Semantic Subtyping with an SMT Solver

Gavin M. Bierman
Microsoft Research
gmb@microsoft.com

Andrew D. Gordon
Microsoft Research
adg@microsoft.com

Cătălin Hriţcu
Saarland University
hritcu@cs.uni-saarland.de

David Langworthy
Microsoft Corporation
dlan@microsoft.com

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 small core calculus can express a rich variety of typing idioms; for example, intersection, union, negation, singleton, nullable, variant, recursive, 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, interpreting types as formulas allows us to call the SMT solver at run-time to compute instances of types.

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 [2], 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 release of M[1] 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, a core calculus 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] http://msdn.microsoft.com/en-us/data/default.aspx

1.1 Programming with Type-Test and Refinement

The core types of Dminor are structural types for scalars, (unordered) collections, and records. (We base the syntax and terminology of Dminor on the database-oriented language M, and hence refer to records as *entities*.) We write $S <: T$ for the subtype relation, meaning that every value of type S is also of type T.

Two central primitives of Dminor are the following:

- A *refinement type*, $(x : T \textbf{ where } e)$, consists of the values x of T satisfying the Boolean expression e.
- A *type-test expression*, $e \textbf{ in } T$, returns **true** or **false** depending on whether or not the value of e belongs to the 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 $(x : \mathsf{Any} \textbf{ where } x == v)$, 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 $(x : \mathsf{Any} \textbf{ where } (x \textbf{ in } T) \,||\, (x \textbf{ in } U))$.

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, 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, S-expressions, and JavaScript Object Notation (JSON) [18], 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 kind fields of distinct singleton types.

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

```
((y.kind == "assignment") ? y.var : "NotAssign") : Text
```

We write $e : T$ to assert that the value of e must have type T. One of the purposes of type-checking is to ensure the validity of such assertions statically. Intuitively, the code above 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 $y.\mathsf{var}$, we have $y : \mathsf{Statement}$, know that $y.\mathsf{kind} == \texttt{"assignment"}$, and need to decide $[y] <: \{\mathsf{var} : \mathsf{Text};\}$, 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, 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 [8], Yices [25], and Z3 [20]. There are standard input formats such as Simplify's [22] unsorted S-expression syntax and the SMT-LIB standard [45] 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,[2] 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 is based on deciding subtyping by invoking an external SMT solver. To decide 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* [3, 30], as opposed to the more common alternative, *syntactic subtyping*, which is to define syntax-driven rules for checking subtyping [41].

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 [28, 34], F7 [9], and Dsolve [47], rely on various provers [20, 25] for problems encoded in first-order logic with equality and linear arithmetic. 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 knowledge of first-order logic and the theory of equality; for example, we represent union and intersection types as logical disjunctions and conjunctions, which are efficiently manipulated by the solver. Hence, we avoid the implementation effort of explicit propagation of known equality constraints, and of syntax-driven rules for union and intersection types [43, 24, 23].

1.4 Background: DSLs and Systems Modeling

The language M has many potential applications, but one specific motivation is *configuration management*: the database repository 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, 15] have proven the worth of model-based systems configuration. Each of these systems has a domain-specific language (DSL) for describing intended configurations. (The mechanisms of configuration management are a motivation for M, but are not the topic of this paper; for a comprehensive, comparative discussion see [5].) 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. Moreover (and this is the focus of the paper), M is a small functional language, with a rich set of types for describing data models in general, and systems configurations in particular. For example, the current and intended state of the machines and software in a data center would be described by M expressions. Intended properties of states can be described by M types.

[2] Most important is the SMT-COMP[7] competition held each year in conjunction with CAV and in which more than a dozen SMT solvers contend.

1.5 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.

(4) Devising a systematic way to use the models produced by the SMT solver as evidence of satisfiability in order to provide precise counterexamples to typing, detect empty types and generate 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.

1.6 Structure of the Paper

§2 describes the formal syntax of Dminor together with a small-step operational semantics, $e \to e'$, 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) and Dminor types; each pure expression e is interpreted as a term $\mathbf{T}[\![e]\!]$ and each type T is interpreted as a first-order logic formula $\mathbf{F}[\![T]\!](t)$. 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 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 2 concerns logical soundness of type assignment; if e is assigned type T then formula $\mathbf{F}[\![T]\!](\mathbf{T}[\![e]\!])$ holds. Progress and preservation results (Theorems 3 and 4) relate type assignment to the operational semantics, entailing that well-typed expressions cannot go wrong.

§5 develops additional theory to justify our implementation techniques. First, we present simpler variations of the translations $\mathbf{T}[\![e]\!]$ 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 5 shows soundness of this optimization. Second, since the declarative rules of §4 are not directly algorithmic, we propose type checking and synthesis algorithms, presented as bidirectional rules. Theorem 6 shows these are sound with respect to type assignment. Finally, we show how to check pu-

rity of expressions using a syntactic termination restriction together with a confluence check that relies on the logical semantics. Theorem 7 shows that our algorithmic purity check is indeed a sufficient condition for purity.

§6 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. §7 reports some details of our implementation. Returning to the original motivation for M, in §8 we describe how typing Dminor models may be used to check for systems configuration errors. We survey related work in §9, before concluding in §10.

The appendixes relate the operational and logical semantics using an intermediate big-step semantics (Appendix A); report our intended logical model of Dminor and its formalization in Coq (Appendix B), as well as the axiomatization of this model that is passed to the external solver during type-checking (Appendix C); and prove the correctness of the algorithmic purity check (Appendix D). Our implementation as well as a screencast comparing the effectiveness of Dminor with the standard M type-checker, is available at `http://research.microsoft.com/~adg/dminor.html`.

2. Syntax and Operational Semantics

Dminor is a 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; and (3) 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**. 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 xs : NullableInt*, x : NullableInt, and the check that x != **null**, our type-checking algorithm infers that 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 | Text | 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. Although 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
	$e : T$	type-assertion
	$\{\ell_i \Rightarrow e_i \ ^{i \in 1..n}\}$	entity (ℓ_i distinct, in fixed order)
	$e.\ell$	field selection
	$\{v_1, \ldots, v_n\}$	collection (multiset; unordered)
	$e_1 :: e_2$	adding element e_1 to collection e_2
	from x **in** e_1 **let** $y = e_2$ **accumulate** e_3	iteration over collection
	$f(e_1, \ldots, e_n)$	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. A *type-assertion*, $e : T$, requires that the result of the expression e be a value of type T. If this is not the case execution "goes wrong". In this paper, type-assertions are verified during type-checking, and hence ignored at

run-time. A future approach may be a hybrid system [28] where the assertions that can not be verified at compile-time are left to be checked at run-time.

The accumulate expression can encode all the usual operations on collections: counting the number of elements or of 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 [14].

Derived Collection Expressions:

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

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

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

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

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

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

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

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

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

In example code, we often rely on the following derived syntax for from-where-select expressions in the style of LINQ [37].

Derived LINQ Queries:

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

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 $f(e_1,\ldots,e_n)$ in a source program that there is a corresponding function definition for f with arity n.

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

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

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.

Reduction Contexts:

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

Reduction Rules for Standard Constructs:

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

Reduction Rules for Type-Test and Type-Assertion:

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

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

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

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

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

$v : T \to (v\ \mathbf{in}\ T)?v : \mathsf{stuck} \qquad$ where $\mathsf{stuck} \triangleq \{\}.\ell$

Each primitive operator is a partial function represented by a set of equations $\oplus(v_1,\ldots,v_n) \mapsto v_0$ where each v_i is a value. The equations for $==$ take the form $(v == v') \mapsto \mathbf{true}$ for all v and v' so that $v = v'$, and $(v == v') \mapsto \mathbf{false}$ for all v, v' with $v \neq v'$. 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.

The reduction rules for type-test expressions, $e\ \mathbf{in}\ T$, first reduce e to a value v and then proceed by case analysis on the structure of the type T. In case T is a refinement type $(x : T'\ \mathbf{where}\ e')$ then v is a value of T if and only if v is a value of type T' and $e'\{v/x\}$ reduces to the value $\mathbf{true}$. Nondeterminism arises from the second reduction rule for accumulate expressions, which binds x to any member of the collection $\{v_1,\ldots,v_n\}$. For example, consider the expression $\mathsf{pick}\ e_1\ e_2 \triangleq \mathbf{from}\ x\ \mathbf{in}\ \{e_1,e_2\}\ \mathbf{let}\ y = \mathbf{null}\ \mathbf{accumulate}\ x$; we have both $\mathsf{pick}\ \mathbf{true}\ \mathbf{false} \to^* \mathbf{true}$ and $\mathsf{pick}\ \mathbf{true}\ \mathbf{false} \to^* \mathbf{false}$.

Next, we use reduction to define an evaluation relation, which relates an expression to its return values, or to $\mathbf{Error}$, in case reduction gets stuck before reaching a value.

Stuckness, Results, and Evaluation: $e \Downarrow v$

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

For example, we have that $\mathsf{stuck} \Downarrow \mathbf{Error}$, where $\mathsf{stuck} = \{\}.\ell$ is for some label ℓ. Let closed expression e *go wrong* if and only if $e \Downarrow \mathbf{Error}$. For instance, a failed type-assertion $e : T$ goes wrong if e is not an element of T. In the presence of type-test and refinement types, expressions can go wrong in unusual ways. For example, given the refinement type $T = (x : \mathsf{Any}\ \mathbf{where}\ \mathsf{stuck})$, any type-test $v\ \mathbf{in}\ T$ goes wrong. The main goal of our type system is to ensure that no closed well-typed expression goes wrong.

Calling a function with arguments that do not have their declared types does *not* necessarily go wrong. One can, however, explicitly enforce that the declared types are respected by rewriting any function definition $f(x_1 : T_1,\ldots,x_n : T_n) : U\{e\}$ into $f(x_1 : T_1,\ldots,x_n : T_n) : U\{(x_1 : T_1\&\&\ldots\&\&x_n : T_n)?e : \mathsf{stuck}\}$. Our type system enforces that declared types are respected.

2.3 Pure Expressions and Refinement Types

A typical problem in languages with refinement types $(x : T \textbf{ where } e)$ is that the refinement expression e, even though well-typed, may make no sense as a boolean condition. This could be because e diverges or is nondeterministic. In Dminor calls to recursive functions can cause divergence, and since collections are unordered, iterating over them with accumulate is 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 §4 requires that for $(x : T \textbf{ where } e)$ 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 §5.3.

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 $e \Downarrow r$, and (3) for every subexpression $f(e_1, \ldots, e_n)$ of e, the function f is labeled-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 assume that the body of every labeled-pure function is a pure expression.

LEMMA 1 (Reduction Preserves Purity). *If e is pure and $e \to e'$ then e' is pure.*

2.4 Derived Types

We end this section by exploring the expressiveness of the primitive types introduced above. We show that the range of derivable types is rather wide. We begin with some basic examples.

Encoding of Empty, Singleton, and Ok Types:

$$\mathsf{Empty} \triangleq (x : \mathsf{Any} \textbf{ where } \textbf{false})$$
$$[e : T] \triangleq \begin{cases} (x : T \textbf{ where } x == e) & (x \notin fv(e)) \quad \text{if } e \text{ pure} \\ T & \text{otherwise} \end{cases}$$
$$[e] \triangleq [e : \mathsf{Any}]$$
$$\mathsf{Ok}(e) \triangleq \begin{cases} (x : \mathsf{Any} \textbf{ where } e) & (x \notin fv(e)) \quad \text{if } e \text{ pure} \\ \mathsf{Any} & \text{otherwise} \end{cases}$$

The type Empty has no elements; it is a subtype of all other types. The *singleton type*, $[e]$, contains only the value of e. For example, the type [**null**] consists just of the **null** value. The actual value of an type $\mathsf{Ok}(e)$ is arbitrary, the point is simply to record that condition e holds [31], provided it is pure. When e is not pure, $\mathsf{Ok}(e)$ is equivalent to Any. (We make use of Ok-types in the rule (Exp Cond) in §4.)

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:

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

A value of the *union type*, $T \mid U$, is a value of either T or 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:

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

As in Forsythe [46], *multiple-field entity types* are examples of intersection types. 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). We derive the general form of entity types as an intersection type. One advantage of this approach is that it immediately entails subtyping in width 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 do not validate subtyping in width. For this we constrain the multiple-field entity types above to additionally satisfy the eta law for records.

Encoding of Closed Entity Types:

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

Given refinement types, closed entity types and type-test, we can encode *dependent pair types* $\Sigma x : T.\, U$ where x is bound in U.

Encoding of Dependent Pair Types:

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

A *conditional type*, **if** e **then** T **else** U, is a type equal to T if the boolean e holds, and otherwise equal to U. We can use conditional and dependent pair types to encode *sum types* (tagged unions). (Throughout, wewrite _ for a fresh variable that occurs nowhere else.)

Encoding of Conditional and Sum Types:

$$\textbf{if } e \textbf{ then } T \textbf{ else } U \triangleq \begin{cases} (\_ : T \textbf{ where } e) \mid (\_ : U \textbf{ where } !e) & \text{if } e \text{ pure} \\ T \mid U & \text{otherwise} \end{cases}$$
$$T + U \triangleq \Sigma x : \mathsf{Logical}.\, (\textbf{if } x \textbf{ then } T \textbf{ else } U))$$

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

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

We only use this encoding when the body of $f_{\mu X.T}$ is pure. We believe that the contractiveness of $\mu X.T$ is a sufficient condition for this, but have not worked out the full details. Instead we can check the purity of the body of $f_{\mu X.T}$ directly, using the algorithmic purity check from §5.3. In a first step we can transform the function body into an equivalent form that makes the termination argument more simple. Consider the case of the list type $\mathsf{List}_T \triangleq \mu X.((\Sigma\_ : T.X) + [\textbf{null}])$. The recursive function we generate for testing this type is

$f_{\mathsf{List}_T}(x:\mathsf{Any}):\mathsf{Logical}\ \{x\ \mathbf{in}\ ((\Sigma\_:T.(x:\mathsf{Any}\ \mathbf{where}\ f_{\mathsf{List}_T}(x)))+[\mathbf{null}])\}$, for which it is not easy to tell whether the recursive call is to a smaller argument or not. If we rewrite this to the equivalent function f'_{List_T} below, then showing termination becomes routine ($x.\mathsf{snd}.\mathsf{snd}$ is structurally smaller than x).

Encoding Lists (transformed)

$$
\begin{array}{l}
f'_{\mathsf{List}_T}(x:\mathsf{Any}):\mathsf{Logical}\ \{\\
\quad x\ \mathbf{in}\ \{\mathsf{fst}:\mathsf{Any}\}\ \&\&\ x.\mathsf{fst}\ \mathbf{in}\ \mathsf{Logical}\ \&\&\ x\ \mathbf{in}\ \{\mathsf{snd}:\mathsf{Any}\}\\
\quad\ \&\&\ x == \{\mathsf{fst}\Rightarrow x.\mathsf{fst},\mathsf{snd}\Rightarrow x.\mathsf{snd}\}\\
\quad\ \&\&\ ((x.\mathsf{snd}\ \mathbf{in}\ \{\mathsf{fst}:\mathsf{Any}\}\ \&\&\ x.\mathsf{snd}.\mathsf{fst}\ \mathbf{in}\ T\\
\quad\ \&\&\ x.\mathsf{snd}\ \mathbf{in}\ \{\mathsf{snd}:\mathsf{Any}\}\ \&\&\ f_{\mathsf{List}_T}(x.\mathsf{snd}.\mathsf{snd})\\
\quad\ \&\&\ x.\mathsf{snd} == \{\mathsf{fst}\Rightarrow x.\mathsf{snd}.\mathsf{fst},\mathsf{snd}\Rightarrow x.\mathsf{snd}.\mathsf{snd}\}\ \&\&\ z == \mathbf{true})\\
\qquad ||\ (x.\mathsf{snd} == \mathbf{null}\ \&\&\ z == \mathbf{false}))\\
\}
\end{array}
$$

Lists can be used to encode XML and JSON. Hence, Dminor can be also be seen as a richly typed functional notation for manipulating data in XML format. In fact, 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 last 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 *self type*, $\mathbf{Self}(s:T)U$, is essentially the intersection of T and U, except that the variable s is bound to the underlying value, with scope U. The type T cannot mention s, and we need to rely on $s:T$ when checking well-formedness of U.

Encoding of Self Types:

$$\mathbf{Self}(s:T)U \triangleq (s:T\ \mathbf{where}\ s\ \mathbf{in}\ U)$$

With these constructions, we can define the following type T_s to be the type of entities where the Y field is either Text or Logical, depending on whether the X field is **true** or **false**. Here is a collection of sample data of type T_s*.

$T_s \triangleq$ **Self**(s: {X:Logical;}){Y:**if** s.X **then** Text **else** Integer;}
{ {X⇒**true**, Y⇒"Hello"}, {X⇒**false**, Y⇒42} } : T_s*

We say that a type like T_s is *self-aware* to mean that the type of one field depends on the value of another. We define below a generalized form $\mathbf{Ent}\{\ell_i:T_i;\ ^{i\in 1..n}\}$ of self-aware entity types by dividing the fields into those whose type depends on the self variable (the fields indexed by J), and those that do not.

Encoding of Self-Aware Entity Types:

$$
\begin{array}{l}
\mathbf{Ent}\{\ell_i:T_i;\ ^{i\in 1..n}\} \triangleq\\
\quad \mathbf{Self}(\mathsf{self}:\{\ell_i:T_i;\ ^{i\in(1..n)-J}\})\{\ell_j:T_j;\ ^{j\in J}\}\\
\quad \text{where } J=\{j\in 1..n \mid \mathsf{self}\in fv(T_j)\}
\end{array}
$$

In this notation, our example type T_s above can be written as:
Ent {X:Logical; Y:**if** self.X **then** Text **else** Integer}

Entity types in M [2] are self-aware by default, and as we have shown, we can reduce this feature to the primitives of Dminor.

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 $\{v_1,\ldots,v_n\}:T*$ 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:

$$
\begin{array}{l}
\mathsf{all}(x:T)e \triangleq (x:T\ \mathbf{where}\ e)*\\
\mathsf{exists}(x:T)e \triangleq T* \ \&\ !(\mathsf{all}(x:T)!e)
\end{array}
$$

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:

$$\mathsf{Mem}(e_i,e_c) \triangleq (e_c\ \mathbf{in}\ \mathsf{exists}(x:\mathsf{Any})(x==e_i))$$

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 example 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.

Ent {Songs: Text*; Default: Text **where** Mem(value,self.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 many-sorted first-order logic (FOL) formulas that 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 implication. We use Coq to state our model and to derive 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 with inductive types Value and Result given as grammars in §2 (for brevity we omit the corresponding Coq definitions; they are given in Appendix B). 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 | Error ⇒ false | Return v ⇒ true end.
Program Definition out_V (res : Result) : Value :=
  match res with | Error ⇒ G G_Null | Return v ⇒ v end.
```

Our semantics uses many-sorted first-order logic (each sort is interpreted by a Coq type of the same name). We write predicates as functions to sort bool, with truth values **true** and **false**. 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 true in our intended model.

Our semantics consists of three translations:

- For any pure expression e, we have the FOL term $\mathbf{T}[\![e]\!]:\mathsf{Result}$.
- For any Dminor type T and FOL term $t:\mathsf{Value}$, we have the 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 any Dminor type T and FOL term $t:\mathsf{Value}$, we have the formula $\mathbf{W}[\![T]\!](t)$, which holds if and only if a type-test that the value denoted by t is a member of the type T goes wrong.

These (mutually recursive) translations are defined below. We rely on notations for let-binding within terms (**let** $x=t$ **in** t'), and terms conditional on formulas (**if** F **then** t **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 $\models (\textbf{Bind}\ x \Leftarrow \textbf{Return}(v)\ \textbf{in}\ t) = t\{v/x\}$ and $\models (\textbf{Bind}\ x \Leftarrow \textbf{Error}\ \textbf{in}\ t) = \textbf{Error}$.

Notation: Monadic Bind for Propagating Errors:

$$\textbf{Bind}\ x \Leftarrow t_1\ \textbf{in}\ t_2 \triangleq (\textbf{if}\ \neg\mathsf{Proper}(t_1)\ \textbf{then Error else let}\ x = \mathsf{out\_V}(t_1)\ \textbf{in}\ t_2)$$

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

Semantics: Core Types and Expressions:

$$\mathbf{F}[\![\mathsf{Any}]\!](t) = \textbf{true}$$
$$\mathbf{W}[\![\mathsf{Any}]\!](t) = \textbf{false}$$

$$\mathbf{F}[\![(x:T\ \textbf{where}\ e)]\!](t) = \mathbf{F}[\![T]\!](t) \wedge \textbf{let}\ x = t\ \textbf{in}\ (\mathbf{T}[\![e]\!] = \textbf{Return}(\textbf{true}))$$
$$\mathbf{W}[\![(x:T\ \textbf{where}\ e)]\!](t) = \mathbf{W}[\![T]\!](t) \vee \textbf{let}\ x = t\ \textbf{in}\ (\neg(\mathbf{T}[\![e]\!] = \textbf{Return}(\textbf{false}) \vee \mathbf{T}[\![e]\!] = \textbf{Return}(\textbf{true})))$$

$$\mathbf{T}[\![x]\!] = \textbf{Return}(x)$$
$$\mathbf{T}[\![e_1\,?e_2 : e_3]\!] = \textbf{Bind}\ x \Leftarrow \mathbf{T}[\![e_1]\!]\ \textbf{in}\ (\textbf{if}\ x = \textbf{true}\ \textbf{then}\ \mathbf{T}[\![e_2]\!]\ \textbf{else}\ (\textbf{if}\ x = \textbf{false}\ \textbf{then}\ \mathbf{T}[\![e_3]\!]\ \textbf{else Error}))$$
$$\mathbf{T}[\![\textbf{let}\ x = e_1\ \textbf{in}\ e_2]\!] = \textbf{Bind}\ x \Leftarrow \mathbf{T}[\![e_1]\!]\ \textbf{in}\ \mathbf{T}[\![e_2]\!]$$
$$\mathbf{T}[\![e\ \textbf{in}\ T]\!] = \textbf{Bind}\ x \Leftarrow \mathbf{T}[\![e]\!]\ \textbf{in}\ (\textbf{if}\ \mathbf{W}[\![T]\!](x)\ \textbf{then Error else Return}((\textbf{if}\ \mathbf{F}[\![T]\!](x)\ \textbf{then true else false})))$$
$$\mathbf{T}[\![e:T]\!] = \textbf{Bind}\ x \Leftarrow \mathbf{T}[\![e\ \textbf{in}\ T]\!]\ \textbf{in}\ (\textbf{if}\ x = \textbf{true}\ \textbf{then}\ \mathbf{T}[\![e]\!]\ \textbf{else Error})$$

Next, we specify the semantics of scalar types and values.

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:

$$\mathbf{F}[\![\mathsf{Integer}]\!](t) = \mathsf{In\_Integer}(t) \qquad \mathbf{T}[\![c]\!] = \textbf{Return}(c)$$
$$\mathbf{F}[\![\mathsf{Text}]\!](t) = \mathsf{In\_Text}(t) \qquad \mathbf{W}[\![G]\!](t) = \textbf{false}$$
$$\mathbf{F}[\![\mathsf{Logical}]\!](t) = \mathsf{In\_Logical}(t)$$

$$\mathbf{T}[\![\oplus(e_1,\ldots,e_n)]\!] = \textbf{Bind}\ x_1 \Leftarrow \mathbf{T}[\![e_1]\!]\ \textbf{in}\ \ldots \textbf{Bind}\ x_n \Leftarrow \mathbf{T}[\![e_n]\!]\ \textbf{in}\ (\textbf{if}\ \mathbf{F}[\![T_1]\!](x_1) \wedge \cdots \wedge \mathbf{F}[\![T_n]\!](x_n)\ \textbf{then Return}(\mathsf{O}_\oplus(x_1,\ldots,x_n))\ \textbf{else Error})$$

where $\oplus : T_1,\ldots,T_n \to T$

The semantics of primitive operators on simple values is defined uniformly. We state below the signature $\oplus : T_1,\ldots,T_n \to T$ 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 $\mathbf{T}[\![\oplus(e_1,\ldots,e_n)]\!]$ of operator expressions. Each of the functions $\mathsf{O}_\oplus$ is defined in Appendix B.

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

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

LEMMA 2. *Suppose* $\oplus : T_1,\ldots,T_n \to T$.

(1) *If* $\models \mathbf{F}[\![T_i]\!](v_i)$ *for each* $i \in 1..n$ *then there is* v *such that* $\oplus(v_1,\ldots,v_n) \mapsto v$.
(2) *If* $\oplus(v_1,\ldots,v_n) \mapsto v$ *then* $\models \mathbf{F}[\![T_i]\!](v_i)$ *for each* $i \in 1..n$, *and* $\models \mathbf{F}[\![T]\!](v)$ *and* $\models \mathsf{O}_\oplus(v_1,\ldots,v_n) = v$.

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

LEMMA 3. *If* $\oplus : G_1,\ldots,G_n \to G$ *then* $dom(\oplus) = K(G_1) \times \cdots \times K(G_n)$.

The value denoted by term t has an entity type $\{\ell : T\}$ if it is an entity that has field ℓ ($\mathsf{v\_has\_field}(\ell,t)$), and this field contains a value of type T ($\mathbf{F}[\![T]\!](\mathsf{v\_dot}(t,\ell))$).

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 (* arbitrary *) end.
```

Semantics: Entity Types and Expressions:

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

The semantics of $\textbf{from}\ x\ \textbf{in}\ e_1\ \textbf{let}\ y = e_2\ \textbf{accumulate}\ e_3$ 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 **Error**. The model of the sort ClosureRes2 is the set of functions from Value to Value to Result. We write the lambda-abstraction $\textbf{fun}\ x\ y \to \mathbf{T}[\![e_3]\!]$ for such a function. There are several standard techniques for representing lambda-abstractions in first-order logic [38]. Our implementation generates a fresh function symbol to represent each lambda-abstraction occurring in its input as a closure of sort ClosureRes2. Since the accumulate expression is pure it produces the same result no matter what order is used when folding.

Model: Functions and Predicates on Collections:

```
Program Definition v_mem (v cv : Value) : bool :=
  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 collections ($\mathsf{is\_C}(t)$) containing only elements of type T ($\forall x.\mathsf{v\_mem}(x,t) \Rightarrow \mathbf{F}[\![T]\!](x)$).

Semantics: Collection Types and Expressions:

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

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

In order to give a semantics to function applications we recall that pure expressions may 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 4. *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:

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

Our semantics has the following substitution property.

LEMMA 5.

(1) *For all v and pure e,*

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

(2) *For all T and v and pure e' and term t:*

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

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

The operational semantics preserves logical meaning:

PROPOSITION 1. *For all closed pure expressions e and e', if $e \to e'$ then $\models \boldsymbol{T}[\![e]\!] = \boldsymbol{T}[\![e']\!]$.*

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

THEOREM 1 (Full Abstraction). *For all closed pure expressions e and e', $\models \boldsymbol{T}[\![e]\!] = \boldsymbol{T}[\![e']\!]$ if and only if, for all r, $e \Downarrow r \Leftrightarrow e' \Downarrow r$.*

Proofs of Proposition 1 and Theorem 1 are in Appendix A.

We calculate the semantics of some example types from §2.4.

Semantics of Derived Forms:

$$\models \mathbf{T}[\![e_1 == e_2]\!] = \mathbf{Bind}\ x_1 \Leftarrow \mathbf{T}[\![e_1]\!]\ \mathbf{in}\ \mathbf{Bind}\ x_2 \Leftarrow \mathbf{T}[\![e_2]\!]\ \mathbf{in}\ \mathbf{Return}(\mathsf{v\_logical}(\mathbf{T}[\![e_1]\!] = \mathbf{T}[\![e_2]\!]))$$
$$\models \mathbf{F}[\![\mathsf{Empty}]\!](t) \Leftrightarrow \mathbf{false}$$
$$\models \mathbf{F}[\![[e : T]]\!](t) \Leftrightarrow \begin{cases} (\mathbf{T}[\![e]\!] = \mathbf{Return}(t)) \wedge \mathbf{F}[\![T]\!](t) & \text{if } e \text{ pure} \\ \mathbf{F}[\![T]\!](t) & \text{otherwise} \end{cases}$$
$$\models \mathbf{F}[\![\mathsf{Ok}(e)]\!](t) \Leftrightarrow \begin{cases} (\mathbf{T}[\![e]\!] = \mathbf{Return}(\mathbf{true})) & \text{if } e \text{ pure} \\ \mathbf{true} & \text{otherwise} \end{cases}$$
$$\models \neg\mathbf{W}[\![T]\!](t) \wedge \neg\mathbf{W}[\![U]\!](t) \implies (\mathbf{F}[\![T \mid U]\!](t) \Leftrightarrow (\mathbf{F}[\![T]\!](t) \vee \mathbf{F}[\![U]\!](t)))$$
$$\models \neg\mathbf{W}[\![U]\!](t) \implies (\mathbf{F}[\![T\ \&\ U]\!](t) \Leftrightarrow (\mathbf{F}[\![T]\!](t) \wedge \mathbf{F}[\![U]\!](t)))$$
$$\models \neg\mathbf{W}[\![T]\!](t) \implies (\mathbf{F}[\![!T]\!](t) \Leftrightarrow \neg\mathbf{F}[\![T]\!](t))$$
$$\models \textstyle\bigwedge_{i\in 1..n} \neg\mathbf{W}[\![T_i]\!](\mathsf{v\_dot}(t,\ell_i)) \implies (\mathbf{F}[\![\{\ell_i : T_i;\ ^{i\in 1..n}\}]\!](t) \Leftrightarrow \mathsf{is\_E}(t) \wedge \textstyle\bigwedge_{i\in 1..n}(\mathsf{v\_has\_field}(\ell_i,t) \wedge \mathbf{F}[\![T_i]\!](\mathsf{v\_dot}(t,\ell_i))))$$
$$\models \mathsf{Proper}(\mathbf{T}[\![e]\!]) \wedge \mathsf{In\_Logical}(\mathsf{out\_V}(\mathbf{T}[\![e]\!])) \wedge \neg\mathbf{W}[\![(x : T\ \mathbf{where}\ e)]\!](t) \wedge \neg\mathbf{W}[\![(x : U\ \mathbf{where}\ e)]\!](t) \implies \mathbf{F}[\![\mathbf{if}\ e\ \mathbf{then}\ T\ \mathbf{else}\ U]\!](t) \Leftrightarrow$$
$$\begin{cases} (\mathbf{F}[\![T]\!](t) \wedge (\mathbf{T}[\![e]\!] = \mathbf{Return}(\mathbf{true}))) \vee & \text{if } e \text{ pure} \\ (\mathbf{F}[\![U]\!](t) \wedge (\mathbf{T}[\![e]\!] = \mathbf{Return}(\mathbf{false}))) & \\ \mathbf{F}[\![T]\!](t) \vee \mathbf{F}[\![U]\!](t) & \text{otherwise} \end{cases}$$
$$\models \neg\mathbf{W}[\![U]\!](t) \implies (\mathbf{F}[\![\mathbf{Self}(s : T)U]\!](t) \Leftrightarrow \mathbf{F}[\![T]\!](t) \wedge \mathbf{let}\ s = t\ \mathbf{in}\ \mathbf{F}[\![U]\!](t))$$

4. 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
$dom(x_1 : T_1,\ldots,x_n : T_n) = \{x_1,\ldots,x_n\}$
$\mathbf{F}[\![x_1 : T_1,\ldots,x_n : T_n]\!] \triangleq \mathbf{F}[\![T_1]\!](x_1) \wedge \cdots \wedge \mathbf{F}[\![T_n]\!](x_n)$

Environments and 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 $f(x_1 : T_1,\ldots,x_n : T_n) : U\{e_f\}$ we assume that $x_1 : T_1,\ldots,x_n : T_n \vdash e_f : U$.

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

(Env Empty) $\dfrac{}{\varnothing \vdash \diamond}$ (Env Var) $\dfrac{E \vdash T \quad x \notin dom(E)}{E, x : T \vdash \diamond}$ (Type Any) $\dfrac{E \vdash \diamond}{E \vdash \mathsf{Any}}$ (Type Scalar) $\dfrac{E \vdash \diamond}{E \vdash G}$

(Type Collection) $\dfrac{E \vdash T}{E \vdash T*}$ (Type Entity) $\dfrac{E \vdash T}{E \vdash \{\ell : T\}}$ (Type Refine) $\dfrac{E, x : T \vdash e : \mathsf{Logical} \quad e \text{ pure}}{E \vdash (x : T\ \mathbf{where}\ e)}$

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

Rule of Semantic Subtyping:

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

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

(Exp Singular Subsum) $\dfrac{E \vdash e : T \quad E \vdash [e : T] <: T'}{E \vdash e : T'}$ (Exp Var) $\dfrac{E \vdash \diamond \quad (x : T) \in E}{E \vdash x : T}$ (Exp Const) $\dfrac{E \vdash \diamond}{E \vdash c : \mathsf{Any}}$

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

(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 Assert)
$$\frac{E \vdash e : T}{E \vdash (e\colon T) : T}$$

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

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

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

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

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

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

In the rule (Exp App), we require that each e_i in a dependent function application $f(e_1,\ldots e_n)$ is pure. This allows us to substitute these expressions into U. To form, say, $f(e)$ where e is impure, we can avoid this restriction by writing **let** $x = e$ **in** $f(x)$ instead.

The rule (Exp Cond) records the appropriate test expression in the environment, when typing the branches. Recall that the judgment $E,\_ : \mathsf{Ok}(e_1) \vdash e_2 : T$ is shorthand for $E,x : \mathsf{Ok}(e_1) \vdash e_2 : T$ where $x \notin fv(E,e_1,e_2,T)$, and the Ok-type $\mathsf{Ok}(e)$ is defined as (Any **where** e) if e is pure, and as Any otherwise (see §2.4).

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 2 (Logical Soundness).

(1) *If e is pure and $E \vdash e : T$ then:*
- $\models \boldsymbol{F}[\![E]\!] \implies \mathit{Proper}(\boldsymbol{T}[\![e]\!])$
- $\models \boldsymbol{F}[\![E]\!] \implies \boldsymbol{F}[\![T]\!](\mathit{out\_V}(\boldsymbol{T}[\![e]\!]))$

(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. □

We have the following basic properties.

LEMMA 6 (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 7 (All Values Typable). *For any v we have $E \vdash v : \mathsf{Any}$.*

Proof: By induction on the structure of v.

LEMMA 8 (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 $E,E' \vdash T$ and $E,E' \vdash S <: T$ and $E,E' \vdash e : T$. □

LEMMA 9 (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 induction on derivations. □

LEMMA 10.

(1) *For all e', x, pure e so that $E \vdash e : V$ we have that*
$\models \boldsymbol{F}[\![E]\!] \implies \boldsymbol{T}[\![e'\{\mathit{out\_V}(\boldsymbol{T}[\![e]\!])/x\}]\!] = \boldsymbol{T}[\![e'\{e/x\}]\!]$;
(2) *For all T, t, pure e so that $E \vdash e : V$ we have that*
$\models \boldsymbol{F}[\![E]\!] \implies \boldsymbol{F}[\![T\{\mathit{out\_V}(\boldsymbol{T}[\![e]\!])/x\}]\!](t) \Leftrightarrow \boldsymbol{F}[\![T\{e/x\}]\!](t)$.

Proof By mutual induction on the structure of e' and T.

LEMMA 11. *For all E, E', pure e, and x, if $E \vdash e : T$ then $\models \boldsymbol{F}[\![E]\!] \implies \boldsymbol{F}[\![E']\!]\{\mathit{out\_V}(\boldsymbol{T}[\![e]\!])/x\} \Leftrightarrow \boldsymbol{F}[\![E'\{e/x\}]\!]$.*

Proof: By induction on the structure of E', with appeal to Lemma 10. □

LEMMA 12 (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 13 (Substitution). *Suppose $E \vdash e' : T'$ and e' 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 (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 $\mathsf{out\_V}(\mathbf{T}[\![e']\!])$ for x, we obtain:
$$\models (\mathbf{F}[\![E]\!] \wedge \mathbf{F}[\![T']\!](\mathsf{out\_V}(\mathbf{T}[\![e']\!])) \wedge \mathbf{F}[\![E']\!]\{\mathsf{out\_V}(\mathbf{T}[\![e']\!])/x\} \wedge \mathbf{F}[\![S]\!](y)\{\mathsf{out\_V}(\mathbf{T}[\![e']\!])/x\}) \implies \mathbf{F}[\![T]\!](y)\{\mathsf{out\_V}(\mathbf{T}[\![e']\!])/x\}$$
By Lemma 5, Lemma 10, and Lemma 11, we have:
$$\models (\mathbf{F}[\![E]\!] \wedge \mathbf{F}[\![T']\!](\mathsf{out\_V}(\mathbf{T}[\![e']\!])) \wedge \mathbf{F}[\![E'\{e'/x\}]\!] \wedge \mathbf{F}[\![S\{e'/x\}]\!](y)) \implies \mathbf{F}[\![T\{e'/x\}]\!](y)$$
By Theorem 2, $E \vdash e' : T'$ and e' pure imply:
$$\models \mathbf{F}[\![E]\!] \Rightarrow \mathbf{F}[\![T']\!](\mathsf{out\_V}(\mathbf{T}[\![e']\!]))$$

Combining the previous two displayed formulas, we obtain:

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

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 12, $T = T'$.
By assumption, then, $E \vdash e' : T$.
By Lemma 8, 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. □

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 simplicity, (Exp Const) assigns $E \vdash c : \mathsf{Any}$ for any constant c when $E \vdash \diamond$. If $c \in K(G)$ for some $G \in \{\mathsf{Integer}, \mathsf{Text}, \mathsf{Logical}\}$, we can derive that $E \vdash c : G$ by observing the singular subtyping $E \vdash [c : \mathsf{Any}] <: G$, and hence applying (Exp Singular Subsum). For example, to see that $c \in K(\mathsf{Integer})$ implies that $E \vdash [c : \mathsf{Any}] <: \mathsf{Integer}$ note that $\models \mathbf{F}[\![[c : \mathsf{Any}]]\!](x) \Leftrightarrow x = c$ and hence that $\models \mathbf{F}[\![[c : \mathsf{Any}]]\!](x) \implies \mathsf{In\_Integer}(c)$.

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

LEMMA 14 (Singular Subtyping).
Suppose $E \vdash e : T$ and $E \vdash T' <: T$ and $x \notin dom(E)$.

(1) *If e is pure then:*
$E \vdash [e : T] <: T'$ *iff* $\models \boldsymbol{F}[\![E]\!] \wedge \boldsymbol{F}[\![T]\!](\mathit{out\_V}(\boldsymbol{T}[\![e]\!])) \implies \boldsymbol{F}[\![T']\!](\mathit{out\_V}(\boldsymbol{T}[\![e]\!]))$

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

Proof: By expanding definitions. □

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

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

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. 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 3.

LEMMA 15 (Transitivity of Singular Subtyping).
If $E \vdash [e : T] <: T'$ and $E \vdash [e : T'] <: T''$ then $E \vdash [e : T] <: T''$.

Proof: Sketch: an easy application of Lemma 14. □

Before we proceed to the preservation theorem, we first need to prove some inversion lemmas for entities and collections.

LEMMA 16 (Entity 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 $E \vdash \{\ell_i \Rightarrow v_i\ ^{i \in 1..n}\} : \mathit{Any}$ then $E \vdash v_i : \mathit{Any}$, for $i \in 1..n$.*

LEMMA 17 (Collection inversion).

(1) *If $E \vdash \{v_1, \ldots, v_n\} : T*$ then $E \vdash v_i : T$, for $i \in 1..n$.*
(2) *If $E \vdash \{v_1, \ldots, v_n\} : \mathit{Any}$ then $E \vdash v_i : \mathit{Any}$, 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 leads to a degenerate subtype relation.

LEMMA 18. *If $E \vdash v : T$ then $E, \_ : \mathit{Ok}(!v \textbf{ in } T) \vdash U <: V$, for any types U, V such that $E \vdash U$ and $E \vdash V$.*

THEOREM 3 (Preservation).
If $E \vdash e : T$ and $e \to e'$ then $E \vdash e' : T$.

Proof: By induction on the derivation of $E \vdash e : T$.
□

THEOREM 4 (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 $\varnothing \vdash e : T$. □

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 3, $\varnothing \vdash e : T$ and $e \to^* e'$ imply $\varnothing \vdash e' : T$. By Theorem 4, this implies e' cannot be stuck, a contradiction.

5. Algorithmic Aspects

5.1 Optimizing 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 2 , its logical semantics is a proper value, not **Error**.

This suggests that during type-checking we can optimize the logical semantics given the assumption that expressions are indeed well-typed. In particular, we can apply the following lemma to transform monadic error-checking lets into ordinary lets.

LEMMA 19. *If e pure and $E \vdash e : T$ then $\models \boldsymbol{F}[\![E]\!] \implies (\textbf{\textit{Bind}}\ x \Leftarrow \boldsymbol{T}[\![e]\!]\ \textbf{\textit{in}}\ t) = (\textbf{\textit{let}}\ x = \mathit{out\_V}(\boldsymbol{T}[\![e]\!])\ \textbf{\textit{in}}\ t)$.*

Proof: By definition of notation, **Bind** $x \Leftarrow \mathbf{T}[\![e]\!]$ **in** t is the term (**if** $\neg\mathsf{Proper}(\mathbf{T}[\![e]\!])$ **then Error else let** $x = \mathsf{out\_V}(\mathbf{T}[\![e]\!])$ **in** t). By Theorem 2, $\models \mathbf{F}[\![E]\!] \implies \mathsf{Proper}(\mathbf{T}[\![e]\!])$. Hence the result. □

The following tables present the optimized rules (as used in our checker), and the following theorem states their correctness.

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

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

$\mathbf{F}'[\![(x : T\ \textbf{where}\ e)]\!](t) =$
$\quad \mathbf{F}'[\![T]\!](t) \wedge \textbf{let}\ x = t\ \textbf{in}\ \mathbf{T}'[\![e]\!] = \textbf{true} \quad x \notin fv(T, t)$

Optimized Semantics of Pure Typed Expressions: $\mathbf{T}'[\![e]\!]$

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

(In this version we omit the definition of the function v_accumulate, which is a variant of res_accumulate that works with values rather than results.)

THEOREM 5 (Soundness of Optimized Semantics).

(1) *If* $E \vdash T$ *and* $x \notin dom(E)$ *then:*
$\models (\boldsymbol{F}[\![E]\!] \implies (\boldsymbol{F}[\![T]\!](x) \Leftrightarrow \boldsymbol{F}'[\![T]\!](x))$.
(2) *If* $E \vdash e : T$ *then:*
$\models \boldsymbol{F}[\![E]\!] \implies (\boldsymbol{T}[\![e]\!] = \textbf{\textit{Return}}(\boldsymbol{T}'[\![e]\!]))$.

Proof: The proof is by simultaneous induction on the derivations of $E \vdash T$ and $E \vdash e : T$, with appeal to Theorem 2 and Lemma 19.□

5.2 Bidirectional Typing Rules

The Dminor type system is implemented as a *bidirectional* type system [42]. 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 without the use of long-distance constraints such as unification variables (as used in, e.g., ML).

Judgments of the Algorithmic Type System:

$E \vdash e \to T$	in E, expression e synthesizes type T
$E \vdash e \leftarrow T$	in E, expression e checks against type T

Bidirectional type systems are simple to implement, and expressive; for example, the type system for C♯ can be defined as a bidirectional type system [13], and several dependently-typed languages have bidirectional type systems [36, 34].

The algorithmic nature of bidirectional type systems makes them predictable to programmers. This is an important feature, not least because it makes type error reporting easy—a disadvantage of languages that use ML-style inference [35].

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

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

(Synth Const)
$$\frac{E \vdash \diamond}{E \vdash c \to [c]}$$

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

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

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

(Synth Test)
$$\frac{E \vdash e \leftarrow \mathsf{Any} \quad E \vdash T}{E \vdash e \textbf{ in } T \to \mathsf{Logical}}$$

(Synth Assert)
$$\frac{E \vdash e \leftarrow T}{E \vdash (e : T) \to T}$$

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

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

(Synth Zero)
$$\frac{E \vdash \diamond}{E \vdash \{\} \to [\{\} : \mathsf{Empty}*]}$$

(Synth Coll)
$$\frac{E \vdash v_i \to T_i \quad \forall i \in 1..n \quad n > 0}{E \vdash \{v_i,\ldots,v_n\} \to (T_1 \mid \ldots \mid T_n)*}$$

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

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

The rules (Synth Var), and (Synth Const) yield singleton types for all variables and constants. The (Synth Cond) rule synthesizes a type for the conditional expression $e_1 ? e_2 : e_3$. The overall synthesized type is the union of the two types synthesized for the branches, although we record the test expression in the type (if it is pure). This allows for more precise typing. Rule (Synth Entity) uses intersection types to encode record types.

In a number of the type synthesis rules we need to inspect components of intermediate types. In simple type systems this is straightforward as we can rely on the syntactic structure of types, but for rich type systems this is not possible. In existing implementations of dependently-typed languages, either the programmer is required to insert casts to force the type into the appropriate syntactic shape [53], or types are inspected after being evaluated to some form of (weak-head) normal form [6]. 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 potentially very large data sets. One pragmatic possibility is to attempt type normalization but place some ad hoc bound on evaluation (e.g. SAGE takes this approach [34]). 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.

Normal Types (DNF) and Normalization:

$D ::= R_1 \mid \ldots \mid R_n$	normal disjunction (Empty if $n = 0$)
$R ::= x : C \textbf{ where } e$	normal refined conjunction
$C ::= f_1 A_1 \,\&\, \ldots \,\&\, f_n A_n$	normal conjunction (Any if $n = 0$)
$f ::= \varepsilon \mid !$	optional negation
$A ::= G \mid T* \mid \{\ell : T\}$	atomic type

$norm(\mathsf{Any}) \triangleq x : \mathsf{Any} \textbf{ where true}$
$norm(G) \triangleq x : G \textbf{ where true}$
$norm(T*) \triangleq x : T* \textbf{ where true}$
$norm(\{\ell : T\}) \triangleq x : \{\ell : T\} \textbf{ where true}$

$$
\begin{array}{l}
\mathit{norm}(x:T \textbf{ where } e) \triangleq \\
\quad |_{i=1}^{n}\ \mathit{Conj}_{DD}(x_i:C_i \textbf{ where } e_i, \mathit{norm}_r(x:C_i \textbf{ where } e)) \\
\quad\ \text{where } |_{i=1}^{n}\ (x_i:C_i \textbf{ where } e_i) = \mathit{norm}(T) \\
\mathit{norm}_r(x:C \textbf{ where } x \textbf{ in } T) \triangleq \mathit{norm}(C \ \&\ T) \quad \text{where } x \notin \mathit{fv}(T) \\
\mathit{norm}_r(x:C \textbf{ where } e_1 \mid\mid e_2) \triangleq \\
\quad \mathit{norm}_r(x:C \textbf{ where } e_1) \mid \mathit{norm}_r(x:C \textbf{ where } e_2) \\
\mathit{norm}_r(x:C \textbf{ where } e_1 \ \&\&\ e_2) \triangleq \\
\quad \mathit{Conj}_{DD}(\mathit{norm}_r(x:C \textbf{ where } e_1), \mathit{norm}_r(x:C \textbf{ where } e_2)) \\
\mathit{norm}_r(x:C \textbf{ where } !e) \triangleq \mathit{Neg}_D(\mathit{norm}_r(x:C \textbf{ where } e)) \\
\mathit{norm}_r(x:C \textbf{ where } e) \triangleq (x:C \textbf{ where } e) \qquad \text{otherwise} \\
\\
\mathit{Conj}_{DD}((R_1 \mid \ldots \mid R_n), D) \triangleq \mathit{Conj}_{RD}(R_1, D) \mid \ldots \mid \mathit{Conj}_{RD}(R_n, D) \\
\mathit{Conj}_{RD}(R, (R_1 \mid \ldots \mid R_n)) \triangleq \mathit{Conj}_{RR}(R, R_1) \mid \ldots \mid \mathit{Conj}_{RR}(R, R_n) \\
\mathit{Conj}_{RR}(x_1:C_1 \textbf{ where } e_1, x_2:C_2 \textbf{ where } e_3) \triangleq \\
\quad y:C_1 \ \&\ C_2 \textbf{ where } e_1\{y/x_1\} \ \&\&\ e_2\{y/x_2\} \\
\qquad\qquad \text{where } y \notin \mathit{fv}(C_1, C_2, e_1, e_2) \\
\\
\mathit{Neg}_D(R_1 \mid \ldots \mid R_n) \triangleq \mathit{Conj}_{DD}(\mathit{Neg}_R(R_1), \ldots, \mathit{Neg}_R(R_n)) \\
\mathit{Neg}_R(x:C \textbf{ where } e) \triangleq \mathit{Neg}_C(C) \mid x:C \textbf{ where } !(e) \\
\mathit{Neg}_C(f_1 C_1 \ \&\ \ldots \ \&\ f_n C_n) \triangleq \neg(f_1) C_1 \mid \ldots \mid \neg(f_n) C_n \\
\neg(\varepsilon) \triangleq\ ! \qquad \neg(!) \triangleq \varepsilon
\end{array}
$$

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

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

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

$$
\begin{array}{ll}
\text{(Field Disj)} & \text{(Field Refine)} \\
\dfrac{R_i.\ell \rightsquigarrow U_i \quad \forall i \in 1..n}{(R_1 \mid \ldots \mid R_n).\ell \rightsquigarrow (U_1 \mid \ldots \mid U_n)} & \dfrac{C.\ell \rightsquigarrow U}{(x:C \textbf{ where } e).\ell \rightsquigarrow U} \\
\\
\text{(Field Conj)} & \text{(Field Atom)} \\
\dfrac{S = \{U_i \mid A_i.\ell \rightsquigarrow U_i \wedge f_i = \varepsilon \wedge i \in 1..n\} \neq \varnothing \quad S_! = \{U_i \mid A_i.\ell \rightsquigarrow U_i \wedge f_i = ! \wedge i \in 1..n\} = \varnothing}{(f_1 A_1 \ \&\ \ldots \ \&\ f_n A_n).\ell \rightsquigarrow (\&\ S)} & \dfrac{}{\{\ell:T\}.\ell \rightsquigarrow T}
\end{array}
$$

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

$$
\begin{array}{ll}
\text{(Items Disj)} & \text{(Items Refine)} \\
\dfrac{R_i.\mathsf{Items} \rightsquigarrow U_i \quad \forall i \in 1..n}{(R_1 \mid \ldots \mid R_n).\mathsf{Items} \rightsquigarrow (U_1 \mid \ldots \mid U_n)} & \dfrac{C.\mathsf{Items} \rightsquigarrow U}{(x:C \textbf{ where } e).\mathsf{Items} \rightsquigarrow U} \\
\\
\text{(Items Conj)} & \text{(Items Atom)} \\
\dfrac{S = \{U_i \mid A_i.\mathsf{Items} \rightsquigarrow U_i \wedge f_i = \varepsilon \wedge i \in 1..n\} \neq \varnothing \quad S_! = \{U_i \mid A_i.\mathsf{Items} \rightsquigarrow U_i \wedge f_i = ! \wedge i \in 1..n\} = \varnothing}{(f_1 A_1 \ \&\ \ldots \ \&\ f_n A_n).\mathsf{Items} \rightsquigarrow (\&\ S)} & \dfrac{}{(T*).\mathsf{Items} \rightsquigarrow T}
\end{array}
$$

Type checking expressions: $E \vdash e \leftarrow T$

$$
\begin{array}{ll}
\text{(Swap)} & \text{(Check Cond)} \\
\dfrac{E \vdash e \rightarrow T \quad E \vdash [e:T] <: T'}{E \vdash e \leftarrow T'} & \dfrac{E \vdash e_1 \leftarrow \mathsf{Logical} \quad E, \_ : \mathsf{Ok}(e_1) \vdash e_2 \leftarrow T \quad E, \_ : \mathsf{Ok}(!e_1) \vdash e_3 \leftarrow T}{E \vdash e_1 ? e_2 : e_3 \leftarrow T} \\
\\
\text{(Check Let)} & \text{(Check Dot)} \\
\dfrac{E \vdash e_1 \rightarrow T \quad E, x:T \vdash e_2 \leftarrow U \quad x \notin \mathit{fv}(U)}{E \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \leftarrow U} & \dfrac{E \vdash e \leftarrow \{\ell:T\}}{E \vdash e.\ell \leftarrow T}
\end{array}
$$

Typically (e.g. SAGE [34]), the type checking relation for a bidirectional type system consists only of the following rule.

$$\frac{E \vdash e \rightarrow T \quad E \vdash T <: T'}{E \vdash e \leftarrow T'}$$

However, we have found in practice that this rule is too strict. In particular, it entails a subtype test for every occurrence of type checking. In the cases where the expression is a conditional or a let-expression, it is more efficient to pass 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.

We also observe that the rule given in the previous paragraph is rather strong, as it requires an inclusion between the types T and T'. The (Swap) rule used in Dminor tests only for singular subsumption.

LEMMA 20 (Synthesis Checkable). *If $E \vdash e \rightarrow T$ then $E \vdash e \leftarrow T$.*

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

THEOREM 6 (Soundness of Bidirectional Typing Rules).

(1) *If $E \vdash e \rightarrow T$ then $E \vdash e : T$.*
(2) *If $E \vdash e \leftarrow T$ then $E \vdash e : T$.*

Proof: By simultaneous induction over the derivations of $E \vdash e \rightarrow T$ and $E \vdash e \leftarrow T$.

□

5.3 Algorithmic Purity Check

In Dminor calls to recursive functions can cause divergence so we use a syntactic termination condition on the functions that are used inside refinements: the recursive calls can only be on syntactically smaller arguments. Also, since collections are unordered, iterating over them with accumulate is nondeterministic, and can in general produce more than one result, therefore we need to impose conditions on the accumulate expressions occurring inside refinements which guarantee that the order in which the elements are processed is irrelevant for the final result.

An expression e is *algorithmically pure* if and only if these conditions hold:

(1) if e is a function application $f(e_1, \ldots, e_n)$ then f is labeled-pure, and and only calls f (directly or indirectly) on structurally smaller arguments;
(2) if e is of the form $\textbf{from } x \textbf{ in } e_1 \textbf{ let } y = e_2 \textbf{ accumulate } e_3$ then
   (a) $\models \mathbf{T}[\![e_3\{x_1/x\}\{x_2/y\}]\!] = \mathbf{T}[\![e_3\{x_2/x\}\{x_1/y\}]\!]$, and
   (b) $\models \mathbf{T}[\![\textbf{let } z = e_3\{x_1/x\}\{x_2/y\} \textbf{ in } e_3\{z/x\}\{x_3/y\}]\!]$
   $= \mathbf{T}[\![\textbf{let } z = e_3\{x_2/x\}\{x_3/y\} \textbf{ in } e_3\{x_1/x\}\{z/y\}]\!]$
   (where the variables x_1, x_2, x_3 and z do not appear free in e_3);
(3) all the proper subexpressions of e are algorithmically pure.

Condition (1) enforces termination of recursive functions, while condition (2) only allows accumulate expressions in a pure expression if their body is commutative (2a) and associative (2b), which is a sufficient condition for uniqueness of evaluation results of syntactically pure expressions. For simplicity we phrased commutativity and associativity in terms of the error-tracking logical semantics from §3. It is possible, albeit more complicated, to phrase them in terms of the optimized semantics from §5.1 (we work this out in detail in Appendix **??**). Finally, we assume that the body of every labeled-pure function is an algorithmically pure expression.

THEOREM 7. *If e is algorithmically pure then e is pure.*

Proof: Immediate from Lemmas 35 and 36.

6. 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[3] 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.

6.1 Precise Counterexamples to Type-checking

The type-checking algorithm from §5.2 crucially relies on subtyping, as in the rule (Swap), and our semantic subtyping relation $E \vdash T <: T'$ 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) Each of the expressions $(y\sigma \textbf{ in } U\sigma)$, for all $(y:U) \in E$, and also the expression $(x\sigma \textbf{ in } (T \,\&!T')\sigma)$ are pure;

(2) $(y\sigma \textbf{ in } U\sigma) \to^* \textbf{true}$, for all $(y:U) \in E$;

(3) $(x\sigma \textbf{ in } (T \,\&!T')\sigma) \to^* \textbf{true}$.

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.

LEMMA 21. *If the three checks above succeed then $E \not\vdash T <: T'$.*

Proof: 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 Proposition 1 it follows that $\models \mathbf{T}[\![y_i\sigma \textbf{ in } U_i\sigma]\!] = \textbf{true}$ 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 Proposition 1 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.

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.

[3] 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.

6.2 Finding Elements of Types Statically

Type emptiness can be phrased in terms of subtyping as $E \vdash T <:$ 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. Moreover, if the SMT solver cannot prove that a type is empty we again obtain an assignment σ, which we can validate as in §6.1. 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.

LEMMA 22. *If the three checks in §6.1 succeed for $T' =$ 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 7 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 Proposition 1 and basic reasoning in first-order logic.

6.3 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 that tries to find an inhabitant of T (this primitive is not present in the M language). 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 can be found.)

Operational Semantics for Finding Elements of Types:

elementof $T \to v$ where v **in** $T \to^*$ **true**
elementof $T \to$ **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:

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

LEMMA 23. *If $E \vdash T$ then the expression v **in** T is pure.*

COROLLARY 1. *If **elementof** $T \to v$ and $\varnothing \vdash T$ then we have that $\varnothing \vdash v : T \mid [\textbf{null}]$.*

Proof: By Lemma 23 we have that v **in** T is pure. By the operational semantics we have that, either $v =$ **null** in which case the conclusion is immediate, or we know that v **in** $T \to^*$ **true**, which allows us to apply Lemma 22 for $E = \varnothing$ and obtain that $\varnothing \vdash v : T$.

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 very 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 (semi-)decision procedures and heuristics to perform the computation. If this fails or is too slow one can just implement the required functionality manually. There is only very 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.

7. Implementation

Our Dminor implementation is approximately 2700 lines of F♯ code, excluding the lexer and parser. Our type-checker implements the optimized logical semantics from §5.1, the bidirectional typing rules from §5.2, and the algorithmic purity check from §5.3. We use Z3 [20] 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 [45], which uses the theories of integers, datatypes and extensional arrays (see Appendix C). The definition of our intended model of Dminor values, results and operations adds an additional 2000 lines of Coq (see Appendix B).

We have tested our type-checker on a test suite consisting of about 130 files, some hand-crafted by us and some transliterated from the M preliminary release. Even without any aggressive optimization the type-checker is fast. Checking each of the 130 files in our test suite on a normal laptop takes from under 1 second (for just startup and parsing) to around 20 seconds (for type-checking an 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,[4] 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 §6. We built a plugin for the Microsoft Intellipad text editor that displays precise counterexamples to typing, flags empty types and otherwise displays one element of each type defined in the code (see the screencast mentioned in §1.6). Moreover, our interpreter for Dminor supports **elementof** for dynamically generating instances of types (§6.3). 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.

8. Safe Systems Configurations by Typing

To conclude our development, we argue that type-based modeling in M may be of practical benefit in an intended application area such as systems administration.

Note first that 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.

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

Consider a concrete example, the WSE Policy Advisor [12, 11]. (There are many such advisors; we select WSE Policy Advisor because it is familiar to us.) WSE Policy Advisor is an XSLT stylesheet 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).

8.1 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.

```
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*;}*;};
```

8.2 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. Hence, static type-checking of systems models can go beyond the current generation of advisors.

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 [40], set comprehensions in VDM [33], and predicate subtypes in PVS [48]. In PVS, constraints found when checking predicate subtypes become proof obligations to be proved interactively. In future work, it may be useful to allow the Dminor user to supply proof scripts in the event that the automatic solver fails to prove implications generated by semantic subtyping. More recently, Sozeau [51] extends Coq with subset types; his system implements syntactic subtyping and lacks type-test; on the other hand, Dminor is based on classical logic, and does not support proof objects for certification.

Pratt [44] 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 [29] extended ML with a form of refinement type, and Xi and Pfenning [53] 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 [36] 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 [9, 47] 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 [34]. 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 [28] 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 [39] and DrScheme [26], and static typing. Compared to purely static typing this can reduce the number of false alarms generated by type-checking.

X10 [49] 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. However, in comparison with Dminor, X10 combines semantic and syntactic subtyping and the constraint language [49, §2.11] does not support type-test expressions.

In spite of early work on semantic subtyping by Aiken and Wimmers [3] and Damm [19], 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 [41]. Unfortunately, deriving an algorithm from such a set of rules can be difficult, especially for advanced features such as intersection and union types [23, 24].

The introduction of XML and XML query languages lead 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 operations (such as `*`, `?`, and `|`) and subtyping between two types becomes inclusion between the set of sequences that are denoted by the regular expression types. Hosoya and Pierce first defined such a type system for XML [32] and their language, XDuce. Frisch, Castagna, and Benzaken [30] extended semantic subtyping to function types and propositional types, with type-test, but not refinement types, resulting in the language CDuce [10]. (An excellent overview of the use of semantic subtyping in the context of querying XML documents was given by Castagna [17].) In the end, the XQuery working group resorted to a more conventional (but less precise) nominal, structural type system [50]. Neither XDuce nor CDuce provides general refinement types, and their subtype algorithm is purpose-built. As far as we are aware, our use of a general-purpose theorem prover to determine Dminor's very general notion of semantic subtyping is novel.

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 also issues warnings if empty types are detected. These tasks are much simpler in CDuce then they are in our setting, since we additionally have to deal with general refinement types.

Typed Scheme [52] makes use of type-test expressions, union types and notions of visible and latent predicates to type-check Scheme programs. It would be interesting to see if these idioms can be internalized in the Dminor type system using refinements.

PADS [27] 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.

10. Conclusions

We have described Dminor, a simple, yet flexible, functional language for defining data models and queries over these data models.

The main novelty of Dminor is its especially rich type system. The combination of refinement types and type-test appears to be new. On top of familiar arithmetic constraints on types (analo-

gous to the sort checked dynamically by other data modeling languages) we have given examples of how this type system can, in addition, encode singleton, nullable, union, intersection, negation, and algebraic types, although without first-class functions. We have sketched how such a rich type system is useful for scripting systems configuration, a key application area M [2].

The other 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 a theorem prover to perform semantic subtyping. (Other systems have either devised special purpose algorithms for semantic subtyping, or used theorem provers only for refinement types.) The design of our bidirectional type system to enable precise typing of programs appears novel. We have implemented our type system in F♯ using the Z3 SMT solver. SMT solvers are now of sufficient maturity that they can realistically be thought of as a platform upon which many applications, including type systems, may be built.

In the future we intend to extend our implementation to cover more of the features proposed for the M Modeling Language, including database update. This will also enable us to test our implementation on more of the samples that are already available in the M CTP. In particular, we plan a more systematic study of extending Dminor to represent graphs by adding labels and pointers to labels. We intend to add support for first-class functions by generalizing the mixture of syntactic and semantic subtyping introduced by Calcagno, Cardelli, and Gordon [16]. We also intend to implement a form of hybrid type checking, as supported by SAGE, to allow particular programs that can not be type-checked at compile-time, to be checked at run-time instead.

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 [28] 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 incompletenesses of the SMT solver. The techniques used in SAGE can be applied to Dminor without any great difficulty.

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 type-test expressions when programming with first-order data, and the viability of type-checking such programs with an SMT solver.

Acknowledgments We thank Nikolaj Bjørner for his invaluable help in using Z3. Paul Anderson, Ioannis Baltopoulos, Johannes Borgström, and Tim Harris commented on a draft. Discussions with Martín Abadi, Cliff Jones, and Benjamin Pierce were useful.

A. Relating Operational and Logical Semantics

We develop proofs for Proposition 1 and Theorem 1.

We begin by considering the following direct inductive definition of the relation $e \Downarrow r$.

Evaluation Semantics: $e \Downarrow r$

(Eval Const)

$$\frac{}{c \Downarrow \textbf{Return}(c)}$$

(Eval Operator 1)

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

(Eval Operator 2)

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

(Eval Operator 3)

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

(Eval Cond 1)

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

(Eval Cond 2)

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

(Eval Let 1)

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

(Eval Let 2)

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

(Eval Entity 1)

$$\frac{e_j \Downarrow \textbf{Error} \quad j \in 1..n}{\{\ell_i \Rightarrow e_i\ ^{i\in 1..n}\} \Downarrow \textbf{Error}}$$

(Eval Entity 2)

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

(Eval Dot 1)

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

(Eval Dot 2)

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

(Eval Collection)

$$\frac{}{\{v_1,\ldots,v_n\} \Downarrow \textbf{Return}(\{v_1,\ldots,v_n\})}$$

(Eval Add 1)

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

(Eval Add 2)

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

(Eval Add 3)

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

(Eval Appl 1)

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

(Eval Appl 2)

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

(Eval Accum 1)

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

(Eval Accum 2)

$$\frac{e_2 \Downarrow \textbf{Error}}{\textbf{from } x \textbf{ in } e_1 \textbf{ let } y = e_2 \textbf{ accumulate } e_3 \Downarrow \textbf{Error}}$$

(Eval Accum 3)

$$\frac{e_1 \Downarrow \textbf{Return}(\{v_1,\ldots,v_n\}) \quad e_2 \Downarrow \textbf{Return}(v) \quad e_3\{v_i/x\}\{v/y\} \Downarrow \textbf{Error} \quad i \in 1..n}{\textbf{from } x \textbf{ in } e_1 \textbf{ let } y = e_2 \textbf{ accumulate } e_3 \Downarrow \textbf{Error}}$$

(Eval Accum 4)
$$\frac{e_1 \Downarrow \textbf{Return}(\{v_1,\ldots,v_n\}) \quad e_2 \Downarrow \textbf{Return}(v) \quad e_3\{v_j/x\}\{v/y\} \Downarrow \textbf{Return}(v') \quad j \in 1..n \quad \textbf{from } x \textbf{ in } \{v_i \,{}^{i\in(1..n)-\{j\}}\} \textbf{ let } y = v' \textbf{ accumulate } e_3 \Downarrow r}{\textbf{from } x \textbf{ in } e_1 \textbf{ let } y = e_2 \textbf{ accumulate } e_3 \Downarrow r}$$

(Eval Accum 5)
$$\frac{e_1 \Downarrow \textbf{Return}(\{\}) \quad e_2 \Downarrow \textbf{Return}(v)}{\textbf{from } x \textbf{ in } e_1 \textbf{ let } y = e_2 \textbf{ accumulate } e_3 \Downarrow \textbf{Return}(v)}$$

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

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

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

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

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

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

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

(Test Collection 2)
$$\frac{e \Downarrow \textbf{Return}(\{v_1,\ldots,v_n\}) \quad v_j \textbf{ in } T \Downarrow \textbf{Error} \quad j \in 1..n}{e \textbf{ in } T* \Downarrow \textbf{Error}}$$

(Test Collection 3)
$$\frac{e \Downarrow \textbf{Return}(\{v_1,\ldots,v_n\}) \quad v_j \textbf{ in } T \Downarrow \textbf{Return}(u_i) \quad \forall i \in 1..n}{e \textbf{ in } T* \Downarrow (\textbf{Return}(\textbf{true}) \text{ if each } u_i = \textbf{true}, \text{ otherwise } \textbf{Return}(\textbf{false}))}$$

(Test Refine 1)
$$\frac{e_1 \textbf{ in } T \Downarrow \textbf{Error}}{e_1 \textbf{ in } (x : T \textbf{ where } e_2) \Downarrow \textbf{Error}}$$

(Test Refine 2)
$$\frac{e_1 \Downarrow \textbf{Return}(v) \quad v \textbf{ in } T \Downarrow \textbf{Return}(v_1) \quad e_2\{v/x\} \Downarrow r \quad r \notin \{\textbf{Return}(\textbf{true}), \textbf{Return}(\textbf{false})\}}{e_1 \textbf{ in } (x : T \textbf{ where } e_2) \Downarrow \textbf{Error}}$$

(Test Refine 3)
$$\frac{e_1 \Downarrow \textbf{Return}(v) \quad v \textbf{ in } T \Downarrow \textbf{Return}(u_1) \quad e_2\{v/x\} \Downarrow \textbf{Return}(u_2)}{e_1 \textbf{ in } (x : T \textbf{ where } e_2) \Downarrow (\textbf{Return}(\textbf{true}) \text{ if each } u_i = \textbf{true}, \text{ otherwise } \textbf{Return}(\textbf{false}))}$$

(Eval Assert 1)
$$\frac{e \Downarrow \textbf{Error}}{e : T \Downarrow \textbf{Error}}$$

(Eval Assert 2)
$$\frac{e \Downarrow v \quad (v \textbf{ in } T)?v : \mathsf{stuck} \Downarrow r}{e : T \Downarrow r}$$

LEMMA 24. *If v is a value then $v \Downarrow$ **Return**(v).*

Proof: By induction on the structure of v. □

LEMMA 25. *Suppose e is closed. If $e \to e'$ 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 $e \to e'$. We omit the details. □

LEMMA 26. *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 proposition, 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.

PROPOSITION 2. *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 $e \Downarrow \textbf{Return}(v)$ and $e \Downarrow \textbf{Error}$.

For the reverse direction of (1), we have $e = e_1 \to \cdots \to e_n \to v$. By Lemma 24, we have $v \Downarrow \textbf{Return}(v)$. By repeated applications of Lemma 25, we have $e_i \Downarrow \textbf{Return}(v)$ for each i from n down to 1, and indeed $e_i \Downarrow \textbf{Return}(v)$.

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 26, we have $e_n \Downarrow \textbf{Error}$. By repeated applications of Lemma 25, we have $e_i \Downarrow \textbf{Error}$ for each i from n down to 1, and indeed $e \Downarrow \textbf{Error}$. □

The following asserts that if a closed pure expression evaluates to a result, then that is the result of the expression according to the logical semantics.

LEMMA 27. *For closed and pure e and r, if $e \Downarrow r$ then $\models \boldsymbol{T}[\![e]\!] = r$.*

Proof: The proof is by induction on the derivation of $e \Downarrow r$. (Notice that the purity assumption arises explicitly in the case (Eval Appl 2) for function calls, which needs Lemma 4.) □

Restatement of Proposition 1 *For all closed and pure e and e', if $e \to e'$ then $\models \boldsymbol{T}[\![e]\!] = \boldsymbol{T}[\![e']\!]$.*

Proof: Suppose $e \to e'$. By Lemma 1, 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 25, $e \to e'$ and $e' \Downarrow r$ imply $e \Downarrow r$. By Lemma 27, we have both $\models \mathbf{T}[\![e']\!] = r$ and $\models \mathbf{T}[\![e]\!] = r$. By transitivity, $\models \mathbf{T}[\![e]\!] = \mathbf{T}[\![e']\!]$. □

Restatement of Theorem 1 *For all closed pure expressions e and e', we have $\models \boldsymbol{T}[\![e]\!] = \boldsymbol{T}[\![e']\!]$ 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 27, we have $\models \mathbf{T}[\![e]\!] = r$ and $\models \mathbf{T}[\![e']\!] = r'$. Given these facts, we have: $\models \mathbf{T}[\![e]\!] = \mathbf{T}[\![e']\!]$ if and only if $r = r'$ if and only if for all r'', $e \Downarrow r'' \Leftrightarrow e' \Downarrow r''$. □

B. Mechanized Definition of the Intended Dminor Model

We have formalized the definition of the intended Dminor Model in Coq [1].

B.1 Values

We first define scalars 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. This representation is not canonical, i.e. multiple representations for the same value exists, which means we cannot use syntactic equality to compare raw value.

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.
```

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 (a string), and do not contain duplicate fields. Collections in normal form are sorted with respect to a total order on raw values (this order is arbitrary but fixed; note that 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 it is usual for FOL models.[5]

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).
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 [51] of RawValue for which the Normal predicate holds. The sorts SVMap, VBag, and ValueOption are interpreted by similar Coq subset types.

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}.

Definition lift_option (X : Type) (P : X → Prop) : option X → Prop :=
  fun ox ⇒ match ox with None ⇒ True | Some x ⇒ P x end.
Implicit Arguments lift_option [X].
Definition ValueOption :=
  {vo : option RawValue | lift_option Normal vo}.
```

Model: All Values "Good"

```
Definition Good (v : Value) := true.
Definition Good_C := is_C.
Definition Good_E := is_E.
```

B.2 Operations on Simple Values

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).
```

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).
```

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))).
```

B.3 Operations on Entities

Model: Entities Represented as Extensional Arrays

```
Definition SVMapArray := Array string ValueOption.
Definition FiniteE (svm : SVMapArray) : bool :=
  match default svm with exist None _ ⇒ true | _ ⇒ false end.
```

Model: Functions and Predicates on Entities

```
Program Definition v_eempty : Value := E nil.
Program Definition v_eupdate s (nv ne : Value) : Value :=
  E (remove_dup eq_str_dec (insert_in_sorted_svm (s, nv) (out_E ne))).
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 (* arbitrary *) end.
```

B.4 Operations on Collections

Model: Entities Represented as Extensional Arrays

```
Definition VBagArray := Array Value Z.
```

[5] 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 Finite (vb : VBagArray) : bool := beq_int (default vb) 0%Z.
Program Definition cmp_val (v1 v2 : Value) := cmp_rval v1 v2.
Program Definition Positive (vb : VBagArray) : bool :=
  allb_R cmp_val vb (fun (z : Z) ⇒
    (match Zcompare z 0 with Lt ⇒ false | _ ⇒ true end)).
```

Model: Functions and Predicates on Collections

```
Program Definition v_zero : Value := C nil.
Program Definition v_mem (v cv : Value) : bool :=
  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).
```

C. Axiomatization of the Dminor Model

We axiomatize the model of Dminor in sorted first-order logic extended with the background theories of integer arithmetic, algebraic datatypes, and arrays. We only axiomatize the parts of the model that are relevant for the optimized logical semantics is §5.1.

In the following we report all the relevant parts of this axiomatization, directly imported from our implementation. We use the standard SMT-LIB format [45] supported by all recent SMT solvers, together with Z3-specific [20] extensions for algebraic datatypes and arrays [21].

C.1 Values

We begin 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))
```

Values are also defined as a datatype. We use arrays to represent entities and collections, however, since Z3 syntactically restricts arrays from appearing inside datatypes, we use two new sorts SVMap and VBag instead. The sort SVMap is then constrained to be isomorphic to the arrays from Strings to Values, while VBag is required to be isomorphic to the arrays from Values to Int.

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
```

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 later on).

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) })
```

C.2 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 (= 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.

C.3 Operations on Entities

Entities:

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

The operations v_eempty and v_eupdate do not correspond to any Dminor construct, but they allow us to construct entity values in an abstract way (without caring how they are implemented – for example, lists vs. arrays)

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)
```

C.4 Operations on Collections

Collections:

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

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 (Closure Closure2)
:extrafuns ((v_apply Closure Value Value)
            (v_apply2 Closure2 Value Value Value))

:extrafuns ((closure_tag Closure Value))
```

Operations on Collections:

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

: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 v vs)) :pat{ (v_add v vs) })

; v_accumulate iterates over a collection using an AC operator
: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_many v 1 vs) initial)
     (v_accumulate clos vs (v_apply2 clos v initial)))
  :pat { (v_accumulate clos (v_add_many v 1 vs) initial) })
```

D. Algorithmic Purity Check

D.1 Proofs

Operational purity is defined in the same way as algorithmic purity, just that we replace condition (2) with a condition we call *operational commutativity and associativity*:

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

(a) $e_3\{x_1/x\}\{x_2/y\} \approx e_3\{x_2/x\}\{x_1/y\}$, and

(b) **let** $z = e_3\{x_1/x\}\{x_2/y\}$ **in** $e_3\{z/x\}\{x_3/y\}$
$\approx$ **let** $z = e_3\{x_2/x\}\{x_3/y\}$ **in** $e_3\{x_1/x\}\{z/y\}$

(where the variables x_1, x_2, x_3 and z do not appear free in e_3).

The $\approx$ symbol denotes *observational equivalence*, which is defined as follows. Two expressions e_1 and e_2 are observationally equivalent if for all closing substitutions σ we have that $e_1\sigma \Downarrow r$ if and only if $e_2\sigma \Downarrow r$.

THEOREM 8 (Full Abstraction for Open Expressions). *For all pure expressions e_1 and e_2 we have that $e_1 \approx e_2$ if and only if $\models \boldsymbol{T}[\![e_1]\!] = \boldsymbol{T}[\![e_2]\!]$.*

Proof: Immediate from Theorem 1 and Lemma 5.

LEMMA 28. *If e is algorithmically pure and $e \to e'$ then e' is also algorithmically pure.*

LEMMA 29. *The reduction relation is terminating on algorithmically pure expressions. (i.e. all reduction sequences starting from 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 each proper accumulate step decreases the size of the collection by one, so since collections are finite this will again always terminate.

LEMMA 30. *If **from** x **in** $\{v_1,\ldots,v_n,v'_1,\ldots,v'_m\}$ **let** $y = u$ **accumulate** $e_3 \Downarrow v$ then there exists a value w so that **from** x **in** $\{v_1,\ldots,v_n\}$ **let** $y = u$ **accumulate** $e_3 \Downarrow w$ and **from** x **in** $\{v'_1,\ldots,v'_m\}$ **let** $y = w$ **accumulate** $e_3 \Downarrow v$.*

LEMMA 31. *If **from** x **in** $\{v_1,\ldots,v_n\}$ **let** $y = u$ **accumulate** $e_3 \Downarrow w$ and **from** x **in** $\{v'_1,\ldots,v'_m\}$ **let** $y = w$ **accumulate** $e_3 \Downarrow v$ then **from** x **in** $\{v_1,\ldots,v_n,v'_1,\ldots,v'_m\}$ **let** $y = u$ **accumulate** $e_3 \Downarrow v$.*

LEMMA 32. *If **from** x **in** e_1 **let** $y = e_2$ **accumulate** $e_3 \Downarrow u_n$ then $e_1 \Downarrow \{v_1,\ldots,v_n\}$ and $e_2 \Downarrow u_0$ and there exists $i_1,\ldots,i_n$ a permutation of $1,\ldots,n$ and there exist intermediate values $u_1,\ldots,u_{n-1}$ so that for all $k \in 1,\ldots,n$ we have that $e_3\{v_{i_k}/x\}\{u_{i_{k-1}}/y\} \Downarrow u_k$*

LEMMA 33. *If **from** x **in** e_1 **let** $y = e_2$ **accumulate** $e_3 \Downarrow$ wrong then at least one of the following conditions holds:*

(1) $e_1 \Downarrow$ wrong
(2) $e_1 \Downarrow v$ *but* $\neg\exists v_1,\ldots,v_n.(o = \{v_1,\ldots,v_n\})$
(3) $e_2 \Downarrow$ wrong
(4) $e_1 \Downarrow \{v_1,\ldots,v_n\}$ *and* $e_2 \Downarrow u_0$ *and there exists* $i_1,\ldots,i_n$ *a permutation of* $1,\ldots,n$ *and there exists a number of good steps* $j \in 0,\ldots,n-1$, *and there exist intermediate values* $u_1,\ldots,u_j$ *so that for all* $k \in 1,\ldots,j$ *we have that* $e_3\{v_{i_k}/x\}\{u_{i_{k-1}}/y\} \Downarrow u_k$ *and additionally* $e_3\{v_{i_{j+1}}/x\}\{u_j/y\} \Downarrow$ wrong.

LEMMA 34. *If e is operationally pure, $e \Downarrow r_1$ and $e \Downarrow r_2$ then $r_1 = r_2$.*

Proof Sketch By induction on the sum of the sizes of the derivations of $e \Downarrow r_1$ and $e \Downarrow r_2$. The only interesting case is when e is an accumulate expression, in which case we use Lemmas 32 and 33 to break the evaluation into two sequences of intermediate results corresponding to two permutations of $1,\ldots,n$. We bubble-sort one of the permutations to reach the other, while preserving the same result as the original sequence. The only operation used when bubble-sorting is swapping two adjacent elements, which preserves the result since the body of the accumulate is operationally commutative and associative.

LEMMA 35. *If e is operationally pure then e is pure.*

Proof Sketch Follows from Lemma 34 and Lemma 29.

LEMMA 36. *If e is algorithmically pure then e is operationally pure.*

Proof Sketch By induction on the structure of e, using Theorem 8 and Lemma 35.

References

[1] The Coq proof assistant, 2009. Version 8.2.

[2] *The Microsoft code name "M" Modeling Language Specification*. Microsoft Corporation, Nov. 2009.

[3] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Proceedings of ICFP*, 1993.

[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 and M. Hofmann. Dependent types. In *Advanced Topics in Types and Programming Languages*, chapter 2. MIT Press, 2005.

[7] C. Barrett, M. Deters, A. Oliveras, and A. Stump. Design and results of the 3rd Annual SMT Competition. *International Journal on Artificial Intelligence Tools*, 17(4):569–606, 2008.

[8] C. Barrett and C. Tinelli. CVC3. In *Proceedings of CAV*, 2007.

[9] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *Proceedings of CSF*, 2008.

[10] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-friendly general purpose language. In *Proceedings of ICFP*, 2003.

[11] K. Bhargavan, C. Fournet, and A. D. Gordon. Policy advisor for WSE 3.0. In *Web Service Security*, pages 324–330. Microsoft Press, 2006.

[12] 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.

[13] G. Bierman, E. Meijer, and M. Torgersen. Lost in translation: Formalizing proposed extensions to C♯. In *Proceedings of OOPSLA*, 2007.

[14] 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.

[15] 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.

[16] C. Calcagno, L. Cardelli, and A. D. Gordon. Deciding validity in a spatial logic for trees. *Journal of Functional Programming*, 15:543–572, 2005.

[17] G. Castagna. Patterns and types for querying XML documents. In *Proceedings of DBPL*, 2005.

[18] D. Crockford. The application/json media type for JavaScript Object Notation (JSON), July 2006. RFC 4627.

[19] F. Damm. Subtyping with union types, intersection types and recursive types. In *Proceedings of TACS*, 1994.

[20] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of TACAS*, 2008.

[21] L. M. de Moura and N. Bjørner. Generalized, efficient array decision procedures. In *FMCAD*, 2009.

[22] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[23] J. Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, Aug. 2007. CMU-CS-07-129.

[24] J. Dunfield and F. Pfenning. Tridirectional typechecking. In *Proceedings of POPL*, pages 281–292, 2004.

[25] B. Dutertre and L. de Moura. The YICES SMT solver. Available at `http://yices.csl.sri.com/tool-paper.pdf`, 2006.

[26] R. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP*, 2002.

[27] K. Fisher, Y. Mandelbaum, and D. Walker. The next 700 data description languages. In *Proceedings of POPL*, 2006.

[28] C. Flanagan. Hybrid type checking. In *Proceedings of POPL*, 2006.

[29] T. Freeman and F. Pfenning. Refinement types for ML. In *Proceedings of PLDI*, 1991.

[30] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *J. ACM*, 55(4), 2008.

[31] A. D. Gordon and A. Jeffrey. Typing one-to-one and one-to-many correspondences in security protocols. In *Proceedings of ISSS*, 2002.

[32] H. Hosoya, J. Vouillon, and B. Pierce. Regular expression types for XML. In *Proceedings of ICFP*, 2000.

[33] C. Jones. *Systematic software development using VDM*. Prentice-Hall Englewood Cliffs, NJ, 1986.

[34] 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.

[35] B. Lerner, M. Flower, D. Grossman, and C. Chambers. Searching for type-error messages. In *Proceedings of PLDI*, 2007.

[36] W. Lovas and F. Pfenning. A bidirectional refinement type system for LF. In *Proceedings of LFMTP*, 2007.

[37] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling objects, relations and XML in the .NET framework. In *Proceedings of SIGMOD*, 2007.

[38] J. Meng and L. C. Paulson. Translating higher-order problems to first-order clauses. *Journal of Automated Reasoning*, 40(1):35–60, 2008.

[39] B. Meyer. *Eiffel: the language*. Prentice Hall, 1992.

[40] B. Nordström and K. Petersson. Types and specifications. In *IFIP'83*, 1983.

[41] B. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[42] B. Pierce and D. Turner. Local type inference. In *Proceedings of POPL*, 1998.

[43] B. C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, 1991.

[44] V. Pratt. Five paradigm shifts in programming language design and their realization in Viron, a dataflow programming environment. In *Proceedings of POPL*, 1983.

[45] S. Ranise and C. Tinelli. *The SMT-LIB Standard: Version 1.2*, 2006.

[46] J. C. Reynolds. Design of the programming language Forsythe. In *Algol-Like Languages*, chapter 8. Birkhäser, 1996.

[47] P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *Proceedings of PLDI*, 2008.

[48] J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998.

[49] V. Saraswat, N. Nystrom, J. Palsberg, and C. Grothoff. Constrained types for object-oriented languages. In *Proceedings of OOPSLA*, 2008.

[50] J. Siméon and P. Wadler. The essence of XML. In *Proceedings of POPL*, 2003.

[51] M. Sozeau. Subset coercions in Coq. In *Proceedings of TYPES*, 2006.

[52] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *Proceedings of POPL*, 2008.

[53] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of POPL*, 1999.