Semantic Subtyping with an SMT Solver

Gavin M. Bierman
Andrew D. Gordon
Microsoft Research

Cătălin Hriţcu
Saarland University

David Langworthy
Microsoft Corporation

Abstract

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

Categories and Subject Descriptors F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Denotational semantics; Operational semantics; Program analysis

General Terms Languages, Theory, Verification

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 [1], whose types correspond to relational schemas, and whose expressions compile to SQL queries. Refinement types are used to express SQL table constraints within a type system, and type-tests are useful for processing relational data, for example, by discriminating dynamically between different forms of union types. Still, although useful and extremely expressive, the combination of type-test and refinement is hard to type-check using conventional syntax-driven subtyping rules. The preliminary implementation of M uses such subtyping rules and has difficulty with certain sound idioms (such as uses of singleton and union types). Hence, type safety is enforced by dynamic checks, or not at all.

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

1.1 Programming with Type-Test and Refinement

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

Two central primitives of Dminor are the following:

- A *refinement type*, $(x : T \textbf{ where } e)$, 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 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 [10], 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) [11], 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:

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

Intuitively, this code is type-safe because it checks the kind field before accessing the var field, which is only present for assignment statements. More precisely, to type-check the then-branch $y.\mathsf{var}$ to type Text, we have $y : \mathsf{Statement}$ (i.e. a union type encoded using refinements and type-test), know that $y.\mathsf{kind} ==$ `"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 [5], Yices [17], and Z3 [13]. There are common input formats such as Simplify's [15] unsorted S-expression syntax and the SMT-LIB standard [36] for sorted logic. Hence, first-order logic with standard theories is emerging as a computing platform. Software written to generate problems in a standard format can rely on a wide range of back-end solvers, which get better over time due in part to healthy competition,[1] and which may even be run in parallel when sufficient cores are available. There are limitations, of course, as first-order validity is undecidable even without any theories, so solvers may fail to terminate within a reasonable time, but recent progress has been remarkable.

1.3 Semantic Subtyping with an SMT Solver

The central idea in this paper is a type-checking algorithm for Dminor that 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* [2, 22], as opposed to the more common alternative, *syntactic subtyping*, which is to define syntax-driven rules for checking subtyping [34].

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 [20, 26], F7 [6], and Dsolve [38], rely on various SMT solvers. However, these systems all rely on syntactic subtyping, with the solver being used as a subroutine to check constraints during subtyping.

To the best of our knowledge, our proposal to implement semantic subtyping by calling an external SMT solver is new. Semantic subtyping nicely exploits the solver's 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 [16]. Moreover, we exploit the theories of extensional arrays [14], integer arithmetic, and algebraic datatypes.

1.4 Contributions of the Paper

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

[1] Most important is the SMT-COMP [4] competition held each year in conjunction with CAV and in which more than a dozen SMT solvers contend.

(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 (integers, datatypes and extensional arrays).

(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.

1.5 Structure of the Paper

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

§3 presents a logical semantics of pure expressions (those without side-effects) and Dminor types; each pure expression e is interpreted as a term $\mathbf{R}[\![e]\!]$ and each type T is interpreted as a first-order logic formula $\mathbf{F}[\![T]\!](t)$. 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. We describe how to show purity of expressions using a syntactic termination restriction together with a confluence check that relies on the logical semantics. Theorem 2 shows that our algorithmic purity check is indeed a sufficient condition for purity.

§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 3 concerns logical soundness of type assignment; if e is assigned type T then formula $\mathbf{F}[\![T]\!](\mathbf{R}[\![e]\!])$ holds. Progress and preservation results (Theorems 4 and 5) relate type assignment to the operational semantics, entailing that well-typed expressions cannot go wrong.

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

§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. We survey related work in §8, before concluding in §9.

A technical report [8] contains additional details and proofs.

2. Syntax and Operational Semantics

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

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

```
NullableInt ≜ Integer | [null]

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

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

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

2.1 Expressions and Types

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

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

Scalar Types, Constants, and Operators:

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

A *value* may be a *simple value* (an integer, string, boolean, or **null**), a *collection* (a finite multiset of values), or an *entity* (a finite set of fields, each consisting of a value with a distinct label).

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
$\{\ell : T\}$	(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 $\{\ell : T\}$ 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**.

Syntax of Expressions:

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

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

The accumulate primitive can encode all the usual operations on collections: counting the number of elements of or occurrences of a certain element, checking membership, removing duplicates and elements, multiset union and difference, as well as LINQ [30] queries and comprehensions in the style of the nested relational calculus [9]. The precise definitions are in the technical report.

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 $fv(e) \subseteq \{x_1, \ldots, x_n\}$ and $fv(U) \subseteq \{x_1, \ldots, x_n\}$.

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.

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 $==$ operator implements syntactic equality, which for collections and entities is up to reordering of elements. Apart from $==$, the other operators only act on scalar values.

Reduction Contexts:

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

Reduction Rules for Standard Constructs:

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

Reduction Rules for Type-Test:

$v\ \textbf{in}\ \mathsf{Any} \to \textbf{true}$
$$v\ \textbf{in}\ G \to \begin{cases} \textbf{true} & \text{if } v \in K(G) \\ \textbf{false} & \text{otherwise} \end{cases}$$
$$v\ \textbf{in}\ \{\ell_j : T_j\} \to \begin{cases} v_j\ \textbf{in}\ T_j & \text{if } v = \{\ell_i \Rightarrow v_i\ ^{i\in 1..n}\} \wedge j \in 1..n \\ \textbf{false} & \text{otherwise} \end{cases}$$
$$v\ \textbf{in}\ T* \to \begin{cases} v_1\ \textbf{in}\ T\ \&\&\dots\&\&\ v_n\ \textbf{in}\ T & \text{if } v = \{v_1,\dots,v_n\} \\ \textbf{false} & \text{otherwise} \end{cases}$$
$v\ \textbf{in}\ (x : T\ \textbf{where}\ e) \to v\ \textbf{in}\ T\ \&\&\ e\{v/x\}$

The reduction rules for type-test expressions, e **in** U, first reduce e to a value v and then proceed by case analysis on the structure of the type U. In case U is a refinement type $(x : T\ \textbf{where}\ e)$ then v is a value of U if and only if v is a value of type T and $e\{v/x\}$ reduces to the value **true**. Nondeterminism arises from the reduction rule for accumulate expressions. Since collections are unordered, the rule applies for any permutation of $\{v_1,\dots,v_n\}$. For example, consider the expression $\mathsf{pick}\ v_1\ v_2 \triangleq \textbf{from}\ x\ \textbf{in}\ \{v_1,v_2\}\ \textbf{let}\ y = \textbf{null}\ \textbf{accumulate}\ x$; we have both $\mathsf{pick}\ \textbf{true}\ \textbf{false} \to^* \textbf{true}$ and $\mathsf{pick}\ \textbf{true}\ \textbf{false} \to^* \textbf{false}$.

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

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

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

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

2.3 Pure Expressions and Refinement Types

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

To address this problem, we define the set of *pure* expressions, the ones that may be used as refinements. The details, below, are a little technical, but the gist is that pure expressions must be terminating, have a unique result (which may be **Error**), and must only call functions whose bodies are pure. The typing rule (Type Refine) in §4 requires that for $(x : T\ \textbf{where}\ e)$ to be well-formed, the expression e must be pure and of type $\mathsf{Logical}$ (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 §3.1.

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 $e \to e_1 \to e_2 \to \dots$. Let a closed expression e be *pure* if and only if (1) e is terminating, (2) there exists a unique result r such that $e \Downarrow r$, (3) for every subexpression $f(e_1,\dots,e_n)$ of e, the function f is labeled-pure, and (4) all subexpressions of e are pure. Let an arbitrary expression e be *pure* if and only if $e\sigma$ is pure for all closing substitutions σ that assign a value to each free variable in e. Finally, we require that the body of every labeled-pure function is a pure expression.

2.4 Derived Types

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

Encoding of Empty and Singleton Types:

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

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

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

Encoding of Union, Intersection, and Negation Types:

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

A value of the *union type*, $T \mid U$, is a value of T or of U. A value of the *intersection type*, $T\ \&\ U$, is a value of both T and U. A value of the *negation type*, $!T$, is a value that is not a value of T.

Next, we define the types of simple values, collections, and entities. We rely on the primitive types $\mathsf{Integer}$, Text, and $\mathsf{Logical}$, 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 $\mathsf{General} \mid \mathsf{Collection}$.

Encoding of Supertypes:

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

The primitive type of entities is unary: the type $\{\ell : T\}$ is the set of entities with a field ℓ whose value belongs to T (and possibly other fields). As in Forsythe [37], we derive *multiple-field entity types* as an intersection type. One advantage of this approach is that it immediately entails width subtyping for entities.

Encoding of Multiple-Field Entity Types:

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

We can also derive *closed entity types*, which only contain entities with a fixed set of labels and therefore do allow width subtyping. To do so we constrain the multiple-field entity types above to additionally satisfy an eta law.

Encoding of Closed Entity Types:

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

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

Encoding of Pair Types and Dependent Pair Types:

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

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

Encoding of Sum and Variant Types:

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

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 List Types

$\mathsf{List}_T \triangleq (T * (x : \mathsf{Any}\ \textbf{where}\ f_{\mathsf{List}_T}(x))) + [\textbf{null}]$
where $f_{\mathsf{List}_T}(x)$ is a new labeled pure function defined by
$f_{\mathsf{List}_T}(x : \mathsf{Any}) : \mathsf{Logical}\ \{$
$x\ \textbf{in}\ ((T * (x : \mathsf{Any}\ \textbf{where}\ f_{\mathsf{List}_T}(x))) + [\textbf{null}])\}$

Lists can be used to encode XML and JSON. Hence, Dminor can be viewed as a richly typed functional notation for manipulating data in XML format. 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 final field holding a list of all the items in the body of the element.

Next, we show how to derive entity types for the common situation where the type of one field depends on the value of another. A *dependent intersection type* $(s : T \,\&\, U)$ [27] 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, but we can rely on $s : T$ when checking well-formedness of U.

Encoding of Dependent Intersection Types:

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

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

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:

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

3. Logical Semantics

In this section we give a set-theoretic semantics for types and pure expressions. Pure expressions are interpreted as first-order terms, while types are interpreted as formulas in many-sorted first-order logic (FOL). These formulas are interpreted in a fixed model, which we formalize in Coq. We represent a Dminor subtyping problem as a logical implication, supply our SMT solver with a set of axioms that are true in our intended model, and ask the solver to prove the validity of the implication. We use Coq to state our model and to 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 the technical report [8]). We define a predicate Proper that is true for results that are not **Error**, and a function $\mathsf{out\_V}$ that returns the value inside if the result passed as argument is proper and **null** otherwise.

Model: Proper Results:

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

Our semantics uses many-sorted first-order logic (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 valid in our intended model.

Our semantics consists of three translations:

- For any pure expression e, we have the FOL term $\mathbf{R}[\![e]\!] : \mathsf{Result}$.
- For any Dminor type T and FOL term $t : \mathsf{Value}$, we have the FOL formula $\mathbf{F}[\![T]\!](t)$, which is valid in the intended model if and only if the value denoted by t is a member of the type T.
- For type T and FOL term $t : \mathsf{Value}$, we have the formula $\mathbf{W}[\![T]\!](t)$, which holds if and only if a type-test goes wrong when showing that the value denoted by t is a member of T. For instance, we have $\models \mathbf{W}[\![(x : \mathsf{Any}\ \textbf{where}\ \mathsf{stuck})]\!](\textbf{null}) \Leftrightarrow \textbf{true}$, but $\models \mathbf{W}[\![\mathsf{Any}]\!](\textbf{null}) \Leftrightarrow \textbf{false}$.

These three (mutually recursive) translations are defined below. We rely on notations for let-binding within terms ($\textbf{let}\ x = t\ \textbf{in}\ t'$), and terms conditional on formulas ($\textbf{if}\ F\ \textbf{then}\ t\ \textbf{else}\ t'$). These notations are supported directly by most SMT solvers. 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 $\mathbf{R}[\![e\ \textbf{in}\ T]\!]$ 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:

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

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

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

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:

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

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

The notation $\oplus : T_1,\dots,T_n \to T$ defines type signatures for each primitive operator $\oplus$. We omit the details, as well as the definitions of the functions $\mathsf{O}_\oplus$ interpreting each primitive operator $\oplus$.

The semantics of an entity type $\{\ell : T\}$ is the set of all values (denoted by t) that are entities ($\mathsf{is\_E}(t)$) having the field ℓ ($\mathsf{v\_has\_field}(\ell,t)$), which contains a value of type T ($\mathbf{F}\llbracket T\rrbracket(\mathsf{v\_dot}(t,\ell))$).

Model: Functions and Predicates on Entities:

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

Semantics: Entity Types and Expressions:

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

The semantics of **from** x **in** e_1 **let** $y = e_2$ **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 **fun** $x\ y \to \mathbf{R}\llbracket e_3\rrbracket$ for such a function. There are several standard techniques for representing lambda-abstractions in first-order logic [31]. 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 ($\forall x.\mathsf{v\_mem}(x,t) \Rightarrow \mathbf{F}\llbracket T\rrbracket(x)$).

Semantics: Collection Types and Expressions:

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

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

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 $f(x_1 : T_1,\dots,x_n : T_n) : U\{e\}$, the model of the symbol f is the total function $\underline{f} \in \mathsf{Value}^n \to \mathsf{Result}$ such that $\underline{f}(v_1,\dots,v_n)$ is the result r such that $e\{v_1/x_1\}\dots\{v_n/x_1\} \Downarrow r$. (We know that there is a unique r such that $e\{v_1/x_1\}\dots\{v_n/x_1\} \Downarrow r$ because e is pure.) Hence, the following holds by definition:

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

Semantics: Function Application:

$$\mathbf{R}\llbracket f(e_1,\dots,e_n)\rrbracket =$$
$$\mathbf{Bind}\ x_1 \Leftarrow \mathbf{R}\llbracket e_1\rrbracket\ \mathbf{in}\ \dots \mathbf{Bind}\ x_n \Leftarrow \mathbf{R}\llbracket e_n\rrbracket\ \mathbf{in}\ \underline{f}(x_1,\dots,x_n)$$

The operational semantics preserves logical meaning:

PROPOSITION 1. *For all closed pure expressions e and e', if $e \to e'$ then $\models \mathbf{R}\llbracket e\rrbracket = \mathbf{R}\llbracket e'\rrbracket$.*

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

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

3.1 Algorithmic Purity Check

The purity property defined in §2.3 is undecidable. We use a syntactic termination condition on the applied functions together with a restriction on the accumulate expressions to make the purity checks tractable.

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

(1) if e is a function application $f(e_1,\dots,e_n)$ then f is labeled-pure, and only calls f (directly or indirectly) on structurally smaller arguments;

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

$$\models \mathbf{R}\llbracket \mathbf{let}\ y = e_3\{x_1/x\}\{y_1/y\}\ \mathbf{in}\ e_3\{x_2/x\}\rrbracket =$$
$$\mathbf{R}\llbracket \mathbf{let}\ y = e_3\{x_2/x\}\{y_1/y\}\ \mathbf{in}\ e_3\{x_1/x\}\rrbracket$$

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

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

Condition (1) enforces termination of algorithmically pure expressions: only labeled-pure functions can be called and if these functions are recursive then recursive calls can only be on syntactically smaller arguments. Condition (2) only allows accumulates in an algorithmically pure expression if the order in which the elements are processed is irrelevant for the final result. In general we call a (mathematical) function $f : X \times Y \to Y$ *order-irrelevant* if $f(x_1, f(x_2, y)) = f(x_2, f(x_1, y))$ for all x_1, x_2 and y. Enforcing that the semantics of the body of accumulate expressions is an order-irrelevant function is a sufficient condition for the uniqueness of evaluation results. We phrase this condition in terms of the logical semantics and check it using the SMT solver. Order-irrelevance is less restrictive than conditions found in the literature such as associativity and commutativity [28]. If f is associative and commutative then f is also order-irrelevant, but the converse fails in general. If f is order-irrelevant its two arguments need not even have the same type.

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

The logical semantics is defined only on pure expressions. Given the logical semantics, we obtain algorithmic purity, a sufficient condition for purity. In the remainder of the paper we rely only on algorithmic purity.

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

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

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

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

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

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

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

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

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

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

Rule of Semantic Subtyping:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The rule (Exp Cond) records the appropriate test expression in the environment, when typing the branches. The actual value of a type $\mathsf{Ok}(e)$ is arbitrary, the point is simply to record that condition e holds [23], provided it is pure. When e is not pure, $\mathsf{Ok}(e)$ is equivalent to Any.

Typed Singleton Types and Ok Types:

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

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

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

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

(Exp Singleton)
$$\frac{E \vdash e : T}{E \vdash e : [e : T]}$$

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

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

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

THEOREM 3 (Logical Soundness).

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

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

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

LEMMA 2 (Singular Subtyping).
Suppose $E \vdash e : T$ and $E \vdash T'$ and $x \notin dom(E)$.

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

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

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

LEMMA 3 (Transitivity of Singular Subtyping).
If $E \vdash [e : T] <: T'$ and $E \vdash [e : T'] <: T''$ then $E \vdash [e : T] <: T''$.

We have proved standard derived judgment, weakening, bound weakening, and substitution lemmas for the type system, which are used in the proofs of the progress and preservation theorems.

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

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

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 3, its logical semantics is a proper value, not **Error**.

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

LEMMA 4. *If e alg. pure and $E \vdash e : T$ then $\models \boldsymbol{F}[\![E]\!] \implies (\boldsymbol{Bind}\ x \Leftarrow \boldsymbol{R}[\![e]\!]\ \boldsymbol{in}\ t) = (\boldsymbol{let}\ x = \mathsf{out\_V}(\boldsymbol{R}[\![e]\!])\ \boldsymbol{in}\ t)$.*

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

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

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

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

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

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

We omit the definition of the function $\mathsf{v\_accumulate}$, which is a variant of $\mathsf{res\_accumulate}$ that works with values rather than results. See the technical report for the full details [8].

THEOREM 6 (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:*
$\models \boldsymbol{F}[\![E]\!] \implies (\boldsymbol{R}[\![e]\!] = \boldsymbol{Return}(\boldsymbol{V}[\![e]\!]))$.

Proof: The proof is by simultaneous induction on the derivations of $E \vdash T$ and $E \vdash e : T$, with appeal to Theorem 3 and Lemma 4. □

5.2 Bidirectional Typing Rules

The Dminor type system is implemented as a *bidirectional* type system [35]. 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
$E \rhd \diamond$	environment E is alg. well-formed
$E \rhd T$	in E, type T is alg. well-formed
$E \rhd S <: T$	in E, type S is alg. a subtype of type T

Both subtyping and well-formedness rely on type-checking, so we need to distinguish versions of these judgments that use the declarative typing rules from versions that use the bidirectional typing rules (and in the case of subtyping, the optimized semantics). For brevity we omit the definitions, which may be found in the technical report [8].

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

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

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

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

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

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

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

(Synth Ascribe)
$$\frac{E\vdash e\leftarrow T}{E\vdash (e\colon T)\rightarrow T}$$

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

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

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

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

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

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

The rules (Synth Var), and (Synth Const) yield singleton types for all variables and constants, where the function *typeof* returns the type of a given constant. Rule (Synth Entity) uses intersection types to encode record types.

The (Synth Cond) rule synthesizes a conditional type, which is the union of the two types synthesized for the branches along with the test expression (if it is pure) to allow more precise typing.

Encoding of Conditional Types:

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

The (Synth Ascribe) rule allows the user to provide hints to the type-checker in the form of type annotations $(e : T)$. Such type annotations have no operational significance (in the small-step semantics $e : T \rightarrow e$), and are necessary in case the type-checker cannot infer the loop invariants of accumulate expressions.

In several of the type synthesis rules we need to inspect components of intermediate types. In simple type systems this is straightforward as one can rely on the syntactic structure of types, but for rich type systems such as the one of Dminor this is not possible. In other dependently-typed languages, either the programmer is required to insert casts to force the type into the appropriate syntactic shape [43], or types are first executed until a normal form is reached [3]. 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 [26]. 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):

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

We can define two partial functions to extract field and item types from normalized entity and collection types. These are written $D.\ell \rightsquigarrow U$ and $D.\mathsf{Items} \rightsquigarrow U$, respectively. For example $(\{\ell\colon \mathsf{Integer}\}\mid\{\ell\colon \mathsf{Logical}\}).\ell \rightsquigarrow \mathsf{Integer}\mid\mathsf{Logical}$ and $((\mathsf{Text}*\ \&\ \mathsf{Logical})\mid \mathsf{Integer}*).\mathsf{Items}\rightsquigarrow \mathsf{Text}\mid\mathsf{Integer}$. Note that both these functions are partial, e.g. $(\{\ell\colon \mathsf{Integer}\}\mid\{\ell'\colon \mathsf{Logical}\}).\ell \not\rightsquigarrow$. The simple definitions of these functions are in the technical report [8].

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

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

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

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

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

The (Swap) rule tests for singular subsumption and applies if the expression to be type-checked is not a conditional, let-expression or a field selection. Typically (e.g. SAGE [26]), the type checking relation for a bidirectional type system consists of a single rule of the form:

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

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

LEMMA 5 (Synthesis Checkable). *If* $E\vdash e\rightarrow T$ *then* $E\vdash e\leftarrow T$.

THEOREM 7 (Soundness of Algorithmic Type System).

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

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[2] 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) $E \vdash T$ and $E \vdash T'$

(2) $(y\sigma$ **in** $U\sigma) \rightarrow^*$ **true**, for all $(y : U) \in E$;

(3) $(x\sigma$ **in** $(T\ \&!T')\sigma) \rightarrow^*$ **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 6. *If the three checks above succeed then* $E \not\vdash T <: T'$.

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.

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. This is useful, since one can make mistakes when writing types containing complicated constraints. Moreover, if the SMT solver cannot prove that a type is empty we again obtain an assignment σ, which we can validate as in §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.

[2] 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.

LEMMA 7. *If the three checks in §6.1 succeed for* $T' = \mathsf{Empty}$ *then* $\varnothing \vdash x\sigma : T\sigma$ *and* $\varnothing \vdash y\sigma : U\sigma$ *for all* $(y : U) \in E$.

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 (not present in the M language) which tries to find an inhabitant of T. If successful the expression returns such a value, but otherwise it returns **null**. (We can always choose T so that **null** is not a member, so that returning **null** unambiguously signals that no member of T was found.)

Operational Semantics for Finding Elements of Types:

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

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

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

Typing rules for elementof:

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

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

LEMMA 8. *If* ***elementof*** $T \rightarrow v$ *and* $\varnothing \vdash T$ *then* $\varnothing \vdash v : T \mid [\mathbf{null}]$.

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

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

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

7. Implementation

Our prototype Dminor implementation is approximately 2700 lines of F$^\sharp$ code, excluding the lexer and parser. Our type-checker implements the algorithmic purity check from §3.1, the optimized logical semantics from §5.1, and the bidirectional typing rules from §5.2. We use Z3 [13] 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 [36], which uses the theories of integers, datatypes and extensional arrays. The formal definition of our intended model of Dminor is just over 4000 lines of Coq.

We have tested our type-checker on a test suite consisting of about 130 files, some type-correct and some type-incorrect, some hand-crafted by us and some transliterated from the M preliminary release. Even without serious optimization the type-checker is fast. Checking each of the 130 files in our test suite on a typical laptop takes from under 1 second (for just startup and parsing) to around 3 seconds (for type-checking 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,[3] 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 [1] that displays precise counterexamples to typing, flags empty types and otherwise displays one element of each type defined in the code. Moreover, our interpreter for Dminor supports **elementof** for dynamically generating instances of types (§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. 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 [33], set comprehensions in VDM [25], and predicate subtypes in PVS [39]. In PVS, constraints found when checking predicate subtypes become proof obligations to be proved interactively. More recently, Sozeau [41] extends Coq with subset types; as in PVS the proofs of subset type membership have to be constructed using tactics.

Freeman and Pfenning [21] extended ML with a form of refinement type, and Xi and Pfenning [43] 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 [29] 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 [6, 38] 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 [26]. 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 [20] 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 [32] and DrScheme [18], and static typing. Compared to purely static typing this can reduce the number of false alarms generated by type-checking.

In spite of early work on semantic subtyping by Aiken and Wimmers [2] and Damm [12], 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 [34]. Unfortunately, deriving an algorithm from such a set of rules can be difficult, especially for advanced features such as intersection and union types [16].

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

X10 [40] is an object-oriented language that supports refinement types. A class `C` can be refined with a constraint `c` on the immutable state of `C`, resulting in a type written `C(:c)`. The base language supports only simple equality constraints but further constraints can be added and multiple constraint solvers can be integrated into the compiler. In comparison with Dminor, X10 uses a mixture of semantic and syntactic subtyping, while its constraint language [40, §2.11] does not support type-test expressions.

The introduction of XML and XML query languages led to renewed (practical) interest in semantic subtyping. In the context of XML documents, there is a natural generalization of DTDs where the structures in XML documents can be described using regular expression 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 [24] and their language, XDuce. Frisch, Castagna, and Benzaken [22] extended semantic subtyping to function types and propositional types, with type-test, but not refinement types, resulting in the language CDuce [7].

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 than they are in our setting, since we additionally have to deal with general refinement types.

Typed Scheme [42] 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 [19] 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.

9. 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 (analogous 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.

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 an SMT solver 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 $F^\sharp$ 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.

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 [20] 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 should apply 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. James Margetson helped with F# programming issues. Paul Anderson, Ioannis Baltopoulos, Johannes Borgström, Nate Foster, Tim Harris, and Thorsten Tarrach commented on a draft. Discussions with Martín Abadi, Cliff Jones, and Benjamin Pierce were useful, as were the comments of the anonymous reviewers. Cătălin Hriţcu is supported by a fellowship from Microsoft Research and the IMPRS.

References

[1] *The Microsoft code name "M" Modeling Language Specification Version 0.5*. Microsoft Corporation, Oct. 2009. Preliminary implementation available as part of the *SQL Server Modeling CTP (November 2009)*.

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

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

[4] 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.

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

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

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

[8] G. M. Bierman, A. D. Gordon, C. Hriţcu, and D. Langworthy. Semantic subtyping with an SMT solver. Technical Report MSR–TR–2010–99, Microsoft Research, July 2010.

[9] 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.

[10] R. M. Burstall, D. B. MacQueen, and D. Sannella. HOPE: An experimental applicative language. In *LISP Conference*, pages 136–143, 1980.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[26] 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.

[27] A. Kopylov. Dependent intersection: A new way of defining records in type theory. In *LICS*, pages 86–95. IEEE Computer Society, 2003.

[28] K. R. M. Leino and R. Monahan. Reasoning about comprehensions with first-order SMT solvers. In *Proceedings of SAC*, 2009.

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

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

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

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

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

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

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

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

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

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

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

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

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

[42] S. Tobin-Hochstadt and M. Felleisen. Logical types for Scheme. In *Proceedings of ICFP*, 2010.

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