The Boogie 2 Type System: Design and Verification Condition Generation

K. Rustan M. Leino[1] and Philipp Rümmer[2]

[1] Microsoft Research, Redmond, WA, USA, `leino@microsoft.com`
[2] Chalmers University of Technology, Göteborg, Sweden, `philipp@chalmers.se`

Abstract. Intermediate languages are a paradigm to separate concerns in software verification systems when bridging the gap between (real-world) 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 the Boogie 2 language, which is used in several program verifiers. The paper gives two encodings of types and formulae in simply typed (or untyped) logic such that ordinary theorem provers and SMT solvers can be used to discharge verification conditions. Extensive empirical evidence is provided showing that the impact of the additional typing information on the verification performance is negligible.

1 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, 13, 12, 4, 5]) is to make use of an *intermediate verification language* [18, 1, 11].

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*, the successor of BoogiePL [8, 1]. 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 [11], Boogie 2 features polymorphic maps, higher-rank polymorphism, and impredicativity which are useful in modeling the semantics of a type-safe heap (as in Spec# or Java.)

We also describe our translation of the polymorphic Boogie 2 into multi-sorted logic, which is used by many satisfiability modulo theories (SMT) solvers that support the SMT-LIB format [2]. Alternatively, our translation can target unsorted logic, like the input to Simplify [9]. In fact, we give two different translations into multi-sorted logic, and we present performance figures from substantial benchmarks that compare these. The benchmarks come from the Spec# program verifier [1], the VCC [5] and HAVOC [4] verifiers for C, and Dafny [16], all of which build on Boogie 2. All of the benchmarks make extensive use of so-called triggers required for e-matching [9], 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 into multi-sorted logics suitable for SMT solvers, (iii) experimental data comparing the performance of the two translations with each other and with an (unsound) translation ignoring types.

2 Boogie 2 Types and Expressions

A Boogie program is a set of declarations. The language offers seven kinds of declarations, which can be divided into two categories. The *imperative declarations* introduce variables, procedure specifications, and procedure implementations. The procedure implementations are in Boogie what gives rise to proof obligations. The implementations can use familiar statements like assignments, loops, procedure calls, and gotos, as well as **assert** statements, which (along with loop invariants and procedure pre- and postconditions) prescribe the proof obligations, and **assume** statements (that is, *partial commands*, which are used to limit nondeterminism and feasible paths. Though unsuitable for execution, the partial commands are essential to modeling the semantics of source languages. The implementations are transformed using *weakest preconditions* [1] into expressions, which together with types are our main concern in this paper.

The *mathematical declarations* introduce types, constants, functions, and axioms. These define first-order structures that can be used in expressions and declarations. All expressions in Boogie are total; that is, every well-typed expression yields some appropriately typed value that is a function of its subexpressions.

In the rest of this section, we introduce salient features of Boogie 2's mathematical declarations, along with some motivating examples. For further details of the language, we refer to the Boogie 2 language reference manual [15].

2.1 Type Declarations

The built-in types of Boogie 2 are booleans (**bool**), mathematical integers (**int**), and bit-vector types of every size ($\mathbf{bv0}, \mathbf{bv1}, \mathbf{bv2}, \ldots$). In addition, there are map

```
type Wicket;      const unique a: Wicket;   function Age(Wicket) returns (int);
type Pair α β;    const unique b: Wicket;   axiom (∀ w: Wicket • 0 ⩽ Age(w) );
                  const c: Wicket;          axiom Age(b) = 25;
```

Fig. 1. Example Boogie 2 declarations.

```
function Left⟨α, β⟩(α, β) returns (α);
axiom (∀ ⟨α, β⟩ a: α, b: β • Left(a, b) = a );

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

function GimmieAnything⟨α⟩() returns (α);
```

Fig. 2. Examples of polymorphic functions and quantifications over types in Boogie 2.

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 *Wicket* and a binary type constructor *Pair*. The sets of individuals denoted by *Wicket*, *Pair* **int int**, and *Pair Wicket* **int** are all disjoint.

2.2 Constants and Functions

A Boogie program can introduce constants of any type and functions over types. Properties of these constants and functions can be described using axiom declarations. For example, Fig. 1 introduces a, b, and c as names of wickets and *Age* as a function from wickets to integers, and postulates the age of any wicket to be non-negative and the age of b to be 25. As a convenience, the constants declared with **unique** are defined to be pairwise distinct; for example, the declarations above imply $a \neq b$, but say nothing about the relation between c and the other constants. The responsibility of making sure the axioms are logically consistent rests with the user; for instance, **axiom false**; is a legal declaration that will make all proof obligations hold trivially.

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. For example, a function that retrieves the left-hand element of a pair can be declared and axiomatized as in Fig. 2. Type parameters and bound type variables are introduced inside angle brackets. 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.

Instantiations of type parameters are inferred. Often, they can be inferred from the types of the a function's arguments. However, Boogie also allows a function to declare type parameters that are used only in the return type. For example, functions like *EmptySequence* in Fig. 2 are useful in many domains. In fact, the language even allows functions like *GimmieAnything* in Fig. 2, which illustrates that the type parameter can affect the result value of the function.

When type inference cannot determine type parameters uniquely, it reports an error. 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 $\mathit{Length}(\mathit{EmptySequence}()) = 0$ is ill-formed because of the ambiguous type-parameter instantiation; in contrast, the axiom about *Length* in Fig. 2 is well-typed and says that the length of any empty sequence is 0. Note that this quantifier ranges only over types, not over any individuals.

2.3 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.

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) [20]. For example, if m is a map of type $[\mathbf{int}, \mathbf{bool}]\,\mathit{Wicket}$, that is, a map type with domain types **int** and **bool** and result type *Wicket*, then the expression $m[5, \mathbf{false}]$ denotes a wicket.

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 [20]. Using common notation for arrays, 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. Where extensionality is needed, users can supply the required axioms themselves.

A key feature of maps in Boogie 2 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 [1, 16]. Since the result type of such a map depends on which field name is selected, it is natural to declare the heap to be of a polymorphic map type. For example, a source-language declaration **class** *Person* { **int** *age*; **bool** *isMarried*; } can be modeled in Boogie as shown in Fig. 3, where *Ref* denotes the type of all object references and $\mathit{Field}\ \alpha$

```
type Ref;                                 const unique age: Field int;
type Field α;                             const unique isMarried: Field bool;
type HeapType = ⟨α⟩[Ref, Field α] α;      var Heap: HeapType;
function IsWellFormedHeap(HeapType) returns (bool);
const unique snapshot: Field HeapType;
```

Fig. 3. Example of how to model object-oriented program features in Boogie 2. Type synonym *HeapType* is defined as the polymorphic map type that represents the heap.

denotes the type of field names that in the heap retrieve values of type α (see also [1, 16]). It is also common for functions to take such polymorphic maps as parameters, like function *IsWellFormedHeap* in Fig. 3 (an example of a higher-rank type). Type parameters may themselves be instantiated with polymorphic maps, which is known as impredicativity; for example, with *snapshot* from Fig. 3, the assignment $\mathit{Heap}[o, \mathit{snapshot}] := \mathit{Heap}$; will store a copy of the entire heap in the *snapshot* field of object reference o. As for functions, it is an error if the type parameters in a map type cannot be uniquely determined from the domain arguments and context.

Type equality among maps does not depend on the names or order of type parameters. For example, the type $\langle\alpha, \beta\rangle[\alpha, \beta]\textbf{int}$ is equal to $\langle\gamma, \delta\rangle[\delta, \gamma]\textbf{int}$. Note, however, that polymorphism is significant. For example, the types $[\textbf{int}]\textbf{bool}$ and $\langle\alpha\rangle[\alpha]\textbf{bool}$ are not compatible.

2.4 Equality Expressions

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 type checking 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 **old**(*Heap*), and its heap on return, written *Heap* (see, *e.g.*, [16]). For instance, to encode that a procedure's effect on the heap in the source language is limited to $p.\mathit{age}$ and $p.\mathit{isMarried}$, one can in Boogie use a postcondition like

$$\begin{aligned}(\forall \langle\alpha\rangle\ o\colon \mathit{Ref},\ f\colon \mathit{Field}\ \alpha \bullet\ & \mathit{Heap}[o,f] = \textbf{old}(\mathit{Heap}[o,f]) \lor \\ & (o = p \land f = \mathit{age}) \lor (o = p \land f = \mathit{isMarried})\ )\end{aligned}$$

In order to type check this expression, it is necessary for the type system to consider the possible instantiation $\alpha := \textbf{int}$ for $f = \mathit{age}$ and $\alpha := \textbf{bool}$ for $f = \mathit{isMarried}$. If the two sides of an equation evaluate to objects of different types, the equation as a whole evaluates to **false**, since different types in Boogie represent disjoint sets of individuals.

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

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

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

Fig. 4. The typing rules for Boogie 2 expressions. The context of type judgments is a partial mapping $\mathcal{V} : \mathcal{X} \rightharpoonup \mathit{Type}$ that assigns types to variables. The rules marked with '*' impose the side condition $\mathit{dom}(\sigma) = \{\bar{\alpha}\}$.

2.5 Matching Triggers

A common way for SMT solvers to handle universal quantifications is to selectively instantiate the quantifiers. The instantiations can be 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 [9]. Boogie has support for specifying matching triggers for quantifications. For example,

$$\textbf{axiom}\ (\,\forall\, x\colon\, t \bullet\;\; \{f(x)\}\ \mathit{fInverse}(f(x)) = x\ );$$

specifies the trigger $f(x)$ and says to instantiate the quantifier with any value appearing among the ground terms as an argument to function f. (For an application that uses quantifiers and an explanation of the design of triggers for that application, see [17].) All Boogie front ends make heavy use of triggers.

2.6 Formalization of the Type System and Type Checking

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

$$\mathit{Type}\ ::=\quad \alpha \quad|\quad C\ \mathit{Type}^* \quad|\quad \langle\alpha^*\rangle\,[\mathit{Type}^*]\ \mathit{Type}$$

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

For two types $s, t \in \mathit{Type}$, 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 $\sigma : \mathcal{A} \to \mathit{Type}$ from type variables to types. Substitutions are canonically extended on all types, whereby we assume that variable capture is avoided by renaming bound type variables whenever 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 4 shows the typing rules for variables, type coercions, function applications, equality, quantifiers, map select, and map update. All other operators are typed as in the rule for function application. In the figure and the whole paper, the set of functions and constants defined in the Boogie program at hand is denoted with $\mathcal{F}$, whereas $\mathcal{X}$ denotes an infinite set of variables.

Note that for a map m of a type like $\langle\alpha\rangle[\mathbf{int}]\alpha$, the expression $m[E]$ can have any type, and similarly for applications of functions like *GimmieAnything* in Fig. 2. As for the type system, Boogie just requires that the context determines a unique type for each occurrence of such expressions. As for their meaning, it is as if the map (or function) m is really a family of maps (or functions) $\bar{m}$. Thus, semantically, $m_{\mathbf{int}}[E]$ has a different value than $m_{\mathbf{bool}}[E]$.

3 Representation of Types

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 [3], which provides a polymorphic type system). When using such provers as a verification back-end, the expressions from the richer language have to be translated into the simpler logic. That is the subject of Section 4. The translation needs to take the richer types into account. For example, consider the following Boogie declarations, which define function *Mojo* for different types:

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

A translation that simply erases the types would not preserve the meaning of these axioms, because one would get $(\forall\, x \bullet\ \mathit{Mojo}(x) = x\ ) \wedge (\forall\, g \bullet\ \mathit{Mojo}(g) = 68\ )$, from which one can derive **false**. Therefore, we encode Boogie's types as terms in the simpler logic, and that is the subject of this section.

To ease the presentation, from now on we impose two simplifying assumptions on how type variables can be introduced (in Section 4.3, we discuss how to remove these assumptions): (i) each of the type parameters $\bar{\alpha}$ of a polymorphic function $f\langle\bar{\alpha}\rangle(\bar{s})$ **returns** (t) has to occur in the argument types $\bar{s}$ (*i.e.*, it is not enough that a type parameter occurs in the result type); and (ii) a map type $\langle\bar{\alpha}\rangle[\bar{s}]t$ is considered well-formed only if the parameters $\bar{\alpha}$ occur in the index types $\bar{s}$.

Preliminaries. Given two types $s, t \in \mathit{Type}$, 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 only differ in the names of free variables. The induced equivalence relation is denoted with $\cong$: for $s, t \in \mathit{Type}$, we define $s \cong t$ iff $s \sqsubseteq t$ and $t \sqsubseteq s$. Due to the definition of $\sqsubseteq$, it is the case that $\equiv\, \subseteq\, \cong$.

The pre-order $\sqsubseteq$ is canonically extended on $\mathit{TypeC} = \mathit{Type}/\!\cong$ and partially orders the set. In fact, $(\mathit{TypeC}, \sqsubseteq)$ is a join-semi-lattice 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 the type encoding in Section 3.1.

Type Encoding. 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, it is rather straightforward to translate expressions in this simply-typed language further into an untyped logic. We introduce a function symbol $\mathit{type} : U \to T$ 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. Here, we encode the algebraic datatypes by functions and axioms. Each type constructor $C \in \mathcal{C}$ gives rise to a function symbol $C^{\#} : T^{\mathit{arity}(C)} \to T$, 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 $\mathit{Ctor} \colon T \to \mathbf{int}$ and, for each type constructor C, a unique constant n_C. Injectivity is achieved by defining inverse functions $C^1, \ldots, C^n \colon T \to T$ for each n-ary type constructor C:

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

These are the axioms that are practically used in the Boogie 2 implementation. Theoretically, further axioms are needed for a faithful model of the type system, which are discussed in Section C in the appendix.

3.1 Representation of Types using Map Reduction

Besides ordinary type constructors, we must also encode Boogie's polymorphic map types, which is more difficult. Let $\mathcal{M}_C \subseteq \mathit{TypeC}$ 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:

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

The property of $\mathcal{M}$ that is central for us is the following: for every type t that starts with a map type constructor, there is a unique type $m = \mathrm{skel}(t) \in \mathcal{M}$ such that $t \sqsubseteq m$. 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 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 skel(t). For example, 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 the free type variables $\alpha_1, \ldots, \alpha_n$ we introduce a new n-ary function symbol $m_t^{\#} : T^n \to T$. We will use the notation $\mathrm{Skel}^{\#}(s) := m^{\#}_{\mathrm{skel}(s)}$ for the skeleton symbol of an arbitrary map type s, and $\mathrm{Skel}^{i}(s) := m^{i}_{\mathrm{skel}(s)}$ for the inverses. Given a mapping $\mu : \mathcal{A} \to \mathit{Term}$, types can be translated to terms:

$$\begin{aligned} [\![\alpha]\!]_\mu &= \mu(\alpha) \qquad\qquad [\![C\, t_1 \ldots t_n]\!]_\mu = C^{\#}(\ [\![t_1]\!]_\mu, \ldots, [\![t_n]\!]_\mu\ ) \\ [\![\langle\bar\alpha\rangle[\bar s]\, t]\!]_\mu &= \mathrm{Skel}^{\#}(\langle\bar\alpha\rangle[\bar s]\, t)(\ [\![\mathrm{flesh}(\langle\bar\alpha\rangle[\bar s]\, t)(\beta_1)]\!]_\mu, \ldots, [\![\mathrm{flesh}(\langle\bar\alpha\rangle[\bar s]\, t)(\beta_n)]\!]_\mu) \end{aligned}$$

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

$$\begin{aligned} [\![C\ T]\!]_\mu &= C^{\#}(T^{\#}) & [\![\,[\mathbf{int}]\ T]\!]_\mu &= m^{\#}_{[\alpha]\beta}(\mathit{int}^{\#}, T^{\#}) \\ [\![\,[T]\, S]\!]_\mu &= m^{\#}_{[\alpha]\beta}(T^{\#}, S^{\#}) & [\![\langle\alpha\rangle[\alpha]\, S]\!]_\mu &= m^{\#}_{\langle\alpha\rangle[\alpha]\beta}(S^{\#}) \end{aligned}$$

3.2 Symbols and Axioms of Maps with Map Reduction

The reduction of map types to ordinary type constructors entails that also the access functions *select* and *store* can be seen and axiomatised as ordinary functions, based on the axioms of the first-order theory of arrays [20]. 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:

$$\mathit{select}_m\langle\bar\alpha, \bar\beta\rangle(m, \bar s)\ \mathbf{returns}\ (t) \qquad \mathit{store}_m\langle\bar\alpha, \bar\beta\rangle(m, \bar s, t)\ \mathbf{returns}\ (m)$$

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 [20]:

$$\begin{aligned} &(\forall\, \langle\bar\alpha, \bar\beta\rangle\ h\colon m,\ \bar x\colon \bar s,\ z\colon t \bullet\ \ \mathit{select}_m(\mathit{store}_m(h, \bar x, z), \bar x) = z\ ) \wedge \\ &(\forall\, \langle\bar\alpha, \bar\alpha', \bar\beta\rangle\ h\colon m,\ \bar x\colon \bar s,\ \bar y\colon [\bar\alpha/\bar\alpha']\bar s,\ z\colon t \bullet \\ &\qquad\qquad \bar x = \bar y \vee \mathit{select}_m(\mathit{store}_m(h, \bar x, z), \bar y) = \mathit{select}_m(h, \bar y)\ ) \end{aligned}$$

4 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, and one that encodes type parameters of polymorphic functions as ordinary (additional) arguments. 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 [6]).

4.1 Translation using Type Guards

There is a long tradition of encoding type information using type guards, *e.g.*, [19, 6, 7]. As this translation is rather naive and has the disadvantage of complicating the propositional structure of formulae, it has been claimed [6] that its performance impact is prohibitive for many applications. We are able to show in Section 5, however, that this is no longer the case with state-of-the-art SMT solvers. The Mojo example given in Section 3 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.

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

Function Axioms. In the course of the translation, typed (user-defined) Boogie functions are replaced with U-typed functions. Suppose $f \in \mathcal{F}$ has the typing $\langle \alpha_1, \ldots, \alpha_m \rangle(\bar{s})$ **returns** (t). The corresponding function $f^{\#}$ has type $U^n \to U$. We will capture the original typing with an axiom that has the shape:

$$(\forall \bar{x}\colon \bar{U} \bullet\ \mathit{type}(f^{\#}(\bar{x})) = [\![t]\!]_{\mu}\ ) \tag{1}$$

Note that this axiom does not contain any quantifiers corresponding to the type parameters $\alpha_1, \ldots, \alpha_m$, which is advantageous for SMT solvers because the formula does not offer good triggers for these variables. Instead, the mapping $\mu : \mathcal{A} \to \mathit{Term}$ that determines the values of type parameters plays a prominent role. We define this mapping with the help of *extractor terms*, which are recursively defined over types and describe how the type parameter values can be reconstructed from the actual arguments $\bar{x}$. This is possible because, by our simplifying assumptions, each parameter α_i occurs in some argument type t_j.

Suppose that $\alpha \in \mathcal{A}$ is a type variable. Under the assumption such that the term E encodes the type $t \in \mathit{Type}$, the set $\mathit{extractors}_{\alpha}(E, t)$ specifies terms that compute the value of α:

$$\begin{aligned} \mathit{extractors}_{\alpha}(E, \beta) &= \textbf{if}\ \alpha = \beta\ \textbf{then}\ \{E\}\ \textbf{else}\ \emptyset \\ \mathit{extractors}_{\alpha}(E, C\, t_1 \ldots t_n) &= \bigcup_{i=1}^{n} \mathit{extractors}_{\alpha}(C^i(E),\ t_i) \qquad (C \in \mathcal{C}) \\ \mathit{extractors}_{\alpha}(E, \langle\bar{\beta}\rangle[\bar{s}]t) &= \bigcup_{i=1}^{m} \mathit{extractors}_{\alpha}(\mathrm{Skel}^i(\langle\bar{\beta}\rangle[\bar{s}]\, t)(E),\ \mathrm{flesh}(\langle\bar{\beta}\rangle[\bar{s}]\, t)(\gamma_i)) \end{aligned}$$

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

$$\begin{aligned} \mathit{extractors}_{\alpha}(x, C\,\beta\,\alpha) &= \{C^2(x)\} \\ \mathit{extractors}_{\alpha}(x, \langle\beta\rangle[C\,\beta\,\alpha]\,\alpha) &= \{C^2(m^1_{\langle\beta\rangle[C\,\beta\,\gamma]\,\delta}(x)),\ m^2_{\langle\beta\rangle[C\,\beta\,\gamma]\,\delta}(x)\} \end{aligned}$$

With this machinery, we can define $\mu : \mathcal{A} \rightarrow \mathit{Term}$ in (1) to be an arbitrary mapping that satisfies $\mu(\alpha_i) \in \bigcup_{j=1}^{n} \mathit{extractors}_{\alpha_i}(\mathit{type}(x_j), t_j)$ for $i \in \{1, \ldots, m\}$.

A simple optimization (that is implemented in Boogie 2 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 $\mathit{i2u} : \mathbf{int} \rightarrow U$, $\mathit{u2i} : U \rightarrow \mathbf{int}$, $\mathit{b2u} : \mathbf{bool} \rightarrow U$, $\mathit{u2b} : U \rightarrow \mathbf{bool}$ and axiomatize them as:

$$\begin{array}{l} (\forall\, x \colon \mathbf{int} \bullet\ \ \mathit{type}(\mathit{i2u}(x)) = \mathit{int}^{\#} \wedge \mathit{u2i}(\mathit{i2u}(x)) = x\ ) \wedge \\ (\forall\, x \colon U \bullet\ \ \mathit{type}(x) = \mathit{int}^{\#} \Rightarrow\ \mathit{i2u}(\mathit{u2i}(x)) = x\ ) \end{array}$$

and analogously for **bool**. The axioms imply that $\mathit{i2u}$ and $\mathit{b2u}$ are embeddings into U, and that $\mathit{u2i}$ and $\mathit{u2b}$ 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 2 implementation as well.

Translation of Expressions. The main cases of the translation are:

$$\begin{array}{rcll} [\![x]\!]_\mu & = & x & (x \in \mathcal{X}) \\ [\![f(E_1, \ldots, E_n)]\!]_\mu & = & f^{\#}([\![E_1]\!]_\mu, \ldots, [\![E_n]\!]_\mu) & \\ [\![E = F]\!]_\mu & = & \mathit{b2u}([\![E]\!]_\mu = [\![F]\!]_\mu) & \\ [\![E + F]\!]_\mu & = & \mathit{i2u}\big(\mathit{u2i}([\![E]\!]_\mu) + \mathit{u2i}([\![F]\!]_\mu)\big) & \cdots \\ [\![E \wedge F]\!]_\mu & = & \mathit{b2u}\big(\mathit{u2b}([\![E]\!]_\mu) \wedge \mathit{u2b}([\![F]\!]_\mu)\big) & \cdots \\ [\![(\,\forall\, \langle\bar{\alpha}\rangle\ \bar{x} \colon \bar{t} \bullet\ E\ )]\!]_\mu & = & (\,\forall\, \bar{x} \colon \bar{U},\ \bar{y} \colon \bar{T} \bullet\ \mathit{type}(\bar{x}) = [\![\bar{t}]\!]_{\mu'} \Rightarrow [\![E]\!]_{\mu'}\ ) & \\ [\![(\,\exists\, \langle\bar{\alpha}\rangle\ \bar{x} \colon \bar{t} \bullet\ E\ )]\!]_\mu & = & (\,\exists\, \bar{x} \colon \bar{U},\ \bar{y} \colon \bar{T} \bullet\ \mathit{type}(\bar{x}) = [\![\bar{t}]\!]_{\mu'} \wedge [\![E]\!]_{\mu'}\ ) & \end{array}$$

In the last two equations, $\bar{y}$ is a vector of fresh variables, and $\mu' = (\mu,\ \bar{\alpha} \mapsto \bar{u})$. 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 $\bar{x}$:

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

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

4.2 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. When using type arguments, the Mojo example from Section 3 gets translated as follows:

$$\begin{array}{ll} \textbf{function}\ \mathit{Mojo}(T, U)\ \textbf{returns}\ (U); & \textbf{axiom}\ (\forall\, x\colon U \bullet\ \mathit{Mojo}(\mathit{int}^{\#}, x) = x\ ); \\ \textbf{const}\ \mathit{GuitarPlayer}^{\#}\colon T; & \textbf{axiom}\ (\forall\, g\colon U \bullet\ \mathit{Mojo}(\mathit{GuitarPlayer}^{\#}, g) = \mathit{i2u}(68)\ ); \end{array}$$

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 in the course of the translation replaced by a corresponding 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 any 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.

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

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

The second equation assumes f has typing $\langle\bar{\alpha}\rangle[\bar{s}]t$ 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.

4.3 Extensions and Optimizations

Unrestricted Type Parameters. The simplifying assumptions made in the beginning of Section 3 require that all type parameters of functions and maps occur in the domain types. Without these restrictions, type checking of expressions becomes somewhat more involved and sometimes needs type coercions for disambiguation (see Section 2.2). Concerning the expression translation with type guards (Section 4.1), additional arguments have to be added to functions in the case of type parameters that are only used in the result type. For a function $f\langle\bar{\alpha}\rangle(s_1, \ldots, s_n)$ **returns** (t) such that k of the type parameters $\bar{\alpha}$ do not occur in $\bar{s}$ (but only in t), the post-translation function $f^{\#}$ is now given the typing $T^k \times U^n \to U$. The types of the map functions *select* and *store* and the axioms for function types have to be changed accordingly.

No changes are necessary for the translation using type arguments (Section 4.2), which already adds arguments for all type parameters to functions.

	Type Guards	Type Arguments	No Types
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

Fig. 5. 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).

Generated Types. The type guards introduced in Section 4.1 are sometimes valuable as triggers for e-matching. *E.g.*, consider a type whose domain is generated:

type *Color*;	**const unique** R, G, B: *Color*;
function f(**int**) **returns** (*Color*);	**axiom** $(\forall\, c\colon \mathit{Color} \bullet\ c = R \lor c = G \lor c = B\ )$;

In this form, the domain axiom does not contain any triggers that an SMT solver could use for e-matching. This is remedied by the translation from Section 4.1, which adds an antecedent $\mathit{type}(c) = \mathit{Color}^{\#}$ that provides a trigger (albeit a very unspecific one). Using the translation from Section 4.2, however, the axiom is translated to $(\forall\, c\colon U \bullet\ (c = R \land \mathit{Color}^{\#} = \mathit{Color}^{\#}) \lor\ \dots\ )$, so that no instantiation can be performed. In fact, for this translation it is essential that no instantiation is possible, because otherwise the SMT solver could conclude that the whole type U only has three elements (see [6] for a discussion).

Two solutions to this problem might be to (i) add type guards for generated types even when using the translation from Section 4.2; or to (ii) partially instantiate the domain axiom for each declared function symbol with result type *Color*, for instance: $(\forall\, x\colon \mathbf{int} \bullet\ f(x) = R \lor f(x) = G \lor f(x) = B\ )$.

5 Experimental Results

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 the overhead of Boogie 2 compared to Boogie 1 is measured. The Boogie programs of the last two categories are really BoogiePL (Boogie 1) programs and do not use polymorphism.

- *The Boogie regression test suite:* A collection of correct and incorrect programs written in Boogie, Spec# [1], and Dafny [16] that make use of polymorphism; also parts of the Boogie tool itself (a Spec# program) are included.
- *Hyper-V verification conditions generated by VCC [5]:* Boogie programs that stem from a project to verify the Microsoft hypervisor Hyper-V.
- *Benchmarks from the HAVOC tool [4]:* Regression tests and verification conditions to prove memory safety and invariants of various C programs.

For each of the three collections of programs, 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[3] on the altogether more than 10000 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 machine with 3.16GHz and 4GB.

Figure 5 summarizes the results, while more detailed diagrams about the time differences are shown in Section A in the appendix. As can be seen in the table, 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 declares a large number of functions as part of the generated Boogie programs, which leads to a large number of additional function axioms when using the type guard encoding.

6 Related Work

The intermediate verification language most closely related to Boogie 2 is Why [11], which offers ML-style polymorphism [22]. ML-style polymorphism is more limited than the higher-rank polymorphism in Boogie 2; for example, it does not allow polymorphic map types, nor does allow general quantifications over types.

Using translations similar to ours, Couchot and Lescuyer turn formulas with ML-style polymorphism into multi-sorted and untyped formulas [6], taking advantage of built-in theories. They have implemented their translations as modules of the Why tool [11] and report on some experiments. With Simplify [9], they measure a 200% slow-down with their version of a Type Guard translation, and a 300% slow-down with their other encoding. In comparison, we measure a slow-down of at most 95% with the type guards encoding and at most 45% with the type arguments encoding (in any of our benchmark categories).

Bobot *et al.* show how to incorporate ML-style polymorphism directly into an SMT solver [3]. The machinery they present is essentially that of our Type Arguments translation. It would be interesting to put to the test their conjecture that building in polymorphism in the prover rather than handling it through a translation 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 [21] 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 type-checked and possibly rejected afterwards. Similarly, Hurd [14] 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 [7] for the Mizar language.

[3] `http://research.microsoft.com/en-us/um/redmond/projects/z3/`

Conclusions

We have introduced the type system of Boogie 2, and we have shown how to translate its polymorphic types and expressions into first-order formulae suitable for SMT solvers. Our experimental data support the idea that including such advanced features in an intermediate verification language is both desirable for verifier front ends and feasible for performance. As future work, we would like to investigate further optimizations, such as monomorphization.

Acknowledgments. We thank Stephan Tobies and Shuvendu Lahiri for providing us with Boogie files from VCC and HAVOC for use as benchmarks. We also thank Michał Moskal, Nikolaj Bjørner, and Leonardo de Moura for useful discussions and for help with Z3, and Mike Barnett for making Boogie 2 available as a download at `research.microsoft.com/boogie`.

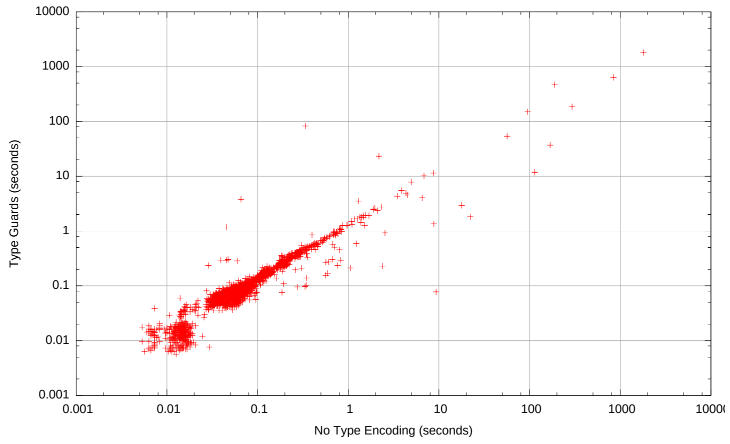
References

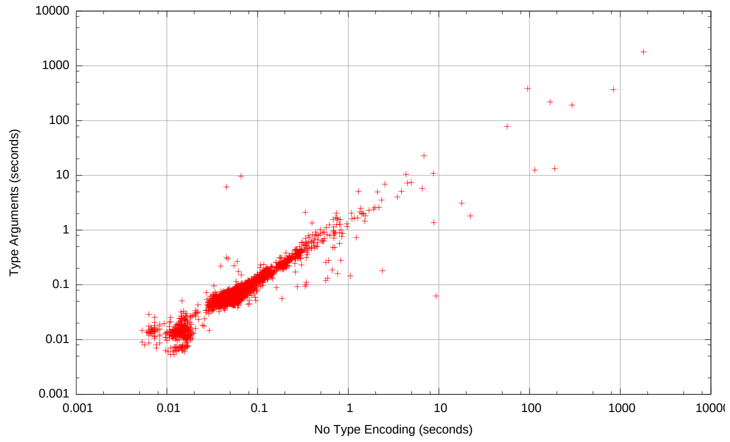
1. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, Sept. 2006.
2. C. Barrett, S. Ranise, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2008.
3. F. Bobot, S. Conchon, E. Contejean, and S. Lescuyer. Implementing polymorphism in SMT solvers. In *SMT 2008*, 2008.
4. S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamarić. A reachability predicate for analyzing low-level software. In *TACAS 2007*, pages 19–33, 2007.
5. E. Cohen, M. Moskal, W. Schulte, and S. Tobies. A practical verification methodology for concurrent programs. MSR-TR 2009-15, Microsoft Research, Feb. 2009.
6. J.-F. Couchot and S. Lescuyer. Handling polymorphism in automated deduction. In *CADE-21*, pages 263–278, 2007.
7. I. Dahn. Interpretation of a mizar-like logic in first-order logic. In *In FTP (LNCS Selection*, pages 137–151. Springer, 1998.
8. R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. MSR-TR 2005-70, Microsoft Research, Mar. 2005.
9. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, May 2005.
10. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, Dec. 1998.
11. J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, Mar. 2003.
12. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV '07*, pages 173–177, 2007.
13. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI 2002*, pages 234–245. ACM, May 2002.
14. J. Hurd. First-order proof tactics in higher-order logic theorem provers. In *Technical Report NASA/CP-2003-212448*, pages 56–68, 2003.
15. K. R. M. Leino. This is Boogie 2. Manuscript KRML 178, 2008. Available at `http://research.microsoft.com/~leino/papers.html`.
16. K. R. M. Leino. Specification and verification of object-oriented software. In *Summer School Marktoberdorf 2008*, NATO ASI Series F. IOS Press, 2009.

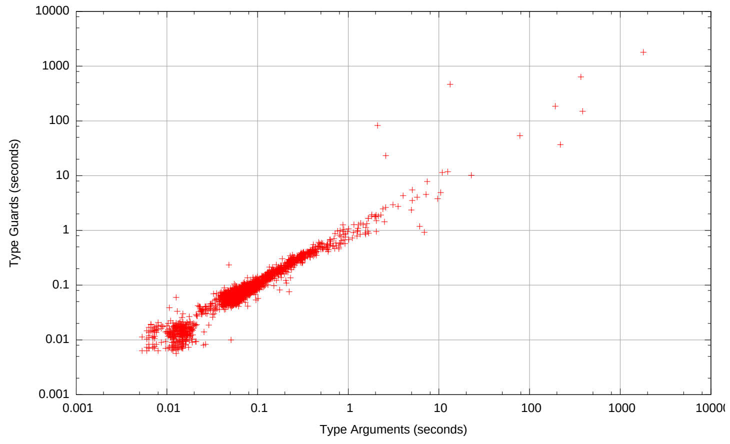
17. K. R. M. Leino and R. Monahan. Reasoning about comprehensions with first-order SMT solvers. In *SAC 2009*. ACM, Mar. 2009.
18. K. R. M. Leino, J. B. Saxe, and R. Stata. Checking Java programs via guarded commands. In *FTfJP 1999*, Tech. Rep. 251. Fernuniversität Hagen, May 1999.
19. M. Manzano. *Extensions of First-Order Logic.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1996.
20. J. McCarthy. Towards a mathematical science of computation. In *IFIP Congress 62*, pages 21–28. North-Holland, Aug.–Sept. 1962.
21. J. Meng and L. C. Paulson. Translating higher-order clauses to first-order clauses. *J. Autom. Reason.*, 40(1):35–60, 2008.
22. B. C. Pierce. *Types and Programming Languages.* The MIT Press, 2002.

A Detailed Benchmark Results

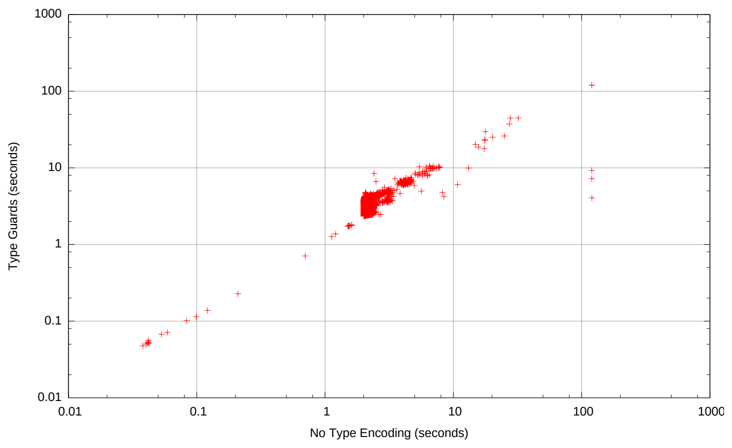
A.1 Boogie Regression Tests

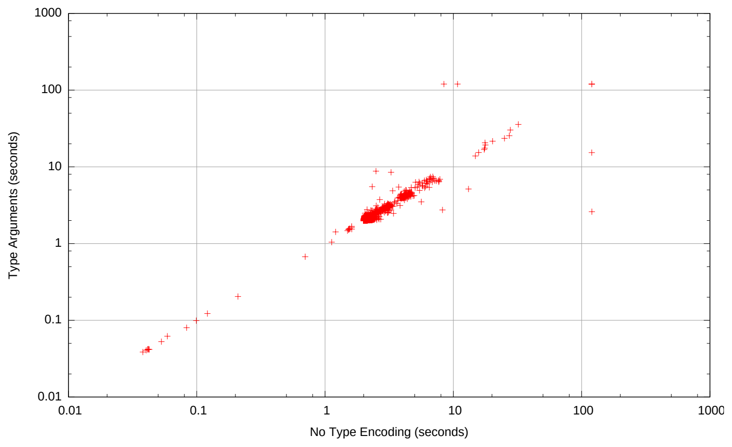


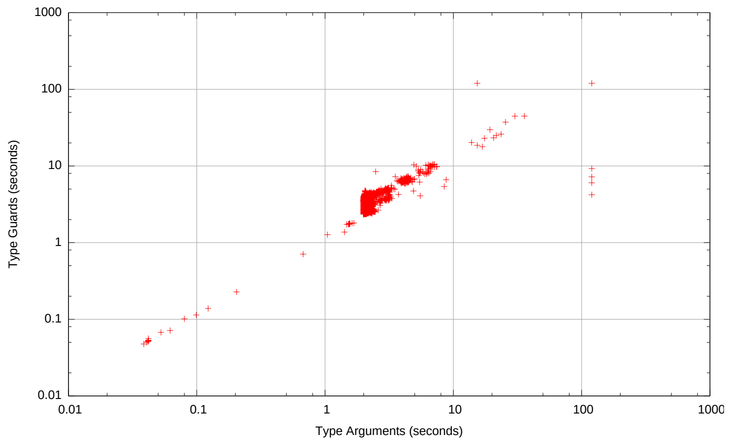




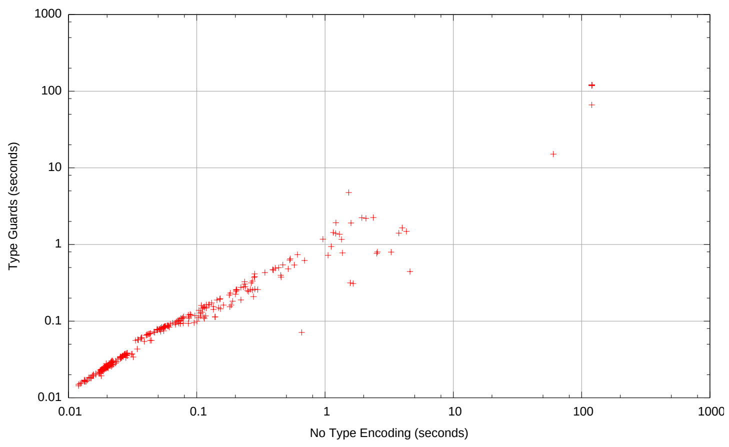
A.2 Hyper-V Verification Conditions Generated by VCC

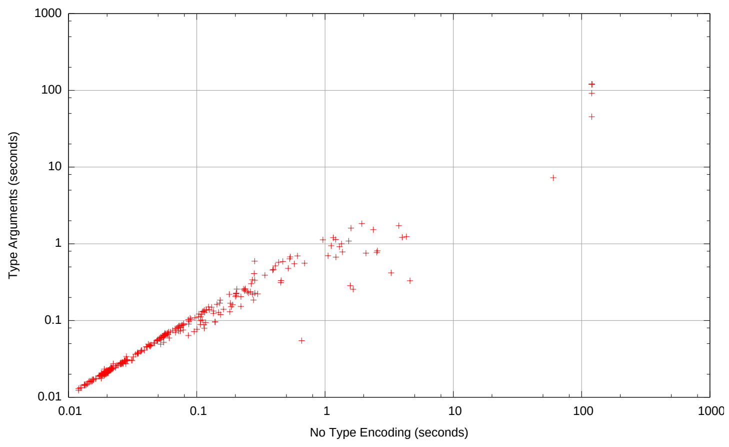


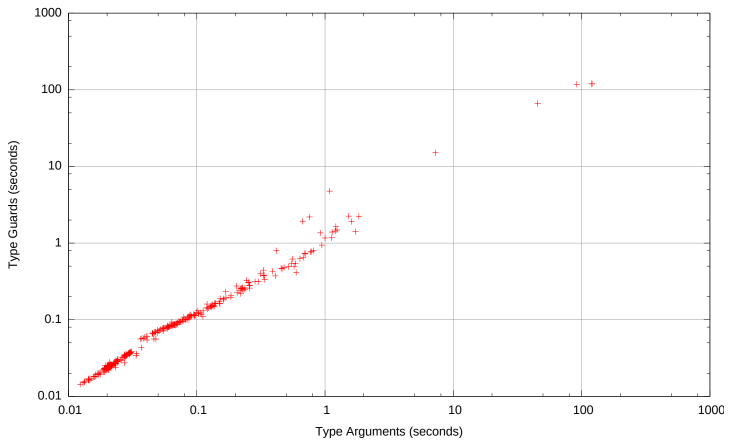




A.3 HAVOC Benchmarks







B Representation of Types using De Brujin Indices

We discuss an alternative representation of types that uses De Brujin indices[4] to encode the binding of type parameters is polymorphic map types. While De Brujin indices allow a rather concise type encoding, they also require quite complicated axioms or axiom schemas. In order to finitely axiomatise the typing of the map functions *select* and *store*, it is even necessary to describe the unification of types as axioms, which is in principle possible but severely slows down the theorem prover. Our implementation therefore uses the solution shown in Section 3.1.

For each $n > 0$, we introduce function symbol $map^n : \mathbf{int} \times T^{n+1} \to T$ to represent map types with n arguments; the first argument of map^n states the number of type parameters, and the last one the value type of the map. Variables are written in De Brujin notation using the unary function $tv : \mathbf{int} \to T$. For instance:

$$[S]\,T \;\rightsquigarrow\; map^1(0, S, T) \qquad\qquad \langle\alpha\rangle[\alpha]\,T \;\rightsquigarrow\; map^1(1, tv(0), T)$$

Translation of Types to Terms. To determine the De Brujin index of bound variables, during the translation a mapping $\beta : \mathcal{A} \to \mathbb{N}$ of variables to natural numbers is maintained. The translation of types is recursively defined as follows:

$$\begin{aligned}
[\![\alpha]\!]_\beta &= tv(\beta(\alpha)) && (\alpha \in \mathcal{A})\\
[\![C\, t_1 \ldots t_n]\!]_\beta &= C^{\#}([\![t_1]\!]_\beta, \ldots, [\![t_n]\!]_\beta) && (C \in \mathcal{C})\\
[\![\langle\bar{\alpha}\rangle[\bar{s}]t]\!]_\beta &= map^{len(\bar{s})}(len(\bar{\alpha}), [\![\bar{s}]\!]_{\beta'}, [\![t]\!]_{\beta'})
\end{aligned}$$

In the last equation, β' is derived from β by "prepending" the bound type variables $\bar{\alpha} = (\alpha_1, \ldots, \alpha_n)$:

$$\beta'(\gamma) \;=\; \begin{cases} \beta(\gamma) + n & \gamma \notin \{\alpha_1, \ldots, \alpha_n\} \\ i - 1 & \gamma = \alpha_i \end{cases}$$

Type Axioms. A number of properties of types have to be stated as axioms to ensure the adequacy of the translation: (i) only well-formed types exist (*e.g.*, the arguments of *tv* are non-negative), (ii) proper types do not contain free type variables, (iii) distinct type constructors construct different types, (iv) type constructors are injective, (v) types are well-founded, and (vi) the domains of proper types are nonempty. While some of these properties can be expressed by pure first-order formulae, properties such as generatedness and wellfoundedness cannot, and we therefore refer to the theory of integer arithmetic in those cases.

Due to lack of space, we do not give the axioms in this paper, but refer the reader to a future Technical Report instead.

[4] N. G. De Bruijn. Lambda Calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indagationes Mathematicae*, pages 381–392, 1972.

C Axioms for the Type Representation with Map Reduction

Distinct and Injective Constructors. The first group specifies that types that start with distinct type constructors are distinct. To this end, we assume an arbitrary but fixed enumeration of the countably infinitely many symbols $\{ C^{\#} : C \in \mathcal{C}\}$ in which each symbol $C^{\#}$ occurs at the n_C'th position, starting with 0. The numbers n_C are identified with the literals $0, 1, 2, \ldots$

$$\begin{aligned} \mathit{TA}_1 = & \quad (\,\forall\, x\colon T \bullet\ \mathit{Ctor}(x) \geq 0\ ) \\ \wedge & \quad \bigwedge_{C \in \mathcal{C}} (\,\forall\, x\colon T \bullet\ (\mathit{Ctor}(x) = n_C) = (\,\exists\, \bar{y}\colon \bar{T} \bullet\ C^{\#}(\bar{y}) = x\ )) \end{aligned}$$

The second group of axioms ensures that all type constructors are injective:

$$\mathit{TA}_2 \;=\; \bigwedge_{C \in \mathcal{C}} \bigwedge_{i=1}^{\mathit{arity}(C)} (\,\forall\, \bar{x}\colon \bar{T} \bullet\ C^i(C^{\#}(\bar{x})) = x_i\ )$$

Nonempty Domains. The third axiom ensures that the domains of types are nonempty:

$$\mathit{TA}_3 \;=\; (\,\forall\, x\colon T \bullet\ (\,\exists\, y\colon U \bullet\ \mathit{type}(y) = x\ ))$$

Type Casts. We define casts to and from the type U in order to integrate built-in types into our framework:

$$\begin{aligned} \mathit{TA}_4 = & \quad (\,\forall\, x\colon \mathbf{int} \bullet\ \mathit{type}(\mathit{i2u}(x)) = \mathit{int}^{\#} \wedge \mathit{u2i}(\mathit{i2u}(x)) = x\ ) \\ \wedge & \quad (\,\forall\, x\colon U \bullet\ \mathit{type}(x) = \mathit{int}^{\#} \Rightarrow \mathit{i2u}(\mathit{u2i}(x)) = x\ ) \\ \wedge & \quad (\,\forall\, x\colon \mathbf{bool} \bullet\ \mathit{type}(\mathit{b2u}(x)) = \mathit{bool}^{\#} \wedge \mathit{u2b}(\mathit{b2u}(x)) = x\ ) \\ \wedge & \quad (\,\forall\, x\colon U \bullet\ \mathit{type}(x) = \mathit{bool}^{\#} \Rightarrow \mathit{b2u}(\mathit{u2b}(x)) = x\ ) \end{aligned}$$

Well-founded Types. The fifth group of axioms ensures that types are well-founded, *e.g.*, for each n-ary type constructor C there is some type that cannot be reached by applying C to n other types. This is done by assigning a non-negative depth to every type (and to every non-type) that grows when type constructors are applied.

$$\begin{aligned} \mathit{TA}_5 = & \quad (\,\forall\, x\colon T \bullet\ \mathit{depth}(x) \geq 0\ ) \wedge \\ \wedge & \quad \bigwedge_{C \in \mathcal{C}} \bigwedge_{i=1}^{\mathit{arity}(C)} (\,\forall\, \bar{x}\colon \bar{T} \bullet\ \mathit{depth}(C^{\#}(\bar{x})) > \mathit{depth}(x_i)\ ) \end{aligned}$$