A Polymorphic Intermediate Verification Language: Design and Logical Encoding

K. Rustan M. Leino [0] and Philipp Rümmer [1]

[0] Microsoft Research, Redmond, `leino@microsoft.com`
[1] Oxford University Computing Laboratory, `philr@comlab.ox.ac.uk`

Abstract. Intermediate languages are a paradigm to separate concerns in software verification systems when bridging the gap between programming languages and the logics understood by theorem provers. While such intermediate languages traditionally only offer rather simple type systems, this paper argues that it is both advantageous and feasible to integrate richer type systems with features like (higher-ranked) polymorphism and quantification over types. As a concrete solution, the paper presents the type system of Boogie 2, an intermediate verification language that is used in several program verifiers. The paper gives two encodings of types and formulae in simply typed logic such that SMT solvers and other theorem provers can be used to discharge verification conditions.

0 Introduction

Building a program verifier is a complex task that requires understanding of many domains. Designing its foundation draws from domains like semantics, specifications, and decision procedures, and constructing its implementation involves knowledge of compilers and software engineering. The task can be made manageable by breaking it into smaller pieces, each of which is simpler to understand. A successful practice (*e.g.*, [10, 3, 4]) is to make use of an *intermediate verification language* [15, 0, 9].

The intermediate verification language serves as a thinking tool in the design of the verifier front end for each particular source language. As such, it must provide a level of abstraction that is high enough to give leverage to the front end. At the same time, there is a risk that the general translations of higher-leverage features become too cumbersome to sustain good decision procedure performance. Some higher-leverage features, like a fancy type system, provide safety to the front end by restricting what intermediate programs are admissible. At the same time, there is a risk that such restrictions lead to cumbersome encodings in the front end, especially compared to the encodings that are possible by directly using the more coarse-grained type system of a decision procedure.

In this paper, we introduce the type system of the intermediate verification language *Boogie 2* developed by the authors, the successor of BoogiePL [7, 0]. Unlike its untyped predecessor, whose type annotations were mainly used for some consistency checks, Boogie 2 features an actual type system. Going beyond the Hindley-Milner style types in the intermediate verification language Why [9], Boogie 2 features polymorphic maps, higher-rank polymorphism, and impredicativity, which are useful in modeling the semantics of a type-safe object store (as in Spec# or Java).

```
class Person { int age; bool isMarried; }
```

Fig. 0. An example code snippet from a source program.

type Ref;	**const unique** age: $Field$ **int**;
type $Field\ \alpha$;	**const unique** $isMarried$: $Field$ **bool**;
type $HeapType = \langle\alpha\rangle[Ref, Field\ \alpha]\ \alpha$;	**var** $Heap$: $HeapType$;
function $IsWellFormed(HeapType)$ **returns** (**bool**);	
const unique $snapshot$: $Field\ HeapType$;	

Fig. 1. An example of how object-oriented program features, like those in Fig. 0, can be modeled in Boogie (the language features used are introduced in detail in Section 1). Ref is a type and $Field$ is a unary type constructor. Type synonym $HeapType$ is defined as the polymorphic map type that represents the heap. $IsWellFormed$ demonstrates that functions can take polymorphic maps as arguments. For any r of type Ref, $Heap[r, snapshot]$ has type $HeapType$, illustrating that polymorphic maps can be arbitrarily nested (an instance of impredicativity). The modifier **unique** is used to say that the constant declared has a different value than all other **unique** constants, which for the 3 constants here also follows from the fact that their types are different.

In addition to introducing the polymorphic features of Boogie, we describe our translation of Boogie's polymorphic logic into simply typed logic, which is used by many satisfiability modulo theories (SMT) solvers that support the SMT-LIB format [1]. In fact, we give two different translations into simply typed logic, and we present performance figures from substantial benchmarks that compare these. The benchmarks come from the Spec# program verifier [0], the VCC [4] and HAVOC [3] verifiers for C, and Dafny [13], all of which build on Boogie. All of the benchmarks make extensive use of so-called triggers required for e-matching [8], and our experiments give evidence to that the triggers are properly maintained by our translations.

The contributions of our work are: (i) An impredicative type system for an intermediate verification language, featuring full higher-ranked polymorphism, (ii) two translations of the verification language, and especially its polymorphic maps, into simply typed logic suitable for SMT solvers, (iii) experimental data comparing the performance of the two translations with each other and with an (unsound) translation ignoring types.

1 Boogie 2 Types and Expressions

A Boogie program consists of a set of mathematical and imperative declarations that define a set of execution sequences. The Boogie program is correct if none of those execution sequences contains an error state [12]. Programs can be written by hand, but most Boogie programs are machine generated by various program verifiers to encode the semantics of given source programs. For example, the source-language declaration in Fig. 0 can be modeled in Boogie as shown in Fig. 1, where the object store is represented explicitly by a variable $Heap$ whose type is a map from object references and field names to values (we explain this example in more detail later).

For the purposes of this paper, one can think of the imperative features of Boogie as convenient syntactic shorthands for writing Boogie expressions. Hence, we focus on

Boogie's expressions and their types. For further details of the language, we refer to the Boogie 2 language reference manual [12].

1.0 Type Declarations

The built-in types of Boogie are booleans (**bool**), mathematical integers (**int**), and bit-vector types of every size (**bv0**, **bv1**, **bv2**, . . .). In addition, there are map types, which we describe below, and user-defined type constructors. A program can also declare parameterized *type synonyms*, which are essentially like macros, thus providing syntactic convenience but not adding to the expressiveness of the type system. A type denotes a nonempty set of individuals, and the sets denoted by different types are disjoint. Each different parameterization of a type constructor yields a distinct type, each denoting an uninterpreted set of individuals. For example, the type declarations in Fig. 1 introduce a nullary type constructor Ref and a unary type constructor $Field$. The sets of individuals denoted by Ref, $Field$ **int**, and $Field\ Ref$ are all disjoint.

1.1 Expressions

Boogie expressions include variables and constants, function applications, logical, arithmetic, and relational operators, as well as logical quantifiers, type coercions, and map operations. All expressions are total: every well-typed expression yields some appropriately typed value that is a function of its subexpressions. For the most part, typing of expressions is obvious and straightforward. Let us describe the more salient features.

Polymorphic Functions, Quantifications over Types. Functions can be polymorphic, that is, they can take type parameters. Analogously, the bound variables in universal and existential quantifiers can range over both individuals (of specified types) and types. Polymorphism is useful because it allows a user to provide an axiomatization of, say, pairs that is independent of the pair element types, while maintaining the type guarantee that different types of pairs are not mixed up.

For example, Fig. 2 declares a binary type constructor $Pair$, along with a function $Cons$ for constructing a pair and a function $Left$ that extracts the left element of a pair. Type parameters and bound type variables are introduced inside angle brackets, like in C# or Java. A function declaration in Boogie only defines the signature of the function; properties of functions can be defined by axioms. The figure includes an axiom that defines the relationship between $Cons$ and $Left$. Note that the quantification is over any element types α and β and any elements a and b of those types. Hence, the axiom applies generically to pairs with any element types.

The meaning of a function depends on its type-parameter instantiation. That is, a polymorphic function f is really a family of functions $\bar{f}$, one for each possible instantiation (*e.g.*, $f_{\mathbf{int}}$, f_{Ref}).

Type Coercions. Boogie infers instantiations for type parameters of function applications. Usually, they can be inferred from the types of the function's arguments, but sometimes it is also necessary to consider the context of the function application. In

```
type Pair α β;
function Cons⟨α, β⟩(α, β) returns (Pair α β);
function Left⟨α, β⟩(Pair α β) returns (α);
axiom ( ∀ ⟨α, β⟩ a : α, b : β ● Left(Cons(a, b)) = a );

type Sequence α;
function Length⟨α⟩(Sequence α) returns (int);
function EmptySequence⟨α⟩() returns (Sequence α);
axiom ( ∀ ⟨α⟩ ● Length(EmptySequence() : Sequence α) = 0 );
```

Fig. 2. Examples of polymorphic functions and quantifications over types in Boogie. In the last line, the quantifier ranges only over types, not over any individuals, and the type coercion makes the application $EmptySequence()$ well-typed.

particular, if a type parameter is used among the domain types in the function's signature, then its instantiation in a function application can be inferred from the arguments. But in the case that a type parameter is used only in the return type, then type inference needs to consult the context. Type parameters that are not used in either the domain types or the result type are not allowed.

For example, Fig. 2 declares a function that gives the length of a generic sequence. Function $EmptySequence$ returns a zero-length sequence of any type. Type parameter $α$ is used only in the return type of $EmptySequence$, which is common and useful for this and similar functions. Hence, to infer the type parameter in an application $EmptySequence()$, the context surrounding the application must be used.

An error is reported if an instantiation for type parameters cannot be determined uniquely. To deal with such cases, the language offers a type coercion expression $e : t$, which has type t, provided t is a possible typing for expression e. For example, the expression $Length(EmptySequence())$ is ill-formed because of the ambiguous type-parameter instantiation; but with the type coercion in Fig. 2, the ambiguity is resolved.

Because the meaning of a polymorphic function is really that of a family of functions, note that $EmptySequence_{\textbf{int}}()$ has a different value than $EmptySequence_{Ref}()$.

Maps. In addition to functions, Boogie offers *maps*. Like functions, maps have a list of domain types and a result type and can be polymorphic. The difference is that maps are themselves expressions (they are "first class"), unlike functions, which can appear in an expression only when applied to arguments. This means that program variables can hold maps (like $Heap$ in Fig. 1).

Though they may have the appearance of higher-order values, maps are but first-order individuals, and to "apply" them to arguments, one applies Boogie's built-in map-select operator, written with square brackets (to be suggestive of retrieving an element at a given index of an array) [18]. For example, if m is a map of type $[\textbf{int}, \textbf{bool}]Ref$, that is, a map type with domain types $\textbf{int}$ and $\textbf{bool}$ and result type Ref, then the expression $m[5, \textbf{false}]$ denotes a value of type Ref. Due to maps, Boogie can in many situations be used like a higher-order language (where functions can be passed around as values), but still allows the use of efficient first-order reasoners.

If m is an expression denoting a map, i is a list of expressions whose types correspond to the domain types of m, and x is an expression of the result type of m, then the map-update expression $m[i := x]$ denotes the map that is like m, except that it maps i to x [18]. Using common notation for arrays, the imperative part of Boogie allows the assignment statement $m[i] := x$; as a shorthand for $m := m[i := x]$;.

Boogie does not promise *extensionality* of maps, that is, the property that maps with all the same elements are equal; for example, m and $m[i := m[i]]$ are not provably equal, but they are provably equal at all values of the domains. From our experience, extensionality is not required for most applications; the motivation to exclude extensionality by default is the better performance of decision procedures for non-extensional maps. Where extensionality is needed, users can supply the required axioms themselves.

A novel and key feature of maps in Boogie is that they can be polymorphic. To motivate this feature, let us consider one of the most important modeling decisions that the designer of a program verifier faces: how to model the memory operated on by the source language. For example, for a type-safe object-oriented language, one may choose to model the object store (the *heap*) as a two-dimensional map from object references and field names to values [22, 0, 13]. Since the result type of such a map depends on the selected field name, it is natural to declare the heap to be of a polymorphic map type. (Without polymorphic maps, one either needs to introduce explicit cast functions or split the one heap variable into several.)

As we already alluded to, Fig. 1 shows by example some Boogie declarations that a verifier might use to encode the semantics of the object-oriented program in Fig. 0 (*cf.* [0, 13]). In the example, Ref is used to denote the type of all object references, $Field \, \alpha$ denotes the type of field names that in the heap retrieve values of type α, and $\langle \alpha \rangle [Ref, Field \, \alpha] \, \alpha$ is the polymorphic map type of the heap itself. For instance, if r is a reference, then $Heap[r, age]$ is an integer and $Heap[r, isMarried]$ is a boolean.

Boogie's type system allows advanced uses of polymorphic maps, which is useful for the kind of semantic models one defines in a program verifier. For example, it is common to want to define properties of heaps, for example distinguishing heaps that satisfy some sort of well-formedness condition from heaps that do not. A natural way to do that is to start by defining a function on heaps, like $IsWellFormed$ in Fig. 1. This is an example of a higher-rank type.

Type parameters of maps are like those of functions: each type parameter must be used in either the domain types or the result type of the map type, and it is an error if type inference cannot uniquely determine the instantiations of type parameters. And as for functions, a polymorphic map is really a family of maps, one for each possible type-parameter instantiation. For example, a map m of type $\langle \alpha \rangle [\mathbf{int}] \alpha$ really denotes a family of maps $\bar{m}$, and $m_{\mathbf{int}}[E]$ has a different value than $m_{\mathbf{bool}}[E]$. It should also be noted that the types $[\alpha] \, T$ and $\langle \alpha \rangle [\alpha] \, T$ are different: the first is a type with a free type parameter α and can be instantiated to any (monomorphic) map type $[s] \, T$, while the second describes polymorphic maps from *any* type to T.

Equality among map types does not depend on the names or order of type parameters. For example, the type $\langle \alpha, \beta \rangle [\alpha, \beta] \mathbf{int}$ is equal to $\langle \gamma, \delta \rangle [\delta, \gamma] \mathbf{int}$. Polymorphism, however, is significant: the types $[\mathbf{int}] \mathbf{bool}$ and $\langle \alpha \rangle [\alpha] \mathbf{bool}$ are incompatible.

Equality. Equality in Boogie is standard mathematical equality, but the typing of equality expressions in Boogie is more liberal than is absolutely the standard. The equality expression $E = F$ is allowed if there is some instantiation of enclosing type parameters that makes the types of E and F equal. Let us motivate this typing rule.

A common way to specify the effects of a source-language procedure is to use a **modifies** clause that lists the object-field locations in the heap that the procedure is allowed to modify. The **modifies** clause is then encoded into Boogie as a procedure postcondition that specifies a relation between the procedure's heap on entry, written $\mathbf{old}(Heap)$, and its heap on return, written $Heap$ (see, *e.g.*, [13]). For instance, to encode that a procedure's effect on the heap in the source language is limited to $p.age$ and $p.isMarried$, one can in Boogie use a postcondition like

$$(\forall \langle \alpha \rangle \ r \colon Ref, \ f \colon Field \ \alpha \bullet \ Heap[r,f] = \mathbf{old}(Heap[r,f]) \ \lor$$
$$(r = p \land f = age) \lor (r = p \land f = isMarried) \ )$$

In order to type check this expression, it is necessary for the type system to consider the possible instantiation $\alpha := \mathbf{int}$ for $f = age$ and $\alpha := \mathbf{bool}$ for $f = isMarried$, and Boogie does exactly that. Being liberal in this typing rule does not cause any semantical problems in Boogie: because different types represent disjoint sets of individuals, an equation simply evaluates to **false** if the two sides of the equation evaluate to individuals of different types. For example, for the f in the quantifier above, $f = age \land f = isMarried$ type checks but always evaluates to **false**.

1.2 Formalization of the Type System and Type Checking

The abstract syntactic category of types is described by the following grammar:

$$Type \ ::= \ \alpha \ | \ C \ Type^* \ | \ \langle \alpha^* \rangle \, [Type^*] \, Type$$

in which $C \in \mathcal{C}$ ranges over type constructors (with a fixed arity $arity(C)$) and $\alpha \in \mathcal{A}$ over an infinite set of type variables. We assume that $\mathcal{C}$ always contains the pre-defined nullary constructors $\mathbf{bool}, \mathbf{int}, \mathbf{bv0}, \mathbf{bv1}, \mathbf{bv2}, \ldots$. Only those types are well-formed in which type constructors receive the correct number of argument types, and in which type parameters of polymorphic map types occur in the map domain or result types.

For two types $s, t \in Type$, we write $s \equiv t$ iff s and t are equal modulo renaming or reordering of bound type parameters. A *type substitution* is a mapping $\sigma : \mathcal{A} \rightarrow Type$ from type variables to types. Substitutions are canonically extended on all types, assuming that variable capture is avoided by renaming bound type variables when necessary.

Formalizing the typing of expressions, the judgment $\mathcal{V} \Vdash E : t$ says that in a context with variable-type bindings $\mathcal{V}$, expression E can be typed as type t. Figure 3 shows the most important typing rules. All other operators are typed as in the rule for function application. In the figure and the whole paper, $\mathcal{F}$ denotes the set of declared functions and constants, whereas $\mathcal{X}$ denotes an infinite set of variables.

Note that for any type-correct program, all type-parameter instantiations have been resolved. But this does not mean that the application of a polymorphic function or map can easily be replaced by a specific monomorphic instance, because of quantifications over types. For example, the application of $EmptySequence$ in Fig. 2 is resolved to $EmptySequence_\alpha$, but α is a quantified type variable that refers to any type; hence, the axiom says something about every member of the $EmptySequence$ family.

$$\frac{x \mapsto t \ \in \mathcal{V}}{\mathcal{V} \Vdash x : t} \qquad \frac{\mathcal{V} \Vdash E : t}{\mathcal{V} \Vdash E{:}t : t} \qquad \frac{\begin{array}{c} f\langle\bar{\alpha}\rangle(\bar{s}) \ \mathbf{returns} \ (t) \ \in \mathcal{F} \\ \mathcal{V} \Vdash E_i : \sigma(s_i) \quad (\text{for } (E_i, s_i) \in (\bar{E}, \bar{s})) \end{array}}{\mathcal{V} \Vdash f(\bar{E}) : \sigma(t)} \ *$$

$$\frac{\begin{array}{c} \mathcal{V} \Vdash E : s \qquad \mathcal{V} \Vdash F : t \\ \sigma(s) \equiv \sigma(t) \end{array}}{\mathcal{V} \Vdash E = F : \mathbf{bool}} \qquad\qquad \frac{(\mathcal{V}, \bar{x} \mapsto \bar{t}) \Vdash E : \mathbf{bool} \qquad Q \in \{\forall, \exists\}}{\mathcal{V} \Vdash (\ Q \ \langle\bar{\alpha}\rangle \ \bar{x} : \bar{t} \bullet E \ ) : \mathbf{bool}}$$

$$\frac{\begin{array}{c} \mathcal{V} \Vdash m : \langle\bar{\alpha}\rangle[\bar{s}]t \\ \mathcal{V} \Vdash E_i : \sigma(s_i) \quad (\text{for } (E_i, s_i) \in (\bar{E}, \bar{s})) \end{array}}{\mathcal{V} \Vdash m[\bar{E}] : \sigma(t)} \ * \qquad \frac{\begin{array}{c} \mathcal{V} \Vdash m : \langle\bar{\alpha}\rangle[\bar{s}]t \qquad \mathcal{V} \Vdash F : \sigma(t) \\ \mathcal{V} \Vdash E_i : \sigma(s_i) \quad (\text{for } (E_i, s_i) \in (\bar{E}, \bar{s})) \end{array}}{\mathcal{V} \Vdash m[\bar{E} := F] : \langle\bar{\alpha}\rangle[\bar{s}]t} \ *$$

Fig. 3. The typing rules for Boogie expressions. The context of type judgments is a partial mapping $\mathcal{V} : \mathcal{X} \rightharpoonup \textit{Type}$ that assigns types to variables. The rules marked with '*' impose the side condition $dom(\sigma) = \{\bar{\alpha}\}$. The typing rules show what it means for expressions to be type correct; they abstract over how type inference is done.

1.3 Matching Triggers

We have one more thing to say about expressions in Boogie, and it concerns the way many SMT solvers handle universal quantifications, namely by selective instantiation. Instantiations are based on (user-supplied or inferred) *matching triggers*, which indicate which patterns of ground terms in the prover's state are to give rise to instantiations [8]. Boogie has support for specifying matching triggers for quantifications. For example,

$$\mathbf{axiom} \ (\forall x : t \bullet \ \{f(x)\} \ \textit{fInverse}(f(x)) = x \ );$$

specifies the trigger $f(x)$ and says to instantiate the universally quantified variable with any value appearing among the ground terms as an argument of function f. In an SMT solver based solely on triggers, these are the only instantiations there will ever be. All Boogie front ends make heavy use of triggers. (For an application that uses quantifiers and an explanation of the design of triggers for that application, see [14].)

A trigger is a set of expressions, each of which will undergo the encoding into the underlying logic that we are about to describe. However, it is important that the logical encoding not interfere with user-defined triggers or automatically inferred triggers, since that might lead to poor performance (too many instantiations) or incompleteness (too few instantiations).

2 Representation of Types as Terms

Automated theorem provers and SMT solvers typically offer only untyped or simple multi-sorted logics as their input language (with the notable exception of Alt-Ergo [2], which provides a polymorphic type system). With such a prover as the verification back end, the expressions from the richer language have to be translated into the simpler logic. We describe two approaches to this translation in Section 3: one that captures type information using logical guards and one that encodes type parameters of polymorphic functions as additional function arguments. In both cases, it is necessary to encode

Boogie's types as terms (so that typing conditions can be expressed as formulae), which is the subject of this section.

As a simply typed target language, we use a subset of the Boogie expression language, restricting the available types to (i) the built-in types **bool** and **int** (other types supported directly by the simply typed logic can be treated analogously to **int**), (ii) a type U for (non-**bool**, non-**int**) individuals, and (iii) a type T for (encoded) types. If necessary, expressions in this simply-typed language could be translated further to an untyped logic by adding domain predicates and guards for the types **bool**, **int**, U, and T. Because current SMT solvers are able to directly handle the four types, however, such a translation will usually not be required. Furthermore, we introduce a function symbol $type: U \to T$ that maps individuals to their type.

We encode types so that T forms an algebraic datatype. If the target logic has direct support for algebraic datatypes, one may be able to build on it; in the scope of this paper, we use functions and axioms to describe the encoding.

2.0 Type Constructors

Each type constructor $C \in \mathcal{C}$ gives rise to a function symbol $C^{\#}: T^{arity(C)} \to T$, as well as an axiomatization of a number of properties, including distinctness and injectivity. To formalize that the images of different type-constructor functions $C^{\#}$ are disjoint, we introduce a function $Ctor: T \to$ **int** and, for each type constructor C, a unique constant n_C. Injectivity is achieved by defining selector functions $C^{-1}, \ldots, C^{-n}: T \to T$ for each n-ary type constructor C:

$$(\forall \bar{x}: T \bullet \ Ctor(C^{\#}(\bar{x})) = n_C) \ \wedge \ \bigwedge_{i=1}^{arity(C)} (\forall \bar{x}: T \bullet \ C^{-i}(C^{\#}(\bar{x})) = x_i)$$

Theoretically, further axioms are needed for a faithful model of the type system. However, because these additional axioms are of a kind that cannot be expected to be useful for SMT solvers (e.g., statements about well-foundedness), we practically use only the axioms shown above in the Boogie implementation.

2.1 Reduction of Map Types to Ordinary Type Constructors

The encoding of Boogie's polymorphic map types is done by a reduction to normal type constructors: a map type t containing the free type variables $\alpha_1, \ldots, \alpha_n$ (and arbitrary bound variables) can be encoded like a type expression $C_t \, \alpha_1, \ldots, \alpha_n$, for some fresh constructor C_t. The access functions can then be seen and axiomatized as ordinary functions $select_t$, $store_t$, based on the axioms of the first-order theory of arrays [18].

There is a caveat in this construction: if two map types s, t have common instances $u = \sigma_s(s) = \sigma_t(t)$, then an encoding of u using either C_s or C_t will be overly restrictive. In particular, it might happen that u is encoded as C_s in one part, and as C_t in another part of the same formula, leading to incompleteness:

> **function** $f\langle\alpha\rangle(\alpha)$ **returns** (**int**);
> **axiom** $(\forall \langle\alpha\rangle \ m: [\alpha]$ **int** $\bullet \ f(m) = 0)$; **axiom** $(\forall \langle\alpha\rangle \ m: [$**int**$]\alpha \bullet \ f(m) = 1)$;

If $s = [\alpha]\,\mathbf{int}$ in the first axiom happens to be encoded as $C_s\,\alpha$, and $t = [\mathbf{int}]\,\alpha$ in the second axiom as $C_t\,\alpha$, then the inconsistency of the two axioms will be lost: $C_s\,\alpha$ and $C_t\,\alpha$ do not have any common instances. The solution is to define larger classes of type constructors for map types: we abstract over map types and define constructors only for "most general" map types. Let us be more precise.

Given two types $s, t \in Type$, we write $t \sqsubseteq s$ and say that t is an *instance* of s iff there is a substitution σ such that $\sigma(s) \equiv t$. Observe that $\sqsubseteq$ is a pre-order on types, but not a partial order because anti-symmetry is violated for types that differ only in the names of free variables. The induced equivalence relation is denoted with $\cong$: for $s, t \in Type$, we define $s \cong t$ iff $s \sqsubseteq t$ and $t \sqsubseteq s$. It is the case that $\equiv\, \subseteq\, \cong$.

The pre-order $\sqsubseteq$ is canonically extended to $TypeC = Type/{\cong}$ and partially orders the set. In fact, $(TypeC, \sqsubseteq)$ is a join-semi-lattice (*i.e.*, any two types have a least common upper bound) whose $\top$-element is the class of type variables α. The strict order $\sqsubset$ satisfies the *ascending chain condition (ACC)*: every ascending chain of types in $TypeC$ eventually becomes stationary. This is important, because it justifies the existence of most-general map types that are the basis for our map-type encoding.

Let $\mathcal{M}_C \subseteq TypeC$ be the set of $\sqsubseteq$-maximal type classes whose elements start with the map type constructor, and let $\mathcal{M}$ be a set of unique representatives for all classes in $\mathcal{M}_C$. The elements of $\mathcal{M}$ can be seen as skeletons of map types and determine the binding and occurrences of bound type variables. Examples of types in $\mathcal{M}$ are:

$$[\alpha]\,\beta \qquad [\alpha, \beta]\,\gamma \qquad [\alpha, \beta, \gamma]\,\delta \qquad \langle\alpha\rangle[\alpha]\,\alpha \qquad \langle\alpha\rangle[\alpha]\,\beta \qquad \langle\alpha\rangle[\alpha]\,(C\,\alpha)$$

For every type t that starts with a map type constructor, there is a unique type $m = \mathrm{skel}(t) \in \mathcal{M}$ with $t \sqsubseteq m$. For example, $\mathrm{skel}(\langle\alpha\rangle[C\,\alpha, \mathbf{int}]\mathbf{bool}) = \langle\alpha\rangle[C\,\alpha, \beta]\gamma$. This means that every map type t (also types containing free variables) can be represented in the form $\sigma(\mathrm{skel}(t))$, whereby the substitution σ is uniquely determined for all variables that occur free in $\mathrm{skel}(t)$. We write $\mathrm{flesh}(t)$ for the unique substitution satisfying $\mathrm{flesh}(t)(\mathrm{skel}(t)) = t$ whose domain is a subset of $\{\alpha_1, \ldots, \alpha_n\}$, where $\alpha_1, \ldots, \alpha_n$ are the free variables in $\mathrm{skel}(t)$. For example, $\mathrm{flesh}(\langle\alpha\rangle[C\,\alpha, \mathbf{int}]\mathbf{bool}) = (\beta \mapsto \mathbf{int}, \gamma \mapsto \mathbf{bool})$.

Translation of Types to Terms. In order to encode types, for each type $t \in \mathcal{M}$ that contains n free type variables $\alpha_1, \ldots, \alpha_n$, we introduce a new n-ary function symbol $\textsc{m}_t^{\#} : T^n \to T$. We will use the notation $\mathrm{Skel}^{\#}(s) := \textsc{m}_{\mathrm{skel}(s)}^{\#}$ for the skeleton symbol of an arbitrary map type s, and $\mathrm{Skel}^{-i}(s) := \textsc{m}_{\mathrm{skel}(s)}^{-i}$ for the selectors. Given an instantiation $\mu : \mathcal{A} \to Term$ of type variables, types can then be translated to terms:

$$[\![\alpha]\!]_\mu = \mu(\alpha) \qquad [\![C\,t_1 \ldots t_n]\!]_\mu = C^{\#}([\![t_1]\!]_\mu, \ldots, [\![t_n]\!]_\mu)$$
$$[\![m]\!]_\mu = \mathrm{Skel}^{\#}(m)([\![\mathrm{flesh}(m)(\beta_1)]\!]_\mu, \ldots, [\![\mathrm{flesh}(m)(\beta_n)]\!]_\mu)$$

In the last equation, m is a map type $\langle\bar{\alpha}\rangle[\bar{s}]\,t$ such that $\mathrm{skel}(m)$ contains the free type variables $\beta_1, \ldots, \beta_n$ (in this order of occurrence). Some examples are:

$$[\![C\,T]\!]_\mu = C^{\#}(T^{\#}) \qquad\qquad [\![[\mathbf{int}]\,T]\!]_\mu = \textsc{m}_{[\alpha]\beta}^{\#}(int^{\#}, T^{\#})$$
$$[\![[T]\,S]\!]_\mu = \textsc{m}_{[\alpha]\beta}^{\#}(T^{\#}, S^{\#}) \qquad [\![\langle\alpha\rangle[\alpha]\,S]\!]_\mu = \textsc{m}_{\langle\alpha\rangle[\alpha]\beta}^{\#}(S^{\#})$$

Symbols and Axioms of Maps with Map Reduction. The access functions *select* and *store* can be seen and axiomatized as ordinary functions, based on the axioms of the first-order theory of arrays [18]. For each map type $m \in \mathcal{M}$, we introduce separate symbols $select_m$ and $store_m$. Suppose that $m = \langle \bar{\alpha} \rangle [\bar{s}] \, t \in \mathcal{M}$ contains the free type variables $\bar{\beta} = (\beta_1, \ldots, \beta_n)$ (in this order of occurrence). Then, the access functions have the following types:

$$select_m \langle \bar{\alpha}, \bar{\beta} \rangle (m, \bar{s}) \textbf{ returns } (t) \qquad store_m \langle \bar{\alpha}, \bar{\beta} \rangle (m, \bar{s}, t) \textbf{ returns } (m)$$

It is necessary to include both $\bar{\alpha}$ and $\bar{\beta}$ as type parameters, because m is parametric in the latter, and $\bar{s}$ and t might be parametric in both. The semantics of maps is defined by axioms similar to the standard axioms of non-extensional arrays [18] ($\bar{\alpha}'$ is a vector of fresh type variables, and $\bar{\alpha} \mapsto \bar{\alpha}'$ the substitution that replaces $\bar{\alpha}$ with $\bar{\alpha}'$):

$$(\forall \langle \bar{\alpha}, \bar{\beta} \rangle \; h \colon m, \; \bar{x} \colon \bar{s}, \; z \colon t \bullet \; select_m(store_m(h, \bar{x}, z), \bar{x}) = z \;) \; \wedge$$
$$(\forall \langle \bar{\alpha}, \bar{\alpha}', \bar{\beta} \rangle \; h \colon m, \; \bar{x} \colon \bar{s}, \; \bar{y} \colon (\bar{\alpha} \mapsto \bar{\alpha}')\bar{s}, \; z \colon t \bullet$$
$$\bar{x} = \bar{y} \vee select_m(store_m(h, \bar{x}, z), \bar{y}) = select_m(h, \bar{y}) \;)$$

3   Translation of Expressions

We define two main approaches to translating typed Boogie expressions into equivalent simply typed expressions: one that captures type information using logical guards (Section 3.0) and one that encodes type parameters of polymorphic functions as ordinary (additional) arguments (Section 3.1). The second encoding relies on the usage of e-matching to instantiate quantifiers (in contrast to methods like superposition used in first-order theorem provers), because typing information is generated such that triggers can only match on expressions of the right type (also see [5]).

The following Boogie program is used as running example for the translations:

function $Mojo\langle \alpha \rangle (\alpha)$ **returns** (**int**); **axiom** $(\forall x \colon \textbf{int} \bullet \; Mojo(x) = x \;)$;
type $GuitarPlayer$;   **axiom** $(\forall g \colon GuitarPlayer \bullet \; Mojo(g) = 68 \;)$;

Note that it is essential to take the types of the quantified variables into account to not introduce inconsistent axioms.

3.0   Translation using Type Guards

There is a long tradition of encoding type information using type guards, *e.g.*, [16, 5, 6]. As this translation is rather naive and has the disadvantage of complicating the propositional structure of formulae, it has been claimed [5] that its performance impact is prohibitive for many applications. We are able to show in Section 4, however, that this is no longer the case with state-of-the-art SMT solvers.

The Mojo example is complemented with type guards as follows. Because the quantified formulae are now guarded and only concern individuals of the right types, no contradiction is introduced. The function $i2u$ is defined below.

function $Mojo^{\#}(U)$ **returns** (U);   **const** $GuitarPlayer^{\#} \colon T$;
axiom $(\forall x \colon U \bullet \; type(Mojo^{\#}(x)) = int^{\#} \;)$;   // function axiom
axiom $(\forall x \colon U \bullet \; type(x) = int^{\#} \Rightarrow Mojo^{\#}(x) = x \;)$;
axiom $(\forall g \colon U \bullet \; type(g) = GuitarPlayer^{\#} \Rightarrow Mojo^{\#}(g) = i2u(68) \;)$;

Function Axioms. During the translation, user-defined Boogie functions are replaced with U-typed functions. For a function $f\langle\alpha_1,\ldots,\alpha_m\rangle(s_1,\ldots,s_n)$ **returns** (t) such that $\alpha_1,\ldots,\alpha_k$ do not occur in $s_1,\ldots,s_n$ (but only in t), while $\alpha_{k+1},\ldots,\alpha_m$ occur in $s_1,\ldots,s_n$ (and possibly in t), this post-translation function $f^{\#}$ has the type $T^k \times U^n \to U$. We will capture the original typing with an axiom of the shape:

$$(\forall\,\bar{x}\colon \bar{U}, \bar{y}\colon \bar{T} \bullet\; type(f^{\#}(\bar{y},\bar{x})) = [\![t]\!]_\mu\;) \tag{0}$$

This axiom does not contain any quantifiers corresponding to $\alpha_{k+1},\ldots,\alpha_m$ that occur in $s_1,\ldots,s_n$, which is advantageous for SMT solvers because the formula does not offer good triggers for $\alpha_{k+1},\ldots,\alpha_m$. Instead, the mapping $\mu : \mathcal{A} \to \mathit{Term}$ that determines the values of type parameters plays a prominent role. We define this mapping using *extractor terms*, which are recursively defined over types and describe how the type parameter values can be reconstructed from the actual arguments $\bar{x}$ with the help of the selector functions C^{-i} defined in Section 2.0.

Suppose that $\alpha \in \mathcal{A}$ is a type variable. Assuming that the term E encodes the type $t \in \mathit{Type}$, the set $\mathit{extractors}_\alpha(E, t)$ specifies terms that compute α's value:

$$\mathit{extractors}_\alpha(E, \beta) = \textbf{if } \alpha = \beta \textbf{ then } \{E\} \textbf{ else } \emptyset$$

$$\mathit{extractors}_\alpha(E, C\,t_1\ldots t_n) = \bigcup_{i=1}^{n} \mathit{extractors}_\alpha(C^{-i}(E),\; t_i) \qquad (C \in \mathcal{C})$$

$$\mathit{extractors}_\alpha(E, m) = \bigcup_{i=1}^{n} \mathit{extractors}_\alpha(\mathrm{Skel}^{-i}(m)(E),\; \mathrm{flesh}(m)(\gamma_i))$$

In the last equation, m is a map type $\langle\bar{\beta}\rangle[\bar{s}]\,t$ such that $\mathrm{skel}(m)$ contains the free type variables $\gamma_1,\ldots,\gamma_n$ (in this order of occurrence). Some examples are:

$$\mathit{extractors}_\alpha(x, C\,\beta\,\alpha) \;=\; \{C^{-2}(x)\}$$

$$\mathit{extractors}_\alpha(x, \langle\beta\rangle[C\,\beta\,\alpha]\,\alpha) \;=\; \{C^{-2}(\mathrm{M}^{-1}_{\langle\beta\rangle[C\,\beta\,\gamma]\,\delta}(x)),\; \mathrm{M}^{-2}_{\langle\beta\rangle[C\,\beta\,\gamma]\,\delta}(x)\}$$

The extractor $C^{-2}(x)$, for instance, can derive α's value from the instance $C\,\textbf{int}\,\textbf{bool}$ of $C\,\beta\,\alpha$, resulting in $C^{-2}([\![C\,\textbf{int}\,\textbf{bool}]\!]) = C^{-2}(C^{\#}(\textbf{int}^{\#},\textbf{bool}^{\#})) = \textbf{bool}^{\#}$.

A simple optimization (that is implemented in Boogie but left out from this paper for reasons of presentation) is to keep argument or result types **int** and **bool** of functions, instead of replacing them with U. This can reduce the number of casts to and from U later needed in the translation.

Embedding of Built-in Types. SMT solvers offer built-in types like booleans, integers, and bit vectors, whose usage is crucial for performance. We define casts to and from the type U in order to integrate built-in types into our framework. For the built-in types **bool** and **int**, we introduce the cast functions $i2u : \textbf{int} \to U$, $u2i : U \to \textbf{int}$, $b2u : \textbf{bool} \to U$, $u2b : U \to \textbf{bool}$ and axiomatize them as:

$$(\forall\,x\colon \textbf{int} \bullet\; type(i2u(x)) = int^{\#} \wedge u2i(i2u(x)) = x\;) \;\wedge$$
$$(\forall\,x\colon U \bullet\; type(x) = int^{\#} \Rightarrow i2u(u2i(x)) = x\;)$$

and analogously for **bool**. The axioms imply that $i2u$ and $b2u$ are embeddings into U, and that $u2i$ and $u2b$ are their inverses. For simplicity, in the following translation we insert casts in each place where operators over **bool** or **int** occur, although many of the casts could directly be eliminated using the axioms. Such optimizations are present in the Boogie implementation as well.

Translation of Expressions. Given an instantiation $\mu : \mathcal{A} \to Term$ of type variables, the main cases of the translation are:

$$
\begin{aligned}
\llbracket x \rrbracket_\mu &= x & (x \in \mathcal{X}) \\
\llbracket f(E_1, \ldots, E_n) \rrbracket_\mu &= f^\#(\llbracket E_1 \rrbracket_\mu, \ldots, \llbracket E_n \rrbracket_\mu) \\
\llbracket E = F \rrbracket_\mu &= b2u(\llbracket E \rrbracket_\mu = \llbracket F \rrbracket_\mu) \\
\llbracket E + F \rrbracket_\mu &= i2u\big(u2i(\llbracket E \rrbracket_\mu) + u2i(\llbracket F \rrbracket_\mu)\big) & \cdots \\
\llbracket E \wedge F \rrbracket_\mu &= b2u\big(u2b(\llbracket E \rrbracket_\mu) \wedge u2b(\llbracket F \rrbracket_\mu)\big) & \cdots \\
\llbracket (\forall \langle \bar{\alpha} \rangle \, \bar{x} : \bar{t} \bullet E) \rrbracket_\mu &= b2u(\forall \bar{x} : \bar{U}, \bar{y} : \bar{T} \bullet type(\bar{x}) = \llbracket \bar{t} \rrbracket_{\mu'} \Rightarrow u2b(\llbracket E \rrbracket_{\mu'})) \\
\llbracket (\exists \langle \bar{\alpha} \rangle \, \bar{x} : \bar{t} \bullet E) \rrbracket_\mu &= b2u(\exists \bar{x} : \bar{U}, \bar{y} : \bar{T} \bullet type(\bar{x}) = \llbracket \bar{t} \rrbracket_{\mu'} \wedge u2b(\llbracket E \rrbracket_{\mu'}))
\end{aligned}
$$

In the last two equations, $\bar{y}$ is a vector of fresh variables, and $\mu' = (\mu, \bar{\alpha} \mapsto \bar{y})$. In the case that a type parameter α_i occurs in some of the types $\bar{t}$, a more efficient translation is possible by extracting the value of α_i from the bound variables $\bar{x}$:

$$\mu'(\alpha_i) \in \bigcup_{j=1}^m extractors_{\alpha_i}(type(x_j), t_j)$$

The optimization is particularly relevant with e-matching-based SMT solvers, because the formula resulting from the original translation often does not contain good triggers for the variables $\bar{y}$: type parameters $\bar{\alpha}$ are used only in types, which usually do not provide good discrimination for instantiation.

3.1 Translation using Type Arguments

Our second translation works by explicitly passing the values of type parameters to functions. In the context of SMT solvers, this allows us to completely leave out type guards and leads to formulae with a simpler propositional structure, albeit functions have a higher arity and more terms occur in the formulae. It has to be noted that this second translation crucially depends on the usage of an SMT solver with e-matching: such solvers are not able to exploit missing type guards, because typing information is inserted in expressions in such a way that triggers can only match on expressions of the right type. The translation trades generality for performance: while it is not applicable with most first-order theorem provers (*e.g.*, superposition provers), the experimental evaluation in Section 4 shows a clear performance gain compared to the type guard translation from the previous section. A similar observation is made in [5].

When using type arguments, the Mojo example gets translated as follows:

function $Mojo^\#(T, U)$ **returns** (U);   **axiom** $(\forall x : U \bullet Mojo^\#(int^\#, x) = x)$;
const $GuitarPlayer^\# : T$;   **axiom** $(\forall g : U \bullet Mojo^\#(GuitarPlayer^\#, g) = i2u(68))$;

The Typing of Functions. A function $f\langle \alpha_1, \ldots, \alpha_m \rangle(s_1, \ldots, s_n)$ **returns** $(t) \in \mathcal{F}$ is during the translation replaced by a function $f^\#$ with the type $T^m \times U^n \to U$, *i.e.*, the type parameters are given the status of ordinary function arguments. It is unnecessary to generate typing axioms for $f^\#$, since typing information is inserted everywhere in terms during the translation and does not have to be derived by the SMT solver.

		Type Guards	Type Arguments	No Types
		Z3 2.0		
Boogie	(2598)	2002/595/1, 0.781s	2000/597/1, 0.651s	1984/613/1, 0.813s
VCC	(7840)	6999/839/2, 3.447s	6999/836/5, 2.181s	6999/836/5, 2.196s
HAVOC	(385)	353/16/16, 0.709s	351/18/16, 0.524s	350/17/18, 0.367s
		Z3 1.3		
Boogie	(2590)	1978/609/3 1.107	1974/611/5 1.212	1961/626/3 2.385

Fig. 4. Results for the different benchmark categories. In each cell, we give the number of times the outcome valid/invalid/timeout occurred, as well as the average time needed for successful proof attempts (*i.e.*, counting cases with the outcome valid or invalid).

Translation of Expressions. We maintain both an instantiation $\mu : \mathcal{A} \to Term$ and an environment $\mathcal{V} : \mathcal{X} \to Type$ that assigns types to variables during the translation:

$$
\begin{aligned}
[\![x]\!]_{\mu,\mathcal{V}} &= x & (x \in \mathcal{X}) \\
[\![f(\bar{E})]\!]_{\mu,\mathcal{V}} &= f^{\#}([\![\sigma(\bar{\alpha})]\!]_{\mu,\mathcal{V}}, \ [\![\bar{E}]\!]_{\mu,\mathcal{V}}) \\
[\![E = F]\!]_{\mu,\mathcal{V}} &= b2u([\![E]\!]_{\mu,\mathcal{V}} = [\![F]\!]_{\mu,\mathcal{V}} \wedge [\![t_E]\!]_{\mu} = [\![t_F]\!]_{\mu}) \\
[\![E + F]\!]_{\mu,\mathcal{V}} &= i2u(u2i([\![E]\!]_{\mu,\mathcal{V}}) + u2i([\![F]\!]_{\mu,\mathcal{V}})) \quad \cdots \\
[\![E \wedge F]\!]_{\mu,\mathcal{V}} &= b2u(u2b([\![E]\!]_{\mu,\mathcal{V}}) \wedge u2b([\![F]\!]_{\mu,\mathcal{V}})) \quad \cdots \\
[\![(Q \langle \bar{\alpha} \rangle \ \bar{x} \colon \bar{t} \bullet E)]\!]_{\mu,\mathcal{V}} &= b2u(Q \ \bar{x} \colon \bar{U}, \ \bar{y} \colon \bar{T} \bullet u2b([\![E]\!]_{(\mu, \bar{\alpha} \mapsto \bar{y}),(\mathcal{V}, \bar{x} \mapsto \bar{t})}))
\end{aligned}
$$

The second equation assumes f has typing $\langle \bar{\alpha} \rangle (\bar{s})$ **returns** (t) and that σ is the instantiation of the type parameters $\bar{\alpha}$ that is inferred when applying f to $\bar{E}$. The types t_E, t_F in the third equation are determined by $\mathcal{V} \Vdash E : t_E$ and $\mathcal{V} \Vdash F : t_F$. In the last equation, $\bar{y}$ is a vector of fresh variables, and $Q \in \{\forall, \exists\}$ is a quantifier.

4 Experimental Results and Related Work

We quantitatively evaluate the two different translations of Boogie expressions, together with a third unsound translation that simply erases all type information. The third translation is close to the translation used by the Boogie 1 tool, so that a comparison between Boogie 2 and Boogie 1 is possible. The evaluated Boogie programs are:

- *The Boogie and SscBoogie regression test suites:* A collection of correct and incorrect programs written in Boogie, Spec# [0], and Dafny [13] that make use of polymorphism; also parts of the Boogie tool itself (a Spec# program) are included.
- *Hyper-V verification conditions generated by VCC [4]:* Boogie programs that stem from a project to verify the Microsoft hypervisor Hyper-V.
- *Benchmarks from the HAVOC tool [3]:* Regression tests and verification conditions to prove memory safety and invariants of various C programs.

Because the programs of the last two categories do not use polymorphism, the overhead of our translations for simple problems (that could really be handled with the "No Types" translation) is measured.

For each of the categories, we used Boogie 2 to generate verification conditions with the different translations and write them to separate files. We then measured the

performance of the state-of-the-art SMT solver Z3 2.0[0] on the altogether more than 10,000 verification conditions. The prover was run on each verification condition with a timeout of 120s (1800s for the Boogie tests), measuring the average time needed over three runs. All experiments were made on an Intel Core 2 Duo, 3.16GHz, with 4GB.

Figure 4 summarizes the results. The time difference between the type argument encoding and the translation without types is always very small, the argument encoding is even faster in two categories. The type guard encoding is close to the other translations on the Boogie tests, but is on average about 55% slower on the VCC examples, and performs similarly on the HAVOC examples. One explanation for this phenomenon is that (in particular) VCC generates a large number of Boogie functions, which leads to a large number of additional axioms in the type guard encoding.

Related Work. The intermediate verification language Boogie is most closely related to Why [9], which offers ML-style polymorphism [21]. ML-style polymorphism (or "let polymorphism") is more limited than the higher-rank polymorphism in Boogie; for example, it does not allow polymorphic map types nor general quantifications over types, both of which are used heavily by some Boogie front ends. Our typing rule for equality is similar to the "heterogeneous equality" introduced in [17]. Meanwhile, compilers have also explored the benefits of using typed intermediate languages [20].

Couchot and Lescuyer turn formulae with ML-style polymorphism into multi-sorted and untyped formulae [5], taking advantage of built-in theories. They have implemented their translations as modules of the Why tool [9] and report on some experiments. With Simplify [8], they measure a 200% slowdown with their version of a type guard translation, and a 300% slowdown with their other encoding (which is somewhat similar to our type argument encoding). In contrast, we measure a slowdown of at most 95% with the type guards encoding and at most 45% with the type arguments encoding.

Bobot *et al.* show how to incorporate ML-style polymorphism directly into an SMT solver [2]. Our type arguments translation is quite similar to the machinery they present. It would be interesting to put to test their conjecture that building polymorphism into a prover is a better solution than handling it through a pre-processing step.

There is a large body of work on the encoding of (typed) higher-order logic (HOL) in first-order logic (FOL). Such translations primarily target FOL provers, in contrast to SMT solvers as in our case. Meng and Paulson [19] enrich terms with type annotations in the form of first-order functions and describe different translations, some of which are sound, while others require proofs to be typechecked and possibly rejected afterwards. Similarly, Hurd [11] describes translations from HOL to FOL in which type information can be included in the operator for function application, which is similar to our type argument encoding (and in particular the handling of map types). Translations in the same spirit as our type guard encoding have been studied [6] for the Mizar language.

5 Conclusions

We have introduced the type system of Boogie 2, shown how its advanced type features are useful to program verifiers in encoding program semantics, and shown how to translate its polymorphic types and expressions into first-order formulae suitable for SMT

---

[0] http://research.microsoft.com/projects/z3/