \end{array}}{E \vdash e_1 ? e_2 : e_3 \leftarrow T}$$

(Check Let)
$$\frac{E \vdash e_1 \rightarrow T \quad E, x : T \vdash e_2 \leftarrow U \quad x \notin fv(U)}{E \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \leftarrow U}$$

(Check Dot)
$$\frac{E \vdash e \leftarrow \{\ell : T\}}{E \vdash e.\ell \leftarrow T}$$

The (Swap) rule tests for singular subsumption. In our implementation we apply this rule only if the expression to be type-checked is not a conditional, let-expression or a field selection. Typically (for example, SAGE [57]), the type checking relation for a bidirectional type system consists of a single rule of the form:

$$\frac{E \vdash e \rightarrow S \quad E \rhd S <: T}{E \vdash e \leftarrow T}$$

However, we have found in practice that in the cases where the expression is a conditional or a let-expression, we get better precision of type checking by following Pierce and Turner [73] and passing the type through to the subexpressions, as shown in the (Check Cond) and (Check Let) rules. Similarly, we can pass through an entity type in the (Check Dot) rule.

THEOREM 7 (Soundness of Algorithmic Type System).

(1) *If $E \rhd \diamond$ then $E \vdash \diamond$.*
(2) *If $E \rhd T$ then $E \vdash T$.*
(3) *If $E \rhd S <: T$ and $E \vdash S$ then $E \vdash S <: T$.*
(4) *If $E \vdash e \rightarrow T$ then $E \vdash e : T$.*
(5) *If $E \vdash e \leftarrow T$ then $E \vdash e : T$.*

PROOF: The details of this proof are in Appendix C.5. □

7. Exploiting SMT Models

SMT solvers such as Z3 can produce a potential model in case they fail to prove the validity of a proof obligation (that is, when they show the satisfiability of its negation, or when they give up). In our case such models can be automatically converted into assignments mapping program variables to Dminor values. Because of the inherent incompleteness of the SMT solver[4] and of the axiomatization we feed to it, the obtained assignment is not guaranteed to be correct. However, given a way to validate assignments, one can use the correct ones to provide very precise counterexamples when type-checking fails, and to find inhabitants of types statically or dynamically, in a way that amounts to a new style of constraint logic programming [51].

7.1 Precise Counterexamples to Type-checking

The type-checking algorithm from §6.2 crucially relies on subtyping, as in the rule (Swap), and our algorithmic semantic subtyping relation $E \rhd T <: T'$ produces proof obligations of the form

$$\models (\mathbf{F}'[\![E]\!] \wedge \mathbf{F}'[\![T]\!](x)) \implies \mathbf{F}'[\![T']\!](x)$$

for some fresh variable x. If the SMT solver fails to prove such an obligation, it produces a potential model, from which we can extract an assignment σ mapping x and all variables in E to Dminor values. To verify that σ is a valid counterexample, we check the following three conditions:

(1) $E \rhd T$ and $E \rhd T'$
(2) $(y\sigma \textbf{ in } U\sigma) \rightarrow^* \textbf{true}$, for all $(y : U) \in E$;
(3) $(x\sigma \textbf{ in } (T \,\&!T')\sigma) \rightarrow^* \textbf{true}$.

Condition (1) enforces that we only evaluate pure expressions therefore ensuring termination and confluence of the reduction. Condition (2) enforces that the values for all variables in E have their corresponding (possibly dependent) types. Condition (3) checks whether the value assigned to x by σ is an element of T but not an element of T'. If these three checks succeed, σ is a valid counterexample to typing that we display to the user.

Since the type-checker is itself over-approximating, there is no guarantee that an expression e that fails to type-check is going to get stuck when evaluated. The best we might do is to evaluate $e\sigma$ for a fixed number of steps, a fixed number of times (remember that e can be non-deterministic), searching for a counterexample trace we can additionally display to the user.

7.2 Finding Elements of Types Statically

Type emptiness can be phrased in terms of subtyping as $E \vdash T <: \mathsf{Empty}$, or equivalently $\models \neg(\mathbf{F}[\![E]\!] \wedge \mathbf{F}[\![T]\!](x))$ for some fresh x. We additionally check that $\mathbf{F}[\![E]\!]$ is satisfiable (and the model the SMT solver produces is a correct one) to exclude the case that the environment is inconsistent and therefore any subtyping judgment holds vacuously. Hence, we can detect empty types during type-checking and issue a warning to the user if an empty type is found. This is useful, since one can make mistakes when writing types containing complicated constraints. Moreover, if the SMT solver cannot prove that a type is empty we again obtain an assignment σ, which we can validate as in §7.1. If validation succeeds we know that $x\sigma$ is an element of $T\sigma$, and we can display this information if the user hovers over a type.

[4] Other than background theories with a non-recursively enumerable set of logical consequences such as integer arithmetic, other sources of incompleteness in SMT solvers are quantifiers (which are usually heuristically instantiated) and user-defined time-outs. All these three sources of incompleteness are affect Dminor.

7.3 Finding Elements of Types Dynamically

We can use the same technique to find elements of types dynamically. We augment the calculus with a new primitive expression **elementof** T (not present in the M language) which tries to find an inhabitant of T. If successful the expression returns such a value, but otherwise it returns **null**. (We can always choose T so that **null** is not a member, so that returning **null** unambiguously signals that no member of T was found.)

Operational Semantics for Finding Elements of Types:

elementof $T \rightarrow v$ where v **in** $T \rightarrow^*$ **true**
elementof $T \rightarrow$ **null**

Finding elements of types is actually simpler to do dynamically than statically: at run-time all variables inside types have already been substituted by values, so there are fewer checks to perform.

The outcome of **elementof** T is in general non-deterministic, and depends in practice on the computational power and load of the system as well as on the timeout used when calling the SMT solver. Because of this **elementof** T expressions are considered algorithmically impure, and therefore cannot appear inside types.

Typing rules for elementof:

(Exp elementof)
$$\frac{E \vdash T}{E \vdash \textbf{elementof}\ T : (T \mid [\textbf{null}])}$$

(Synth elementof)
$$\frac{E \vdash T}{E \vdash \textbf{elementof}\ T \rightarrow (T \mid [\textbf{null}])}$$

The new **elementof** T construct enables a form of constraint programming in Dminor, in which we iteratively change the constraints inside types in order to explore a large state space. For instance the following recursive function computes all correct configurations of a complex system when called with the empty collection as argument. Correctness is specified by some type GoodConfig.

```
allGoodConfigs(avoid : GoodConfig*) : GoodConfig* {
    let m = elementof (GoodConfig where !(value in avoid)) in
    (m == null) ? {} : (m :: (allGoodConfigs(m :: avoid)))
}
```

Programming in this purely declarative style can be appealing for rapid prototyping or other tasks where efficiency is not the main concern. One only needs to specify *what* has to be computed in the form of a type. It is up to the SMT solver to use the right decision procedures and heuristics to perform the computation. If this fails or is too slow one can instead implement the required functionality manually. There is little productivity loss in this case since the types one has already written will serve as specification for the code that needs to be written manually.

8. Implementation

Our prototype Dminor implementation is approximately 2700 lines of F$^\sharp$ code, excluding the lexer and parser. Our type-checker implements the optimized logical semantics from §6.1, and the bidirectional typing rules from §6.2. We use Z3 [33] to discharge the proof obligations generated by semantic subtyping. Together with the proof obligations we feed to Z3 a 500 line axiomatization of our intended model in SMT-LIB format [75], which uses the theories of integers, datatypes and extensional arrays (see Appendix B). Our Coq formalization is just over 4000 lines of Coq, out of which the definition of the intended model of Dminor and the proof of its well-definedness are about 2000 lines (see Appendix A). The rest encompasses formalizations of our definitions and mechanized versions of some of the proofs in the paper (Lemmas 12-14, 19, 22, 23, 43, 44, and Theorem 5).

We have tested our type-checker on a test suite consisting of about 130 files, some type-correct and some type-incorrect, some hand-crafted by us and some transliterated from the M preliminary release. Even without serious optimization the type-checker is fast. Checking each of the 130 files in our test suite on a typical laptop takes from under 1 second (for just startup and parsing) to around 3 seconds (for type-checking a 175 lines long interpreter for while-programs—see §1.1—that discharges more than 300 proof obligations). Also, our experience with Z3 has been very positive so far—whilst it is possible to craft subtyping tests that cannot be efficiently checked,[5] Z3 has performed very well on the idioms in our test suite. Still, we cannot draw firm conclusions until we have studied bigger examples.

We have also implemented the techniques for exploiting SMT solver models described in §7. We built a plugin for the Microsoft Intellipad text editor [69] that displays precise counterexamples to typing, flags empty types and otherwise displays one element of each type defined in the code. Moreover, our interpreter for Dminor supports **elementof** for dynamically generating instances of types (§7.3). This works well for simple constraints involving equalities, datatypes and simple arithmetic, and types that are not too deeply nested. However, scaling this up to arbitrary Dminor types is a challenge that will require additional work, as well as further progress in SMT solvers.

9. Related Work

Whilst Dminor's combination of refinement types and type-tests is new and highly expressive, it builds upon a large body of related work on advanced type systems. Refinement types have their origins in early work in theorem proving systems and specification languages, such as subset types in constructive type theory [70], set comprehensions in VDM [54], and predicate subtypes in PVS [78]. In PVS, constraints found when checking predicate subtypes become proof obligations to be proved interactively. More recently, Sozeau [81] extends Coq with subset types; as in PVS the proofs of subset type membership have to be constructed using tactics.

Pratt [74] argued for a semantic notion of "predicate types," where objects intrinsically belong to many types. His proposed language Viron has an early notion of refinement type. Freeman and Pfenning [42] extended ML with a form of refinement type, and Xi and Pfenning [90] considered applications of dependent types in an extension of ML. In both of these systems, decidability of type checking is maintained by restricting which expressions can appear in types. Lovas and Pfenning [64] presented a bidirectional refinement type system for LF, where a restriction on expressions leads to an expressive yet decidable type system.

Other work has combined refinement types with syntactic subtyping [12, 77] but none includes type-test in the refinement language. Closest to our type system is the work of Flanagan et al. on hybrid types and SAGE [57]. SAGE also uses an SMT solver to check the validity of refinements but not for subtyping (checked by traditional syntactic techniques), and does not allow type-test expressions in refinements. However, SAGE supports a dynamic type and employs a particular form of hybrid type checking [41, 58] that allows particular expressions to have their type-check deferred until run-time. The idea of hybrid types is to strike a balance between runtime checking of contracts, as in Eiffel [68] and Racket [39], and static typing. Compared to purely static typing this can reduce the number of false alarms generated by type-checking.

In spite of early work on semantic subtyping by Aiken and Wimmers [2] and Damm [31], most programming and query languages instead use a *syntactic* notion of subtyping. This syntactic approach is typically formalized by an inductively or co-inductively defined set of rules [72]. Unfortunately, deriving an algorithm from

[5] Z3 gets at most 1 second for each proof obligation by default.

such a set of rules can be difficult, especially for advanced features such as intersection and union types [36, 37].

Although by definition semistructured data (such as from the SSD model [1] or, more recently, XML and JSON) has no schema, Buneman and Pierce [22] show that it can be checked using a flexible enough type system. They propose a combination of collection types, record types, and untagged union types, along with a sophisticated notion of structural subtyping including certain distributivity axioms.

The introduction of XML and XML query languages led to renewed (practical) interest in semantic subtyping. In the context of XML documents, there is a natural generalization of DTDs where the structures in XML documents can be described using regular expression types. These types capture and generalize regular expression notation (such as `*`, `?`, and `|`) and subtyping becomes inclusion between the sets of documents denoted by two regular expression types.[6] Hosoya and Pierce first defined such a type system for XML [50] and an XML-processing language, XDuce, based on this type system [49]. Frisch, Castagna, and Benzaken [43] extended semantic subtyping to function types and propositional types, with type-test, resulting in the language CDuce [13]. (An excellent overview of the use of semantic subtyping in the context of querying XML documents was given by Castagna [27].) In the end, the XQuery working group resorted to a more conventional pure named type system [80] with a simpler notion of subtyping based on ordinary regular expression inclusion (as opposed to XDuce's use of tree regular expressions). Neither XDuce nor CDuce provides general refinement types, and their subtype algorithm is purpose-built. Geneves et al. [44] consider a related problem of XML path containment. They translate XPath expressions and XML regular tree types into a particular logic and hence containment becomes implication. They use binary decision diagrams to check satisfiability; however, their type system does not provide general refinement types.

CDuce allows expressions to be pattern-matched against types and statically detects if a pattern-matching expression is non-exhaustive or if a branch is unreachable. If this is the case a counterexample XML document is generated that exhibits the problem. CDuce can also generate a counterexample document when subtyping fails, and issues warnings if empty types are detected. These tasks are much simpler in CDuce than they are in our setting, since we additionally have to deal with general refinement types. In particular the models produced by the SMT solver are not guaranteed to be real counterexamples, so we perform additional validation as explained in §7.

Complete functional synthesis [61] is a recent technique closely related to our **elementof** expressions. It involves generating specialized decision procedures to find concrete values that satisfy predicates expressed as Boolean expressions in a predictible way, while our **elementof** expressions find concrete values that satisfy predicates expressed as refinement types by calling an SMT solver. It would be interesting to investigate whether we can compile **elementof** expressions into code implementing specialized decision procedures, following the techniques of Kuncak et al. [61].

X10 [79] is an object-oriented language that supports refinement types. A class `C` can be refined with a constraint `c` on the immutable state of `C`, resulting in a type written `C(:c)`. The base language supports only simple equality constraints but further constraints can be added and multiple constraint solvers can be integrated into the compiler. In comparison with Dminor, X10 uses a mixture of semantic and syntactic subtyping, while its constraint language [79, §2.11] does not support type-test expressions.

Soft typing systems [26, 3, 88] infer types that represent program invariants, including shapes of S-expressions, but are not value-dependent. Typed Scheme [84, 85] makes use of shallow type-test expressions, union types and notions of visible and latent predicates to type-check Scheme programs. Typed Scheme records information from previous conditional expressions in a similar way we do in our rule (Exp Cond). It would be interesting to see if these idioms can be internalized in the Dminor type system using refinements—preliminary experiments are encouraging: Dminor can already handle all the first-order challenge examples introduced in the paper of Tobin-Hochstadt and Felleisen [85]. We list our typechecked translation of some of these examples below.

```
f(x:Integer | Text) : Integer {
  (x in Integer) ? (x + 1) : string_length(x) }

g(x : Integer | Text | Logical) : Any {
  (let temp = (x in Integer) in
     ((temp) ? temp :(x in Text)))
  ?f(x)
  :0 }

h(input : (Integer | Text), extra : {fst: Any; snd: Any;}) : Integer {
  ((input in Integer) && (extra.fst in Integer)) ?
      (input + extra.fst)
      : ((extra.fst in Integer) ?
           (string_length(input) + extra.fst)
           : 0) }
```

More recently, Greenberg et al. [47] have considered the use of refinement types combined with dynamic checks and first-class blame as a higher-order contracts language in the sense of Findler and Felleisen [39]. It remains future work to similarly consider an extension of Dminor with first-class blame as a contracts language. Some other prior work on dependent type systems has specifically targeted correct access to union types in COBOL [59] and in C [53].

PADS [40] develops a type theory for ad hoc data formats such as system traces, together with a rich range of tools for learning such formats and integrating into existing programming languages. The PADS type theory has refinement types, dependent pairs, and intersection types, but not type-test. There is a syntactic notion of type equivalence, but not subtyping. Dminor would be a useful language for programming transformations on data parsed using PADS, as our type system would enforce the constraints in PADS specifications, and hence guarantee statically that transformed data remains well-formed. Existing interfaces of PADS to C or to OCaml do not offer this guarantee.

DVerify [8] is a recent tool that verifies Dminor programs by translating them into a standard while language and then using Boogie [9] for generating verification conditions. DVerify directly uses our logical semantics from §3 to generate assertions in the while program that faithfully represent the typing constraints in the original Dminor program. Experimental evidence suggests that DVerify achieves very similar precision and efficiency compared to our prototype type-checker.

We do not consider type inference for Dminor; we assume that all function definitions have explicit type signatures. There has been considerable recent progress in inference algorithms for refinement types [77, 55, 87, 83, 52], some of which may be applicable to inferring type signatures for Dminor functions.

SMT solvers are widely used to find concrete inputs to imperative programs, by using forms of symbolic execution [56]. Our use of an SMT solver to find concrete values of types is in the same spirit. Both ideas amount to asking the SMT solver to find concrete counterexamples to formulas. To relate these approaches, it

[6] More precisely, regular expression types correspond exactly to tree automata and thus subtyping reduces to the inclusion problem between tree automata.

would be interesting future work to consider a symbolic execution of our operational semantics for type-test, and to compare the resulting formulas with our direct interpretation of refinement types as formulas.

10. Conclusions

We have described Dminor, a simple, first-order functional language for data processing that features an especially expressive type system. The novel combination of refinement types and type-test allows us to encode a rich variety of typing idioms; for example, intersection, union, negation, singleton, nullable, variant, and algebraic types are all derivable.

The main contribution of this paper is a technique to type-check Dminor programs *statically*: we combine the use of a bidirectional type system with the use of an SMT solver to perform semantic subtyping. Previous type systems have either devised special purpose algorithms for semantic subtyping, or used theorem provers only for refinement types. As far as we are aware, our use of an SMT solver to determine Dminor's very general notion of semantic subtyping is novel. We have implemented our type system in F$^\sharp$ using the Z3 SMT solver. We consider that SMT solvers are now of sufficient maturity that they can realistically be thought of as a platform upon which many applications may be built, including expressive type systems.

Our type-checker, like all static analyzers, has the potential to generate false negatives, that is, rejecting programs as type incorrect that are, in fact, type correct. As any SMT solver is incomplete for the first-order theories that we are interested in, it is possible that the solver is unable to determine an answer to a logical statement. SAGE [41, 57] avoids these problems by catching these cases and inserting a cast so that the test is performed again at run-time. This has the pleasant effect of not penalizing the developer for any possible incompleteness of the SMT solver. The techniques used in SAGE should apply to Dminor without any great difficulty.

We leave as future work the project of adding support for first-class functions; one direction is to generalize the mixture of syntactic and semantic subtyping introduced by Calcagno, Cardelli, and Gordon [25].

Finally, the implications of this work go beyond the core calculus Dminor. PADS, JSON, and M, for example, show the significance of programming languages for first-order data. Our work establishes the usefulness of combining refinement types and dynamic type-tests when programming with first-order data, and the viability of statically type-checking such programs with the aid of an SMT solver.

Acknowledgments We thank Nikolaj Bjørner for his invaluable help in using Z3. James Margetson helped with F$^\sharp$ programming issues. Paul Anderson, Ioannis Baltopoulos, Johannes Borgström, Nate Foster, Tim Harris, Ranjit Jhala, and Thorsten Tarrach commented on drafts. Discussions with Martín Abadi, Cliff Jones, Christoph Koch, Benjamin Pierce, and Dominique Unruh were useful, as were the comments of the anonymous reviewers. Cătălin Hriţcu was supported by a fellowship from Microsoft Research and the International Max Planck Research School for Computer Science.

A. Mechanized Definition of the Intended Model

In our logical semantics from §3 and its optimized version from §6.1, the semantics of a Dminor type is a first-order logic formula that is interpreted in a specific model. In this section we present the formal definition of this model in the Coq proof assistant [86]. Sorts are encoded as Coq types, and function symbols are interpreted as Coq functions. We focus only on the interpretation of the types and function symbols used by the optimized logical semantics from §6.1.

The formalization of the intended Dminor model is valuable for two main reasons. First, the formalization ensures that the model is properly defined. The recursive functions in the model are checked by Coq to be total and terminating. Additionally, we have proved in Coq that all functions preserve the logical invariants of the types on which they operate. For instance the FOL sort Value is interpreted as the Coq subset [81] type $\{x : \mathsf{RawValue} \mid \mathsf{Normal}\ x\}$, which required us to prove that all functions in the model only produce values in normal form (from values in normal form).

Second, having a natural model that is defined independently of the axioms that are fed to the SMT solver (see Appendix B) allows us to reason about the soundness of these axioms with respect to the model. The soundness of the axioms is not always obvious since the axioms are meant to have good performance in the SMT solver, rather than to be easy to trust by a human. Proving soundness in a standard model of axioms provided to an automated prover is not new, of course; it was done for instance by Boehme, Leino and Wolff [18] who proved in Isabelle/HOL that the axiomatization underlying their verification method for C code is correct. The Boyer-Moore family of theorem provers [20] allows both writing logical definitions for models and executing them efficiently [48].

A.1 Values

We first define scalars (sort General) and "raw" values as inductive types. Entities are represented as lists of string-raw-value pairs, while collections are represented as lists of raw values.

Model: Raw Values

```
Inductive General : Type :=
  | G_Integer : Z → General
  | G_Text : string → General
  | G_Logical : bool → General
  | G_Null : General.
Inductive RawValue : Type :=
  | G : General → RawValue
  | E : list (string * RawValue) → RawValue
  | C : list RawValue → RawValue.
```

This representation is not canonical, that is, multiple representations for the same value exist, which means we cannot use syntactic equality to compare raw values.

Instead of working directly with raw values, we only consider raw values that are in a normal form. Entities in normal form are sorted by their field name (a string), and do not contain duplicate field names. Collections in normal form are sorted with respect to a total order on raw values (this order is arbitrary but fixed; this order is irrelevant for the semantics of *pure* expressions). The main advantage of using values in normal form is that FOL equality can be interpreted as syntactic equality, as is usual for FOL models.[7]

Model: Sorted String-value Maps and Value Bags

```
Definition leAll (x : A) (ys : list A) := forall y, In y ys → le x y.
Inductive Sorted: list A → Prop :=
  | Sorted_nil: Sorted nil
  | Sorted_cons: forall hd tl, leAll hd tl → Sorted tl → Sorted (hd :: tl).
```

[7] If we had gone with a more complicated interpretation of equality, we would have needed to restrict the interpretation of function symbols to equality-respecting functions since the interpretation of equality needs to be a congruence.

```
Definition le_sv (sv1 sv2 : (string ∗ RawValue)) : Prop :=
  match sv1, sv2 with
  (s1,_), (s2,_) ⇒ cmp_str s1 s2 = Lt ∨ cmp_str s1 s2 = Eq
  end.
Definition Sorted_svm (svm : list (string ∗ RawValue)) : Prop :=
  Sorted le_sv svm.
Definition le_rval (v1 v2 : RawValue) : Prop :=
  cmp_rval v1 v2 = Lt ∨ cmp_rval v1 v2 = Eq.
Definition Sorted_vb (vb : list RawValue) : Prop := Sorted le_rval vb.
```

Model: Normal Values

```
Inductive Normal : RawValue → Prop :=
  | normal_G : forall g, Normal (G g)
  | normal_E : forall svm,
     NoDup (fst (split svm)) →
     Sorted_svm svm →
     IndAll Normal (snd (split svm)) →
       Normal (E svm)
  | normal_C : forall vb,
     Sorted_vb vb → IndAll Normal vb → Normal (C vb).
```

We define the Coq type Value (the interpretation of the FOL sort Value) as the subset [81] of RawValue for which the Normal predicate holds. The sorts SVMap and VBag are interpreted by similar Coq subset types. The elements of SVMap are association lists, lists of key/value pairs; the three conditions in the definition of SVMap require that the list of keys contain no duplicates (so that it forms a finite map), that the list of keys is ordered (so that it is in normal form), and that each of the contained values is itself in normal form.

Model: Coq Types Interpreting FOL Sorts

```
Definition Value := {x : RawValue | Normal x}.
Definition SVMap :=
  {svm : list (string ∗ RawValue) | NoDup (fst (split svm))
    ∧ Sorted_svm svm ∧ IndAll Normal (snd (split svm)) }.
Definition VBag :=
  {vb : list RawValue | Sorted_vb vb ∧ IndAll Normal vb}.
```

We define testers and accessors for values and scalars in the usual way. For instance, is_E checks whether its argument is an entity, and if this is the case out_E can be used to obtain the association list corresponding to the entity.

Model: Testers and Accessors

```
Program Definition is_E (v : Value) : bool :=
  match v with | E _ ⇒ true | _ ⇒ false end.
Program Definition out_E (v : Value) : SVMap :=
  match v with | E svm ⇒ svm | _ ⇒ nil end.
```

When the argument to out_E is not an entity the function returns the empty association list. This choice is arbitrary, but it is necessary that all the functions in the model are total. The testers and accessors for the other constructors are defined in the same way, and for the sake of brevity we omit their definitions here. For collections and entities we further wrap the testers, to permit more flexibility in the way they are axiomatized.

Model: Good Entities and Collections

```
Definition Good_C := is_C.
Definition Good_E := is_E.
```

A.2 Operations on Simple Values

The functions In_Logical, In_Integer, and In_Text test whether a value is in the corresponding scalar type.

Model: Testers for Simple Values

```
Definition In_Logical v := (is_G v) && is_G_Logical (out_G v).
Definition In_Integer v := (is_G v) && is_G_Integer (out_G v).
Definition In_Text v := (is_G v) && is_G_Text (out_G v).
```

We also define shorthand notation for constructing simple values.

Model: Constructors for Simple Values

```
Program Definition v_tt : Value := G (G_Logical true).
Program Definition v_ff : Value := G (G_Logical false).
Program Definition v_logical (b : bool) : Value := G (G_Logical b).
Program Definition v_int i : Value := G (G_Integer i).
Program Definition v_text s : Value := G (G_Text s).
Program Definition v_null : Value := G (G_Null).
```

The operations on simple values are straightforward, and are implemented directly by their Coq counterparts.

Model: Operators on Simple Values

```
Definition O_Sum v1 v2 :=
  v_int (Zplus (of_G_Integer(out_G v1)) (of_G_Integer(out_G v2))).
Definition O_Minus v1 v2 :=
  v_int (Zminus (of_G_Integer(out_G v1)) (of_G_Integer(out_G v2))).
Definition O_Mult v1 v2 :=
  v_int (Zmult (of_G_Integer(out_G v1)) (of_G_Integer(out_G v2))).
Definition O_GT v1 v2 :=
  match Zcompare (of_G_Integer(out_G v1)) (of_G_Integer(out_G v2))
  with Gt ⇒ v_tt | _ ⇒ v_ff end.
Definition O_LT v1 v2 :=
  match Zcompare (of_G_Integer(out_G v1)) (of_G_Integer(out_G v2))
  with Lt ⇒ v_tt | _ ⇒ v_ff end.
Definition O_EQ v1 v2 := v_logical (syn_beq_val v1 v2).
Definition O_Not v := v_logical (negb (of_G_Logical (out_G v))).
Definition O_And v1 v2 := v_logical (andb (of_G_Logical (out_G v1))
                                          (of_G_Logical (out_G v2))).
Definition O_Or v1 v2 := v_logical (orb (of_G_Logical (out_G v1))
                                        (of_G_Logical (out_G v2))).
```

A.3 Operations on Entities

The model provides two operations for creating entities: v_eempty creates an empty entity, and v_eupdate creates a new entity from an existing one by updating one field. If the updated field already exists in the original entity then the value of this field will be lost in the new entity. The implementation of v_eupdate uses an auxiliary function update_in_sorted_svm that implements insertion sorting for association lists. If the key to be added is, however, already present in the association list, then update_in_sorted_svm additionally removes the old entry. The v_eupdate operation does not correspond to any Dminor construct (although it would be easy to add functional entity updates to Dminor) but it allows us to construct entity values in an abstract way (without caring how they are implemented—for example, lists versus arrays).

Model: Creating Entities

```
Program Definition v_eempty : Value := E nil.
Program Definition v_eupdate (s : string) (v e : Value) : Value :=
  E (update_in_sorted_svm (s, v) (out_E e)).
```

The two basic operations on entities are: v_has_field that tests whether an entity has a certain field, and v_dot that given an entity that has a certain field selects the value of this field. Functions v_has_field and v_dot use the Coq library function TheoryList.assoc to obtain the value associated with a given key in a list of pairs.

Model: Basic Operations on Entities

```
Program Definition v_has_field (s : string) (v : Value) : bool :=
  match TheoryList.assoc eq_str_dec s (out_E v) with
  | Some v ⇒ true | None ⇒ false end.
Program Definition v_dot (s : string) (v : Value) : Value :=
  match TheoryList.assoc eq_str_dec s (out_E v) with
  | Some v ⇒ v | None ⇒ v_null end.
```

A.4 Operations on Collections

The constant v_zero represents the empty collection. The boolean function v_mem tests whether a value is present in a collection using the TheoryList.mem function from the Coq standard library. The function v_add adds an element to a collection using an auxiliary function insert_in_sorted_vb, which implements insertion sorting for collections. In turn v_add is used to define v_add_many, which adds i instances of a given value to a collection.

Model: Functions and Predicates on Collections

```
Program Definition v_zero : Value := C nil.

Program Definition v_mem (v cv : Value) : bool :=
  TheoryList.mem eq_rval_dec v (out_C cv).
Program Definition v_add (v cv : Value) : Value :=
  (C (insert_in_sorted_vb v (out_C cv))).

Fixpoint v_add_many' (v : Value) (n : nat) (cv : Value) : Value :=
  match n with 0 ⇒ cv | S n' ⇒ v_add_many' v n' (v_add v cv) end.
Definition v_add_many (v : Value) (i : Z) (cv : Value) : Value :=
  v_add_many' v (Zabs_nat i) cv.

Definition Closure2 := Value → Value → Value.
Definition v_apply2 (c : Closure2) v1 v2 := c v1 v2.

Program Fixpoint v_acc_fold (f : Closure2) (vb : VBag) (a : Value)
  {measure List.length vb} : Value :=
  match vb with nil ⇒ a | v :: vb' ⇒ v_acc_fold vb' (f a v) end.

Definition v_accumulate (clos:Closure2) c := v_acc_fold clos (out_C c).
```

Finally, v_accumulate implements folding over the elements of a collection using the fixed order on raw values. For the semantics of pure expressions, the order cannot influence the final result.

B. Axiomatization of the Dminor Model

We axiomatize the model of Dminor in sorted first-order logic extended with the background theories of equality, integer arithmetic, algebraic datatypes, and extensional arrays. We only axiomatize the parts of the model that are relevant for the optimized logical semantics in §6.1.

In the following we report all the relevant parts of this axiomatization, directly imported from our implementation (file Dminor-FoundationSmtLib.smt in the Dminor release). We use the standard SMT-LIB 1.2 format [75] supported by all recent SMT solvers, together with Z3-specific [33] extensions for algebraic datatypes and arrays [34].

We leave it as future work to prove formally that these axioms are properties of the model from Appendix A.

B.1 An Overview on Z3 Arrays

We use arrays in our axiomatization to represent collections (multisets) and entities (maps). An array is a function, with finite support, from one sort (the domain) into another (the range). The domain can be infinite, but the array can differ from a default element only on a finite subset of the domain. As a simple illustrative example (which is not part of our axiomatization for Dminor), we can define an array from integers to booleans, which basically represents a set of integers using its characteristic function.

Defining an Array Sort Representing Sets of Integers in Z3:

```
:define_sorts ((IntSetArray (Array Int bool)))
```

The basic theory of arrays was introduced by McCarthy [65] and characterizes functions store and select, using the following two axioms:

$$\forall a,i,v.\ \mathsf{select}(\mathsf{store}(a,i,v),i) = v$$
$$\forall a,i,j,v.\ i = j \vee \mathsf{select}(\mathsf{store}(a,i,v),j) = \mathsf{select}(a,j)$$

These axioms can be written in Z3 syntax (for our array sort IntSetArray above) as follows:

The Basic Theory of Arrays in Z3 Syntax

```
:assumption (forall (a IntSetArray) (i Int) (v bool)
  (= (select (store a i v) i) v))
:assumption (forall (a IntSetArray) (i Int) (j Int) (v bool)
  (or (= i j) (= (select (store a i v) j) (select a j))))
```

One additional property that is often desirable is extensionality, that two arrays are equal when they agree on all elements.

Extensionality of Arrays in Z3 Syntax:

```
:assumption (forall (a1 IntSetArray) (a2 IntSetArray)
  (implies (forall (i Int) (= (select a1 i) (select a2 i))) (= a1 a2)))
```

The select and store function symbols can for instance be used to implement a predicate set_contains that checks whether an integer is an element of a set, and a function set_remove to remove an element from a set.

Set Membership and Removing an Element from a Set:

```
:extrafuns ((set_contains IntSetArray Int bool)
  (set_remove IntSetArray Int IntSetArray))
:assumption (forall (a IntSetArray) (i Int)
  (= (set_contains a i) (select a i)) :pat{ (set_contains a i) })
:assumption (forall (a IntSetArray) (i Int)
  (= (set_remove a i) (store a i false)) :pat{ (set_remove a i) })
```

In the axioms above we use use quantifier patterns [32, 62] to restrict the number of quantifier instantiations. So the SMT solver will replace set_remove by a store, but not the other way around. Such careful fine-tuning allows one to choose the right trade-off between performance and completeness. More general quantifier patterns lead to more instantiations of the axioms, which can be expensive, and can lead to non-termination. On the other hand, too specific patterns can prevent the SMT solver from even trying to prove useful proof obligations.

Instead of relying directly on the axioms above, Z3 provides an efficient saturation procedure for the extensional array theory as well as a powerful extension called combinatory array logic [34]. The extension defines three new combinators, const, default, and map[f], which satisfy the following axioms.

Combinatory Array Logic Operations in Z3 Syntax:

```
:assumption (forall (i Int) (v bool)
  (= (select (const[IntSetArray] v) i) v))
:assumption (forall (v Bool) (= (default (const[IntSetArray] v)) v))
:assumption (forall (a IntSetArray) (i Int) (v bool)
  (= (default (store a i v)) (default a)))
:extrafuns ((f bool bool bool))
:assumption (forall (a1 IntSetArray) (a2 IntSetArray) (i Int)
  (= (select (map[f] a1 a2) i) (f (select a1 i) (select a2 i))))
```

For example, we can use these new combinators for defining a constant set_empty representing the empty set, a predicate set_finite capturing the finiteness of sets, and a function set_union that computes the union of two sets.

Additional Operations on Sets:

```
:extrafuns ((set_empty IntSetArray) (set_finite IntSetArray bool)
  (set_union IntSetArray IntSetArray IntSetArray))
:assumption (set_empty = const[IntSetArray] false)
:assumption (forall (a IntSetArray)
  (equiv (set_finite a) (= (default a) false)) : pat{ (set_finite a) })
:assumption (forall (a1 IntSetArray) (a2 IntSetArray)
  (= (set_union a1 a2) (map[and] a1 a2)) :pat{ (set_union a1 a2) })
```

In Z3 all the array axioms above are built-in (so they should not be added manually) and are efficiently implemented [34].

B.2 Values

We begin our axiomatization of Dminor by defining simple values. For strings and the labels of entities we define a new sort named String. The semantics of sorted first-order logic ensures that this sort is non-empty and disjoint from all other sorts. Since strings and labels are constants and we have no operation on them we do not further constrain this sort.

The sort General is defined as an algebraic datatype with four constructors: G_Integer taking a (built-in) integer as argument, G_Text taking a String, G_Logical taking a (built-in) boolean, and the constant G_Null.

Simple Values:

```
:extrasorts (String)
:datatypes ((General
  (G_Integer (of_G_Integer Int))
  (G_Text (of_G_Text String))
  (G_Logical (of_G_Logical bool))
  G_Null))
```

This declaration implicitly defines three accessor functions, named of_G_Integer, of_G_Text, and of_G_Logical, which are inverses to G_Integer, G_Text, and G_Logical. Given an argument of the form (G_Integer i), the function of_G_Integer returns i; of_G_Text and of_G_Logical act similarly. In addition, the declaration implicitly defines tester functions by adding the is_ prefix to the names of each constructor, so (is_G_Integer g) tests whether g is of the form (G_Integer i), and similarly for is_G_Text, is_G_Logical and is_G_Null.

Values also are defined as a datatype. We use extensional arrays to represent entities and collections. However, since Z3 syntactically restricts arrays from appearing inside datatypes, and since we need to restrict the arrays so that they represent only finite maps and bags, we use two new (abstract) sorts SVMap and VBag instead. The sort SVMap is then constrained to be isomorphic to the arrays from Strings to Values for which a finiteness condition holds, while VBag is required to be isomorphic to the arrays from Values to non-negative Ints, again with an additional finiteness condition[8].

Values:

```
:extrasorts (SVMap VBag)
:datatypes ((Value
    (G (out_G General)) ;; simple value (scalar)
    (E (out_E SVMap)) ;; entity: finite map from String to Value
    (C (out_C VBag)))) ;; collection: finite multiset of Value
```

[8] Because of this additional indirection, our axiomatization of sort Value captures not only the values of Dminor, but also infinite values that contain themselves (for example, a collection that has itself as an element). This is sound, since if a property can be proved of this larger set of values, then it also holds for the actual values. In practice it happens very rarely that the SMT solver manages to falsify a property by constructing a cyclic value; still, our code to process a Z3 model and to extract a counterexample (see §7) keeps track of cycles and aborts if one is encountered.

Since arrays can in general be infinite we further restrict the set of values to contain only finite collections and entities using the predicates Good_C and Good_E (defined below).

Good Values:

```
:assumption (forall (v Value)
  (implies (Good v)
        (and (implies (is_C v) (Good_C v))
             (implies (is_E v) (Good_E v))))
  :pat{ (Good v) })
```

B.3 Operations on Simple Values

We define several functions that test whether a value is a boolean (In_Logical), an integer (In_Integer), or a string (In_Text). These functions are trivial to implement because Z3 already provides testers for datatypes.

Testers for Simple Values:

```
:assumption (forall (v Value)
  (iff (In_Logical v) (and (is_G v) (is_G_Logical (out_G v))))
  :pat { (In_Logical v) })
:assumption (forall (v Value)
  (iff (In_Integer v) (and (is_G v) (is_G_Integer (out_G v))))
  :pat { (In_Integer v) })
:assumption (forall (v Value)
  (iff (In_Text v) (and (is_G v) (is_G_Text (out_G v))))
  :pat { (In_Text v) })
```

We also define more convenient constructors for simple values.

Constructors for Simple Values:

```
:assumption (= v_tt (G(G_Logical true)))
:assumption (= v_ff (G(G_Logical false)))
:assumption (forall (b bool) (= (v_logical b) (G(G_Logical b)))
  :pat { (v_logical b) })
:assumption (= v_null (G(G_Null)))
:assumption (forall (n Int) (= (v_int n) (G(G_Integer n)))
  :pat { (v_int n) } :pat { (G(G_Integer n)) } )
:assumption (forall (s String) (= (v_text s) (G(G_Text s)))
  :pat { (v_text s) } :pat { (G(G_Text s)) })
```

The operators on integers and booleans are easy to define using the built-in SMT-LIB functions.

Operators on Simple Values:

```
:assumption (forall (i1 Int) (i2 Int)
  (= (O_Sum (v_int i1) (v_int i2)) (v_int (+ i1 i2)))
  :pat { (O_Sum (v_int i1) (v_int i2)) })
:assumption (forall (v1 Value) (v2 Value)
  (= (O_EQ v1 v2) (ite (= v1 v2) v_tt v_ff))
  :pat { (O_EQ v1 v2) })
:assumption (forall (v Value)
  (= (O_Not v) (ite (not (= v v_tt)) v_tt v_ff))
  :pat { (O_Not v) })
:assumption (forall (v1 Value) (v2 Value)
  (= (O_And v1 v2) (ite (and (= v1 v_tt) (= v2 v_tt)) v_tt v_ff))
  :pat{ (O_And v1 v2) })
:assumption (forall (v1 Value) (v2 Value)
  (= (O_Or v1 v2) (ite (or (= v1 v_tt) (= v2 v_tt)) v_tt v_ff))
  :pat{ (O_Or v1 v2) })
```

We omit the definitions for O_NE, O_Minus, O_Mult, O_GT, and O_LT, which follow the same pattern.

B.4 Operations on Entities

Entities:

```
:datatypes ((ValueOption
  NoValue
  (SomeValue (of_SomeValue Value))))
:define_sorts ((SVMapArray (Array String ValueOption)))
:extrafuns ((alpham SVMap SVMapArray)
            (betam SVMapArray SVMap))
```

We represent entities as arrays from strings to the datatype ValueOption, which contains Values, as well as a special NoValue marker. We call such an array finite (FiniteE) if it has NoValue as the default element. We use the functions alpham and betam as the witnesses of the isomorphism between the abstract sort SVMap and the finite part of the array sort SVMapArray. The axiomatization of entities uses these witness functions intensively.

Operations on Entities:

```
;; SVMap and the finite arrays in SVMapArray are isomorphic
:assumption (forall (am SVMapArray)
  (implies (FiniteE am) (= (alpham (betam am)) am)))
:assumption (forall (svm SVMap)
  (and (FiniteE (alpham svm)) (= (betam (alpham svm)) svm)))

:assumption (forall (svm SVMapArray) (iff (FiniteE svm)
   (= (default svm) NoValue)) :pat{ (FiniteE svm) })

:assumption (forall (v Value)
  (iff (Good_E v) (and (is_E v) (FiniteE (alpham (out_E v)))))
  :pat{ (Good_E v) })

:assumption (= v_eempty (E (betam (const[SVMapArray] NoValue))))

:assumption (forall (l String) (v Value) (svm SVMap)
  (= (v_eupdate l v (E svm))
     (E (betam (store (alpham svm) l (SomeValue v)))))
  :pat{ (v_eupdate l v (E svm)) })

:assumption (forall (l String) (svm SVMap)
  (iff (v_has_field l (E svm)) (not(= (select (alpham svm) l) NoValue)))
  :pat { (v_has_field l (E svm)) }) ;;:pat (select (alpham svm) l)

:assumption (forall (l String) (svm SVMap)
  (= (v_dot l (E svm)) (of_SomeValue (select (alpham svm) l)))
  :pat { (v_dot l (E svm)) }) ;; :pat (select (alpham svm) l)
```

B.5 Operations on Collections

Collections:

```
:define_sorts ((VBagArray (Array Value Int)))
:extrafuns ((alphab VBag VBagArray)
            (betab VBagArray VBag))
```

We represent collections as arrays from Values to integers. We call such a collection good (Good_C) when it is finite (default value of the array is zero) and the cardinality of the elements is non-negative. Good collections correspond to finite multisets over values. We use the functions alphab and betab to define an isomorphism between the abstract sort VBag and the good collections in VBagArray.

Constraints on Bags:

```
;; VBag and the finite and positive arrays in VBagArray are isomorphic
:assumption (forall (ab VBagArray)
  (implies (and (Finite ab) (Positive ab)) (= (alphab (betab ab)) ab))
  :pat{ (alphab (betab ab)) })
:assumption (forall (vb VBag)
  (and (Finite (alphab vb)) (Positive (alphab vb))
    (= (betab (alphab vb)) vb))
  :pat{ (betab (alphab vb)) })

;; Good collections are finite and positive
:assumption (forall (v Value)
  (iff (Good_C v)
       (and (is_C v)
            (Finite (alphab (out_C v)))
            (Positive (alphab (out_C v)))))
  :pat{ (Good_C v) })

;; Finiteness of bags
:assumption (forall (a VBagArray)
  (iff (Finite a) (= (default a) 0))
  :pat{ (Finite a) })

;; Only positive indices in bags
:assumption (forall (a VBagArray)
  (iff (Positive a) (forall (v Value) (>= (select a v) 0)
  :pat{ (select a v) })) :pat{ (Positive a) })
```

Closures:

```
:extrasorts (Closure2)
:extrafuns ((v_apply2 Closure2 Value Value Value))
```

For axiomatizing v_accumulate we use an abstract sort for closures of two arguments (Closure2). The v_apply2 operation defines how each of the closures behaves on its arguments. When giving semantics to an expression **from** x **in** e_1 **let** $y = e_2$ **accumulate** e_3 a fresh closure is generated for e_2 (called f_e_2 below), and the following axiom is added for it:

```
:extrafuns ((f_e2 Closure2))
:assumption (forall (x y Value)
      (= (apply2 f_e2 x y) V[[e2]]) :pat{(apply2 f_e2 x y)})
```

The v_apply2 function is used in the second axiom for v_accumulate.

Operations on Collections:

```
:assumption (= v_zero (C (betab (const[VBagArray] 0))))

:assumption (forall (v Value) (vb VBag)
  (iff (v_mem v (C vb)) (> (select (alphab vb) v) 0))
  :pat { (v_mem v (C vb)) }
  :pat{ (select (alphab vb) v)} )

:assumption (forall (v Value) (i Int) (vb VBag)
  (= (v_add_many v i (C vb))
     (C (betab (store (alphab vb) v (+ i (select (alphab vb) v))))))
  :pat{ (v_add_many v i (C vb)) })

:assumption (forall (v Value) (vs Value)
  (= (v_add v vs) (v_add_many v 1 vs)) :pat{ (v_add v vs) })

;; v_accumulate iterates using an order-preserving function
:assumption (forall (clos Closure2) (initial Value)
  (= (v_accumulate clos v_zero initial) initial)
  :pat{ (v_accumulate clos v_zero initial) })

:assumption (forall (clos Closure2) (initial Value) (v Value) (vs Value)
  (= (v_accumulate clos (v_add v vs) initial)
     (v_accumulate clos vs (v_apply2 clos v initial)))
  :pat { (v_accumulate clos (v_add v vs) initial) })
```

C. Proofs

It is immediate from the definition of purity in §2.3 that purity is preserved by small-step reduction. This property is used in the proof of Theorem 1.

LEMMA 4 (Reduction Preserves Purity).
If e is pure and $e \rightarrow e'$ then e' is pure.

C.1 Relating Operational and Logical Semantics

In this section we develop proofs for Lemma 2 and Theorem 1 (Full Abstraction) from §3.

We begin with a direct inductive definition of the relation $e \Downarrow r$; that is, an error-tracking big-step operational semantics.

Evaluation Semantics: $e \Downarrow r$

(Eval Const)
$$\frac{}{c \Downarrow \mathbf{Return}(c)}$$

(Eval Operator 1)
$$\frac{e_i \Downarrow \mathbf{Return}(v_i) \quad \forall i \in 1..j-1 \quad e_j \Downarrow \mathbf{Error} \quad j \in 1..n}{\oplus(e_1,\dots,e_n) \Downarrow \mathbf{Error}}$$

(Eval Operator 2)
$$\frac{e_i \Downarrow \mathbf{Return}(v_i) \quad i \in 1..n \quad \neg\exists v.(\oplus(v_1,\dots,v_n) \mapsto v)}{\oplus(e_1,\dots,e_n) \Downarrow \mathbf{Error}}$$

(Eval Operator 3)
$$\frac{e_i \Downarrow \mathbf{Return}(v_i) \quad \forall i \in 1..n \quad \oplus(v_1,\dots,v_n) \mapsto v}{\oplus(e_1,\dots,e_n) \Downarrow \mathbf{Return}(v)}$$

(Eval Cond 1)
$$\frac{e_1 \Downarrow r \quad r \notin \{\mathbf{Return}(\mathbf{true}), \mathbf{Return}(\mathbf{false})\}}{e_1 ? e_{\mathbf{true}} : e_{\mathbf{false}} \Downarrow \mathbf{Error}}$$

(Eval Cond 2)
$$\frac{e_1 \Downarrow \mathbf{Return}(b) \qquad b \in \{\mathbf{true}, \mathbf{false}\} \qquad e_b \Downarrow r}{e_1 ? e_{\mathbf{true}} : e_{\mathbf{false}} \Downarrow r}$$

(Eval Let 1)
$$\frac{e_1 \Downarrow \mathbf{Error}}{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Downarrow \mathbf{Error}}$$

(Eval Let 2)
$$\frac{e_1 \Downarrow \mathbf{Return}(v) \quad e_2\{v/x\} \Downarrow r}{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Downarrow r}$$

(Eval Entity 1)
$$\frac{e_i \Downarrow \mathbf{Return}(v_i) \quad \forall i \in 1..j-1 \quad e_j \Downarrow \mathbf{Error} \quad j \in 1..n}{\{\ell_i \Rightarrow e_i\ {}^{i\in 1..n}\} \Downarrow \mathbf{Error}}$$

(Eval Entity 2)
$$\frac{e_i \Downarrow \mathbf{Return}(v_i) \quad \forall i \in 1..n}{\{\ell_i \Rightarrow e_i\ {}^{i\in 1..n}\} \Downarrow \mathbf{Return}(\{\ell_i \Rightarrow v_i\ {}^{i\in 1..n}\})}$$

(Eval Dot 1)
$$\frac{e \Downarrow r \quad \neg\exists v_1,\dots,v_n.(r = \mathbf{Return}(\{\ell_i \Rightarrow v_i\ {}^{i\in 1..n}\}) \wedge j \in 1..n)}{e.\ell_j \Downarrow \mathbf{Error}}$$

(Eval Dot 2)
$$\frac{e \Downarrow \mathbf{Return}(\{\ell_i \Rightarrow v_i\ {}^{i\in 1..n}\}) \quad j \in 1..n}{e.\ell_j \Downarrow \mathbf{Return}(v_j)}$$

(Eval Collection)
$$\frac{}{\{v_1,\dots,v_n\} \Downarrow \mathbf{Return}(\{v_1,\dots,v_n\})}$$

(Eval Add 1)
$$\frac{e_1 \Downarrow \mathbf{Error}}{e_1 :: e_2 \Downarrow \mathbf{Error}}$$

(Eval Add 2)
$$\frac{e_1 \Downarrow \mathbf{Return}(v) \quad e_2 \Downarrow r \quad \neg\exists v_1,\dots,v_n.(r = \mathbf{Return}(\{v_1,\dots,v_n\}))}{e_1 :: e_2 \Downarrow \mathbf{Error}}$$

(Eval Add 3)
$$\frac{e_1 \Downarrow \mathbf{Return}(v) \quad e_2 \Downarrow \mathbf{Return}(\{v_1,\dots,v_n\})}{e_1 :: e_2 \Downarrow \mathbf{Return}(\{v, v_1,\dots,v_n\})}$$

(Eval Appl 1)
$$\frac{e_i \Downarrow \mathbf{Return}(v_i) \quad \forall i \in 1..j-1 \quad e_j \Downarrow \mathbf{Error} \quad j \in 1..n}{f(e_1,\dots,e_n) \Downarrow \mathbf{Error}}$$

(Eval Appl 2)
$$\frac{\begin{array}{l} e_i \Downarrow \mathbf{Return}(v_i) \quad \forall i \in 1..n \quad e\{v_1/x_1\}\dots\{v_n/x_n\} \Downarrow r \\ \text{given function definition } f(x_1 : T_1,\dots,x_n : T_n) : U\{e\} \end{array}}{f(e_1,\dots,e_n) \Downarrow r}$$

(Eval Accum 1)
$$\frac{e_1 \Downarrow r \quad \neg\exists v_1,\dots,v_n.(r = \mathbf{Return}(\{v_1,\dots,v_n\}))}{\mathbf{from}\ x\ \mathbf{in}\ e_1\ \mathbf{let}\ y = e_2\ \mathbf{accumulate}\ e_3 \Downarrow \mathbf{Error}}$$

(Eval Accum 2)
$$\frac{\begin{array}{l} e_1 \Downarrow \mathbf{Return}(\{v_1,\dots,v_n\}) \\ \mathbf{let}\ y = e_2\ \mathbf{in}\ \mathbf{let}\ y = e_3\{v_1/x\}\ \mathbf{in} \dots \mathbf{let}\ y = e_3\{v_n/x\}\ \mathbf{in}\ y \Downarrow r \end{array}}{\mathbf{from}\ x\ \mathbf{in}\ e_1\ \mathbf{let}\ y = e_2\ \mathbf{accumulate}\ e_3 \Downarrow r}$$

(Test Wrong)
$$\frac{e \Downarrow \mathbf{Error}}{e\ \mathbf{in}\ T \Downarrow \mathbf{Error}}$$

(Test Any)
$$\frac{e \Downarrow \mathbf{Return}(v)}{e\ \mathbf{in}\ \mathsf{Any} \Downarrow \mathbf{Return}(\mathbf{true})}$$

(Test G 1)
$$\frac{e \Downarrow \mathbf{Return}(v) \quad v \in K(G)}{e\ \mathbf{in}\ G \Downarrow \mathbf{Return}(\mathbf{true})}$$

(Test G 2)
$$\frac{e \Downarrow \mathbf{Return}(v) \quad v \notin K(G)}{e\ \mathbf{in}\ G \Downarrow \mathbf{Return}(\mathbf{false})}$$

(Test Entity 1)
$$\frac{e \Downarrow \mathbf{Return}(v) \quad v = \{\ell_i \Rightarrow v_i\ {}^{i\in 1..n}\} \wedge j \in 1..n \quad v_j\ \mathbf{in}\ T_j \Downarrow r}{e\ \mathbf{in}\ \{\ell_j : T_j\} \Downarrow r}$$

(Test Entity 2)
$$\frac{e \Downarrow r \quad \neg\exists v_1,\dots,v_n.(r = \mathbf{Return}(\{\ell_i \Rightarrow v_i\ {}^{i\in 1..n}\}) \wedge j \in 1..n)}{e\ \mathbf{in}\ \{\ell_j : T_j\} \Downarrow \mathbf{Return}(\mathbf{false})}$$

(Test Collection 1)
$$\frac{e \Downarrow \mathbf{Return}(v) \quad \neg\exists v_1,\dots,v_n.(v = \{v_1,\dots,v_n\})}{e\ \mathbf{in}\ T* \Downarrow \mathbf{Return}(\mathbf{false})}$$

(Test Collection 2)
$$\frac{e \Downarrow \mathbf{Return}(\{v_1,\dots,v_n\}) \quad v_1\ \mathbf{in}\ T\ \&\&\ \dots\ \&\&\ v_n\ \mathbf{in}\ T \Downarrow r}{e\ \mathbf{in}\ T* \Downarrow r}$$

(Test Refine)
$$\frac{e_1 \Downarrow \mathbf{Return}(v) \quad v\ \mathbf{in}\ T\ \&\&\ e_2\{v/x\} \Downarrow r}{e_1\ \mathbf{in}\ (x : T\ \mathbf{where}\ e_2) \Downarrow r}$$

LEMMA 5. *If v is a value then $v \Downarrow$ **Return**(v).*

PROOF: By induction on the structure of v. □

LEMMA 6. *Suppose e is closed. If $e \rightarrow e'$ and $e' \Downarrow r$ then $e \Downarrow r$.*

PROOF: By induction on the derivation of $e' \Downarrow r$, with a case analysis of the reduction $e \rightarrow e'$. We omit the details. □

LEMMA 7. *If e is closed and stuck then $e \Downarrow$ **Error**.*

PROOF: By induction on the structure of e. We omit the details. □

By the following lemma, we obtain an independent definition of the relation $e \Downarrow r$ in terms of the reduction relation and stuckness. This is the definition used in §2. The equivalent inductive definition given in this section is convenient for proofs.

LEMMA 8. *Suppose that e is closed.*

(1) *$e \Downarrow$ **Return**(v) if and only if $e \to^* v$.*

(2) *$e \Downarrow$ **Error** if and only if there is e' with $e \to^* e'$ and e' is stuck.*

PROOF: The forwards direction follow by straightforward inductions on the derivations of $e \Downarrow$ **Return**(v) and $e \Downarrow$ **Error**.

For the reverse direction of (1), we have $e = e_1 \to \cdots \to e_n \to v$. By Lemma 5, we have $v \Downarrow$ **Return**(v). By repeated applications of Lemma 6, we have $e_i \Downarrow$ **Return**(v) for each i from n down to 1, and indeed $e_i \Downarrow$ **Return**(v).

For the reverse direction of (2), suppose there is e' such that $e = e_1 \to \cdots \to e_n = e'$ and e' is stuck. By Lemma 7, we have $e_n \Downarrow$ **Error**. By repeated applications of Lemma 6, we have $e_i \Downarrow$ **Error** for each i from n down to 1, and indeed $e \Downarrow$ **Error**. □

LEMMA 9. *Suppose $\oplus : T_1, \dots, T_n \to T$.*

(1) *If $\models \boldsymbol{F}[\![T_i]\!](v_i)$ for each $i \in 1..n$ then there is v such that $\oplus(v_1, \dots, v_n) \mapsto v$.*

(2) *If $\oplus(v_1, \dots, v_n) \mapsto v$ then $\models \boldsymbol{F}[\![T_i]\!](v_i)$ for each $i \in 1..n$, and $\models \boldsymbol{F}[\![T]\!](v)$ and $\models O_\oplus(v_1, \dots, v_n) = v$.*

PROOF: By inspection. □

The following applies to each operator apart from equality $==$. As mentioned previously, equality is defined on any pair of closed values.

LEMMA 10. *If $\oplus : G_1, \dots, G_n \to G$ then $dom(\oplus) = K(G_1) \times \cdots \times K(G_n)$.*

Our semantics has the following substitution property.

LEMMA 11.

(1) *For all values v and all expressions e that only call labeled-pure functions,*

$$\models \boldsymbol{R}[\![e]\!]\{v/x\} = \boldsymbol{R}[\![e\{v/x\}]\!]$$

(2) *For all types T, values v, and FOL terms t:*

$$\models \boldsymbol{F}[\![T]\!](t)\{v/x\} \Leftrightarrow \boldsymbol{F}[\![T\{v/x\}]\!](t\{v/x\})$$

(3) *For all types T, values v, and FOL terms t:*

$$\models \boldsymbol{W}[\![T]\!](t)\{v/x\} \Leftrightarrow \boldsymbol{W}[\![T\{v/x\}]\!](t\{v/x\})$$

PROOF: By simultaneous induction on the structure of e and T. □

We would like to show that if a closed pure expression evaluates to a result, then that is the result of the expression according to the logical semantics (if $e \Downarrow r$ then $\models \mathbf{R}[\![e]\!] = r$). Intuitively this proof should proceed by induction on the structure of the derivation of $e \Downarrow r$. This works for all cases other than (Eval Accum 2), so it is instructive to observe a failed proof attempt. In this case we know that $e =$ **from** x **in** e_1 **let** $y = e_2$ **accumulate** e_3, $e_1 \Downarrow$ **Return**$(\{v_1, \dots, v_n\})$ and **let** $y = e_2$ **in let** $y = e_3\{v_1/x\}$ **in** ... **let** $y = e_3\{v_n/x\}$ **in** $y \Downarrow r$, for some arbitrary ordering $v_1, \dots, v_n$. So the induction hypothesis gives us that $\mathbf{R}[\![e_1]\!] =$ **Return**$(\{v_1, \dots, v_n\})$ and **Bind** $y \Leftarrow \mathbf{R}[\![e_2]\!]$ **in** $\mathbf{R}[\![$**let** $y = e_3\{v_1/x\}$ **in** ... **let** $y = e_3\{v_n/x\}$ **in** $y]\!] = r$. We need to show that $\mathbf{R}[\![$**from** x **in** e_1 **let** $y = e_2$ **accumulate** $e_3]\!] = r$. By purity we know that for any permutation of $v_1, \dots, v_n$, including the canonical one used by the model $v_{i_1}, \dots, v_{i_n}$ we have that **let** $y = e_2$ **in let** $y = e_3\{v_{i_1}/x\}$ **in** ... **let** $y = e_3\{v_{i_n}/x\}$ **in** $y \Downarrow r$. However, we cannot apply the induction hypothesis to this (possibly) different permutation.

In order to obtain a strong enough induction hypothesis in the accumulate case we define an auxiliary judgment $e \Downarrow_D r$, which has the same rules as $e \Downarrow r$, with the exception of (Eval Accum 2) that is replaced by the following rule:

Auxiliary Evaluation Relation: $e \Downarrow_D r$

(Eval Accum D)

$$\frac{\begin{array}{l} e_1 \Downarrow_D \textbf{Return}(\{v_1, \dots, v_n\}) \\ \forall k.\ v_{i_1^k}, \dots, v_{i_n^k} \text{ is a permutation of } v_1, \dots, v_n \\ \textbf{let } y = e_2 \textbf{ in let } y = e_3\{v_{i_1^k}/x\} \textbf{ in } \dots \textbf{let } y = e_3\{v_{i_n^k}/x\} \textbf{ in } y \Downarrow_D r_k \end{array}}{\textbf{from } x \textbf{ in } e_1 \textbf{ let } y = e_2 \textbf{ accumulate } e_3 \Downarrow_D r_j}$$

The new rule (Eval Accum D) does not pick an arbitrary ordering from the start, but instead it evaluates using all orderings and only in the end picks one of the results. It is very easy to show that if $e \Downarrow_D r$ then also $e \Downarrow r$.

LEMMA 12. *If e is closed and $e \Downarrow_D r$ then also $e \Downarrow r$.*

PROOF: By induction on the structure of the derivation of $e \Downarrow_D r$. □

If e is additionally pure then also the implication in the other direction holds.

LEMMA 13. *If e is closed and pure and $e \Downarrow r$ then $e \Downarrow_D r$.*

PROOF: By induction on the structure of the derivation of $e \Downarrow r$. The proof uses the fact that pure expressions have to terminate on all paths. □

LEMMA 14. *For closed and pure e and r, if $e \Downarrow_D r$ then $\models \boldsymbol{R}[\![e]\!] = r$.*

PROOF: The proof is by induction on the derivation of $e \Downarrow_D r$. Notice that the purity assumption arises explicitly in the case (Eval Accum D), as well as (Eval Appl 2) for function calls, which also uses Lemma 1. We list representative cases of the proof, but omit some details.

(Eval Appl 1)

$$\frac{e_i \Downarrow \textbf{Return}(v_i) \quad \forall i \in 1..j-1 \quad e_j \Downarrow \textbf{Error} \quad j \in 1..n}{f(e_1, \dots, e_n) \Downarrow \textbf{Error}}$$

By induction hypothesis, we have $\models \mathbf{R}[\![e_i]\!] =$ **Return**(v_i) for each $i \in 1..j-1$ and $\models \mathbf{R}[\![e_j]\!] =$ **Error**. We calculate as follows:

$$\begin{array}{rl} & \models \mathbf{R}[\![f(e_1, \dots, e_n)]\!] \\ = & \textbf{Bind } x_1 \Leftarrow \mathbf{R}[\![e_1]\!] \textbf{ in } \dots \textbf{Bind } x_n \Leftarrow \mathbf{R}[\![e_n]\!] \textbf{ in} \\ & \underline{f}(x_1, \dots, x_n) \\ = & \textbf{Error} \end{array}$$

(Eval Appl 2)

$$\frac{\begin{array}{c} e_i \Downarrow \textbf{Return}(v_i) \quad \forall i \in 1..n \quad e\{v_1/x_1\}\dots\{v_n/x_n\} \Downarrow r \\ \text{given function definition } f(x_1 : T_1, \dots, x_n : T_n) : U\{e\} \end{array}}{f(e_1, \dots, e_n) \Downarrow r}$$

By induction hypothesis, we have $\models \mathbf{R}[\![e_i]\!] =$ **Return**(v_i) for each $i \in 1..n$. Since $f(e_1, \dots, e_n)$ is pure, it must be that f is a pure-labeled function, and therefore that its body e is pure. Hence, by Lemma 1, the definition $f(x_1 : T_1, \dots, x_n : T_n) : U\{e\}$ and e is pure and $e\{v_1/x_1\}\dots\{v_n/x_n\} \Downarrow r$ imply $\models \underline{f}(v_1, \dots, v_n) = r$. We calculate as follows:

$$\begin{array}{rl} & \models \mathbf{R}[\![f(e_1, \dots, e_n)]\!] \\ = & \textbf{Bind } x_1 \Leftarrow \mathbf{R}[\![e_1]\!] \textbf{ in } \dots \textbf{Bind } x_n \Leftarrow \mathbf{R}[\![e_n]\!] \textbf{ in} \\ & \underline{f}(x_1, \dots, x_n) \\ = & \textbf{Bind } x_1 \Leftarrow \textbf{Return}(v_1) \textbf{ in } \dots \textbf{Bind } x_n \Leftarrow \textbf{Return}(v_n) \textbf{ in} \\ & \underline{f}(x_1, \dots, x_n) \\ = & \underline{f}(v_1, \dots, v_n) \\ = & r \end{array}$$

(Eval Accum D)

$$\frac{\begin{array}{l} e_1 \Downarrow_D \mathbf{Return}(\{v_1,\ldots,v_n\}) \\ \forall k.\ v_{i_1^k},\ldots,v_{i_n^k} \text{ is a permutation of } v_1,\ldots,v_n \\ \quad \mathbf{let}\ y = e_2\ \mathbf{in\ let}\ y = e_3\{v_{i_1^k}/x\}\ \mathbf{in} \ldots \mathbf{let}\ y = e_3\{v_{i_n^k}/x\}\ \mathbf{in}\ y \Downarrow_D r_k \end{array}}{\mathbf{from}\ x\ \mathbf{in}\ e_1\ \mathbf{let}\ y = e_2\ \mathbf{accumulate}\ e_3 \Downarrow_D r_j}$$

The induction hypothesis gives us that $\mathbf{R}\llbracket e_1 \rrbracket = \mathbf{Return}(\{v_1,\ldots,v_n\})$ and for any $v_{i_1^k},\ldots,v_{i_n^k}$ permutation of $v_1,\ldots,v_n$ $\mathbf{Bind}\ y \Leftarrow \mathbf{R}\llbracket e_2 \rrbracket\ \mathbf{in}\ \mathbf{R}\llbracket \mathbf{let}\ y = e_3\{v_{i_1^k}/x\}\ \mathbf{in}\ \ldots \mathbf{let}\ y = e_3\{v_{i_n^k}/x\}\ \mathbf{in}\ y \rrbracket = r_k$. We choose this to be the canonical permutation used by the model $v_{i_1^c},\ldots,v_{i_n^c}$ and obtain a result r_c for which $\mathbf{let}\ y = e_2\ \mathbf{in\ let}\ y = e_3\{v_{i_1^c}/x\}\ \mathbf{in} \ldots \mathbf{let}\ y = e_3\{v_{i_n^c}/x\}\ \mathbf{in}\ y \Downarrow_D r_c$. From Lemma 12 by point (2) in the definition of purity we obtain that $r_c = r_j$. We can thus calculate as follows:

$$\begin{array}{ll} \multicolumn{2}{l}{\models \mathbf{R}\llbracket \mathbf{from}\ x\ \mathbf{in}\ e_1\ \mathbf{let}\ y = e_2\ \mathbf{accumulate}\ e_3 \rrbracket} \\ = & \mathsf{res\_accumulate}((\mathbf{fun}\ x\ y \to \mathbf{R}\llbracket e_3 \rrbracket), v, v') \\ = & \mathbf{Bind}\ y \Leftarrow \mathbf{R}\llbracket e_2 \rrbracket\ \mathbf{in} \\ & \qquad \mathbf{R}\llbracket \mathbf{let}\ y = e_3\{v_{i_1^c}/x\}\ \mathbf{in} \ldots \mathbf{let}\ y = e_3\{v_{i_n^c}/x\}\ \mathbf{in}\ y \rrbracket \\ = & r_c = r_j \end{array}$$

(Eval Operator 1)

$$\frac{e_i \Downarrow \mathbf{Return}(v_i) \quad \forall i \in 1..j-1 \quad e_j \Downarrow \mathbf{Error} \quad j \in 1..n}{\oplus(e_1,\ldots,e_n) \Downarrow \mathbf{Error}}$$

Suppose that $\oplus : T_1,\ldots,T_n \to T$. By induction hypothesis, $\models \mathbf{R}\llbracket e_i \rrbracket = \mathbf{Return}(v_i)$ for each i and $\models \mathbf{R}\llbracket e_j \rrbracket = \mathbf{Error}$.
We calculate as follows:

$$\begin{array}{ll} \multicolumn{2}{l}{\models \mathbf{R}\llbracket \oplus(e_1,\ldots,e_n) \rrbracket} \\ = & \mathbf{Bind}\ x_1 \Leftarrow \mathbf{R}\llbracket e_1 \rrbracket\ \mathbf{in}\ \ldots \mathbf{Bind}\ x_n \Leftarrow \mathbf{R}\llbracket e_n \rrbracket\ \mathbf{in} \\ & \quad (\mathbf{if}\ \mathbf{F}\llbracket T_1 \rrbracket(x_1) \wedge \cdots \wedge \mathbf{F}\llbracket T_n \rrbracket(x_n) \\ & \quad\ \mathbf{then\ Return}(\mathsf{O}_\oplus(x_1,\ldots,x_n))\ \mathbf{else\ Error}) \\ = & \mathbf{Error} \end{array}$$

(Eval Operator 2)

$$\frac{e_i \Downarrow \mathbf{Return}(v_i) \quad i \in 1..n \quad \neg\exists v.(\oplus(v_1,\ldots,v_n) \mapsto v)}{\oplus(e_1,\ldots,e_n) \Downarrow \mathbf{Error}}$$

Suppose that $\oplus : T_1,\ldots,T_n \to T$. By induction hypothesis, $\models \mathbf{R}\llbracket e_i \rrbracket = \mathbf{Return}(v_i)$ for each i. By Lemma 9, $\neg\exists v.(\oplus(v_1,\ldots,v_n) \mapsto v)$ implies that $\models \neg(\mathbf{F}\llbracket T_1 \rrbracket(v_1) \wedge \cdots \wedge \mathbf{F}\llbracket T_n \rrbracket(v_n))$.
We calculate as follows:

$$\begin{array}{ll} \multicolumn{2}{l}{\models \mathbf{R}\llbracket \oplus(e_1,\ldots,e_n) \rrbracket} \\ = & \mathbf{Bind}\ x_1 \Leftarrow \mathbf{R}\llbracket e_1 \rrbracket\ \mathbf{in}\ \ldots \mathbf{Bind}\ x_n \Leftarrow \mathbf{R}\llbracket e_n \rrbracket\ \mathbf{in} \\ & \quad (\mathbf{if}\ \mathbf{F}\llbracket T_1 \rrbracket(x_1) \wedge \cdots \wedge \mathbf{F}\llbracket T_n \rrbracket(x_n) \\ & \quad\ \mathbf{then\ Return}(\mathsf{O}_\oplus(x_1,\ldots,x_n))\ \mathbf{else\ Error}) \\ = & (\mathbf{if}\ \mathbf{F}\llbracket T_1 \rrbracket(v_1) \wedge \cdots \wedge \mathbf{F}\llbracket T_n \rrbracket(v_n) \\ & \quad \mathbf{then\ Return}(\mathsf{O}_\oplus(v_1,\ldots,v_n))\ \mathbf{else\ Error}) \\ = & \mathbf{Error} \end{array}$$

(Eval Operator 3)

$$\frac{e_i \Downarrow \mathbf{Return}(v_i) \quad \forall i \in 1..n \quad \oplus(v_1,\ldots,v_n) \mapsto v}{\oplus(e_1,\ldots,e_n) \Downarrow \mathbf{Return}(v)}$$

Suppose that $\oplus : T_1,\ldots,T_n \to T$. By induction hypothesis, $\models \mathbf{R}\llbracket e_i \rrbracket = \mathbf{Return}(v_i)$ for each i. By Lemma 9, $(\oplus(v_1,\ldots,v_n) \mapsto v$ implies that $\models \mathbf{F}\llbracket T_1 \rrbracket(v_1) \wedge \cdots \wedge \mathbf{F}\llbracket T_n \rrbracket(v_n)$ and $\models \mathbf{F}\llbracket T \rrbracket(v)$ and $\models \mathsf{O}_\oplus(v_1,\ldots,v_n) = v$.

We calculate as follows:

$$\begin{array}{ll} \multicolumn{2}{l}{\models \mathbf{R}\llbracket \oplus(e_1,\ldots,e_n) \rrbracket} \\ = & \mathbf{Bind}\ x_1 \Leftarrow \mathbf{R}\llbracket e_1 \rrbracket\ \mathbf{in}\ \ldots \mathbf{Bind}\ x_n \Leftarrow \mathbf{R}\llbracket e_n \rrbracket\ \mathbf{in} \\ & \quad (\mathbf{if}\ \mathbf{F}\llbracket T_1 \rrbracket(x_1) \wedge \cdots \wedge \mathbf{F}\llbracket T_n \rrbracket(x_n) \\ & \quad\ \mathbf{then\ Return}(\mathsf{O}_\oplus(x_1,\ldots,x_n))\ \mathbf{else\ Error}) \\ = & (\mathbf{if}\ \mathbf{F}\llbracket T_1 \rrbracket(v_1) \wedge \cdots \wedge \mathbf{F}\llbracket T_n \rrbracket(v_n) \\ & \quad \mathbf{then\ Return}(\mathsf{O}_\oplus(v_1,\ldots,v_n))\ \mathbf{else\ Error}) \\ = & \mathbf{Return}(\mathsf{O}_\oplus(v_1,\ldots,v_n)) \\ = & \mathbf{Return}(v) \end{array}$$

(Eval Cond 1)

$$\frac{e_1 \Downarrow r \quad r \notin \{\mathbf{Return}(\mathbf{true}), \mathbf{Return}(\mathbf{false})\}}{e_1 ? e_{\mathbf{true}} : e_{\mathbf{false}} \Downarrow \mathbf{Error}}$$

By induction hypothesis, $\models \mathbf{R}\llbracket e_1 \rrbracket = r$. Since we have that $r \notin \{\mathbf{Return}(\mathbf{true}), \mathbf{Return}(\mathbf{false})\}$, either $r = \mathbf{Error}$ or $r = \mathbf{Return}(v)$ and $v \notin \{\mathbf{true}, \mathbf{false}\}$.
In either case, we calculate as follows:

$$\begin{array}{ll} \multicolumn{2}{l}{\models \mathbf{R}\llbracket e_1 ? e_{\mathbf{true}} : e_{\mathbf{false}} \rrbracket} \\ = & \mathbf{Bind}\ x \Leftarrow \mathbf{R}\llbracket e_1 \rrbracket\ \mathbf{in} \\ & (\mathbf{if}\ x = \mathbf{true\ then}\ \mathbf{R}\llbracket e_{\mathbf{true}} \rrbracket\ \mathbf{else} \\ & (\mathbf{if}\ x = \mathbf{false\ then}\ \mathbf{R}\llbracket e_{\mathbf{false}} \rrbracket\ \mathbf{else\ Error})) \\ = & \mathbf{Error} \end{array}$$

(Eval Cond 2)

$$\frac{e_1 \Downarrow \mathbf{Return}(b) \qquad b \in \{\mathbf{true}, \mathbf{false}\} \quad e_b \Downarrow r}{e_1 ? e_{\mathbf{true}} : e_{\mathbf{false}} \Downarrow r}$$

By induction hypothesis, we have $\models \mathbf{R}\llbracket e_1 \rrbracket = \mathbf{Return}(b)$ and $\models \mathbf{R}\llbracket e_b \rrbracket = r$.
We calculate as follows:

$$\begin{array}{ll} \multicolumn{2}{l}{\models \mathbf{R}\llbracket e_1 ? e_{\mathbf{true}} : e_{\mathbf{false}} \rrbracket} \\ = & \mathbf{Bind}\ x \Leftarrow \mathbf{R}\llbracket e_1 \rrbracket\ \mathbf{in} \\ & (\mathbf{if}\ x = \mathbf{true\ then}\ \mathbf{R}\llbracket e_{\mathbf{true}} \rrbracket\ \mathbf{else} \\ & (\mathbf{if}\ x = \mathbf{false\ then}\ \mathbf{R}\llbracket e_{\mathbf{false}} \rrbracket\ \mathbf{else\ Error})) \\ = & (\mathbf{if}\ b = \mathbf{true\ then}\ \mathbf{R}\llbracket e_{\mathbf{true}} \rrbracket\ \mathbf{else} \\ & (\mathbf{if}\ b = \mathbf{false\ then}\ \mathbf{R}\llbracket e_{\mathbf{false}} \rrbracket\ \mathbf{else\ Error})) \\ = & \mathbf{R}\llbracket e_b \rrbracket \\ = & r \end{array}$$

(Eval Let 1)

$$\frac{e_1 \Downarrow \mathbf{Error}}{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Downarrow \mathbf{Error}}$$

By induction hypothesis, $\models \mathbf{R}\llbracket e_1 \rrbracket = \mathbf{Error}$.
In either case, we calculate as follows:

$$\begin{array}{ll} \multicolumn{2}{l}{\models \mathbf{R}\llbracket \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \rrbracket} \\ = & \mathbf{Bind}\ x \Leftarrow \mathbf{R}\llbracket e_1 \rrbracket\ \mathbf{in}\ \mathbf{R}\llbracket e_2 \rrbracket \\ = & \mathbf{Error} \end{array}$$

(Eval Let 2)

$$\frac{e_1 \Downarrow \mathbf{Return}(v) \quad e_2\{v/x\} \Downarrow r}{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Downarrow r}$$

By induction hypothesis, we have $\models \mathbf{R}\llbracket e_1 \rrbracket = \mathbf{Return}(v)$ and $\models \mathbf{R}\llbracket e_2\{v/x\} \rrbracket = r$.

Given Lemma 11, we calculate as follows:

$$\begin{aligned}
&\models \mathbf{R}[\![\textbf{let } x = e_1 \textbf{ in } e_2]\!] \\
&\quad= \textbf{Bind } x \Leftarrow \mathbf{R}[\![e_1]\!] \textbf{ in } \mathbf{R}[\![e_2]\!] \\
&\quad= \mathbf{R}[\![e_2]\!]\{v/x\} \\
&\quad= \mathbf{R}[\![e_2\{v/x\}]\!] \\
&\quad= r
\end{aligned}$$

(Eval Add 1)

$$\frac{e_1 \Downarrow \textbf{Error}}{e_1 :: e_2 \Downarrow \textbf{Error}}$$

By induction hypothesis, $e_1 \Downarrow \textbf{Error}$ implies $\models \mathbf{R}[\![e_1]\!] = \textbf{Error}$.

$$\begin{aligned}
&\models \mathbf{R}[\![e_1 :: e_2]\!] \\
&\quad= \textbf{Bind } x_1 \Leftarrow \mathbf{R}[\![e_1]\!] \textbf{ in Bind } x_2 \Leftarrow \mathbf{R}[\![e_2]\!] \textbf{ in} \\
&\qquad (\textbf{if } \mathsf{is\_C}(x_2) \textbf{ then Return}(\mathsf{v\_add}(x_1,x_2)) \textbf{ else Error}) \\
&\quad= \textbf{Error}
\end{aligned}$$

(Eval Add 2)

$$\frac{e_1 \Downarrow \textbf{Return}(v) \quad e_2 \Downarrow r \quad \neg\exists v_1,\ldots,v_n.(r = \textbf{Return}(\{v_1,\ldots,v_n\}))}{e_1 :: e_2 \Downarrow \textbf{Error}}$$

By induction hypothesis, $e_1 \Downarrow \textbf{Return}(v)$ implies $\models \mathbf{R}[\![e_1]\!] = \textbf{Return}(v)$ and $e_2 \Downarrow r$ implies $\models \mathbf{R}[\![e_2]\!] = r$. Given that we have $\neg\exists v_1,\ldots,v_n.(r = \textbf{Return}(\{v_1,\ldots,v_n\}))$, either $r = \textbf{Error}$ or $r = \textbf{Return}(v)$ where $\models \neg\mathsf{is\_C}(v)$. In either case, we have:

$$\begin{aligned}
&\models \mathbf{R}[\![e_1 :: e_2]\!] \\
&\quad= \textbf{Bind } x_1 \Leftarrow \mathbf{R}[\![e_1]\!] \textbf{ in Bind } x_2 \Leftarrow \mathbf{R}[\![e_2]\!] \textbf{ in} \\
&\qquad (\textbf{if } \mathsf{is\_C}(x_2) \textbf{ then Return}(\mathsf{v\_add}(x_1,x_2)) \textbf{ else Error}) \\
&\quad= \textbf{Error}
\end{aligned}$$

(Eval Add 3)

$$\frac{e_1 \Downarrow \textbf{Return}(v) \quad e_2 \Downarrow \textbf{Return}(\{v_1,\ldots,v_n\})}{e_1 :: e_2 \Downarrow \textbf{Return}(\{v,v_1,\ldots,v_n\})}$$

By induction hypothesis, we have $e_1 \Downarrow \textbf{Return}(v)$ implies $\models \mathbf{R}[\![e_1]\!] = \textbf{Return}(v)$ and we also have that $e_2 \Downarrow \{v_1,\ldots,v_n\}$ implies $\models \mathbf{R}[\![e_2]\!] = \textbf{Return}(\{v_1,\ldots,v_n\})$.

$$\begin{aligned}
&\models \mathbf{R}[\![e_1 :: e_2]\!] \\
&\quad= \textbf{Bind } x_1 \Leftarrow \mathbf{R}[\![e_1]\!] \textbf{ in Bind } x_2 \Leftarrow \mathbf{R}[\![e_2]\!] \textbf{ in} \\
&\qquad (\textbf{if } \mathsf{is\_C}(x_2) \textbf{ then Return}(\mathsf{v\_add}(x_1,x_2)) \textbf{ else Error}) \\
&\quad= (\textbf{if } \mathsf{is\_C}(\{v_1,\ldots,v_n\}) \textbf{ then} \\
&\qquad \textbf{Return}(\mathsf{v\_add}(v,\{v_1,\ldots,v_n\})) \textbf{ else Error}) \\
&\quad= \textbf{Return}(\{v,v_1,\ldots,v_n\})
\end{aligned}$$

(Eval Accum 1)

$$\frac{e_1 \Downarrow r \quad \neg\exists v_1,\ldots,v_n.(r = \textbf{Return}(\{v_1,\ldots,v_n\}))}{\textbf{from } x \textbf{ in } e_1 \textbf{ let } y = e_2 \textbf{ accumulate } e_3 \Downarrow \textbf{Error}}$$

By induction hypothesis we have that $\mathbf{R}[\![e_1]\!] = r$. If $r = \textbf{Error}$ then we calculate

$$\begin{aligned}
&\models \mathbf{R}[\![\textbf{from } x \textbf{ in } e_1 \textbf{ let } y = e_2 \textbf{ accumulate } e_3]\!] \\
&\quad= \textbf{Bind } x_1 \Leftarrow \textbf{Error in Bind } x_2 \Leftarrow \mathbf{R}[\![e_2]\!] \textbf{ in} \\
&\qquad \mathsf{res\_accumulate}((\textbf{fun } x\ y \to \mathbf{R}[\![e_3]\!]), x_1, x_2) \\
&\quad= \textbf{Error}
\end{aligned}$$

If instead $r = \textbf{Return}(v)$ for some value that is not a collection then, either $\mathbf{R}[\![e_2]\!] = \textbf{Error}$ in which case we are easily done, or $\mathbf{R}[\![e_2]\!] = \textbf{Return}(v')$ in which case we calculate

$$\begin{aligned}
&\models \mathbf{R}[\![\textbf{from } x \textbf{ in } e_1 \textbf{ let } y = e_2 \textbf{ accumulate } e_3]\!] \\
&\quad= \mathsf{res\_accumulate}((\textbf{fun } x\ y \to \mathbf{R}[\![e_3]\!]), v, v') \\
&\quad= (\textbf{if } \mathsf{is\_C}(v) \textbf{ then } \mathsf{res\_acc\_fold}(f, \mathsf{out\_C}(v), \textbf{Return}(v')) \textbf{ else Error}) \\
&\quad= \textbf{Error}
\end{aligned}$$

The omitted cases proceed similarly. □

LEMMA 15. *For closed and pure e and r, if $e \Downarrow r$ then $\models \mathbf{R}[\![e]\!] = r$.*

PROOF: Immediate from Lemma 13 and Lemma 14.

RESTATEMENT OF LEMMA 2.
For all closed and pure e and e', if $e \to e'$ then $\models \mathbf{R}[\![e]\!] = \mathbf{R}[\![e']\!]$.

PROOF: Suppose $e \to e'$. By Lemma 4, since e is pure, so is e'. By point (2) of the definition of purity, there exists a unique result r such that $e' \Downarrow r$. By Lemma 6, $e \to e'$ and $e' \Downarrow r$ imply $e \Downarrow r$. By Lemma 15, we have both $\models \mathbf{R}[\![e']\!] = r$ and $\models \mathbf{R}[\![e]\!] = r$. By transitivity, $\models \mathbf{R}[\![e]\!] = \mathbf{R}[\![e']\!]$. □

RESTATEMENT OF THEOREM 1 (FULL ABSTRACTION).
For all closed pure expressions e and e', we have $\models \mathbf{R}[\![e]\!] = \mathbf{R}[\![e']\!]$ if and only if, for all r, $e \Downarrow r \Leftrightarrow e' \Downarrow r$.

PROOF: Since e and e' are closed and pure, by point (2) of the definition of purity, there exist unique results r and r' such that $e \Downarrow r$ and $e' \Downarrow r'$. By Lemma 15, we have $\models \mathbf{R}[\![e]\!] = r$ and $\models \mathbf{R}[\![e']\!] = r'$. Given these facts, we have: $\models \mathbf{R}[\![e]\!] = \mathbf{R}[\![e']\!]$ if and only if $r = r'$ if and only if for all r'', $e \Downarrow r'' \Leftrightarrow e' \Downarrow r''$. □

C.2 Algorithmic Purity Implies Purity

Here we develop the proof for Theorem 2 from §4, which shows that algorithmic purity implies purity. We start with a series of useful lemmas.

LEMMA 16. *Values are algorithmically pure.*

PROOF: This is straightforward since none of the restrictions required for algorithmic purity applies to values. □

LEMMA 17. *If e is algorithmically pure then so is $e\{v/x\}$ for any v.*

PROOF: By induction on the structure of e. The variable base case x is handled by Lemma 16. The function application case is straightforward since substitution only applies to function arguments, not function bodies. In the accumulate case we have

$$\begin{aligned}
\models \mathbf{R}[\![\textbf{let } y = e_3\{x_1/x\}\{y_1/y\} \textbf{ in } e_3\{x_2/x\}]\!] = \\
\mathbf{R}[\![\textbf{let } y = e_3\{x_2/x\}\{y_1/y\} \textbf{ in } e_3\{x_1/x\}]\!]
\end{aligned}$$

and need to show the following, where we may assume the bound variable y is distinct from x.

$$\begin{aligned}
\models \mathbf{R}[\![\textbf{let } y = e_3\{v/x\}\{x_1/x\}\{y_1/y\} \textbf{ in } e_3\{v/x\}\{x_2/x\}]\!] = \\
\mathbf{R}[\![\textbf{let } y = e_3\{v/x\}\{x_2/x\}\{y_1/y\} \textbf{ in } e_3\{v/x\}\{x_1/x\}]\!]
\end{aligned}$$

This equation follows from Lemma 11 together with the substitution property of the logic (remember that in FOL free variables are implicitly universally quantified). Condition (3) in the definition of algorithmic purity follows directly by the induction hypothesis. □

LEMMA 18. *If e is algorithmically pure then so is $e\sigma$ for any value substitution σ.*

PROOF: Since e only has finitely many free variables, we only need to consider non-empty finite substitutions of the form: $\sigma = \{v_0/x_0\}\ldots\{v_n/x_n\}$. The proof proceeds by induction on n, using Lemma 17 in the inductive case. □

LEMMA 19. *If e is algorithmically pure and $e \to e'$ then e' is also algorithmically pure.*

PROOF: By induction on the derivation of $e \to e'$, using the following equivalent inductive formulation of algorithmic purity.

Reformulation of Algorithmic Purity: $\vdash T$ **pure** $\vdash e$ **pure**

$\vdash$ Any pure always
$\vdash G$ pure always
$\vdash T*$ pure, if $\vdash T$ pure
$\vdash \{\ell\colon T\}$ pure, if $\vdash T$ pure
$\vdash (x : T$ **where** $e)$ pure, if $\vdash T$ pure and $\vdash e$ pure

$\vdash x$ pure always
$\vdash c$ pure always
$\vdash \oplus(e_1,\ldots,e_n)$ pure, if $\vdash e_i$ pure for each $i \in 1..n$
$\vdash e_1?e_2 : e_3$ pure, if $\vdash e_i$ pure for each $i \in 1..3$
$\vdash$ **let** $x = e_1$ **in** e_2 pure, if $\vdash e_1$ pure and $\vdash e_2$ pure
$\vdash e$ **in** T pure, if $\vdash e$ pure and $\vdash T$ pure
$\vdash \{\ell_i \Rightarrow e_i\ ^{i\in 1..n}\}$ pure, if $\vdash e_i$ pure for each $i \in 1..n$
$\vdash e.\ell$ pure if $\vdash e$ pure
$\vdash \{v_1,\ldots,v_n\}$ pure always
$\vdash e_1 :: e_2$ pure if $\vdash e_1$ pure and $\vdash e_2$ pure
$\vdash$ **from** x **in** e_1 **let** $y = e_2$ **accumulate** e_3 pure
  if $\vdash e_1$ pure and $\vdash e_2$ pure and $\vdash e_3$ pure
  and $\models \mathbf{R}[\![\mathbf{let}\ y = e_3\{x_1/x\}\{y_1/y\}\ \mathbf{in}\ e_3\{x_2/x\}]\!] =$
    $\mathbf{R}[\![\mathbf{let}\ y = e_3\{x_2/x\}\{y_1/y\}\ \mathbf{in}\ e_3\{x_1/x\}]\!]$
(where the variables x_1, x_2, and y_1 do not appear free in e_3)
$\vdash f(e_1,\ldots,e_n)$ pure if $\vdash e_i$ pure for each $i \in 1..n$
  and f is labeled-pure

The rest of the proof is routine and was mechanized in Coq. □

We show that the reduction relation is terminating on algorithmically pure expressions.

LEMMA 20. *All reduction sequences starting from closed algorithmically pure expressions are finite.*

PROOF SKETCH Recursive functions have to decrease the size of their arguments on each recursive call, which guarantees their termination. The only other source of repetitive computation are accumulate expressions. But collections are finite and the accumulates are immediately inlined, so this will again always terminate. □

LEMMA 21. *If e is closed and algorithmically pure then there exists (at least) a result r so that $e \Downarrow r$.*

PROOF: Immediate from Lemma 20. □

The most important step for showing the uniqueness of evaluation results for algorithmically pure expressions is to show that the result of evaluating such expressions coincides with the result provided by the logical semantics.

LEMMA 22. *For all closed and algorithmically pure expressions e and for all results r, if $e \Downarrow r$ then $\models \mathbf{R}[\![e]\!] = r$.*

PROOF: By induction on the structure of the derivation of $e \Downarrow r$, with appeal to the big-step semantics in Appendix C.1. The most interesting case is when e is an accumulate expression of the form: **from** x **in** e_1 **let** $y = e_2$ **accumulate** e_3. We have that $e_1 \Downarrow \mathbf{Return}(\{v_1,\ldots,v_n\})$ and

$$\mathbf{let}\ y = e_2\ \mathbf{in}\ \mathbf{let}\ y = e_3\{v_1/x\}\ \mathbf{in} \ldots \mathbf{let}\ y = e_3\{v_n/x\}\ \mathbf{in}\ y \Downarrow r$$

for some arbitrary ordering $v_1,\ldots,v_n$. So the induction hypothesis gives us that $\mathbf{R}[\![e_1]\!] = \mathbf{Return}(\{v_1,\ldots,v_n\})$ and **Bind** $y \Leftarrow \mathbf{R}[\![e_2]\!]$ **in** $\mathbf{R}[\![\mathbf{let}\ y = e_3\{v_1/x\}\ \mathbf{in} \ldots \mathbf{let}\ y = e_3\{v_n/x\}\ \mathbf{in}\ y]\!] = r$. We need to show that $\mathbf{R}[\![\mathbf{from}\ x\ \mathbf{in}\ e_1\ \mathbf{let}\ y = e_2\ \mathbf{accumulate}\ e_3]\!] = r$. By definition this is equivalent to showing that **Bind** $x_2 \Leftarrow \mathbf{R}[\![e_2]\!]$ **in** res_accumulate$((\mathbf{fun}\ x\ y \to \mathbf{R}[\![e_3]\!]),\{v_1,\ldots,v_n\},x_2) = r$. If $\mathbf{R}[\![e_2]\!] = \mathbf{Error}$ the conclusion is immediate, so we focus on the case when $\mathbf{R}[\![e_2]\!] = u$. We show by a separate induction that res_accumulate$((\mathbf{fun}\ x\ y \to \mathbf{R}[\![e_3]\!]),\{v_1,\ldots,v_n\},x_2)$ is equal to $\mathbf{R}[\![\mathbf{let}\ y = e_3\{v_{i_1}/x\}\ \mathbf{in} \ldots \mathbf{let}\ y = e_3\{v_{i_n}/x\}\ \mathbf{in}\ y]\!]$ for a canonical permutation $v_{i_1},\ldots,v_{i_n}$ of $v_1,\ldots,v_n$ (defined in the model).

So all that remains to be shown in this case is that $\models (\mathbf{let}\ y = u\ \mathbf{in}\ \mathbf{R}[\![\mathbf{let}\ y = e_3\{v_1/x\}\ \mathbf{in} \ldots \mathbf{let}\ y = e_3\{v_n/x\}\ \mathbf{in}\ y]\!]) = (\mathbf{let}\ y = u\ \mathbf{in}\ \mathbf{R}[\![\mathbf{let}\ y = e_3\{v_{i_1}/x\}\ \mathbf{in} \ldots \mathbf{let}\ y = e_3\{v_{i_n}/x\}\ \mathbf{in}\ y]\!])$. Since the sequence $v_{i_1},\ldots,v_{i_n}$ is a permutation of $v_1,\ldots,v_n$ it can be decomposed as a finite sequence of adjacent transpositions. The remainder of the proof goes by induction on the structure of this sequence of transpositions. The base case is trivial, since then $v_{i_1},\ldots,v_{i_n} = v_1,\ldots,v_n$. In the inductive case we use point (2) in the definition of algorithmic purity to conclude the proof. □

LEMMA 23. *If e is closed and algorithmically pure, and $e \Downarrow r_1$, and $e \Downarrow r_2$ then $r_1 = r_2$.*

PROOF: By Lemma 22 we have that $\models \mathbf{R}[\![e]\!] = r_1$ and $\models \mathbf{R}[\![e]\!] = r_2$. By transitivity it follows that $\models r_1 = r_2$, which directly implies that $r_1 = r_2$. □

LEMMA 24 (Subexpressions and Substitution). *For all value substitutions σ, if e' is a subexpression of $e\sigma$ then there exists e'' so that e'' is a subexpression of e and $e' = e''\sigma$.*

RESTATEMENT OF THEOREM 2.
If e is algorithmically pure then e is pure.

PROOF: We prove the following more general statement by mutual induction on e and T:

(a) If e is algorithmically pure then e is pure.
(b) For all T, if e is a subexpression of T and e is algorithmically pure then e is pure.

For proving (a), by definition, e is pure if and only if for any value substitution σ each of the following four properties hold:

(1) $e\sigma$ is terminating,
(2) there exists a unique result r such that $e\sigma \Downarrow r$,
(3) for every subexpression $f(e_1,\ldots,e_n)$ of $e\sigma$, the function f is labeled-pure, and
(4) all subexpressions of $e\sigma$ are pure.

Let σ be an arbitrary value substitution. By Lemma 18 we have that $e\sigma$ is algorithmically pure. The first three properties can be proved immediately without using the induction hypothesis. Property (1) follows from Lemma 20. Property (2) follows from Lemma 21 and Lemma 23. Property (3) is immediate from the definition of algorithmic purity.

The only property that uses the induction hypothesis is (4): all subexpressions of $e\sigma$ are pure. Let e' be an arbitrary subexpression of $e\sigma$. By Lemma 24 we have that there exists e'' so that e'' is a subexpression of e and $e' = e''\sigma$. We need to prove that $e''\sigma$ is pure, and we do this by case analysis on e. All cases follow immediately by applying the induction hypothesis. The only exception is when $e = e_0$ **in** T and e'' is a subexpression of T, but there we can use (b). The proof of (b) is also simple, by case analysis on the T, and uses the main induction hypothesis when T is a refinement type $(x : T_0$ **where** $e_0)$ and e'' is a subexpression of e_0. □

C.3 Logical Soundness

Here we develop the proof for Theorem 3 that relates type assignment to the logical semantics of types and expressions. We start with a series of useful lemmas.

LEMMA 25 (Transitivity of Semantic Subtyping).
If $E \vdash T <: T'$ and $E \vdash T' <: T''$ then $E \vdash T <: T''$.

PROOF: By expanding definitions. □

LEMMA 26. *If e is alg. pure then:*

$$\models \boldsymbol{F}[\![\mathsf{Ok}(e)]\!](t) \Leftrightarrow (\boldsymbol{R}[\![e]\!] = \textbf{\textit{Return}}(\textbf{\textit{true}}))$$

PROOF: We have:

$$\begin{aligned}
&\models \mathbf{F}[\![\mathsf{Ok}(e)]\!](t)\\
&\quad\Leftrightarrow\quad \mathbf{F}[\![(x : \mathsf{Any}\ \textbf{where}\ e)]\!](t) \qquad x \notin fv(e)\\
&\quad\Leftrightarrow\quad \mathbf{F}[\![\mathsf{Any}]\!](t) \land \textbf{let}\ x = t\ \textbf{in}\ (\mathbf{R}[\![e]\!] = \textbf{Return}(\textbf{true}))\\
&\quad\Leftrightarrow\quad (\mathbf{R}[\![e]\!] = \textbf{Return}(\textbf{true}))
\end{aligned}$$

□

LEMMA 27. *If e is alg. pure then:*

$$\models \boldsymbol{F}[\![[e : T]]\!](t) \Leftrightarrow \boldsymbol{F}[\![T]\!](t) \land (\boldsymbol{R}[\![e]\!] = \textbf{\textit{Return}}(t))$$

PROOF: We have:

$$\begin{aligned}
&\models \mathbf{F}[\![[e : T]]\!](t)\\
&\quad\Leftrightarrow\quad \mathbf{F}[\![(x : T\ \textbf{where}\ x == e)]\!](t) \qquad x \notin fv(e)\\
&\quad\Leftrightarrow\quad \mathbf{F}[\![T]\!](t) \land \textbf{let}\ x = t\ \textbf{in}\ (\mathbf{R}[\![x == e]\!] = \textbf{Return}(\textbf{true}))\\
&\quad\Leftrightarrow\quad \mathbf{F}[\![T]\!](t) \land\\
&\qquad\quad (\textbf{Bind}\ y \Leftarrow \mathbf{R}[\![e]\!]\ \textbf{in Return}(\mathsf{v\_logical}(t = y))) = \textbf{Return}(\textbf{true})\\
&\quad\Leftrightarrow\quad \mathbf{F}[\![T]\!](t) \land (\mathbf{R}[\![e]\!] = \textbf{Return}(t))
\end{aligned}$$

□

The following lemma characterizes singular subtyping in terms of the logical semantics.

LEMMA 28 (Singular Subtyping).
Suppose $E \vdash e : T$ and $E \vdash T'$ and $x \notin dom(E)$.

(1) *If e is alg. pure then:*
$E \vdash [e : T] <: T'$ *iff* $\models \boldsymbol{F}[\![E]\!] \land \boldsymbol{F}[\![T]\!](\mathsf{out\_V}(\boldsymbol{R}[\![e]\!]))$
$\Longrightarrow \boldsymbol{F}[\![T']\!](\mathsf{out\_V}(\boldsymbol{R}[\![e]\!]))$

(2) *If e is not alg. pure then:*
$E \vdash [e : T] <: T'$ *iff* $\models \boldsymbol{F}[\![E]\!] \land \boldsymbol{F}[\![T]\!](x) \Longrightarrow \boldsymbol{F}[\![T']\!](x)$

PROOF: In case (1), we have that:

$$\begin{aligned}
&E \vdash [e : T] <: T'\\
&\quad\text{iff}\quad \models (\mathbf{F}[\![E]\!] \land \mathbf{F}[\![[e : T]]\!](x)) \implies \mathbf{F}[\![T']\!](x)\\
&\quad\text{iff}\quad \models (\mathbf{F}[\![E]\!] \land \mathbf{F}[\![T]\!](x) \land (\mathbf{R}[\![e]\!] = \textbf{Return}(x))) \implies \mathbf{F}[\![T']\!](x)\\
&\quad\text{iff}\quad \models (\mathbf{F}[\![E]\!] \land \mathbf{F}[\![T]\!](x) \land (\mathsf{out\_V}(\mathbf{R}[\![e]\!]) = x) \implies \mathbf{F}[\![T']\!](x)\\
&\quad\text{iff}\quad \models (\mathbf{F}[\![E]\!] \land \mathbf{F}[\![T]\!](\mathsf{out\_V}(\mathbf{R}[\![e]\!])) \implies \mathbf{F}[\![T']\!](\mathsf{out\_V}(\mathbf{R}[\![e]\!]))
\end{aligned}$$

In case (2), $[e : T] = T$, and the calculation is immediate. □

By the following lemma, singular subtyping is transitive, and hence we have that any derivation of a type assignment can be seen as one instance of a structural rule plus one instance of (Exp Singular Subsum). This observation is useful, for example, in proving type preservation, Theorem 4.

LEMMA 29 (Transitivity of Singular Subtyping).
If $E \vdash [e : T] <: T'$ and $E \vdash [e : T'] <: T''$ then $E \vdash [e : T] <: T''$.

PROOF: An easy application of Lemma 28. □

We can now prove the logical soundness of the type system.

RESTATEMENT OF THEOREM 3 (LOGICAL SOUNDNESS).

(1) *If e is alg. pure and $E \vdash e : T$ then:*
  (a) $\models \boldsymbol{F}[\![E]\!] \implies \mathsf{Proper}(\boldsymbol{R}[\![e]\!])$
  (b) $\models \boldsymbol{F}[\![E]\!] \implies \boldsymbol{F}[\![T]\!](\mathsf{out\_V}(\boldsymbol{R}[\![e]\!]))$

(2) *If $E \vdash U$ then $\models \boldsymbol{F}[\![E]\!] \implies \forall y.\neg \boldsymbol{W}[\![U]\!](y)$, for $y \notin fv(U)$.*

PROOF: By mutual induction on the derivation of judgments. In the following, we appeal to the following general properties, which follow directly by definition.

- $\models \mathsf{Proper}(\textbf{Return}(t))$
- $\models \mathsf{out\_V}(\textbf{Return}(t)) = t$
- $\models \mathsf{Proper}(t_1) \implies \textbf{Bind}\ x \Leftarrow t_1\ \textbf{in}\ t_2 = t_2\{\mathsf{out\_V}(t_1)/x\}$

For case (1), we consider each of the rules of type assignment.

(Exp Singular Subsum)

$$\frac{E \vdash e : T \quad E \vdash [e : T] <: T'}{E \vdash e : T'}$$

For (a), we get $\models \mathbf{F}[\![E]\!] \implies \mathsf{Proper}(\mathbf{R}[\![e]\!])$, as required, by induction hypothesis. For (b), we get $\models \mathbf{F}[\![E]\!] \implies \mathbf{F}[\![T]\!](\mathsf{out\_V}(\mathbf{R}[\![e]\!]))$ by induction hypothesis. By Lemma 28, since e is algorithmically pure, we get:

$$\models \mathbf{F}[\![E]\!] \land \mathbf{F}[\![T]\!](\mathsf{out\_V}(\mathbf{R}[\![e]\!])) \implies \mathbf{F}[\![T']\!](\mathsf{out\_V}(\mathbf{R}[\![e]\!]))$$

Hence, we get $\models \mathbf{F}[\![E]\!] \implies \mathbf{F}[\![T']\!](\mathsf{out\_V}(\mathbf{R}[\![e]\!]))$, as required.

(Exp Var)

$$\frac{E \vdash \diamond \quad (x : T) \in E}{E \vdash x : T}$$

We have $\mathbf{R}[\![x]\!] = \textbf{Return}(x)$, and so point (a), $\models \mathbf{F}[\![E]\!] \implies \mathsf{Proper}(\textbf{Return}(x))$, follows at once, since $\models \forall x.\mathsf{Proper}(\textbf{Return}(x))$ by definition of Proper. It must be that $E = x_1 : T_1, \ldots, x_n : T_n$ with $x = x_i$ and $T = T_i$ for some $i \in 1..n$, so that $\mathbf{F}[\![E]\!] = \mathbf{F}[\![T_1]\!](x_1) \land \cdots \land \mathbf{F}[\![T_n]\!](x_n)$. Hence, point (b) $\models \mathbf{F}[\![E]\!] \implies \mathbf{F}[\![T]\!](\mathsf{out\_V}(\textbf{Return}(x)))$, amounts to $\models \mathbf{F}[\![T_1]\!](x_1) \land \cdots \land \mathbf{F}[\![T_n]\!](x_n) \implies \mathbf{F}[\![T_i]\!]x_i$, which follows, since $\models \forall x.\mathsf{out\_V}(\textbf{Return}(x)) = x$ by definition of $\mathsf{out\_V}$.

(Exp Const)

$$\frac{E \vdash \diamond}{E \vdash c : \mathsf{Any}}$$

We have $\mathbf{R}[\![c]\!] = \textbf{Return}(c)$, and so we have that point (a), $\models \mathbf{F}[\![E]\!] \implies \mathsf{Proper}(\textbf{Return}(c))$, follows at once. We have $\mathbf{F}[\![\mathsf{Any}]\!](t) = \textbf{true}$, and so point (b), $\models \mathbf{F}[\![E]\!] \implies \mathbf{F}[\![\mathsf{Any}]\!](\mathsf{out\_V}(\textbf{Return}(c)))$, follows at once.

(Exp Eq)

$$\frac{\begin{array}{l}E \vdash e_1 : T_1\\ E \vdash e_2 : T_2\\ T = \mathsf{Logical}\end{array}}{E \vdash e_1 == e_2 : T}$$

The argument for this case is essentially the same as for the following rule, (Exp Operator), together with the observation that:

$$\models \mathbf{F}[\![\mathsf{Logical}]\!](\mathsf{O\_EQ}(t_1, t_2))$$

(Exp Operator)

$$\frac{\begin{array}{l}\oplus \neq (==)\\ \oplus : T_1, \ldots, T_n \to T\\ E \vdash e_i : T_i \quad \forall i \in 1..n\end{array}}{E \vdash \oplus(e_1, \ldots, e_n) : T}$$

In this case we have:

$$\begin{array}{l}\mathbf{R}[\![\oplus(e_1,\dots,e_n)]\!] = \textbf{Bind } x_1 \Leftarrow \mathbf{R}[\![e_1]\!] \textbf{ in } \dots \textbf{Bind } x_n \Leftarrow \mathbf{R}[\![e_n]\!] \textbf{ in} \\ \quad (\textbf{if } \mathbf{F}[\![T_1]\!](x_1) \wedge \dots \wedge \mathbf{F}[\![T_n]\!](x_n) \\ \quad \textbf{then Return}(\mathsf{O}_\oplus(x_1,\dots,x_n)) \textbf{ else Error}) \\ \text{where } \oplus : T_1,\dots,T_n \to T\end{array}$$

By induction hypothesis, we have that $\models \mathbf{F}[\![E]\!] \implies \mathsf{Proper}(e_i)$ and $\models \mathbf{F}[\![E]\!] \implies \mathbf{F}[\![T]\!](\mathsf{out\_V}(\mathbf{R}[\![e_i]\!]))$ for each $i \in 1..n$. By inspection of the definition of $\mathsf{O}_\oplus$, $\oplus : T_1,\dots,T_n \to T$ implies that:

$$\models \mathbf{F}[\![T_1]\!](t_1) \wedge \dots \wedge \mathbf{F}[\![T_n]\!](t_n) \implies \mathbf{F}[\![T]\!](\mathsf{O}_\oplus(t_1,\dots,t_n))$$

Hence, we can calculate the following properties:

$$\begin{array}{l}\models \mathbf{F}[\![E]\!] \implies \mathbf{R}[\![\oplus(e_1,\dots,e_n)]\!] = \\ \qquad \textbf{Return}(\mathsf{O}_\oplus(\mathsf{out\_V}(\mathbf{R}[\![e_1]\!]),\dots,\mathsf{out\_V}(\mathbf{R}[\![e_n]\!]))) \\ \models \mathbf{F}[\![E]\!] \implies \mathbf{F}[\![T]\!](\mathsf{O}_\oplus(\mathsf{out\_V}(\mathbf{R}[\![e_1]\!]),\dots,\mathsf{out\_V}(\mathbf{R}[\![e_n]\!])))\end{array}$$

Hence, we obtain (a) $\models \mathbf{F}[\![E]\!] \implies \mathsf{Proper}(\mathbf{R}[\![\oplus(e_1,\dots,e_n)]\!])$ and (b) $\models \mathbf{F}[\![E]\!] \implies \mathbf{F}[\![T]\!](\mathsf{out\_V}(\mathbf{R}[\![\oplus(e_1,\dots,e_n)]\!]))$.

(Exp Cond)

$$\frac{E \vdash e_1 : \mathsf{Logical} \qquad E, \_ : \mathsf{Ok}(e_1) \vdash e_2 : T \qquad E, \_ : \mathsf{Ok}(!e_1) \vdash e_3 : T}{E \vdash (e_1 ? e_2 : e_3) : T}$$

In this case we have:

$$\begin{array}{l}\mathbf{R}[\![e_1 ? e_2 : e_3]\!] = \textbf{Bind } x \Leftarrow \mathbf{R}[\![e_1]\!] \textbf{ in} \\ \quad (\textbf{if } x = \textbf{true then } \mathbf{R}[\![e_2]\!] \textbf{ else } (\textbf{if } x = \textbf{false then } \mathbf{R}[\![e_3]\!] \textbf{ else Error}))\end{array}$$

By induction hypothesis, we have:

$$\begin{array}{l}\models \mathbf{F}[\![E]\!] \implies \mathsf{Proper}(\mathbf{R}[\![e_1]\!]) \\ \models \mathbf{F}[\![E]\!] \implies \mathbf{F}[\![\mathsf{Logical}]\!](\mathsf{out\_V}(\mathbf{R}[\![e_1]\!]))\end{array}$$

By definition of $\mathbf{F}[\![\mathsf{Logical}]\!](t)$, we have:

$$\models \mathbf{F}[\![\mathsf{Logical}]\!](t) \implies (t = \textbf{true}) \vee (t = \textbf{false})$$

Hence, we can calculate:

$$\begin{array}{l}\models \mathbf{F}[\![E]\!] \implies \\ \quad (\mathbf{R}[\![e_1]\!] = \textbf{Return}(\textbf{true}) \wedge \mathbf{R}[\![e_1 ? e_2 : e_3]\!] = \mathbf{R}[\![e_2]\!]) \vee \\ \quad (\mathbf{R}[\![e_1]\!] = \textbf{Return}(\textbf{false}) \wedge \mathbf{R}[\![e_1 ? e_2 : e_3]\!] = \mathbf{R}[\![e_3]\!])\end{array}$$

By induction hypothesis, with Lemma 26, we have:

$$\begin{array}{l}\models \mathbf{F}[\![E]\!] \wedge \mathbf{R}[\![e_1]\!] = \textbf{Return}(\textbf{true}) \implies \mathsf{Proper}(\mathbf{R}[\![e_2]\!]) \\ \models \mathbf{F}[\![E]\!] \wedge \mathbf{R}[\![e_1]\!] = \textbf{Return}(\textbf{true}) \implies \mathbf{F}[\![T]\!](\mathsf{out\_V}(\mathbf{R}[\![e_2]\!]))\end{array}$$

By induction hypothesis, with Lemma 26, we have:

$$\begin{array}{l}\models \mathbf{F}[\![E]\!] \wedge \mathbf{R}[\![!e_1]\!] = \textbf{Return}(\textbf{true}) \implies \mathsf{Proper}(\mathbf{R}[\![e_3]\!]) \\ \models \mathbf{F}[\![E]\!] \wedge \mathbf{R}[\![!e_1]\!] = \textbf{Return}(\textbf{true}) \implies \mathbf{F}[\![T]\!](\mathsf{out\_V}(\mathbf{R}[\![e_3]\!]))\end{array}$$

We can also calculate:

$$\models \mathbf{F}[\![E]\!] \wedge \mathbf{R}[\![e_1]\!] = \textbf{Return}(\textbf{false}) \implies \mathbf{R}[\![!e_1]\!] = \textbf{Return}(\textbf{true})$$

Altogether, we can calculate points (a) and (b):

$$\begin{array}{l}\models \mathbf{F}[\![E]\!] \implies \mathsf{Proper}(\mathbf{R}[\![e_1 ? e_2 : e_3]\!]) \\ \models \mathbf{F}[\![E]\!] \implies \mathbf{F}[\![T]\!](\mathsf{out\_V}(\mathbf{R}[\![e_1 ? e_2 : e_3]\!]))\end{array}$$

We omit the detailed arguments for the rules (Exp Let), (Exp Test), (Exp Entity), (Exp Dot), (Exp Coll), (Exp Add), (Exp Acc), and (Exp App), which follow similarly. The argument for (Exp Test) depends on case (2) of the mutual induction.

For case (2), we consider each of the rules of type formation. The argument for (Type Refine) depends on case (1) of the mutual induction. □

C.4 Preservation and Progress

Here we develop proofs of Theorems 4 and 5 imply the safety of our declarative type system from §5.

We have the following basic properties.

LEMMA 30 (Implied Judgments).

(1) *If $E \vdash T$ then $E \vdash \diamond$ and $fv(T) \subseteq dom(E)$.*
(2) *If $E \vdash T <: T'$ then $E \vdash T$ and $E \vdash T'$.*
(3) *If $E \vdash e : T$ then $E \vdash T$ and $fv(e) \subseteq dom(E)$.*

PROOF: By simultaneous induction on the derivations of each judgment. □

LEMMA 31 (All Values Typable). *For any v we have $E \vdash v : \mathsf{Any}$.*

PROOF: By induction on the structure of v. □

LEMMA 32 (Weakening). *Suppose $E, x : T' \vdash \diamond$ and $x \notin dom(E')$.*

(1) *If $E, E' \vdash \diamond$ then $E, x : T', E' \vdash \diamond$.*
(2) *If $E, E' \vdash T$ then $E, x : T', E' \vdash T$.*
(3) *If $E, E' \vdash S <: T$ then $E, x : T', E' \vdash S <: T$.*
(4) *If $E, E' \vdash e : T$ then $E, x : T', E' \vdash e : T$.*

PROOF: The proof is by simultaneous induction on the derivation of the judgments $E, E' \vdash \diamond$ and $E, E' \vdash T$ and $E, E' \vdash S <: T$ and $E, E' \vdash e : T$. □

LEMMA 33 (Bound Weakening). *Suppose $E \vdash T <: T'$.*

(1) *If $E, x : T', E' \vdash \diamond$ then $E, x : T, E' \vdash \diamond$.*
(2) *If $E, x : T', E' \vdash S$ then $E, x : T, E' \vdash S$.*
(3) *If $E, x : T', E' \vdash S <: S'$ then $E, x : T, E' \vdash S <: S'$.*
(4) *If $E, x : T', E' \vdash e : S$ then $E, x : T, E' \vdash e : S$.*

PROOF: By simultaneous induction on derivations. □

LEMMA 34 (Semantic Substitution).

(1) *For all e', x, alg. pure e so that $E \vdash e : V$ we have that $\models \mathbf{F}[\![E]\!] \implies \mathbf{R}[\![e']\!]\{\mathsf{out\_V}(\mathbf{R}[\![e]\!])/x\} = \mathbf{R}[\![e'\{e/x\}]\!]$;*
(2) *For all T, t, alg. pure e so that $E \vdash e : V$ we have that*
   (a) $\models \mathbf{F}[\![E]\!] \implies \mathbf{F}[\![T]\!](t)\{\mathsf{out\_V}(\mathbf{R}[\![e]\!])/x\} \Leftrightarrow \mathbf{F}[\![T\{e/x\}]\!](t\{e/x\})$;
   (b) $\models \mathbf{F}[\![E]\!] \implies \mathbf{W}[\![T]\!](y)\{\mathsf{out\_V}(\mathbf{R}[\![e]\!])/x\} \Leftrightarrow \mathbf{W}[\![T\{e/x\}]\!](t\{e/x\})$.

PROOF By mutual induction on the structure of e' and T. For (2)(a), the interesting case is when $T = (z : U \textbf{ where } e')$ for $z \neq x$. By definition of the semantics we have that $\mathbf{F}[\![T]\!](t) = \mathbf{F}[\![U]\!](t) \wedge \textbf{let } z = y \textbf{ in } \mathbf{R}[\![e']\!] = \textbf{Return}(\textbf{true})$, so we have that

$$\begin{array}{l}\mathbf{F}[\![T]\!](t)\{\mathsf{out\_V}(\mathbf{R}[\![e]\!])/x\} = \\ \quad \mathbf{F}[\![U]\!](t)\{\mathsf{out\_V}(\mathbf{R}[\![e]\!])/x\} \\ \quad \wedge \textbf{let } z = t \textbf{ in } \mathbf{R}[\![e']\!]\{\mathsf{out\_V}(\mathbf{R}[\![e]\!])/x\} = \textbf{Return}(\textbf{true}).\end{array}$$

For $\mathbf{F}[\![U]\!](t)\{\mathsf{out\_V}(\mathbf{R}[\![e]\!])/x\}$ we apply part (2)(a) of the induction hypothesis, while for $\mathbf{R}[\![e']\!]\{\mathsf{out\_V}(\mathbf{R}[\![e]\!])/x\}$ we apply part (1) and obtain that

$$\begin{array}{l}\models \mathbf{F}[\![E]\!] \implies \mathbf{F}[\![U]\!](t)\{\mathsf{out\_V}(\mathbf{R}[\![e]\!])/x\} = \\ \quad (\mathbf{F}[\![U\{e/x\}]\!](t\{e/x\}) \\ \quad \wedge \textbf{let } z = t\{e/x\} \textbf{ in } \mathbf{R}[\![e'\{e/x\}]\!] = \textbf{Return}(\textbf{true})).\end{array}$$

By definition of the logical semantics

$$\begin{array}{l}(\mathbf{F}[\![U\{e/x\}]\!](t\{e/x\}) \wedge \\ \quad \textbf{let } z = t\{e/x\} \textbf{ in } \mathbf{R}[\![e'\{e/x\}]\!] = \textbf{Return}(\textbf{true})) = \\ \qquad = \mathbf{F}[\![(z : U\{e/x\} \textbf{ where } e'\{e/x\})]\!](t\{e/x\}).\end{array}$$

which, by the definition of substitution is the same as $\mathbf{F}[\![(z : U \textbf{ where } e')\{e/x\}]\!](t\{e/x\})$, as required to complete the proof of the case.

For proving (1) one interesting case is when $e' = x$, so we need to show that $\models \mathbf{F}[\![E]\!] \implies \textbf{Return}(x)\{\mathsf{out\_V}(\mathbf{R}[\![e]\!])/x\} = \mathbf{R}[\![e]\!]$, or equivalently $\models \mathbf{F}[\![E]\!] \implies \textbf{Return}(\mathsf{out\_V}(\mathbf{R}[\![e]\!])) = \mathbf{R}[\![e]\!]$. This follows by Theorem 3, point (1)(a), using the assumption that e is well-typed in E and the definition of Proper. For the case when $e' = e'' \textbf{ in } T$ we use all three induction hypotheses. We start from the right-hand side and calculate that

$$\begin{array}{l}\mathbf{R}[\![e'\{e/x\}]\!] = \mathbf{R}[\![e''\{e/x\} \textbf{ in } T\{e/x\}]\!] = \\ \quad \textbf{Bind } y \Leftarrow \mathbf{R}[\![e''\{e/x\}]\!] \textbf{ in } (\textbf{if } \mathbf{W}[\![T\{e/x\}]\!](y) \textbf{ then Error else} \\ \quad\quad (\textbf{if } \mathbf{F}[\![T\{e/x\}]\!](y) \textbf{ then Return}(\textbf{true}) \textbf{ else Return}(\textbf{false}))),\end{array}$$

for some $y \neq x$ (so we have that $y = y\{e/x\}$). By all the three parts of the induction hypothesis

$$\begin{array}{l}\models \mathbf{F}[\![E]\!] \implies \mathbf{R}[\![e''\{e/x\}]\!] = \\ \quad \textbf{Bind } y \Leftarrow \mathbf{R}[\![e'']\!]\{\mathsf{out\_V}(\mathbf{R}[\![e]\!])/x\} \textbf{ in} \\ \quad\quad (\textbf{if } \mathbf{W}[\![T]\!](y)\{\mathsf{out\_V}(\mathbf{R}[\![e]\!])/x\} \textbf{ then Error else} \\ \quad\quad\quad (\textbf{if } \mathbf{F}[\![T]\!](y)\{\mathsf{out\_V}(\mathbf{R}[\![e]\!])/x\} \textbf{ then Return}(\textbf{true}) \textbf{ else} \\ \quad\quad\quad\quad \textbf{Return}(\textbf{false}))).\end{array}$$

By the definitions of substitution and the logical semantics this is the same as

$$\models \mathbf{F}[\![E]\!] \implies \mathbf{R}[\![e'' \textbf{ in } T]\!]\{\mathsf{out\_V}(\mathbf{R}[\![e]\!])/x\} = \mathbf{R}[\![e''\{e/x\}]\!].$$

The other cases are similar but easier. □

LEMMA 35. *For all E, E', alg. pure e, x, if $E \vdash e : T$ then:*

$$\models \boldsymbol{F}[\![E]\!] \implies \boldsymbol{F}[\![E']\!]\{\mathsf{out\_V}(\boldsymbol{R}[\![e]\!])/x\} \Leftrightarrow \boldsymbol{F}[\![E'\{e/x\}]\!]$$

PROOF: By induction on the structure of E', with appeal to Lemma 34. □

LEMMA 36 (Lookup).
If $E \vdash \diamond$ and $(x : T) \in E$ and $(x : T') \in E$ then $T = T'$.

PROOF: If $E \vdash \diamond$ all the entries in E are for distinct variables. □

LEMMA 37 (Substitution). *Suppose $E \vdash e' : T'$ and e' alg. pure.*

(1) *If $E, x : T', E' \vdash \diamond$ then $E, E'\{e'/x\} \vdash \diamond$.*
(2) *If $E, x : T', E' \vdash T$ then $E, E'\{e'/x\} \vdash T\{e'/x\}$.*
(3) *If $E, x : T', E' \vdash S <: T$ then $E, E'\{e'/x\} \vdash S\{e'/x\} <: T\{e'/x\}$.*
(4) *If $E, x : T', E' \vdash e : T$ then $E, E'\{e'/x\} \vdash e\{e'/x\} : T\{e'/x\}$.*

PROOF: The proof is by simultaneous induction on the derivation of the judgments.

(1) The case for (Env Empty) is immediate. The case for (Env Var) relies on induction with point (2).
(2) The cases for well-formed types are similar induction steps.
(3) We have $E, x : T', E' \vdash S <: T$. By inverting (Subtype), we have $E, x : T', E' \vdash S$ and $E, x : T', E' \vdash T$ and

$$\models (\mathbf{F}[\![E, x : T', E']\!] \wedge \mathbf{F}[\![S]\!](y)) \implies \mathbf{F}[\![T]\!](y)$$

for some $y \notin dom(E, x : T', E')$.
Since y is fresh, we may assume $y \notin fv(e')$.
By induction with point (2), we have $E, E'\{e'/x\} \vdash S\{e'/x\}$ and $E, E'\{e'/x\} \vdash T\{e'/x\}$.
Expanding definitions, we have:

$$\models (\mathbf{F}[\![E]\!] \wedge \mathbf{F}[\![T']\!](x) \wedge \mathbf{F}[\![E']\!] \wedge \mathbf{F}[\![S]\!](y)) \implies \mathbf{F}[\![T]\!](y)$$

Since $x \notin fv(E, T', y)$, by substituting $\mathsf{out\_V}(\mathbf{R}[\![e']\!])$ for x, we obtain:

$$\begin{array}{l}\models (\mathbf{F}[\![E]\!] \wedge \mathbf{F}[\![T']\!](\mathsf{out\_V}(\mathbf{R}[\![e']\!])) \wedge \mathbf{F}[\![E']\!]\{\mathsf{out\_V}(\mathbf{R}[\![e']\!])/x\} \wedge \\ \quad \mathbf{F}[\![S]\!](y)\{\mathsf{out\_V}(\mathbf{R}[\![e']\!])/x\}) \implies \mathbf{F}[\![T]\!](y)\{\mathsf{out\_V}(\mathbf{R}[\![e']\!])/x\}\end{array}$$

By Lemma 11, Lemma 34, and Lemma 35, we have:

$$\begin{array}{l}\models (\mathbf{F}[\![E]\!] \wedge \mathbf{F}[\![T']\!](\mathsf{out\_V}(\mathbf{R}[\![e']\!])) \wedge \mathbf{F}[\![E'\{e'/x\}]\!] \wedge \\ \quad \mathbf{F}[\![S\{e'/x\}]\!](y)) \implies \mathbf{F}[\![T\{e'/x\}]\!](y)\end{array}$$

By Theorem 3, $E \vdash e' : T'$ and e' alg. pure imply:

$$\models \mathbf{F}[\![E]\!] \Rightarrow \mathbf{F}[\![T']\!](\mathsf{out\_V}(\mathbf{R}[\![e']\!]))$$

Combining the previous two displayed formulas, we obtain:

$$\begin{array}{l}\models (\mathbf{F}[\![E]\!] \wedge \mathbf{F}[\![E'\{e'/x\}]\!] \wedge \\ \quad \mathbf{F}[\![S\{e'/x\}]\!](y)) \implies \mathbf{F}[\![T\{e'/x\}]\!](y)\end{array}$$

To summarize, for some $y \notin dom(E, E'\{e'/x\})$ we have judgments $E, E'\{e'/x\} \vdash S\{e'/x\}$ and $E, E'\{e'/x\} \vdash T\{e'/x\}$ and

$$\models (\mathbf{F}[\![E, E'\{e'/x\}]\!] \wedge \mathbf{F}[\![S\{e'/x\}]\!](y)) \implies \mathbf{F}[\![T\{e'/x\}]\!](y)$$

Hence, by (Subtype), we are done.
(4) In case (Exp Var), we have $E, x : T', E' \vdash x : T$ derived from $E, x : T', E' \vdash \diamond$ with $(x : T) \in (E, x : T', E')$.
By induction with point (2), $E, E'\{e'/x\} \vdash \diamond$.
By Lemma 36, $T = T'$.
By assumption, then, $E \vdash e' : T$.
By Lemma 32, this and $E, E'\{e'/x\} \vdash \diamond$ imply $E, E'\{e'/x\} \vdash e' : T$, as required.
The other cases follow by similar induction steps. □

LEMMA 38.

(1) *If $\varnothing \vdash e : T$, $\varnothing \vdash e' : T$, $e \rightarrow e'$ and e is alg. pure then $\varnothing \vdash [e' : T] <: [e : T]$.*
(2) *If $\varnothing \vdash e : \mathsf{Logical}$, $\varnothing \vdash e' : \mathsf{Logical}$, $e \rightarrow e'$ then $\varnothing \vdash \mathsf{Ok}(e') <: \mathsf{Ok}(e)$.*

PROOF:

(1) From Lemma 19 we have that e' is also alg. pure. We have that $\mathbf{F}[\![[e' : T]]\!](x)$ holds iff $\mathbf{F}[\![T]\!](x) \wedge (\mathbf{R}[\![e']\!] = \textbf{Return}(x))$. By Lemma 2 this is equivalent to $\mathbf{F}[\![T]\!](x) \wedge (\mathbf{R}[\![e]\!] = \textbf{Return}(x))$, which is equivalent to $\mathbf{F}[\![[e : T]]\!](x)$.
(2) We proceed by considering e.

Case e is alg. pure. From Lemma 19 we have that e' is also alg. pure. We have that $\mathbf{F}[\![\mathsf{Ok}(e')]\!](x)$ holds iff $\mathbf{F}[\![\mathsf{Any}]\!](x) \wedge (\mathbf{R}[\![e']\!] = \textbf{Return}(x))$. By Lemma 2 this is equivalent to $\mathbf{F}[\![\mathsf{Any}]\!](x) \wedge (\mathbf{R}[\![e]\!] = \textbf{Return}(x))$, which is equivalent to $\mathbf{F}[\![\mathsf{Ok}(e)]\!](x)$.

Case e is not alg. pure. By definition $\mathsf{Ok}(e)$ is Any, and so our property holds as Any is the top type. □

Another useful lemma relates subtyping with reduction via substitution.

LEMMA 39. *If $\varnothing \vdash e : U$, $\varnothing \vdash e' : U$, $e \rightarrow e'$ and e is alg. pure then $\varnothing \vdash T\{e/x\} <: T\{e'/x\}$ and $\varnothing \vdash T\{e'/x\} <: T\{e/x\}$.*

PROOF: Assume $\mathbf{F}[\![T\{e/x\}]\!](t)$ for some term t. Then by Lemma 34 $\mathbf{F}[\![T\{e/x\}]\!](t)$ holds just if $\mathbf{F}[\![T]\!](t)\{\mathsf{out\_V}(\mathbf{R}[\![e]\!])/x\}$. By Lemma 2 this is equivalent to $\mathbf{F}[\![T]\!](t)\{\mathsf{out\_V}(\mathbf{R}[\![e']\!])/x\}$. Again by Lemma 34 this holds if and only if $\mathbf{F}[\![T\{e'/x\}]\!](t)$, as e' is alg. pure by Lemma 19. □

Before we proceed to the preservation theorem, we first need some inversion lemmas for entity and collection types.

LEMMA 40 (Entity type inversion).

(1) *If $E \vdash \{\ell_i \Rightarrow v_i{}^{i \in 1..n}\} : \{\ell_i : T_i{}^{i \in 1..n}\}$ then $E \vdash v_i : T_i$, for $i \in 1..n$.*
(2) *If $E \vdash \{\ell_i \Rightarrow v_i{}^{i \in 1..n}\} : \mathsf{Any}$ then $E \vdash v_i : \mathsf{Any}$, for $i \in 1..n$.*

LEMMA 41 (Collection type inversion).

(1) *If* $E \vdash \{v_1,\dots,v_n\} : T*$ *then* $E \vdash v_i : T$, *for* $i \in 1..n$.
(2) *If* $E \vdash \{v_1,\dots,v_n\} : \mathsf{Any}$ *then* $E \vdash v_i : \mathsf{Any}$, *for* $i \in 1..n$.

We also need the following lemma, which captures the intuition that if we know that a value inhabits a type, then assuming that it does not inhabit that type leads to a degenerate subtype relation.

LEMMA 42. *If* $E \vdash v : T$ *then* $E, \_ : \mathsf{Ok}(!v\ \mathbf{in}\ T) \vdash U <: V$, *for any types* U,V *such that* $E \vdash U$ *and* $E \vdash V$.

RESTATEMENT OF THEOREM 4 (PRESERVATION)
If $\varnothing \vdash e : T$ *and* $e \to e'$ *then* $\varnothing \vdash e' : T$.

PROOF: By induction on the derivation of $\varnothing \vdash e : T$. For space reasons we give details of three rules; the rest follow in a similar fashion.

(Exp Singular Subsum)

$$\frac{\varnothing \vdash e : T \quad \varnothing \vdash [e:T] <: T'}{\varnothing \vdash e : T'}$$

By induction hypothesis, we have that $\varnothing \vdash e' : T$. We also know that $\varnothing \vdash T'$. We proceed by considering e.

Case e is alg. pure. By Lemma 38 we can conclude that $\varnothing \vdash [e' : T] <: [e : T]$ (noting that by Lemma 19, e' is also alg. pure). From this and the fact that $\varnothing \vdash [e : T] <: T'$ we can conclude $\varnothing \vdash [e' : T] <: T'$ by transitivity of subtyping (Lemma 25). From this and $\varnothing \vdash e' : T$ we can deduce $\varnothing \vdash e' : T'$ from (Exp Singular Subsum).

Case e is not alg. pure. By definition we have $\varnothing \vdash T <: T'$ and by applying the derived rule (Exp Subsum) we can deduce $\varnothing \vdash e' : T'$.

(Exp Cond)

$$\frac{\begin{array}{l}\varnothing \vdash e_1 : \mathsf{Logical}\\ \_ : \mathsf{Ok}(e_1) \vdash e_2 : T\\ \_ : \mathsf{Ok}(!e_1) \vdash e_3 : T\end{array}}{\varnothing \vdash (e_1 ? e_2 : e_3) : T}$$

We proceed by considering the reduction step taken by the expression $e_1 ? e_2 : e_3$.

Case $e_1 ? e_2 : e_3 \to e_1' ? e_2 : e_3$. By induction we have $\varnothing \vdash e_1' : \mathsf{Logical}$. By applying part (2) of Lemma 38 we have $\varnothing \vdash \mathsf{Ok}(e_1') <: \mathsf{Ok}(e_1)$. By Lemma 33 we have that $\_ : \mathsf{Ok}(e_1') \vdash e_2 : T$. By similar reasoning we have $\_ : \mathsf{Ok}(!e_1') \vdash e_3 : T$ and so by (Exp Cond) we can conclude $\varnothing \vdash (e_1' ? e_2 : e_3) : T$.

Case $\mathbf{true} ? e_2 : e_3 \to e_2$. We have that $\_ : \mathsf{Ok}(\mathbf{true}) \vdash e_2 : T$, and by Lemma 37 we can deduce that $\varnothing \vdash e_2 : T$.

Case $\mathbf{false} ? e_2 : e_3 \to e_3$. Similar reasoning to above.

(Exp App)

$$\frac{\begin{array}{l}\text{given } f(x_1 : T_1,\dots,x_n : T_n) : U\{e_f\}\\ \sigma_i = \{e_1/x_1\}\dots\{e_i/x_i\} \quad \forall i \in 0..n\\ e_i \text{ alg. pure} \quad \varnothing \vdash e_i : T_i\sigma_{i-1} \quad \forall i \in 1..n\end{array}}{\varnothing \vdash f(e_1,\dots e_n) : U\sigma_n}$$

We proceed by considering the reduction step taken by the expression $f(e_1,\dots e_n)$.

**Case $f(v_1,\dots,v_{i-1},e_i,e_{i+1},\dots,e_n) \to$
$f(v_1,\dots,v_{i-1},e_i',e_{i+1},\dots,e_n)$**

As $e_i \to e_i'$, we have by induction that $\varnothing \vdash e_i' : T_i\sigma_{i-1}$. We have that $\varnothing \vdash e_{i+1} : T_{i+1}\sigma_i$ where $T_{i+1}\sigma_i$ is equivalent to $(T_{i+1}\sigma_{i-1})\{e_i/x_i\}$. By Lemma 39 we have that $\varnothing \vdash (T_{i+1}\sigma_{i-1})\{e_i'/x_i\} <: (T_{i+1}\sigma_{i-1})\{e_i/x_i\}$. Thus we can deduce that $\varnothing \vdash e_{i+1} : (T_{i+1}\sigma_{i-1})\{e_i'/x_i\}$, and similarly $\varnothing \vdash e_n : (T_{i+1}\sigma_{i-1})\{e_i'/x_i\}\dots\{e_n/x_n\}$. By (Exp App) that $\varnothing \vdash f(v_1,\dots,v_{i-1},e_i',e_{i+1},\dots,e_n) : (U\sigma_i)\{e_i'/x_i\}\dots\{e_n/x_n\}$. By Lemma 39 we can conclude that $\varnothing \vdash f(e_1,\dots e_n) : U\sigma_n$.

Case $f(v_1,\dots,v_n) \to e_f\{v_1/x_1\}\dots\{v_n/x_n\}$. We have by assumption that $x_1 : T_1,\dots,x_n : T_n \vdash e_f : U$. By Lemma 37 we can conclude $\varnothing \vdash e_f\{v_1/x_1\}\dots\{v_n/x_n\} : U\{v_1/x_1\}\dots\{v_n/x_n\}$.

□

LEMMA 43 (Canonical Forms).

(1) *If* $\varnothing \vdash v : \mathsf{Integer}$ *then* $v = \underline{i}$ *for some integer i.*
(2) *If* $\varnothing \vdash v : \mathsf{Text}$ *then* $v = \underline{s}$ *for some string s.*
(3) *If* $\varnothing \vdash v : \mathsf{Logical}$ *then* $v = \mathbf{true}$ *or* $v = \mathbf{false}$.
(4) *If* $\varnothing \vdash v : T*$ *then* $v = \{v_1,\dots,v_n\}$ *for some values* $v_1,\dots,v_n$ *where* $\varnothing \vdash v_i : T$ *for all i.*
(5) *If* $\varnothing \vdash v : \{\ell : T\}$ *then* $v = \{\ell_i \Rightarrow v_i\ ^{i \in 1..n}\}$ *for some values* $v_1,\dots,v_n$ *and field names* $\ell_1,\dots,\ell_n$ *so that* $\ell = \ell_i$ *for some i, and additionally* $\varnothing \vdash v_i : T$.

PROOF: All parts of the lemma are proved by first applying part (1)(a) of Theorem 3 to the typing derivation of v. For instance for (3) we obtain that $\models \mathbf{F}[\![\varnothing]\!] \implies \mathbf{F}[\![\mathsf{Logical}]\!](\mathsf{out\_V}(\mathbf{Return}(v)))$. This is equivalent to $\models \mathsf{In\_Logical}(v)$, which gives us that $v = \mathbf{true}$ or $v = \mathbf{false}$ by the definition of $\mathsf{In\_Logical}$ in the model. The other cases proceed in exactly the same way. □

LEMMA 44 (Progress for Type-tests).
For all types T and values v, $\exists e.\ v\ \mathbf{in}\ T \to e$.

PROOF: By induction on the structure of T using the reduction rules for type-tests. □

RESTATEMENT OF THEOREM 5 (PROGRESS)
If $\varnothing \vdash e : T$ *and e is not a value then* $\exists e'.\ e \to e'$.

PROOF: By induction on the derivation of $\varnothing \vdash e : T$ using Lemma 44 in the (Exp Test) case. The (Exp Operator), (Exp Cond), (Exp Dot), (Exp Add) and (Exp Acc) cases use Lemma 43. □

C.5 Soundness of the Algorithmic Type System

In this section we present the soundness proof for the algorithmic type system from §6.2.

The key property of type normalization is that it preserves the semantics of types. First we state the following properties of the helper functions that are used in type normalization.

LEMMA 45 (Soundness of Helper Functions).

(1) *If* $E \vdash R_1$ *and* $E \vdash R_2$ *then* $E \vdash R_1\ \&\ R_2 <: \mathit{Conj}_{RR}(R_1,R_2)$ *and* $E \vdash \mathit{Conj}_{RR}(R_1,R_2) <: R_1\ \&\ R_2$.
(2) *If* $E \vdash R_1$ *and* $E \vdash D_1$ *then* $E \vdash R_1\ \&\ D_1 <: \mathit{Conj}_{RD}(R_1,D_1)$ *and* $E \vdash \mathit{Conj}_{RD}(R_1,D_1) <: R_1\ \&\ D_1$.
(3) *If* $E \vdash D_1$ *and* $E \vdash D_2$ *then* $E \vdash D_1\ \&\ D_2 <: \mathit{Conj}_{DD}(D_1,D_2)$ *and* $E \vdash \mathit{Conj}_{DD}(D_1,D_2) <: D_1\ \&\ D_2$.

PROOF: All three cases proceed in a similar way, so for space reasons we just consider case 1. From Theorem 3, we know that $\models \forall y_1.\neg\mathbf{W}[\![R_1]\!](y_1)$ and $\models \forall y_2.\neg\mathbf{W}[\![R_2]\!](y_2)$, for $y_i \notin fv(R_i), i \in 1..2$. By definition R_i must be of the form $x_i : C_i$ **where** e_i. Thus we know by the definition of $\mathbf{W}[\![-]\!]$:

(1) $\models \forall y_1.\neg\mathbf{W}[\![C_1]\!](y_1) \wedge (\mathbf{let}\ x_1 = y_1\ \mathbf{in}\ (\mathbf{R}[\![e_1]\!] = \mathbf{Return}(\mathbf{true}) \vee \mathbf{R}[\![e_1]\!] = \mathbf{Return}(\mathbf{false})))$
(2) $\models \forall y_2.\neg\mathbf{W}[\![C_1]\!](y_2) \wedge (\mathbf{let}\ x_2 = y_2\ \mathbf{in}\ (\mathbf{R}[\![e_2]\!] = \mathbf{Return}(\mathbf{true}) \vee \mathbf{R}[\![e_2]\!] = \mathbf{Return}(\mathbf{false})))$

We calculate as follows, for all $x \notin dom(E)$:

$$\begin{array}{ll}
\models \mathbf{F}[\![Conj_{RR}(R_1,R_2)]\!](x) & \\
\quad = & \mathbf{F}[\![x_3 : C_1 \,\&\, C_2 \ \textbf{where}\ (e_1\{x_3/x_1\}\ \&\&\ e_2\{x_3/x_2\})]\!](x) \\
\quad = & \mathbf{F}[\![C_1 \,\&\, C_2]\!](x) \wedge \\
 & \quad \textbf{let}\ x_3 = x\ \textbf{in} \\
 & \qquad (\mathbf{R}[\![e_1\{x_3/x_1\}\ \&\&\ e_2\{x_3/x_2\}]\!] = \textbf{Return}(\textbf{true})) \\
\quad = & \mathbf{F}[\![C_1]\!](x) \wedge \mathbf{F}[\![C_2]\!](x) \wedge \\
 & \quad \textbf{let}\ x_3 = x\ \textbf{in} \\
 & \qquad (\mathbf{R}[\![e_1\{x_3/x_1\}\ \&\&\ e_2\{x_3/x_2\}]\!] = \textbf{Return}(\textbf{true})) \\
\quad = & \mathbf{F}[\![C_1]\!](x) \wedge \mathbf{F}[\![C_2]\!](x) \wedge \\
 & \quad \textbf{let}\ x_3 = x\ \textbf{in} \\
 & \qquad (\textbf{Bind}\ a_1 \Leftarrow \mathbf{R}[\![e_1\{x_3/x_1\}]\!]\ \textbf{in} \\
 & \qquad\ \ \textbf{Bind}\ a_2 \Leftarrow \mathbf{R}[\![e_2\{x_3/x_2\}]\!]\ \textbf{in} \\
 & \qquad\ \ \textbf{Return}(\mathsf{v\_logical}(a_1 \wedge a_2)) = \textbf{Return}(\textbf{true})) \\
\quad = & \mathbf{F}[\![C_1]\!](x) \wedge \mathbf{F}[\![C_2]\!](x) \wedge \\
 & \quad \textbf{let}\ x_3 = x\ \textbf{in} \\
 & \qquad (\textbf{Bind}\ a_1 \Leftarrow \mathbf{R}[\![e_1\{x_3/x_1\}]\!]\ \textbf{in Return}(a_1) = \textbf{Return}(\textbf{true}) \wedge \\
 & \qquad\ \ \textbf{Bind}\ a_2 \Leftarrow \mathbf{R}[\![e_2\{x_3/x_2\}]\!]\ \textbf{in Return}(a_2) = \textbf{Return}(\textbf{true})) \\
\quad = & \mathbf{F}[\![C_1]\!](x) \wedge \mathbf{F}[\![C_2]\!](x) \wedge \\
 & \quad \textbf{let}\ x_3 = x\ \textbf{in} \\
 & \qquad (\mathbf{R}[\![e_1\{x_3/x_1\}]\!] = \textbf{Return}(\textbf{true}) \wedge \\
 & \qquad\ \ \mathbf{R}[\![e_2\{x_3/x_2\}]\!] = \textbf{Return}(\textbf{true}))
\end{array}$$

$$\begin{array}{ll}
\quad = & \mathbf{F}[\![C_1]\!](x) \wedge (\textbf{let}\ x_3 = x\ \textbf{in}\ \mathbf{R}[\![e_1\{x_3/x_1\}]\!] = \textbf{Return}(\textbf{true})) \wedge \\
 & \mathbf{F}[\![C_2]\!](x) \wedge (\textbf{let}\ x_3 = x\ \textbf{in}\ \mathbf{R}[\![e_2\{x_3/x_2\}]\!] = \textbf{Return}(\textbf{true})) \\
\quad = & \mathbf{F}[\![C_1]\!](x) \wedge (\textbf{let}\ x_1 = x\ \textbf{in}\ \mathbf{R}[\![e_1]\!] = \textbf{Return}(\textbf{true})) \wedge \\
 & \mathbf{F}[\![C_2]\!](x) \wedge (\textbf{let}\ x_2 = x\ \textbf{in}\ \mathbf{R}[\![e_2]\!] = \textbf{Return}(\textbf{true})) \\
\quad = & \mathbf{F}[\![x_1 : C_1\ \textbf{where}\ e_1]\!](x) \wedge \mathbf{F}[\![x_2 : C_2\ \textbf{where}\ e_2]\!](x) \\
\quad = & \mathbf{F}[\![R_1 \,\&\, R_2]\!](x)
\end{array}$$

Hence, we obtain $E \vdash R_1 \,\&\, R_2 <: Conj_{RR}(R_1,R_2)$ and $E \vdash Conj_{RR}(R_1,R_2) <: R_1 \,\&\, R_2$. The other cases follow by similar calculations. □

Before we proceed to the soundness of type normalization we state some properties of semantic subtyping that are immediate by definition.

LEMMA 46.

(1) *If $E \vdash S$, $E \vdash T$ and $E \vdash S <: T$ then $E \vdash (x : S$ **where** $e) <: (x : T$ **where** $e)$.*
(2) $E \vdash (x : T_1 \,\&\, T_2$ **where** $e) <: (x : T_1$ **where** $e) \,\&\, (x : T_2$ **where** $e)$.
(3) $E \vdash (x_1 : (x_2 : T$ **where** $e_2)$ **where** $e_1) <: (x_1 : T$ **where** $e_1) \,\&\, (x_2 : T$ **where** $e_2)$.

LEMMA 47 (Soundness of Type Normalization).

(1) *If $E \vdash T$ and $norm(T) = D$ then D is a normal type with $E \vdash T <: D$.*
(2) *If $E \vdash (x : C$ **where** $e)$ and $norm_r(x : C$ **where** $e) = D$ then $E \vdash (x : C$ **where** $e) <: D$.*

PROOF: Proof by mutual induction on T and e. Most of the cases are routine, here we give just two.

Case T is of the form $(x : T$ where $e)$. We have that $norm(T) = |_{i=1}^{n}\ (x_i : C_i$ **where** $e_i)$ and by induction on T, $E \vdash T <: |_{i=1}^{n}\ (x_i : C_i$ **where** $e_i)$. By part (1) of Lemma 46 we have that $E \vdash (x : T$ **where** $e) <: (x : |_{i=1}^{n}\ (x_i : C_i$ **where** $e_i)$ **where** $e)$. By part (2) of Lemma 46 we have that $E \vdash (x : T$ **where** $e) <: (|_{i=1}^{n}\ (x : (x_i : C_i$ **where** $e_i)$ **where** $e))$. By part (3) of Lemma 46 we have that $E \vdash (x : T$ **where** $e) <: (|_{i=1}^{n}\ ((x_i : C_i$ **where** $e_i) \,\&\, (x : C_i$ **where** $e)))$. We also have by mutual induction $E \vdash (x : C_i$ **where** $e) <: norm_r(x : C_i$ **where** $e)$, so we can deduce that $E \vdash (x : T$ **where** $e) <: (|_{i=1}^{n}\ ((x_i : C_i$ **where** $e_i) \,\&\, (norm_r(x : C_i$ **where** $e))))$. By Lemma 45 we can conclude $E \vdash (x : T$ **where** $e) <: (|_{i=1}^{n}\ Conj_{DD}(x_i : C_i$ **where** $e_i, norm_r(x : C_i$ **where** $e)))$ as required.

Case e is of the form x in T. We have by definition that $norm_r(x : C$ **where** $(x$ **in** $T)) = norm(C \,\&\, T)$. By mutual induction we have that $E \vdash (C \,\&\, T) <: norm(C \,\&\, T)$. We assume $\mathbf{F}[\![E]\!]$ and we have that $\mathbf{F}[\![x : C$ **where** $(x$ **in** $T)]\!](t)$ is equal to the following by expanding definitions:

$$\mathbf{F}[\![C]\!](t) \wedge \textbf{let}\ x = t\ \textbf{in}\ (\mathbf{R}[\![x\ \textbf{in}\ T]\!] = \textbf{Return}(\textbf{true}))$$

By further expansion and Theorem 3 this is equivalent to:

$$\begin{array}{l}
\mathbf{F}[\![C]\!](t) \wedge \textbf{let}\ x = t\ \textbf{in} \\
\quad ((\textbf{if}\ \mathbf{F}[\![T]\!](x)\ \textbf{then Return}(\textbf{true})\ \textbf{else Return}(\textbf{false})) = \textbf{Return}(\textbf{true}))
\end{array}$$

which is clearly equivalent to $\mathbf{F}[\![C]\!](t) \wedge \mathbf{F}[\![T]\!](t)$, which by Theorem 3 and the meaning of $E \vdash (C \,\&\, T) <: norm(C \,\&\, T)$ allow us to deduce $\mathbf{F}[\![norm(C \,\&\, T)]\!](t)$ as required. □

LEMMA 48 (Soundness Of Field Type Extraction).

(1) *If $E \vdash A$ and $A.\ell \rightsquigarrow U$ then $E \vdash A <: \{\ell \colon U\}$.*
(2) *If $E \vdash C$ and $C.\ell \rightsquigarrow U$ then $E \vdash C <: \{\ell \colon U\}$.*
(3) *If $E \vdash R$ and $R.\ell \rightsquigarrow U$ then $E \vdash R <: \{\ell \colon U\}$.*
(4) *If $E \vdash D$ and $D.\ell \rightsquigarrow U$ then $E \vdash D <: \{\ell \colon U\}$.*

PROOF: All parts follow from expanding definitions, here we consider part (3). If $E \vdash R$ and $R.\ell \rightsquigarrow U$ then it must be the case that R is of the form $(x : C$ **where** $e)$, and $C.\ell \rightsquigarrow U$. By part (2) we know that $E \vdash C <: \{\ell : U\}$. We know that for any type C with $E \vdash C$ that $E \vdash (x : C$ **where** $e) <: C$, and so by transitivity (Lemma 25) we can conclude $E \vdash (x : C$ **where** $e) <: \{\ell \colon U\}$. □

LEMMA 49 (Soundness of Item Type Extraction).

(1) *If $E \vdash A$ and $A.\mathit{Items} \rightsquigarrow U$ then $E \vdash A <: U*$.*
(2) *If $E \vdash C$ and $C.\mathit{Items} \rightsquigarrow U$ then $E \vdash C <: U*$.*
(3) *If $E \vdash R$ and $R.\mathit{Items} \rightsquigarrow U$ then $E \vdash R <: U*$.*
(4) *If $E \vdash D$ and $D.\mathit{Items} \rightsquigarrow U$ then $E \vdash D <: U*$.*

PROOF: Similar to the proof of the previous lemma; we omit the details. □

LEMMA 50 (Synthesis Checkable). *If $E \vdash e \to T$ then $E \vdash e \leftarrow T$.*

PROOF: By (Swap) and reflexivity of singular subtyping. □

RESTATEMENT OF THEOREM 7 (SOUNDNESS OF ALGORITHMIC TYPE SYSTEM)

(1) *If $E \rhd \diamond$ then $E \vdash \diamond$.*
(2) *If $E \rhd T$ then $E \vdash T$.*
(3) *If $E \rhd S <: T$ and $E \vdash S$ then $E \vdash S <: T$.*
(4) *If $E \vdash e \to T$ then $E \vdash e : T$.*
(5) *If $E \vdash e \leftarrow T$ then $E \vdash e : T$.*

PROOF: By simultaneous induction over the derivations. For space reasons we give just the more interesting cases.

Part (4): (Synth Dot)

$$\frac{E \vdash e \to T \quad norm(T) = D \quad D.\ell \rightsquigarrow U}{E \vdash e.\ell \to [e.\ell : U]}$$

By induction hypothesis we have that $E \vdash e : T$. From Lemma 47 we have that $E \vdash T <: D$, and from Lemma 48 we have that $E \vdash D <: \{\ell \colon U\}$. By transitivity (Lemma 25) and the derived rule (Exp Subsum) we can conclude that $E \vdash e : \{\ell \colon U\}$.

From rule (Exp Dot) we deduce that $E \vdash e.\ell : U$, and by the derived rule (Exp Singleton) we have $E \vdash e.\ell : [e.\ell : U]$ as required.

Part (4): (Synth Add)

$$\frac{E \vdash e_1 \to T_1 \quad E \vdash e_2 \to T_2 \quad norm(T_2) = D_2 \quad D_2.\mathsf{Items} \rightsquigarrow U_2}{E \vdash e_1 :: e_2 \to ([e_1 : T_1] \mid U_2)*}$$

By induction hypothesis we have both $E \vdash e_1 : T_1$ and $E \vdash e_2 : T_2$. It is simple to show that $E \vdash [e_1 : T_1] <: ([e_1 : T_1] \mid U_2)$ and so by rule (Exp Singular Subsum) we can derive $E \vdash e_1 : ([e_1 : T_1] \mid U_2)$. From Lemma 47 we have that $E \vdash T_2 <: D_2$ and from Lemma 49 we have that $E \vdash D_2 <: U_2*$. By transitivity (Lemma 25) and the derived rule (Exp Subsum) we can conclude that $E \vdash e_2 : U_2*$. It is simple to show that $E \vdash U_2* <: ([e_1 : T_1] \mid U_2)*$ and by the derived rule (Exp Subsum) we can conclude that $E \vdash e_2 : ([e_1 : T_1] \mid U_2)*$. From rule (Exp Add) we deduce $E \vdash e_1 :: e_2 : ([e_1 : T_1] \mid U_2)*$ as required.

Part (5): (Swap)

$$\frac{E \vdash e \to T \quad E \rhd [e\colon T] <: T'}{E \vdash e \leftarrow T'}$$

By (simultaneous) induction hypothesis we have that $E \vdash e : T$ and $E \vdash [e\colon T] <: T'$. By rule (Exp Singular Subsum) we have $E \vdash e : T'$ as required.

C.6 Exploiting SMT Models Correct

In this section we show that the operational checks we use to validate the models produced by the SMT solver are correct (Lemmas 51 and 52), and that the **elementof** construct does indeed return a value of the requested type or **null** (Lemma 54).

LEMMA 51.
If the three checks from §7.1 succeed then $E \not\vdash T <: T'$.

PROOF: By inverting rule (Subtype), it suffices to show that $\not\models (\mathbf{F}[\![E]\!] \wedge \mathbf{F}[\![T]\!](x)) \implies \mathbf{F}[\![T']\!](x)$. Since our intended model is not inconsistent it suffices to show that:

$$\models \exists x, y_1, \ldots, y_n, \mathbf{F}[\![U_1]\!](y_1) \wedge \ldots \mathbf{F}[\![U_n]\!](y_n) \wedge \mathbf{F}[\![T]\!](x) \wedge \neg \mathbf{F}[\![T']\!](x).$$

From conditions (1) and (2) by Lemma 2 it follows that $\models \mathbf{R}[\![y_i\sigma \text{ in } U_i\sigma]\!] = \mathbf{true}$ for all $i \in 1..n$. After unfolding the definitions this implies that $\models \mathbf{F}[\![U_i\sigma]\!](y_i\sigma)$ for all $i \in 1..n$. In a similar way, from conditions (1) and (3) by Lemma 2 we have that $\models \mathbf{F}[\![(T\ \&\ !T')\sigma]\!](x\sigma)$, or equivalently that $\models \mathbf{F}[\![T\sigma]\!](x\sigma) \wedge \neg \mathbf{F}[\![T'\sigma]\!](x\sigma)$. Instantiating the existential variables with the values given by σ completes the proof. □

LEMMA 52. *If the three checks in §7.1 succeed for $T' = \mathsf{Empty}$ then $\varnothing \vdash x\sigma : T\sigma$ and $\varnothing \vdash y\sigma : U\sigma$ for all $(y : U) \in E$.*

PROOF: Since $x\sigma$ and $y\sigma$ for all $y \in dom(E)$ are values, by Lemma 31 and (Exp Singular Subsum) it suffices to show that $\varnothing \vdash [x\sigma] <: T\sigma$ and $\varnothing \vdash [y\sigma] <: U\sigma$ for all $y : U \in E$. By (Subtype) it suffices to show that $\models \mathbf{F}[\![T\sigma]\!](x\sigma)$ and $\models \mathbf{F}[\![U\sigma]\!](y\sigma)$ for all $y : U \in E$. These follow from the corresponding checks by Lemma 2 and basic reasoning in first-order logic. □

LEMMA 53.
*If $E \vdash T$ then the expression v **in** T is algorithmically pure.*

LEMMA 54. *If **elementof** $T \to v$ and $\varnothing \vdash T$ then $\varnothing \vdash v : T \mid [\mathbf{null}]$.*

PROOF: By Lemma 53 we have that v **in** T is pure. By the reduction relation for **elementof** T we have that, either $v = \mathbf{null}$ in which case the conclusion is immediate, or we know that v **in** $T \to^* \mathbf{true}$, which allows us to apply Lemma 52 for $E = \varnothing$ and obtain $\varnothing \vdash v : T$. □

D. Safe Systems Configurations by Typing

The language M has many potential applications, but one specific motivation is *configuration management* [5]: a repository database holds a model of a data center, that is, the configuration data for each server, and M can be used to check existing configurations and to compute new ones. Various bespoke systems [4, 23] have proven the worth of model-based systems configuration. Each of these systems has a domain-specific language (DSL) for describing intended configurations. A goal for M is to be a general-purpose modeling language able to subsume DSLs such as these. To this end, M comes with a flexible parser generator able to process the syntax of existing DSLs.

In this appendix, we present the idea that type-based modeling in M may be of practical benefit in an intended application area such as systems administration.

Many systems errors arise from misconfiguration. Administrators make mistakes, in part because configuration formats are often too flexible and allow inconsistent settings. To address this problem, numerous ad hoc tools advise on configuration safety, by finding misconfigurations in firewalls, protocol stacks, etc. Such tools package specialist expertise, are more accessible than best practice papers, and are easy to update as new issues arise.

Consider a concrete example, the WSE Policy Advisor [15, 14]. (There are many such advisors; we select WSE Policy Advisor because it is familiar to us.) WSE Policy Advisor is an XSLT style sheet that generates a report from the configuration data for Web Services Enhancements (WSE), an implementation of some standard web services security protocols. The advisor has dozens of rules advising on various potential vulnerabilities.

Advisors like this are valuable, but XSLT is not a good platform for building such tools. Instead, the M repository should be an effective platform, since it can hold the configuration data for an entire data center. Moreover, the point of this section is that the work of tools such as the policy advisor can be expressed within the rich type system of Dminor (and hence M).

D.1 Representing XML Data

First, we show how to represent configuration data within Dminor.

Here is a snippet of configuration data for WSE. This policy selects version 1.0 of a protocol known as “mutual certificate security”, switches off the establishment of a security context, and selects a particular order of encryption and digital signature.

```
<policies>
  <policy name="policy-CAM-42">
    <mutualCertificate10Security
      establishSecurityContext="false"
      messageProtectionOrder="EncryptBeforeSign">
    </mutualCertificate10Security>
  </policy>
</policies>
```

We can import this XML into Dminor as the following value:

```
{tag⇒"policies",
 body⇒{{tag⇒"policy",
        name⇒"policy-CAM-42",
        body⇒{{tag⇒"mutualCertificate10Security",
               establishSecurityContext⇒"false",
               messageProtectionOrder⇒"EncryptBeforeSign"
                   }}}}}
```

Second, we show how to express the equivalent of an XML schema within Dminor. The following type definitions capture the schema for a mutualCertificate10Security element. (The Dminor keyword **type** introduces a type definition.)

```
type bool : Text where value == "true" || value == "false";
type messageProtectionOrder :
    Text where value == "EncryptBeforeSign" ||
                value == "SignBeforeEncrypt";
type mutualCertificate10Security :
    {tag : Text where value == "mutualCertificate10Security";
     establishSecurityContext:bool;
     messageProtectionOrder:messageProtectionOrder;};
```

Hence, we can define the overall schema for a WSE configuration as follows. We omit the details of other kinds of policies. Our example value has type Config; it is schema-correct.

```
type Policy : mutualCertificate10Security | ... ;
type Config :
    {tag: Text where value == "policies";
     body: {tag : Text where value == "policy";
            body : Policy*;}*;};
```

D.2 Types for Safe Configurations

Third, we go beyond schema-correctness, and define a type for *safe* configurations, configurations that trigger no advisories.

One of the advisor's rules detects a potential "credit-taking attack" on WSE; the details need not concern us, but it turns out that there is a defect in mutual certificate security such that encrypting before signing is vulnerable to this attack. We can write a type of vulnerable policies as follows, and indeed define a union type Advisory of policies that trigger any rule. (Most of the other rules are more complex than this simple example.)

```
type q_credit_taking_attack_10 :
    mutualCertificate10Security
        where value.messageProtectionOrder == "
              EncryptBeforeSign";
type Advisory : q_credit_taking_attack_10 | ... ;
```

Now, we can define a safe policy as one that is schema-correct but that triggers no advisory, and then a safe configuration is one containing only safe policies.

```
type SafePolicy :
    Any where ((value in Policy) && !(value in Advisory));
type SafeConfig :
    {tag: Text where value == "policies";
     body: {tag : Text where value == "policy";
            body : SafePolicy*;}*; };
```

Although our example value has type Config, it does not have type SafeConfig since it is vulnerable to a credit-taking attack.

We have sketched how the type system of Dminor can express rules for detecting security vulnerabilities in real configuration data. We can check such types at run-time, and hence mimic the use of tools like the WSE Policy Advisor. More significantly, we can use such types to check code that generates new configurations, to obtain a static guarantee that no configuration produced dynamically would trigger an advisory. We conclude that static type-checking of systems models can go beyond the current generation of advisors.

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the web*. Morgan Kaufmann, 2000.

[2] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Proceedings of ICFP*, 1993.

[3] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proceedings of POPL*, 1994.

[4] P. Anderson. Towards a high-level machine configuration system. In *Proceedings of LISA*, 1994.

[5] P. Anderson. *System Configuration*, volume 14 of *Short Topics in System Administration*. USENIX Association/SAGE, 2006.

[6] D. Aspinall. Subtyping with singleton types. In *Proceedings of CSL*, 1994.

[7] D. Aspinall and M. Hofmann. Dependent types. In *Advanced Topics in Types and Programming Languages*, chapter 2. MIT Press, 2005.

[8] M. Backes, C. Hriţcu, and T. Tarrach. Automatically verifying typing constraints for a data processing language. In *Proceedings of CPP*, 2011.

[9] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of FMCO*, 2005.

[10] C. Barrett, M. Deters, A. Oliveras, and A. Stump. Design and results of the 3rd Annual SMT Competition (SMT-COMP 2007). *International Journal on Artificial Intelligence Tools*, 17(4):569–606, 2008.

[11] C. Barrett and C. Tinelli. CVC3. In *Proceedings of CAV*, 2007.

[12] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *Proceedings of CSF*, 2008.

[13] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-friendly general purpose language. In *Proceedings of ICFP*, 2003.

[14] K. Bhargavan, C. Fournet, and A. D. Gordon. Policy advisor for WSE 3.0. In *Web Service Security*, pages 324–330. Microsoft Press, 2006.

[15] K. Bhargavan, C. Fournet, A. D. Gordon, and G. O'Shea. An advisor for web services security policies. In *Proceedings of Workshop on Secure Web Services*, 2005.

[16] G. M. Bierman, A. D. Gordon, C. Hriţcu, and D. Langworthy. Semantic subtyping with an SMT solver. In *Proceedings of ICFP*, 2010.

[17] G. M. Bierman, E. Meijer, and M. Torgersen. Lost in translation: Formalizing proposed extensions to C$^\sharp$. In *Proceedings of OOPSLA*, 2007.

[18] S. Böhme, K. R. M. Leino, and B. Wolff. HOL-Boogie – an interactive prover for the Boogie program-verifier. In *Proceedings of TPHOLs*, 2008.

[19] D. Box. Update on SQL Server Modeling CTP (Repository/Modeling Services, "Quadrant" and "M"). Blog post on http://blogs.msdn.com/b/modelcitizen, Sept. 2010.

[20] R. S. Boyer, M. Kaufmann, and J. S. Moore. The Boyer-Moore theorem prover and its interactive enhancement. *Computers & Mathematics with Applications*, 29(2):27–62, 1995.

[21] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, 1995.

[22] P. Buneman and B. C. Pierce. Union types for semistructured data. In *Proceedings of DBPL*, 1999.

[23] M. Burgess and Æ. Frisch. *A System Engineer's Guide to Host Configuration and Maintenance using Cfengine*, volume 16 of *Short Topics in System Administration*. USENIX Association/SAGE, 2007.

[24] R. M. Burstall, D. B. MacQueen, and D. Sannella. HOPE: An experimental applicative language. In *Proceedings of LISP Conference*, 1980.

[25] C. Calcagno, L. Cardelli, and A. D. Gordon. Deciding validity in a spatial logic for trees. *Journal of Functional Programming*, 15:543–572, 2005.

[26] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of PLDI*, 1991.

[27] G. Castagna. Patterns and types for querying XML documents. In *Proceedings of DBPL*, 2005.

[28] G. Castagna and G. Chen. Dependent types with subtyping and late-bound overloading. *Information and Computation*, 168(1):1–67, 2001.

[29] S. Cohen. User-defined aggregate functions: bridging theory and practice. In *Proceedings of SIGMOD*, 2006.

[30] D. Crockford. The application/json media type for JavaScript Object

Notation (JSON), July 2006. RFC 4627.

[31] F. Damm. Subtyping with union types, intersection types and recursive types. In *Proceedings of TACS*, 1994.

[32] L. M. de Moura and N. Bjørner. Efficient E-matching for SMT solvers. In *Proceedings of CADE-21*, pages 183–198, 2007.

[33] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of TACAS*, 2008.

[34] L. M. de Moura and N. Bjørner. Generalized, efficient array decision procedures. In *Proceedings of FMCAD*, 2009.

[35] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of ACM*, 52(3):365–473, 2005.

[36] J. Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, Aug. 2007. CMU-CS-07-129.

[37] J. Dunfield and F. Pfenning. Tridirectional typechecking. In *Proceedings of POPL*, 2004.

[38] B. Dutertre and L. M. de Moura. The YICES SMT solver. Available at `http://yices.csl.sri.com/tool-paper.pdf`, 2006.

[39] R. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of ICFP*, 2002.

[40] K. Fisher, Y. Mandelbaum, and D. Walker. The next 700 data description languages. In *Proceedings of POPL*, 2006.

[41] C. Flanagan. Hybrid type checking. In *Proceedings of POPL*, 2006.

[42] T. Freeman and F. Pfenning. Refinement types for ML. In *Proceedings of PLDI*, 1991.

[43] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *Journal of ACM*, 55(4), 2008.

[44] P. Genevès, N. Layaïda, and A. Schmitt. Efficient static analysis of XML paths and types. In *Proceedings of PLDI*, 2007.

[45] J. Giesl. Termination of nested and mutually recursive algorithms. *Journal of Automated Reasoning*, 19:1–29, 1997.

[46] A. D. Gordon and A. Jeffrey. Typing one-to-one and one-to-many correspondences in security protocols. In *Proceedings of ISSS*, 2002.

[47] M. Greenberg, B. Pierce, and S. Weirich. Contracts made manifest. In *Proceedings of POPL*, 2010.

[48] D. A. Greve, M. Kaufmann, P. Manolios, J. S. Moore, S. Ray, J.-L. Ruiz-Reina, R. Sumners, D. Vroon, and M. Wilding. Efficient execution in an automated reasoning environment. *Journal of Functional Programming*, 18(1):15–46, 2008.

[49] H. Hosoya and B. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[50] H. Hosoya, J. Vouillon, and B. Pierce. Regular expression types for XML. In *Proceedings of ICFP*, 2000.

[51] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic and Algebraic Programming*, 19/20:503–581, 1994.

[52] R. Jhala, R. Majumdar, and A. Rybalchenko. HMC: Verifying functional programs using abstract interpreters. In *Proceedings of CAV*, pages 470–485, 2011.

[53] R. Jhala, R. Majumdar, and R.-G. Xu. State of the union: Type inference via Craig interpolation. In *Proceedings of TACAS*, 2007.

[54] C. Jones. *Systematic software development using VDM*. Prentice-Hall, 1986.

[55] M. Kawaguchi, P. M. Rondon, and R. Jhala. Type-based data structure verification. In *Proceedings of PLDI*, pages 304–315, 2009.

[56] J. C. King. Symbolic execution and program testing. *Communications of The ACM*, 19:385–394, 1976.

[57] K. Knowles, A. Tomb, J. Gronski, S. Freund, and C. Flanagan. SAGE: Unified hybrid checking for first-class types, general refinement types and `Dynamic`. Technical report, UCSC, 2007.

[58] K. W. Knowles and C. Flanagan. Hybrid type checking. *ACM TOPLAS*, 32(2), 2010.

[59] R. Komondoor, G. Ramalingam, S. Chandra, and J. Field. Dependent types for program understanding. In *Proceedings of TACAS*, 2005.

[60] A. Kopylov. Dependent intersection: A new way of defining records in type theory. In *Proceedings of LICS*, 2003.

[61] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *Proceedings of PLDI*, pages 316–329, 2010.

[62] K. R. M. Leino and R. Monahan. Reasoning about comprehensions with first-order SMT solvers. In *Proceedings of SAC*, 2009.

[63] B. Lerner, M. Flower, D. Grossman, and C. Chambers. Searching for type-error messages. In *Proceedings of PLDI*, 2007.

[64] W. Lovas and F. Pfenning. A bidirectional refinement type system for LF. In *Proceedings of LFMTP*, 2007.

[65] J. McCarthy. Towards a mathematical science of computation. In *Proceedings of IFIP Congress*, 1962.

[66] E. Meijer, B. Beckman, and G. M. Bierman. LINQ: Reconciling objects, relations and XML in the .NET framework. In *Proceedings of SIGMOD*, 2007.

[67] J. Meng and L. C. Paulson. Translating higher-order problems to first-order clauses. *Journal of Automated Reasoning*, 40(1):35–60, 2008.

[68] B. Meyer. *Eiffel: the language*. Prentice Hall, 1992.

[69] Microsoft Corporation. *The Microsoft code name "M" Modeling Language Specification Version 0.5*, Oct. 2009. Preliminary implementation available as part of the *SQL Server Modeling CTP (November 2009)*.

[70] B. Nordström and K. Petersson. Types and specifications. In *Proceedings of IFIP*, 1983.

[71] B. C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, 1991.

[72] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[73] B. C. Pierce and D. N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, 2000.

[74] V. Pratt. Five paradigm shifts in programming language design and their realization in Viron, a dataflow programming environment. In *Proceedings of POPL*, 1983.

[75] S. Ranise and C. Tinelli. *The SMT-LIB Standard: Version 1.2*, 2006.

[76] J. C. Reynolds. Design of the programming language Forsythe. In *Algol-Like Languages*, chapter 8. Birkhäser, 1996.

[77] P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *Proceedings of PLDI*, 2008.

[78] J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998.

[79] V. Saraswat, N. Nystrom, J. Palsberg, and C. Grothoff. Constrained types for object-oriented languages. In *Proceedings of OOPSLA*, 2008.

[80] J. Siméon and P. Wadler. The essence of XML. In *Proceedings of POPL*, 2003.

[81] M. Sozeau. Subset coercions in Coq. In *Proceedings of TYPES*, 2006.

[82] N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in Fine. In *Proceedings of ESOP*, 2010.

[83] T. Terauchi. Dependent types from counterexamples. In *Proceedings of POPL*, 2010.

[84] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *Proceedings of POPL*, 2008.

[85] S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped

languages. In *Proceedings of ICFP*, 2010.

[86] TypiCal Project. *The Coq proof assistant*, 2009. Version 8.2. Available at `http://coq.inria.fr`.

[87] H. Unno and N. Kobayashi. Dependent type inference with interpolants. In *Proceedings of PPDP*, 2009.

[88] A. K. Wright and R. Cartwright. A practical soft type system for scheme. *ACM TOPLAS*, 19:87–152, 1997.

[89] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[90] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of POPL*, 1999.