Preface

C and its cousin C++ are among the most popular languages. In contrast, the application of formal methods to C and C++ code are relatively rare. Holes in the type system, the frequent use of type casts and sometimes direct hardware access in C/C++ code make the development of formal methods very challenging.

The aim of the C/C++ verification workshop was to bring together people that are working on the verification or the semantics of C or C++ programs. The workshop provided a forum to discuss aspects of the type system and the semantics of C/C++, present approaches for the verification of C or C++ programs, demonstrate tools and report about (ongoing) verification projects.

The C/C++ verification workshop was organised as a side event to IFM 2007, the conference of Integrated Formal Methods. It took place in Oxford (UK) on July 2nd, 2007.

The organisation committee consisted of

Hendrik Tews, Radboud Universiteit Nijmegen, The Netherlands
Reiner Håhnl, Chalmers University of Technology, Sweden
Wolfgang J. Paul, Universität des Saarlandes, Germany
Ralf Huuck, NICTA, Australia
Norbert Schirmer, Technische Universität München, Germany

June 5, 2007

Hendrik Tews
# Table of Contents

Union and Cast in Deductive Verification ........................................ 1  
  
*Yannick Moy*

Reasoning on Data-Parallel Programs in Isabelle/HOL ......................... 17

*Matthias Daum*

How C differs from Java for Symbolic Program Execution ....................... 29

*Christoph D. Gladisch*

A Dynamic Logic for Deductive Verification of C Programs with KeY-C .......... 43

*Oleg Mürk, Daniel Larsson, and Reiner Hänle*

Formal Methods in the Robin project: Specification and verification of the Nova microhypervisor .............................................. 59

*Hendrik Tews*

Analysing Embedded System Software

«Extended Abstract» ................................................................... 69

*Ansar Fehrer, Ralf Huuck, Felix Rauch, and Sean Seefried*

Verification of C Programs Via Natural Semantics and Abstract Interpretation (Extended Abstract) ....................................... 75

*Roberto Bognara, Patricia M. Hill, Andrea Pescetti, and Enea Zaffanella*

DeHydra Source Analysis Tool ....................................................... 81

*Taras Glek*

Experiments in validating formal semantics for C .............................. 95

*Sandrine Blazy*

Olmar: manipulating C and C++ abstract syntax trees in Ocaml ........... 103

*Hendrik Tews*

**Author Index** ........................................................................ 115
Union and Cast in Deductive Verification *

Yannick Moy\textsuperscript{1,2,3}

\textsuperscript{1} France Télécom, Lannion, F-22307
\textsuperscript{2} INRIA Futurs, ProVal, Parc Orsay Université, F-91893
\textsuperscript{3} Lab de Recherche en Informatique, Univ Paris-Sud, CNRS, Orsay, F-91405

Abstract. Deductive verification based on weakest-precondition calculus has proved effective at proving imperative programs, through a suitable encoding of memory as functional arrays (a.k.a. the Burstall-Borot model). Unfortunately, this encoding of memory makes it impossible to support features like union and cast in C. We show that an interesting subset of these unions and casts can be encoded as structure subtyping, on which it is possible to extend the Burstall-Borot encoding. We present an automatic translation from C to an intermediate language Jessie on which this extended interpretation is performed.

1 Introduction

Deductive verification aims at checking behavioral properties of programs, where behaviors are formally specified by logical annotations: function preconditions and postconditions, assertions, loop invariants, global invariants, etc. Verification tools based on deduction target mainly Java, annotated with JML [14] or C#, annotated with the Spec# language [2]. These tools generate verification conditions (VC) from logical annotations given as stylized comments and program statements, using Hoare-style weakest precondition method. Proof of these verification conditions can be automated in most cases using automatic theorem provers, or otherwise performed manually in an interactive theorem prover. Our background framework is given by the WHY/Krakatoa/Caduceus family of tools [11] for the deductive verification of C and Java programs.

It is common in deductive verification to model the heap as a set of functional arrays indexed by pointers, also known as the Burstall-Borot memory model [3, 10]. This model relies on the strong assumption that memory is strongly typed, as in a Java-like language where objects are allocated on the heap in a typed-way (using \texttt{new}) and accessed only through their original type. This model makes it difficult to consider unions and casts which occur frequently in C, because these features allow reinterpreting typed memory through a different type. However, it is sometimes desirable to use union or cast and possible to do it in a safe way, as shown by the secure implementation for Managed Strings by CEKT [4]. Safe use of union mandates that reading a variant field is only allowed if it was the last field written through. Safe use of cast relies on the layout of structures in memory and will be explained in Section 4.

In this paper, we introduce support for safe union and cast in deductive verification. Our aim is to support as much as possible those unions and casts that do not lead to

\* This research is partly supported by CIFRE contract 2005/973 with France Télécom company, and ANR RNTL "CAT"
reinterpreting memory at the byte level, but are used to simulate higher level constructs like discriminated unions or inheritance in a low level language like C. Considering unions as a special case of cast, these inheritance-related casts account for 99% of casts in C programs, according to [7].

We introduce the intermediate language Jessie as a cast-free, union-free subset of C, with added support for structure subtyping, similar to Java class subtyping. We show how to transform C programs with the kind of casts and unions that we target into Jessie programs, on which we perform deductive verification.

The remaining of this paper is organized as follows. Section 2 presents the Jessie language. Section 3 and Section 4 describe respectively our support of union and cast. Section 5 reports on our current implementation and Section 6 discusses related work. We finally conclude in Section 7.

2 From C to Jessie

Jessie is an intermediate language for the deductive verification of Java and C. To some extent, it incorporates C features not present in Java (e.g., pointer arithmetic) and Java features not present in C (e.g., exception mechanism). Jessie does not incorporate the low-level platform-dependent C features (e.g., pointer cast) as well as the dynamic method call of Java. Overall, this intermediate language is designed for being suitable for deductive verification and a target of translation from C and Java. Logical annotations are part of the language.

2.1 Illustrative Example

The following program illustrates the most important features in Jessie, with keywords in bold type.

```java
type Point = { real x; real y; }

type ColorPoint = Point with {
    integer col;
    invariant color(this) = this.col >= 0 && this.col < 256;
}

unit darkens(Point[0] p) {
    if (p <: ColorPoint) {
        ColorPoint[0] q = p => ColorPoint:
        q.col = q.col / 2;
    }
}
```

First, it is possible to define subtypes of a parent type, using the keyword with. Subtypes inherit all their parent type fields, like in usual object-oriented languages. Arithmetic types integer and real are mathematical infinite precision integers and reals.

Logical annotations can be added in structures as type invariants or in functions as preconditions and postconditions. Defining exactly when a type invariant holds, when it
may be violated and when it must be restored is an orthogonal problem. This should be dealt with a type invariant system, like the one in JML [13] or Boogie [2], the simplest of which adds parameter type invariants as function preconditions and postconditions. We assume we have one such invariant system in this paper.

In the following, we call object a value of a pointer type. By extension, we call object a value of type pointer in C, whose translation is a Jessie object. In Jessie, one may precise a pointer type with a range of valid indices, e.g., Pointer[0] is a pointer that can be safely dereferenced. An instance-of operator, denoted <::, allows testing the type of an object in annotations. Coercions, denoted ->, allows viewing an object of a type as an object of another type in the same hierarchy. If it is an upcast, from a subtype to a parent type, it can be used freely. If it is a downcast, from a parent type to a subtype, a VC is generated to prove its correctness.

C program statements are translated to Jessie statements and C logical annotations to Jessie logical annotations. Our goal is to prove C programs correct, i.e., that they follow their specification and they do not lead to errors (e.g., memory errors).

2.2 Memory Model

The weakest precondition calculus is performed using a model of memory based on functional arrays. This means that a field access $e_i$ is interpreted as select($f$, $e_i$) and a field update $e_i = e'$ is interpreted as store($f$, $e_i$, $e'$) where select and store satisfy the theory of functional arrays:

\[
\begin{align*}
select(update(f, p, v), p) &= v, \\
select(update(f, p, v), q) &= select(f, q) \text{ if } p \neq q.
\end{align*}
\]

The interpretation language we use is strongly typed. The type of a pointer $p$ has the form $S$ pointer where $S$ denotes the root of the Jessie type hierarchy to which $p$ source type belongs. On our example, both pointers to Point and ColorPoint are interpreted as $Point$ pointer. A structure field of type $t$ in any structure of $S$ hierarchy has type $(t, S)$ memory. E.g., field $col$ of structure ColorPoint has type $(integer, Point)$ memory. Functions select and store have polymorphic types:

\[
\begin{align*}
select : (\tau, \sigma) \text{ memory, } \sigma \text{ pointer } \rightarrow \tau, \\
update : (\tau, \sigma) \text{ memory, } \sigma \text{ pointer, } \tau \rightarrow (\tau, \sigma) \text{ memory}.
\end{align*}
\]

To model the type hierarchy, we first introduce a logical type tag. Each source type in Jessie has a corresponding symbol of type tag that represents it. More precisely, for each type hierarchy with root $S$ and each structure name $T$ in that hierarchy, a symbol $T$ tag of type $S$ tag represents source type $T$. E.g., Point_tag and ColorPoint_tag are symbols of type Point_tag that represent respectively source types Point and ColorPoint. Dynamic types of pointers in hierarchy $S$ are represented by a functional array $S$ tag table which associates a $S$ tag to each $S$ pointer. A logical predicate instanceof of type $(\sigma$ tag table, $\sigma$ pointer, $\sigma$ tag) $\rightarrow$ boolean models the operator $: >$. For each type subtype $T'$ of $T$, an axiom is introduced:

\[
\text{instanceof}(a, p, T') \Rightarrow \text{instanceof}(a, p, T).
\]
Casting is just the identity function with a VC. Statement \( p \colon T \) is equal to \( p \) under the assumption that \( p \colon T \). Stated otherwise, \( p \colon T \) is interpreted into:

\[
\text{assert(instanceof(S_tag_table,p,T_tag)); p.}
\]

This axiomatization is simple enough that the VC generated are usually handled by automatic theorem provers. It is the case for our illustrative example, with any of the three automatic theorem provers we use: Simplify [8], Ergo [6] and Yices [9].

3 Union As Subtyping

3.1 Motivating Example

Fig. 1 shows a typical example of a C program where union feature is used in a principled way to simulate a discriminated union. An implicit invariant of type packet is that whenever field \( \text{discr} \) is zero, the variant field \( vall \) of field \( \text{uni} \) is set, and whenever field \( \text{discr} \) is one, the variant field \( \text{val2} \) of field \( \text{uni} \) is set. We manually introduce the following logical annotations as stylized comments in packet definition to denote just that, where \( \text{variantof}(p,i) \) means that pointer \( p \) of a union type \( t \) has currently variant \( f \) of \( t \) set.

```c
struct packet {
    [...]  //\@ invariant discr0(this) = this->discr == 0 ⇒ variantof(this->uni,vall)
    //\@ invariant discr1(this) = this->discr == 1 ⇒ variantof(this->uni,val2)
}
```

>From this annotated C program, it is possible to extract the corresponding Jessie program shown in Fig. 2. Before we describe how to do it automatically, we first do it step-by-step to show how our translation proceeds.

The natural way to see the relation between a union type and its variants in Jessie is through the subtyping relation. Thus, we translate a union type into an empty structure type and each of its variants as a subtype of the empty structure type just defined. In our example, we define an empty type \( \text{packet Uni} \) to represent the union, and two subtypes of \( \text{packet Uni} \) to represent the variants of the union.

```c
type packet uni = {
}

type packet uni vall = packet uni with {
    integer vall;
}

type packet uni val2 = packet uni with {
    real val2;
}
```

Type \( \text{packet} \) is then just a record with a \( \text{discr} \) field of base type \( \text{integer} \) and a \( \text{uni} \) field of object type \( \text{packet Uni} \). Type invariants \( \text{discr}0 \) and \( \text{discr}1 \) are also translated to reflect translated types.
struct packet {
  int descr;
  union {
    int val1;
    float val2;
  } uni;
};
float get(struct packet *p) {
  switch (p->descr) {
    case 0: return p->uni.val1;
    case 1: return p->uni.val2;
  }
}

Fig. 1. Discriminated Union in C

type packet_uni = {
}
type packet_uni_val1 = packet_uni with {
  integer val1;
}
type packet_uni_val2 = packet_uni with {
  real val2;
}
type packet = {
  integer descr;
  packet_uni[0] uni;
  invariant descr0(this) = this.descr == 0 ⇒ this.uni <: packet_uni_val1;
  invariant descr1(this) = this.descr == 1 ⇒ this.uni <: packet_uni_val2;
}
real get(packet[0] p)
  requires descr0(p) && descr1(p);
  ensures descr0(p) && descr1(p);
  {
    switch (p.descr) {
      case 0: return (p.uni > packet_uni_val1).val1;
      case 1: return (p.uni > packet_uni_val2).val2;
    }
  }

Fig. 2. Discriminated Union in Jessie
type packet {
  integer discr;
  packet_uni[0] uni;
  invariant discr0(this) = this.dscr == 0 ⇒ this.uni <: packet_uni_val1;
  invariant discr1(this) = this.dscr == 1 ⇒ this.uni <: packet_uni_val2;
}

Function get can then be translated with the appropriate coercion where accessing a field of the union. We also add the function preconditions and postconditions that are generated with a naive type invariant system. The result of the translation is shown in Fig. 2. On this Jessie program, our VC generator produces 11 VC that are all proved automatically.

3.2 General Translation Scheme

Although C type invariants are necessary for this proof to succeed, they are not needed for the translation from C to Jessie. The automatic translation goes as follows:

1. Translate each union type as a different empty type.
2. Translate variants of a union type as subtypes of its translation.
3. Translate union field reads as coercions to the appropriate type. E.g., \( x = u => f \) becomes \( x = (u : ⇒ T).f \). This applies to both program statements and logical annotations, and leads to the generation of VC later on.
4. Translate union field writes as coercions to the appropriate type. E.g., \( u => f = x \) becomes \( (u : ⇒ T).f = x \). The new operator \( ⇒ \) is the strong coercion operator. It applies only to program statements, and does not lead to the generation of VC. Application of the strong coercion changes the dynamic type of pointer \( u \), and may invalidate fields of \( u \) previously available.

It must be noted that we do not consider here uses of the address-of operator, as well as union assignment (when union is assigned as a whole). Section 5 reports on our current work to support those constructs.

Additionally, when there are type invariants, they need to be translated too. In particular, the notation \( \text{variantof} \) is translated to the instance-of operator. The second argument to \( \text{variantof} \) should be a field of a union. It is translated into the appropriate subtype that represents this variant in Jessie.

With this simple translation scheme, it is the programmer’s responsibility to provide the necessary logical annotations that account for a safe use of union. Ideally, only the type invariant needs to be written, as in our example. All accesses to union fields should be proved correct by extracting from the program text the conditions that ensure a safe access, through weakest precondition computation. In more complex cases, it may be necessary to add function preconditions and postconditions, as well as loop invariants.

4 Cast As Physical Subtyping

4.1 Motivating Example

Casts provide C programmers with additional flexibility. E.g., the program in Fig. 3 implements the same functionality as the one in Fig. 1, except types \( \text{packet1} \) and \( \text{packet2} \)
struct header {
  int discr;
};
struct packet1 {
  int discr;
  int val1;
};
struct packet2 {
  struct header head;
  float val2;
};
float get(struct header *p) {
  switch (p->discr) {
    case 0: return ((struct packet1*) p)->val1;
    case 1: return ((struct packet2*) p)->val2;
  }
}

Fig. 3. Discriminated Cast in C

type header = {
  integer discr;
  invariant discr0(this) = this.discr == 0 ⇒ this < packet1;
  invariant discr1(this) = this.discr == 1 ⇒ this < packet2;
}
and type packet1 = header with {
  integer val1;
}
and type packet2 = header with {
  real val2;
}
real get(header[0] p) {
  requires discr0(p) && discr1(p);
  ensures discr0(p) && discr1(p);
  switch (p.discr) {
    case 0: return (p ⇒ packet1).val1;
    case 1: return (p ⇒ packet2).val2;
  }
}

Fig. 4. Discriminated Cast in Jesse
seem unrelated in this version with cast, whereas they corresponded to variants of the same union in the version with union. However, in this version too, casts are used in a principled way to simulate a discriminated union. This is what we call discriminated casts in C. Again, there is an implicit typing invariant for objects of type header: whenever the field discr of an object is zero, the object is of type packet1, and whenever the field discr of an object is one, the object is of type packet2. This can be made explicit using logical annotations as follows, where typeof(p,t) means that object p has type t.

```c
struct packet1; // forward declaration
struct packet2; // forward declaration
struct header {
  int discr;
  //@ invariant discr0(this) = this->discr == 0 ⇒ typeof(this,struct packet1)
  //@ invariant discr1(this) = this->discr == 1 ⇒ typeof(this,struct packet2)
}
```

>From this annotated C program, it is possible to extract the corresponding Jessie program shown in Fig. 4. Before we describe how to do it automatically, we first do it step-by-step to show how our translation proceeds.

In the version with cast, it is not obvious which type should be root of the hierarchy. More generally, the form of the type hierarchy must be extracted from the program. In our example, types packet1 and packet2 can be seen as subtypes of type header, for different reasons. The sequence of field types of type header is a prefix of the sequence of field types of type packet1. Although this is not mandated by the C standard [12] (contrary to a common belief, e.g., see [16]), but according to all application binary interfaces (ABI) implemented in C compilers we know of, their layout on this common prefix is the same. Therefore, it is safe\(^4\) to view an object of type packet1 as an object of type header, which expresses a subtyping relation. For type packet2, things are simpler, as the standard mandates (6.7.2.1 §13) that an object of type packet2 can be safely casted into an object of its first field type header. Again, this expresses a subtyping relation. Both subtyping relations rely on some physical correspondence at the byte-level, which is also known as physical subtyping [18,5]. Finally, in Jessie, type header has two subtypes packet1 and packet2. To allow the translation of type invariant in type header to refer to its subtypes, we make all three types mutually recursive.

```c
typedef header = {
  integer discr;
  //@ invariant discr0(this) = this->discr == 0 ⇒ this < packet1;
  //@ invariant discr1(this) = this->discr == 1 ⇒ this < packet2;
}
```

```c
and typedef packet1 = header with {
  integer val;
}
```

\(^4\) We do not consider here the possibility for the compiler to optimize code by assuming a restricted form of aliasing, based on types. E.g., we forbid use of gcc with option -fstrict-aliasing.
and type packet2 = header with {
    real val2;
}

Translation of function get is very close to the one already seen in the version with union. The result of the translation is shown in Fig. 4. Like in the version with union, our VC generator produces 11 VC for this program to be correct, which are proved automatically.

4.2 General Translation Scheme

In order to automate the translation for casts, we define formally our notion of physical subtyping. We call base types in C all types that are not C structures. An object may be of structure type or base type. An object of structure type has sub-objects, that are pairs consisting in an offset from the beginning of the object where the sub-object begins and a type, the type of the sub-object. Sub-objects may themselves have sub-objects. We are interested in the set of sub-objects of base type contained transitively in an object. If the object is of base type, we include it in the set, at offset zero. We will call sub-objects of a type the sub-objects of an object of this type.

Definition 1. The layout of type $t$ is the set of its base type sub-objects.

It must be noted that the layout of a type is platform-dependent, as it depends, among other things, on the size and alignment of types that are themselves platform-dependent in C.

Example 1. The layout of type header is $(0,\text{int})$ on every platform. On the contrary, the layout of types packet1 and packet2 can vary according to the platform. E.g., for packet1, it is of the form $(0,\text{int}),(\text{off},\text{int})$ for some integer offset $\text{off}$.

Definition 2. A type $t'$ is a physical subtype of type $t$ if the layout of $t$ is a subset of the layout of $t'$.

This ensures accesses to a base type sub-object in memory through type $t$ are also valid accesses if the original type of the object was type $t'$. Of course, the syntax for the access may not be the same, e.g., field names may differ. Contrary to the work of [5], we do not rely on the actual layout of fields in memory on a particular platform, but on their order and type. More precisely, we assume there is a simple algorithm to compute the layout of a structure.

Definition 3. A layout algorithm takes as input a list of types and returns a layout.

We assume there exists such a layout algorithm $\lambda$ taking as input the list of field types of a structure, that we sketch in Fig. 5. Size and alignment of types are manipulated symbolically, which makes our analysis platform-independent. Function $\text{sizeof}$ returns the size of a base type. Function $\text{alignof}$ returns the alignment of a base type, while we assume for simplicity that $\text{alignof}(\text{struct})$ is the common alignment of structures, which is also the maximum alignment allowed. We could easily adopt a finer
\begin{enumerate}
\item \textbf{init} current-size to 0
\item \textbf{init} current-layout to empty set
\item \textbf{define} \( A \) on input type-list :
\begin{enumerate}
\item increase current-size to multiple of alignof(struct)
\item for each \( t \) in type-list do
\begin{enumerate}
\item align = \textbf{if} \( t \) is a structure \textbf{then} alignof(struct) \textbf{else} alignof(t) \textbf{fi}
\item increase current-size to multiple of align
\item \textbf{if} \( t \) is a structure \textbf{then}
\begin{enumerate}
\item call \( A \) on the field types of \( t \)
\item \textbf{else}
\begin{enumerate}
\item add (current-size, \( t \)) to current-layout
\item increase current-size by sizeof(t)
\end{enumerate}
\end{enumerate}
\end{enumerate}
\item \textbf{fi}
\item increase current-size to multiple of alignof(struct)

\end{enumerate}
\item call \( A \) on field types of input structure
\end{enumerate}

\textbf{Fig.5.} Layout algorithm \( A \)

The notion of structure alignment, e.g., to account for smaller structure alignment on structures that have only \texttt{short} and \texttt{char} fields, as in gcc for x86 [16].

The outermost call to \( A \) on type \( t \) returns when \textit{current-layout} contains the layout of \( t \) and \textit{current-size} the size of \( t \). \( A \) calls itself recursively on line 9 on sub-structures. Its first and last operations on lines 4 and 15 consist in increasing if necessary the current size to a multiple of the structure alignment. The increase is kept as small as possible, as on line 7. Its body is a loop (lines 5 to 14) that considers each type \( t \) in the list, increases the current size to fit the alignment of \( t \) (line 7) and either calls \( A \) recursively if \( t \) is a structure (line 9) or adds the new basic type sub-object to the layout (line 11) before increasing the current size by the size of \( t \) (line 12).

\textbf{Example 2.} Assume the following platform-dependent requirements:

\[ \text{sizeof(int)} = \text{alignof(int)} = 4, \text{alignof(float)} = 4, \text{alignof(struct)} = 4. \]

On this platform, applying algorithm \( A \) gives the actual layouts of types \texttt{header}, \texttt{packet1} and \texttt{packet2}. The layout of \texttt{header} is \((0, \text{int})\), the layout of \texttt{packet1} is \((0, \text{int}), (4, \text{int})\) and the layout of \texttt{packet2} is \((0, \text{int}), (4, \text{float})\).

Consider now a different set of platform-dependent requirements:

\[ \text{sizeof(int)} = \text{alignof(int)} = 4, \text{alignof(float)} = 4, \text{alignof(struct)} = 8. \]

On this other platform, the layout of \texttt{header} is still \((0, \text{int})\), the layout of \texttt{packet1} is still \((0, \text{int}), (4, \text{int})\), but the layout of \texttt{packet2} changes to \((0, \text{int}), (8, \text{float})\). \( \square \)

Given a cast from pointer type to pointer type, we need to determine which of the pointed types is subtype of the other. \( A \) cannot be used directly, as we do not know the exact size and alignment of types. In the following, we build a layout algorithm from \( A \) which makes it possible to decide physical subtyping in a platform-independent way.
Let $t$ and $t'$ be the pointed types. We write $\{t_1\ldots t_n\}$ the type of a structure with field types $t_1$ to $t_n$ in that order. We define a dummy type $t_s$ to stand for the beginning and end of structures, denote $tl$ a list of types, and $\varepsilon$ the empty list of types. We use pattern-matching in the rules we present below, with the rules that appear first being applied first.

Function $\text{flatten}$ takes as input a type and returns a list of types (including pseudo-type $t_s$) that represent the input type. Commas are used to separate types in a list.

\[
\text{flatten}(\{t_1\ldots t_n\}) = t_s, \text{flatten}(t_1)\ldots \text{flatten}(t_n), t_s
\]

\[
\text{flatten}(t) = t
\]

Function $\text{stutter}$ removes repeated occurrences of pseudo-type $t_s$.

\[
\text{stutter}(t_s,t_s,tl) = \text{stutter}(t_s,tl)
\]

\[
\text{stutter}(t,tl) = t, \text{stutter}(tl)
\]

\[
\text{stutter}(\varepsilon) = \varepsilon
\]

Function $\text{trim}$ removes occurrences of $t_s$ at the beginning and end of a list of types.

\[
\text{trim}(t_s,tl) = \text{trim}(tl)
\]

\[
\text{trim}(tl,t_s) = \text{trim}(tl)
\]

\[
\text{trim}(tl) = tl
\]

Function $\text{prefix}$ takes two lists of types in argument (separated by a colon). It returns the common prefix of these two lists.

\[
\text{prefix}(t,tl ; t,t') = t, \text{prefix}(tl,t')
\]

\[
\text{prefix}(tl ; t,t') = \varepsilon
\]

We denote the subtyping relation $\triangleleft$, with the interpretation that $t \triangleleft t'$ means that $t'$ is a subtype of $t$. First we define a type list $tl$ attached to $t$ and a type list $tl'$ attached to $t'$. Using these we can define $\triangleleft$

\[

tl = \text{trim}(\text{stutter}(\text{flatten}(t)))
\]

\[

tl' = \text{trim}(\text{stutter}(\text{flatten}(t')))
\]

\[
t \triangleleft t' \text{ iff } \text{prefix}(tl ; tl') = tl \text{ and } tl \neq tl'
\]

In order to prove that relation $\triangleleft$ is the expected physical subtyping relation, we introduce the notion of layout type list and prove some properties of functions $\text{flatten}$, $\text{stutter}$ and $\text{trim}$.

**Definition 4.** A layout type list for a type $t$ is a list of types for which there exists a layout algorithm returning the layout of $t$.

The list of field types of $t$ is a layout type list for $t$, since algorithm $\mathcal{A}$ applied to it returns the layout of $t$.
**Lemma 1.** \(\text{flatten}(i)\) is a layout type list for \(t\).

**Proof.** Given the layout algorithm \(A\) associated to the list of field types of \(t\), we build a layout algorithm \(A'\) associated to \(\text{flatten}(i)\). Consider every type in the list in turn. When the current type is not \(t_5\), do as in the body of \(A\). When the current type is \(t_5\), increase the current size so that it is a multiple of the structure alignment. The only difference between \(A\) and \(A'\) is that \(A\) increases the current size to a multiple of the structure alignment at recursive calls entry and exit, which correspond to the occurrences of \(t_5\) in \(\text{flatten}(i)\). Thus, algorithm \(A'\) applied to \(\text{flatten}(i)\) produces the same layout and size as algorithm \(A\) applied to the list of field types of \(t\). □

**Lemma 2.** \(\text{stutter}(\text{flatten}(i))\) is a layout type list for \(t\).

**Proof.** The layout algorithm \(A'\) associated to \(\text{flatten}(i)\) is also a layout algorithm associated to \(\text{stutter}(\text{flatten}(i))\). Indeed, calling \(A'\) repeatedly on \(t_5\) does not increase the current size after the first call, since it is already a multiple of structure alignment. □

**Lemma 3.** \(\text{trim}(\text{stutter}(\text{flatten}(i)))\) is a layout type list for \(t\).

**Proof.** Given the layout algorithm \(A\) associated to \(\text{stutter}(\text{flatten}(i))\), we build a layout algorithm \(A''\) associated to \(\text{trim}(\text{stutter}(\text{flatten}(i)))\). \(A''\) simply calls \(A\), but it does not return the size of the structure together with its layout. Initial call to \(A'\) on \(t_5\) does not increase the current size, since it is zero, which is a multiple of structure alignment. Final call to \(A'\) on \(t_5\) might increase the current size, but \(A''\) ignores it, therefore removing \(t_5\) at the end of the list has no effect. □

**Theorem 1.** Subtyping relation \(\prec\) is the most precise platform-independent strict physical subtyping relation, assuming a layout algorithm \(A\).

**Proof.** We show first that the assumptions we made in \(A\), \(\prec\) defines a strict physical subtyping relation, i.e., whenever \(t \prec t'\), \(t'\) is a strict physical subtype of \(t\). Take \(tl\) and \(tl'\) the lists of types obtained from \(t\) and \(t'\) as in the definition of \(\prec\). According to lemma 3, \(tl\) is a layout type list for \(t\) and \(tl'\) is a layout type list for \(t'\), with associated layout algorithm \(A''\). Therefore, \(A''\) applied to \(tl\) returns all base type sub-objects of \(t\), and \(A''\) applied to \(tl'\) returns all base type sub-objects of \(t'\). Since \(tl\) is a prefix of \(tl'\), algorithm \(A''\) applied to \(tl'\) returns a superset of the layout of \(t\). Since \(tl\) is a strict prefix of \(tl'\), and the difference cannot be only pseudo-types \(t_5\) that would have been removed by \(\text{trim}\), algorithm \(A''\) applied to \(tl'\) returns a strict superset of the layout of \(t\). Finally, \(t'\) is a strict physical subtype of \(t\).

We then show that \(\prec\) is the most precise strict physical subtyping relation given the assumptions we made in \(A\). Suppose the contrary. Take \(t\) and \(t'\) such that \(t'\) is a strict physical subtype of \(t\), according to our assumptions, but \(t \not\prec t'\) does not hold. If \(tl = tl'\), it is trivial that \(\text{prefix}(tl ; tl') = tl\), and we reach a contradiction. Therefore, \(tl \neq tl'\) and \(\text{prefix}(tl ; tl') \neq tl\). Then, \(t_5\) and \(t_5'\) share a common prefix, and differ on a first type \(t_1\) for \(tl\) and \(t_2\) for \(tl'\). If \(tl\) and \(tl'\) differ on their first element, it must be a real type, because \(t_5\) would have been removed by \(\text{trim}\). Then, algorithm \(A''\) returns a sub-object of base type \(t_1\) at offset zero for \(t\) and a sub-object of base type \(t_2 \neq t_1\) at offset zero for \(t'\), which violates the fact \(t'\) is a subtype of \(t\). Therefore, the common prefix cannot be
empty. Then, algorithm $A''$ applied to $t_1$ and $t_1'$ up to this prefix returns the same current size, an unknown strictly positive integer, given our lack of assumptions on base type sizes. In particular, we cannot know given the prefix if the current size reached is a multiple of any alignment. Then, either $t_1$ and $t_2$ are both real types, or one of them is the pseudo-type $t_3$. Given our lack of assumptions on base type alignment, it is not possible to decide that $t'$ is a subtype of $t$.

Example 3. Going back to our example, we compute the layout type list for $\text{header}$, $\text{packet1}$ and $\text{packet2}$.

\[
\begin{align*}
\text{trim(stutter(flatten(header)))} &= \text{int} \\
\text{trim(stutter(flatten(packet1)))} &= \text{int}, \text{int} \\
\text{trim(stutter(flatten(packet1)))} &= \text{int}, \text{t3}, \text{float}
\end{align*}
\]

This makes $\text{header}$ a parent type of $\text{packet1}$ and $\text{packet2}$, as expected.  \(\square\)

5 Implementation and Future Work

We have implemented a translation from C to Jessie in FRAMAC, a C analysis toolbox based on Cct. [15]. Our implementation already handles the simple examples presented in this paper. We are currently extending it to support real programs, like the implementation for Managed Strings by CERT [4].

Generation of VC from Jessie programs is handled in the WHY/Krakatoa/Caduceus family of tools [11] for the deductive verification of C and Java programs. The Jessie language is itself currently developed to handle more features from C and Java.

As mentioned in Section 3, we do not support yet uses of the address-of operator and union assignment. We are working to support those, building on what is done for structures in Caduceus [10]. Translating the address-of operator applied to a union field should be straightforward. Following what is done in [10], we can manage to have structures always referred to via pointers. Then, the address-of operator applied on a dereference of pointer $p$ yields the same pointer $p$. It is more complex to translate the address-of operator applied to a subfield of a union field, e.g., $a\rightarrow E\.x$. The translation of [10] that inserts pointers to fields whose address is taken does not work well with our translation of union. For those cases, we consider introducing multiple inheritance in Jessie, so that the same pointer $p$ can be used with a different type than those in the union hierarchy presented in this paper.

6 Related Work

Siff and others [18] collected uses of casts in large C programs. They introduce a platform-dependent notion of physical subtyping that is not directly comparable to ours. We present in Fig. 6 examples to illustrate our comparison. First, our presentation of physical subtyping is platform-independent. E.g., it is possible on some platforms that $\text{struct T}$ is a subtype of $\text{struct S}$ while we cannot in general conclude the same on every platform. Secondly, their algorithm emphasizes equality of field names
and sub-structure boundaries for deciding subtyping, which is not needed for safe access. It is rather an additional requirement they impose as a good software engineering practice. E.g., even on a platform where int and structures have the same alignment (and thus struct T is a physical subtype of struct S), their algorithm fails to detect such subtyping relation. Even more convincing in that matter is the case of struct T and struct U. Our algorithm concludes that struct U is a subtype of struct T in a platform-independent way whereas their algorithm rejects such subtyping relation on any particular platform.

```c
struct S { int i; int j; struct { int j; int k; } s; }
struct T { int i; int j; struct { int a; } s; }
struct U { struct { int a; } s; int d; }
```

**Fig. 6.** Comparing Physical Subtyping Algoritms

In [18], they use this notion of physical subtyping to statically classify casts in upcasts, downcasts or mismatch (remaining cases). Chandra and Reps use it in [5] to devise a physical type checking algorithm for C, that statically rejects programs that cannot be proved correct with respect to physical subtyping. As mentioned in their paper, their algorithm does not target programs which emulate inheritance using discriminated union and cast as we do. Our physical subtyping bears much resemblance with the one in CCured [7], although they only precise that care must be taken to account for structure padding without formal description and proof. In [16], Nita et al. extract from programs the implicit platform-dependent assumptions that ensure all casts are either upcasts or downcasts in the physical subtyping hierarchy. This produces constraints about the layout of structures and the size and alignment of types.

Jhala et al. have described in [17] an algorithm to automatically discover the type invariants that guarantee proper use of union as discriminated union in C. Although quite effective at discovering the proper invariants in large C programs, their method is limited to simple forms of invariants. Moreover, we believe that the overhead of manually annotating union types is not big when doing program verification.

Another direction was taken by Tuch et al. in [20] to allow reasoning about C union and cast in deductive verification. They choose to work with a byte-level memory model (described in [19]), with lifting functions providing a typed view of the heap. It is possible in their model to reason about any cast and union, even those that reinterpret typed memory through a different type, at the cost of manually providing the rules for how the lifted views of the heap change during these unsafe operations. In their case, they do manual proofs in Isabelle/HOL. In the context of Caduceus, Andronick presents in her thesis [1] a similar idea to interpret C unions in deductive verification. Writes to memory through a union type are performed in two steps: first, the original
field is updated, then a synchronization operation lifts the changes to other fields of the
union. Again, rules for reinterpretation must be given manually. Our structure subtyping
feature is more restricted than both approaches, but also better suited for automatic
proof, and sufficient in most cases.

7 Conclusion

The classical Burstall-Bornat memory model for imperative languages does not directly
fit with uses of union and cast in C. Although these features allow arbitrary reinterpre-
tation of typed memory through different types, their use can be restricted to simulate
inheritance in a low-level language like C, thus allowing only correctly typed access
to memory. Union used as discriminated union is easily translated into structure sub-
typing, where each variant of the union becomes a subtype. Then, it is sufficient to
annotate the C program with a type invariant for the union to specify which variant
should be accessible at any time. Cast used as discriminated cast can be translated in a
similar way, although with more difficulties. The main problem is to define a physical
subtyping relation in a platform-independent way. C programs rely on such subtyping
in many cases where the C standard does not enforce it. Therefore, we assume a general
algorithm to compute the layout of structures, similar to what is done in C compilers.
We deduce an algorithm to decide platform-independent physical subtyping, and use
it to build a hierarchy of types based on the casts present in the program. Finally, we
translate C programs with restricted unions and casts into Jessie programs on which we
perform deductive verification.

Acknowledgements I would like to thank Claude Marché and Pierre Créguet for their
advisory work, and the anonymous referee for his useful remarks.

References

1. June Andronick. Modélisation et Vérification Formelles de Systèmes Embraqués dans les
   Cartes à Microprocesseur – Plate-Forme Java Card et Système d’Exploitation. PhD thesis,
   Université Paris-Sud, 2006.
   managedstring.html.
   ergo.lri.fr/.
   CCured in the real world. In Programming Language Design and Implementation, pages


Reasoning on Data-Parallel Programs in Isabelle/HOL*

Matthias Daum
Saarland University, Saarbrücken, Germany
md1@ipserver.cs.uni-sb.de

Abstract This paper describes a dialect of the programming language C for data-parallel applications and its embedding into a verification environment based on Isabelle/HOL. On the one hand, the proposed language eliminates C features that interfere with program verification, while on the other hand, it introduces means to express data-parallelism. The embedding of this language into the existing verification environment shows the flexibility and diversified usability of the used verification environment even for rather special-purpose programming languages.

1 Introduction

Data-Parallel C Extensions (DPCE) [1] define a set of extensions to the programming language C [2] that support the programming of data-parallel applications. The so-extended programming language permits the implementation of concise and portable programs that will efficiently run on data-parallel architectures. DPCE comprise the most fundamental concepts of a consistent data-parallel model. The language adopts a general model that includes the basic concept of parallel data aggregates with an inner structure, a nodal distribution, and a context (constraining the participating elements).

While the extended language permits convenient programming, we encounter obviously the same problems on verification as with C. In order to reduce verification costs, I have restricted this language to a type-safe subset called Data-Parallel C0. For verification, I specified a formal semantics for Data-Parallel C0 (DPC0). This semantics employs Schirmer's verification environment for imperative programming languages [3] as its basis.

The verification environment is implemented on top of the generic theorem prover Isabelle [4,5]. It features a decent abstract syntax permitting to embed a great range of sequential imperative programming languages. Schirmer proposed an embedding for a type-safe C variant. The DPC0 embedding reuses great portions of this work and mainly adds data-parallel specific language features. I have implemented a syntax-translation tool in order to rule out any accidental mistakes during translation.

* This work was partially funded by the German Federal Ministry of Education and Research (BMBF) in the Verisoft project under grant 01 IS C38.
The remaining paper is organized as follows: The basic language features of DPC0 are described in the next section, Section 3 introduces Schirmer’s verification environment. In Section 4, the formal semantics is defined. And finally, Section 5 concludes the paper, gives an outlook on further work, and points to related work.

2 Data-Parallel C0

The programming language Data-Parallel C0 (DPC0) was developed as a variant of Data-Parallel C (DPC) that is just as well suited for verification as for programming of data-parallel applications. The most notable difference is the perfect type safety of DPC0 because type safety tremendously simplifies the verification effort for individual programs. For the same reason, expressions have to be free of side effects.

Other crucial design objectives were compatibility and easy portability of existing code. These objectives do not only help a programmer when switching they were most fundamental because I did not intend to write a new compiler. Instead, the source code has to be compiled with a standard DPC compiler. Hence, DPC0 is almost literally compatible to DPC. The only syntactic difference regards dynamic memory allocation.

Moreover, DPC0 inherited the basic design principles of DPC: The ability to write efficient programs for data-parallel systems in a readable and portable manner. Hence, both languages feature a very general model for parallel data. The key concept is the separation of algorithm and data distribution. The programmer can naturally specify the data distribution in the high level language but will then program independently from it. On the semantic level, there is just a single thread of control.

The remaining section sketches the key concept of data-parallelism in both languages, focusing on DPC0 but pointing out important differences to DPC.

A data-parallel object is an aggregate of data elements of the same data type. The data elements might be distributed over several processors or nodes. Moreover, they have an inner structure, i.e., a rank and dimensions. Hence, a data aggregate is characterized on the one hand by the data type of the elements, and on the other hand by the inner structure and the nodal distribution. The latter two attributes together form a shape. Here is a small example definition of a shape for illustration:

``shape [2][5][3] S;
``

This line defines a new shape called S with rank three. The first dimension has two positions, the second one five, and the third one three. The nodal distribution was not explicitly specified and can thus be chosen by the compiler.

Additionally, a context is attached to each shape. The context limits the scope of operations only to the so-called active positions in the aggregate. While rank,
dimensions, and distribution are static properties, the context might dynamically change during the program execution. Hence, it will never be part of a definition but can be altered with special commands, namely where and everywhere.

The syntax of the where statement reads as follows:

\[
\text{where } (\langle \text{mask} \rangle ) \ (\langle \text{statement block} \rangle )
\]

We assume that \langle mask \rangle is a parallel Boolean expression of some shape \text{S}. In the specified \langle statement block \rangle, all those positions of \text{S} are deactivated, where the element in \langle mask \rangle evaluates to false.\footnote{Strictly speaking, there is no value false in C and hence in DPC. Instead, value 0 is used to represent the logical value false. The standard library header \texttt{<stdbool.h>} defines the necessary macros in order to overcome this shortcoming.} The where statement might optionally have an else branch with a second statement block. During the execution of this block, all those positions are deactivated, where the elements of the mask evaluate to true.

DPC features the statement everywhere that will temporarily reactivate all positions of a shape for a statement block. There was no use-case for this statement in our project and hence it is currently not supported in DPC0. However, it could easily be added. Moreover, the same effect can be enforced by a function call, which will implicitly activate all positions.

A parallel data type is fully described by its element type and its shape. Type and shape are separated by colons, as illustrated in the following example:

\[
\text{int}: \text{S parallel};
\]

This line defines a parallel variable that contains integer elements and is shaped according to shape \text{S}.

Operations on data-parallel objects comprise the usual C operations like arithmetics, logical operations and assignments. Unary operations work just element-wise on all active positions. So do binary operations, given two compatible parallel expressions. If a binary operator is applied to a scalar and a parallel expression, DPC will perform an implicit conversion. In contrast, DPC0 does not support any implicit conversions. However, both language variants feature a type-conversion syntax to construct parallel values from scalars, called replication in data-parallel programming.

Moreover, both languages introduce a number of novel operators. These include reduction operators, a minimum and a maximum operator together with their respective reduction operators as well as so-called parallel indexing.

The latter operation permits on the one hand the selection of a single element just as known from arrays. On the other hand, a parallel value may arbitrarily be reshaped with parallel indexing — a technique that is called scatter and gather in data-parallel programming. DPC permits even a third kind of parallel indexing, which is called sliced indexing. However, this operation is not fully type safe and thus not supported in DPC0.
The common syntax for both in DPC0 supported forms reads as follows:

\[ ([i_0]) \cdots ([i_n]) \langle pepr \rangle \]

We assume that \( \langle pepr \rangle \) is a parallel expression. The expression is only valid if
the number of index expressions is equal to the rank of the parallel expression’s
shape. The index expressions \( \langle i_j \rangle \) can either be scalar or parallel integral values.
In the first case, the expression evaluates to the selected scalar. In the second
case, all index expressions have to be of the same shape; a new parallel value is
constructed with that shape.

The new value might as well be used as left-value, where the evaluation order
effects the result if a position is selected multiple times, e.g.:

\[
\text{shape}[5]\text{ S; long: S p,q; }
\]

\[
\text{[Int:S 1]}\text{ p = q; }
\]

The new value of variable \( p \) is implementation defined in DPC, while DPC0
treats this case as a runtime error.

In analogy to the original syntax of the language, we will denote parallel
indexing in the paper by \( [i_0] \cdots [i_{\text{rankof}(S)-1}]x \) for some parallel expression \( x \)
with shape \( S \).

DPC inherited the weak type system from C and additionally permits par-
tially and even completely unspecified shapes — in particular the generic shape
\textbf{void} —, which are an offence against type safety. However, the specification
of DPC includes several library functions that rely on the weak type system.
Some of these functions had to be dropped in DPC0, for instance the function
\textbf{newshape} that can dynamically create a new shape.

Other library functions were replaced by additional operators. In DPC, dy-
namic memory allocation for non-parallel data objects works C-like with the li-
brary function \texttt{malloc} and likewise for parallel data objects with the additional
library function \texttt{palloc}. The counterparts for memory allocation in DPC0 are
the built-in operators \texttt{new} and \texttt{pnew}, which return a typed pointer in a C++-like
fashion.

Another newly introduced operator is called \texttt{pcoord}. The corresponding li-
brary function in DPC designates a parallel value that enumerates the elements
along an axis, formally:

Assuming \( S \) is a shape and \( a \) denotes an axis, i.e., \( a < \text{rankof}(S) \), we define
\( \texttt{pcoord}(S,a) \equiv P \) such that \( P \) is the parallel value of shape \( S \) with the element
type \texttt{int} and \( \forall i_0, \cdots, i_{\text{rankof}(S)-1} : [i_0] \cdots [i_{\text{rankof}(S)-1}]P = i_a \).

This functionality is frequently used for indexing.

In order to contain the number of newly introduced operators in DPC0,
functions might still take an arbitrary shape as argument. DPC0 treats shapes
as values of a distinct data type \texttt{shape}. A newly defined shape is a constant of
that type. It can be used in the specification of a parallel data type and equally
as an argument to a function. With the parameter of this function, however, it is impossible to specify a new parallel data type because that is not a constant anymore. However, this approach permits the access of shape properties like its rank (rankof) or its elements along an axis (dimof).

3 A Verification Environment for Sequential Imperative Programming Languages

This section shortly introduces the very basics of Schirmer’s verification environment [3]. This environment is seamlessly integrated into the generic proof assistant Isabelle [4] using the higher order logic library HOL [5]. The composed system provides a sound and practically usable environment for reasoning on sequential imperative programs.

The verification environment is based on a general language model for sequential imperative programs, which is called Simpl. This model is not restricted to a particular programming language but covers all common language features of imperative programs: mutually recursive functions, global and stack variables, pointers, heap objects and dynamic memory allocation, runtime faults like array bound violations or dereferencing null pointers, and much more.

The flexibility of Simpl is based on the mixed embedding scheme for statements and expressions: While statements are represented by an abstract data type, the expressions in the language model are represented by ordinary Isabelle/HOL expressions. For this language model, Schirmer has defined an operational semantics and a Hoare logic for both, partial and total correctness. Furthermore, the Hoare logics were proven sound and complete with respect to the operational semantics. Formalisation and proofs were completely conducted in the theorem prover Isabelle/HOL. A verification condition generator (VCG) translates the deep-embedded statements into Isabelle/HOL formulæ. The VCG is integrated in the proof assistant as a proof tactic.

The remaining section sketches the representation of program states and describes the syntax of the language in more detail.

Program States. In his language model, Schirmer does not fix a particular representation of the program state space. However, he proposes the use of records as provided by Isabelle/HOL. Records are essentially tuples but supplemented with selection and update functions for each component. For example,

\[
\text{record } \text{state} = N :: \text{nat} \quad B :: \text{bool}
\]

defines a record type that is represented as a tuple of a natural number and a Boolean value. Additionally, this definition creates two selectors \(N \in \text{state} \rightarrow \text{nat}\) and \(B \in \text{state} \rightarrow \text{bool}\) as well as two update functions \(N\text{-update} \in \text{nat} \rightarrow \text{state} \rightarrow \text{state}\) and \(B\text{-update} \in \text{bool} \rightarrow \text{state} \rightarrow \text{state}\).

A record update \(N\text{-update} n s\) can be abbreviated with \(s(N := n)\) in Isabelle/HOL.
We represent each program variable by a record field. Thus, every variable might have an individual HOL type and the automatic type inference of Isabelle/HOL takes care of typing issues. Moreover, with field selection and update we have convenient means to express state updates and assertions.

**Syntax of the Programming Language.** The abstract statement syntax is defined by the polymorphic data type \((S, P, F)\) over a program state space \(S\), a procedure name space \(P\), and a fault type \(F\). Table 1 lists the constructors of the data type that are used with DPC0. The variables \(c, c_1\) and \(c_2\) are Simpl commands of type \((S, P, F)\). The Boolean condition \(b\) and the guard \(g\) are modelled as state sets of type \(S\).

**Table 1.** Statement syntax

<table>
<thead>
<tr>
<th>Atomic steps</th>
<th>Do nothing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic (f)</td>
<td>Basic command, where (f : S \rightarrow S) is a state update</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Standard control flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Seq \ c_1 \ c_2)</td>
</tr>
<tr>
<td>(Cond b \ c_1 \ c_2)</td>
</tr>
<tr>
<td>(While b \ c)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Runtime faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Guard \ f \ g \ c)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scoping and procedure calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Block \ init \ c \ exit)</td>
</tr>
<tr>
<td>(Call \ init \ p \ exit)</td>
</tr>
</tbody>
</table>

The semantic rules for the common language constructs follow the standard textbook semantics. The command **Basic \(f\)** applies the function \(f\) to the current state. Examples for basic commands are assignments or dynamic memory allocation. As conditions are semantically modelled as state sets, testing the conditions is formulated as set membership.

During expression evaluation, we might encounter runtime faults like division by zero or the dereferencing of a NULL pointer. We use the guarded command **Guard \(f \ g \ c\)** to model these faults. In order to prove a guarded command, we have to ensure that the guard \(g\) holds. The fault flag \(f\) is used for the integration with automatic tools and not of interest throughout the scope of this paper.

The abstract syntax differs much from the original appearance in an ordinary imperative programming language. Hence, Schirmer has employed the powerful

---

2 Schirmer's data-type definition is somewhat more general because he deals with language features like abrupt termination and dynamic method invocation. As these features are not present in DPC0, we may safely simplify matters, here.
syntax-translation mechanisms of Isabelle in order to establish a more readable, so-called concrete syntax.

So, for example, an assignment of variable \( b \) to variable \( a \) looks like

\[
\langle a \mapsto b \rangle
\]

while Isabelle/HOL internally processes it as:

\[
Basic (\lambda s. s\langle a \mapsto b \rangle)
\]

4 Embedding DPC0 Code in Isabelle/HOL

In this section, we explore the semantics of DPC0. We regard usual language concepts like type system, variables, operations, and statements. A few peculiarities of the language design are explained, here, because they can best be understood together with the semantical background.

**Type system.** Type \textit{bool} is naturally represented as type \textit{bool} in Isabelle/HOL. All other unsigned integral types are represented as type \textit{nat} and all signed integral types as type \textit{int}. The respective type ranges are enforced separately via guards.

Array and parallel types are represented as lists. Note that multi-dimensional arrays in C are arrays with an elementary type of an array. Consequently, they are represented as lists of lists. Parallel types, however, are represented as lists of their elementary type regardless of the inner structure. This approach is motivated by the observation that the inner structure does not matter for most parallel operations. Wherever the inner structure is relevant, the syntax-translation tool explicitly provides it as additional parameter to the operator.

As mentioned in Section 2, DPC0 integrates the notion of shapes as one distinct type into the type system. However, the concept of a shape is actually twofold. On the one hand, a shape definition contains static properties, namely rank and dimensions, and optionally a nodal distribution. As the semantics does not depend on the nodal distribution, we can completely disregard it. Rank and dimensions, we preserve as a constant of type \textit{nat list}. On the other hand, a context is set up for each shape, and this context might change during program execution. Consequently, it has to be part of the program state just like an ordinary variable. We represent contexts as components of the state record of type \textit{bool list}.

Hence, the shape definition

\[
\]

is represented in Isabelle/HOL as constant \( s'S \) and record component \( c'S \):

\[
\begin{align*}
\text{constdefs} & \quad s'S \equiv [2, 5, 3] \\
\text{record} & \quad state = \ldots. \quad c'S :: bool list \ldots
\end{align*}
\]
Variables and the address of operator. Usually, variables are components of the state record just as said in Section 3. However, there is one exception to that rule: Like C, DPC0 features an address-of operator that constructs a pointer to a variable. Schirmer did not provide a semantics for such a construct because it leads to aliasing of variables, which cause a considerable overhead in verification. However, he already takes care of aliasing for pointers to dynamically allocated heap objects. Moreover, we can statically check for a given program, where the address-of operator is applied to. Hence, we introduce the notion of addressable variables and store them on the heap. Note that only global variables may be addressable in DPC0 because otherwise, we might introduce dangling pointers.

The very same feature is also used for the call-by-reference semantics of array arguments. We just regard arrays as addressable if they are used as an argument in a function call. The drawback is that hence, arrays must be global if passed as argument to a function.

Variable initialization. Our representation of expressions requires that all memory objects have to be initialized prior to their first use. The undetected use of an uninitialized pointer, for example, would completely destroy type safety. Hence, we must either detect such a case within Isabelle/HOL and guard against it or ensure initialization externally. An explicit modelling of uninitialized values, however, would cause an overhead in verification. So, we take the second option.

C initializes all global variables right upon program start. In contrast, stack variables and dynamic memory objects might be used uninitialized. Here, the programmer—or in our case the verification engineer—has to ensure that initialization takes place properly. For stack variables, we perform a static definite-assignment analysis. This analysis is a safe over-approximation and is implemented in the syntax-translation tool. A caveat on the static analysis is that it cannot recognize an iterative array initialization in a loop. Thus, stack array variables have to be initialized at once with an explicit initializer although, in a particular case, this might lead to less efficiency. The initializers are converted into a conventional assignment.

A comparable check for dynamic memory objects is naturally impossible. Here, we postulate an implicit default-initialization and have to bear possible extra costs. We can enforce the initialization upon dynamic memory allocation because this mechanism is DPC0-specific, anyway. The syntax for memory allocation was specifically designed such that it can easily be wrapped to the usual DPC mechanisms via the C preprocessor.

Data-Parallel Operations. The semantics of all element-wise working operators simply employs the map operator that is build into Isabelle/HOL for element-wise operations on lists. As we have learned in Section 2, the context limits the scope of operations to the active positions. However, for element-wise operations it does actually not matter whether the values for the inactive elements are computed as long as runtime errors are checked context-sensitively,
Reduction operators have to filter the active positions according to the current context; then we can use the built-in list reduction \textit{foldl}.

The inner structure of a parallel expression is only relevant for parallel indexing. We define an auxiliary function \textit{ppos} that computes the position in the flattened list according to some shape for a given list of index expressions. Formally:

\[
    \text{ppos } S \ I \equiv \sum_{i=0}^{\text{length}(I)-1} I!i \cdot \prod_{j=i+1}^{\text{length}(S)-1} S!j
\]

where \(!\) denotes Isabelle/HOL's operator to select an element from a list. For scalar index expressions, we can directly employ \(!\) on the parallel expression, once that we have computed the position in the flattened list. With a few more standard list manipulations, we achieve the same for parallel index expressions.

\textbf{Runtime faults.} In general, we encounter the same runtime faults in DPC0 as Schirmer does in his C dialect: array bound violations, arithmetic over- and underflows, division by 0, and dereferencing of \texttt{NULL} pointers. Additionally, we have to guard against another runtime error: Parallel indexing might be used in assignments, but the specification of DPC does not define an order, in which assignments have to be performed. In order to avoid non-determinism, DPC0 treats it as an error if a single position is selected multiple times.

Unfortunately, we cannot fully reuse the guard generating mechanism of the verification environment because it assumes that each C type is represented by a distinct Isabelle/HOL type. Thus, I decided to generate the guards outside by the syntax-translation tool.

\textbf{Statements.} DPC0 features the same statements as C, hence we can directly adopt the existing framework as is. However, there is one additional statement, \textit{where}, that constrains the context of a shape for the execution of a given statement block.

As we know, a context is represented as pseudo variable. When constraining it for the duration of a statement block, we have to retain the old value of the context and restore it afterwards. Hence, a \textit{where} statement is equivalent to a statement block with a block-local context variable that is initialized with the constrained value. Assuming the following concrete syntax of \textit{where} statements

\[
    \text{WHERE} \langle \text{mask} \rangle \text{ FOR } \langle \text{context} \rangle \text{ DO } \langle \text{statements} \rangle \text{ END}
\]

we can easily express the semantics by

\[
    \begin{align*}
    \text{Block} & (\lambda s, t(s l(\langle \text{context} \rangle := \langle \text{context} \rangle s \land_p \langle \text{mask} \rangle l))} \\
    & \langle \text{statements} \rangle \\
    & (\lambda s, t(s l(\langle \text{context} \rangle := \langle \text{context} \rangle s l))
    \end{align*}
\]

where operator \(\land_p\) performs an element-wise Boolean and. If a \textit{where} statement features an else branch, its semantics is analogously a sequence of two similar \textit{Block} statements.
The current context is relevant for assignments. In the simplest case, there is an assignment to a plain variable within an unconstrained context. Only in this case, we can completely replace the contents of the variable. Otherwise we have to use a selective update, which naturally complicates matters for verification. Happily, we can statically check whether the context is constrained because a function call implicitly activates all positions. The syntax-translation tool distinguishes these cases and supplies the current context to the assignment syntax only when needed.

Figure 1 shows a code fragment together with its translation and illustrates both cases in contrast. The displayed function \textit{elem2par} replicates a scalar value thus constructing a list that represents a parallel value with the given shape.

$$i = (\text{int}:S) \ 0; \quad \ i := \text{elem2par} \ s'S \ 0;$$

\begin{verbatim}
where (j ) (int:S) 3)
{ i = j - (int:S) 3;
}
\end{verbatim}

FOR 'c'S do
\begin{verbatim}
in 'c'S: \ i := 'j - elem2par s'S 3
\end{verbatim}

WHERE 'j >=p elem2par s'S 3

Figure 1. Translation of assignments: unconstrained vs. constrained context

5 Conclusion

Summarizing, this paper has presented a formal semantics of a data-parallel language. The embedding of even this rather special-purpose programming language was possible with few extensions of the well-established verification environment of Schirmer. Most importantly, these extensions provide only semantics for additional expressions and some supplemental syntactic sugar. It was not necessary to touch the kernel of the verification environment, which consists of the abstract syntax and the operational semantics of statements. Hence, this work demonstrates the flexibility and diversified usability of the used verification environment.

The herein presented work comprises on the one hand a syntax-translation tool and on the other hand Isabelle/HOL theories defining the necessary extensions. The tool translates DPC0 source code into its Isabelle/HOL representation and additionally performs several checks ensuring (a) well-typedness, (b) definedness of all variables, type names, functions and struct components, (c) correctness of all function calls with respect to the number and types of their arguments, as well as (d) initialization of all stack variables before their first use. Its implementation consists of about 5,000 lines of code.

The Isabelle/HOL theories define the semantic extensions for expressions as well as some additional concrete syntax for the where statement and the
expressions. Moreover, I have proven a few simple lemmata on the semantic extensions. The library of these lemmata is currently quite small and should be expanded in the future. Currently, the theories count around 500 lines.

As first case studies, I have conducted proofs of the functional correctness for a few rather simple programs. For these small programs, verification could be done easily and quite straightforward. The verification of a larger and more challenging program has yet to be done.

**Further Work.** Many additional features are desirable in the future: Most notably, data-parallel programming is usually employed in a context of extensive numeric computation. In this field, it is hard to find applications that will not heavily use IEEE-compatible floating-point arithmetic, which is currently not supported by DPC0. Harrison [6,7] works since several years on formal verification of floating-point algorithms using the HOL Light theorem prover. Especially exciting is that it might even be possible to import his theory files directly into Isabelle/HOL because there exists a converter.

Moreover, it turned out that perfect type safety immensely contradicts a 100% compatibility with the C language. It might be worthwhile to investigate an alternative, untyped memory model for Simpl. Tuch and Klein [8], for instance, have developed an alternative heap model for better C compliance. This model does not rely on a well-typed memory but still offers a typed view of memory where possible.

**Related Work.** Gordon [9] has first presented the idea to embed a programming language into higher order logic. He introduced a while language with variables ranging over natural numbers.

There exists a great range of formal semantics for C dialects. Gurevich and Huggins [10] have presented a formal operational semantics for C based on evolving algebras. However, this semantics was only formalized on paper and not mechanically checked. Tews et al. [11,12] developed a denotational semantics for a subset of C++ using coalgebras and proposed an embedding in the theorem prover PVS. They used a shallow embedding for both, statements and expressions. The proposed embedding features many advanced language features like arbitrary jumps or side-effects in expressions. While it is impressing to see the formalization of these rather difficult phenomena, the verification of even simple programs suffers from the complex model. The same problem is exposed by Norrish’s formalization of the C language [13] that he carried out in the HOL theorem prover.

Even Schirmer provides a fully deep embedded semantics of his restricted C dialect in his work [3]. This semantics allows him to perform definite assignment and well-typedness checks directly in Isabelle/HOL. A simulation relation links the deep embedding with Simpl and is used to transfer results. This approach is very valuable in the context of pervasive verification and in conjunction with a semantical stack for the embedding of inline assembler code. However, it would have been overkill in our subproject.
To the best of my knowledge, it exists no other embedding of a data-parallel language into a theorem prover. Prensa [14] presented a verification environment for fully parallel programs. However, she deals with true parallelism while DPCO has just a single thread of control on the semantic level. Much more importantly, Prensa’s verification environment lacks procedures.

References

How C differs from Java for Symbolic Program Execution

Christoph D. Gladisch
gladisch@uni-koblenz.de

University of Koblenz-Landau
Computer Science Department
Universitätsstr. 1, 56072 Koblenz, Germany

Abstract. Verification systems, test generators, debuggers, and compilers often implement a symbolic execution framework. We have identified a small set of features in ANSI C, MISRA C, and C0 that may require special treatment or special considerations when migrating a symbolic execution framework for Java to a framework for a C dialect. For some of these language features we have developed symbolic execution rules that are implemented in the KeY-System.

1 Introduction

Verification systems, test generators, debuggers, and compilers often implement a symbolic execution framework to perform static analysis of a program (target program). Many of these tools already exist for Java or JavaCard as the target language for symbolic execution but haven’t been yet extended for symbolic execution of C as the target language. The language C plays an important role because of the existence of a large body of legacy code, many available compilers for different platforms, and the possibility of programming close to the hardware.

We have identified a small set of features in dialects of the C language that may require special treatment or special considerations when migrating a symbolic execution framework for Java to a framework for C. For some of these language features we have developed symbolic execution rules that are implemented in the KeY-System.

The comparison of two programming languages is not a trivial task. On the one hand C and Java have the expressive power of a Turing machine. It is possible to replace every construct of one language by a program written in the other language. Therefore in principle any symbolic execution engine is universal if it supports one of these languages. From this extreme point of view C and Java can be regarded as equal.

On the other hand C and Java can be regarded as completely different languages with no language constructs in common. For instance, arithmetic expressions in both languages differ, because of differences of the integral types that cause different overflow semantics. Also the semantics of statements like the if-statement or while-statement are different, because in Java they can be abruptly terminated by exceptions in contrast to C. Neither of the two views is helpful when a symbolic execution framework is to be extended for the use with another programming language.
The first approach, where the constructs of one language are simulated by the other language, is in particular not useful when a (semi-) interactive system is extended or reports are shown to the user where she sees the source code. We therefore use a perspective for the comparison that lies between these extreme points of view: we regard differences with respect to the functionality and capabilities of a typical symbolic execution framework like the one implemented in the KeY-System. Two components of the framework and respectively two parts of the language semantics are distinguished: the static and the dynamic part. We present a small set of constructs in C that are special with respect to Java concerning the dynamic part. The existence of these features depends on the particular dialect of C and is given by the following list:

- reference and dereference operators (ANSI C, MISRA C, C0)
- assignment of structures (ANSI C, MISRA C, C0)
- evaluation order of expressions and function arguments (ANSI C only)
- explicit object deallocation (ANSI C only)
- unions, pointer arithmetics, and type casts (ANSI C only)
- goto-statement (ANSI C only)

2 About Java and JavaCard

Java [12] is an imperative object oriented programming language with statements, expressions, declarations, and types similar to those of C++. JavaCard [24] can be seen as a subset of Java. The specification of both languages is given in natural language. The syntax and semantic of the languages are unambiguous and fully specified. The languages have strong typing rules in contrast to C and C++. Their object oriented features are interfaces, inheritance, virtual methods, overloading, dynamic object creation and scoping but in contrast to C++ there is multiple inheritance of interface but not of classes. It is also not possible to write purely procedural programs as in C and C++. Common primitive types of Java and JavaCard are: byte, short, and boolean.

Java’s primitive data types also include int, long, double, float, and char. In contrast to JavaCard Java supports multidimensional arrays, and dynamic class loading, threads (concurrency), introspection and Strings as build-in APIs. As a consequence, the class Object which is the super class of all other classes has less methods in JavaCard than in Java. For the programming of SmartCards JavaCard supports transactions by which it can be assured that a SmartCard stops execution in a valid state in case of abrupt termination (e.g., due to powerless).

3 About C Dialects

3.1 ANSI C

The programming language C is a procedural language whose main building blocks are functions. When C was invented and first implemented in the 70’s
and 80's, there was no official specification but only the book *The C Programming Language* that was used as “existing practice”. The creation of the first standard specification began in 1983 [1], was finalized in 1989, and adopted by ANSI and ISO (International Standard Organization) in 1990 [3]. Since a lot of legacy code existed already at this time that was based on different interpretations of the C “practice” the specification was designed to leave room for these interpretations. Therefore much of the language is intentionally undefined, unspecified or implementation dependent in contrast to the specification of the Java language — clearly, ANSI C has ambiguous semantics. This version of C is commonly called C89. In 1996 and 1999 two more versions C96 [4] and C99 [21] were created.

The specifications are written in natural language and are hard to understand. For instance, “members of the standardization committee and other distinguished researchers participating in the discussions often give contradictory answers when asked about the intended semantics of surprisingly small programs” (A case study in specifying the denotational semantics of C by N. Papaspyrou [19]). Also many popular books, e.g., *C: The Complete Reference*, contain “several hundred errors” as reported in [22] and [10]. Similar reviews of many other books are given by ACCU (Association of C & C++ Users) in [11].

### 3.2 MISRA C

From what has been said above it seems to be hard to write safe and secure software in C. The MISRA (Motor Industry Software Reliability Association) consortium has therefore developed a manual with 141 rules which restrict the use of C96 resulting in the subset of C called MISRA C. “The MISRA consortium is not intending to promote the use of C in the automotive industry. Rather it recognizes the already widespread use of C, and this document seeks only to promote the safest possible use of the language.” [15]. In the following we list some of the important rules of the manual to give an impression of the differences between MISRA C and C96 and at the same time report some of the features of C96 that are unspecified, undefined, or implementation dependent. (Rule ids are given in parentheses.)

1. The value of an expression shall be the same under any order of evaluation that the ANSI C Standard permits. (12.1 and 12.2)
2. Directly or indirectly recursive functions are not allowed (16.2). Functions have a constant number of arguments (16.1) and may only have a single point of exit via the `return` statement which is at the end of the function (14.7).
3. Dynamic heap memory allocation (and deallocation) shall not be used (20.4)
4. Pointer arithmetic is restricted to pointers that address elements of the same array (17.1)
5. Use of `goto` (14.4), `continue` (14.5) and the comma-operator (12.10) is prohibited and the use of the `break` statement in loops and in the `switch` statement is restricted.
• Unions shall not be used. (18.4)
• The way expressions may be constructed is restricted regarding the types of
  the expression arguments by the rules (10.1 and 10.2); and type casts be-
  tween pointers are restricted to those that are independent from the memory
  representation (11.1 to 11.5).

3.3 C0

C0 [14, 7] is a C-like language that has a single and formally defined semantics. It
was developed in the Verisoft project [25] as the target language for verification
using Isabel/HOL [20]. C0 is much simpler than ANSI C. Some of the interesting
restrictions of C0 are:

• no side effects in expressions (e.g., operators like i++, i--)  
• no functions as sub expressions  
• size of arrays is statically fixed at compile time  
• exactly one return statement in each function as last statement  
• only one scope for variables inside a function  
• evaluation of expressions in the standard order defined by post order traversal of syntax trees  
• no pointers to local memories or to functions  
• pointers are strictly typed  
• two’s complement representation of integral types

4 The Differences

4.1 Distinguishing the static and the dynamic part of the languages

The static component of the execution framework, also called program context,
consists of tables typically generated by a compiler. The tables, hold information
about type or class declarations, program variable declarations, and declarations
of functions or methods. Besides the tables we also regard the name resolution
or binding component as part of the static component. The binding component
maps identifiers of program variables and functions or methods depending on the
context of usage of the identifier to the correct declarations.

The dynamic component consists of a language in which the operational se-
manitics of the programming language is defined and an execution framework
that can execute a program based on these definitions. This component has typ-
ically means to express the current state of program variables, a path condition,
a stack of executed methods and other program blocks, and a program counter.
In endogenous logics like Temporal Logic this counter is a pointer or index to the
current program instruction. In exogenous logics like Hoare Logic or Dynamic
Logic the program is explicit in the formulas.

The differences of C with respect to Java that we mainly focus on are related
to the dynamic parts of the languages. We assume that macros and preprocessor
directives are resolved by a preprocessor and are not part of the language.
4.2 How C differs from Java regarding the static part of the languages

We briefly address the differences between some static parts of the languages. Java has primitive types, arrays, classes and interfaces with access modifiers and C has similar – but in detail different – elementary types, arrays, typedefs, structs, unions, bitfields, and pointers. The type system of C can be mapped to the type systems of Java by an encoding such that different types can be distinguished and relations like “pointer of” and “substructure of” can be evaluated based on the encoding. Such an encoding is however very obfuscated, we therefore recommend an implementation of the type system that is independent from Java’s type system.

4.3 How C differs from Java regarding the dynamic part of the languages

We have identified a small set of features of the C dialects that likely require special treatment for the formal definition of the language semantics and for the definition of symbolic execution rules with respect to those for Java. Features that exist only in one of the dialects are marked in parentheses.

Reference and Dereference Operators. The reference operator '\&' returns the pointer to a program variable (or C function) and the dereference operator '\*' maps a pointer to the value at the location indicated by the pointer. Similar to the dereference operator is Java’s dot operator '.' for accessing members or methods of objects (target objects). Techniques for solving the aliasing problem in Java can be reused here. However, the dereference operator in ANSI C and MISRA C is more powerful than the dot operator in Java, because the dereference operator can be used together with pointer arithmetic. Array indexing is a commonly used application of this technique.

An important difference between the languages results from the combined usage of the reference and dereference operators in C. In this case program variables in C must be treated like objects in Java. Dereferencing a pointer to a program variable in C behaves like accessing the member of an object in Java.

In ANSI C a pointers to a local program variables can be obtained. The pointer to a local program variable becomes invalid when the function in which the program variable is declared terminates. This can be solved by bookkeeping of the local program variables and upon termination marking the program variables as invalid.

Assignment of Structures. The explicit or implicit (e.g., by passing arguments to a function) assignment between variables of a structure type has copy-by-value semantics. This causes a component-wise assignment between the corresponding members of the structures. If a constant size array is declared within
a structure, then the elements of the array are copied individually to the corresponding array positions of the array that is embedded as a member of the other structure. Thus an assignment in C may represent more than one assignment in contrast to assignments in Java. C0 has copy-by-value semantics also for not embedded arrays.

When using the reference and dereference operators in combination with structures the aliasing problem becomes more complicated. A structure can indirectly not only be manipulated by dereferencing a pointer to the structure and assigning a value to it but also by dereferencing pointers of members of the structure and assigning new values to them.

**Evaluation Order of Expressions and Function Arguments (ANSI C only).** The evaluation order of expressions and of arguments in function calls is unspecified. It is also possible that some subexpressions are not evaluated at all. The evaluation order of expression is not specified in MISRA C but the set of expressions that maybe used in MISRA C is restricted to those expressions that evaluate equally under any order of execution of the subexpressions.

The evaluation of an expression may yield different results upon different orders of evaluation if the subexpressions have side-effects or if one subexpression causes the abrupt termination of the program. An expression has a side-effect if it contains an implicit assignment and thus a change of the execution state during the evaluation of the expression. The evaluation order of expressions with side-effects is not **critical** if no read access to the implicitly modified memory is performed during the expression evaluation.

**Unions, Pointer Arithmetic, and Type Casts (ANSI C only).** The common concept of unions and arbitrary type casts is that one sequence of bytes in memory can be interpreted in different ways.

The *interpretation* of the memory depends on the type or data structure that is virtually laid over the sequence of bytes in the memory. Translations between arbitrary interpretations of the memory require complicated arithmetic. Aliased program variables of different types are seldom used except when the boundaries of the subsequences of the addressed memory are aligned.

The problem of interpreting non-aligned memory blocks is particularly hard when pointer arithmetic is used. A non-critical usage of pointer arithmetic is array indexing.

Even if memory blocks are not interpreted for different types but a runtime type is changed by a type cast or union this causes a problem for quantification over objects of one type. In verification systems it is important to restrict quantification only to created objects of one type. The usage of unions and type casts can change the type dependent sets of created object. This problem occurs for instance when `malloc` is called — the type of the returned pointer is `void` and is usually type casted at some time.
Explicit Object Deallocation (ANSI C only). In ANSI C memory can be deallocated using the API function `free` of the standard C library. A deallocated pointer is `invalid`. When the location indicated by the invalid pointer is subsequently modified via the dereferenced pointer this results typically in modification of subsequently allocated heap memory. In general however the behavior is unspecified.

Due to this feature, several kinds of pointers have to be distinguished. Only `dynamic root pointers`, which are the pointers returned from the allocation function, may be used for deallocation. Pointers to global program variables and to local program variables, called `static pointers`, and `dynamic non-root pointers` like pointers to members of dynamically allocated structures and to elements of dynamically allocated arrays must not be deallocated; otherwise the program behavior is undefined (usually the program will terminate abruptly).

Special care has to be taken if a dynamically allocated structure or array is deallocated, because pointers to members of the structure or to array elements become invalid as well.

goto Statement (ANSI C only). A symbolic execution framework that can handle abrupt termination in Java caused by `break`, `continue`, `return`, or `throw` can also handle the respective jump statements of C. All these statements have in common that the program execution is continued at the beginning or end of the block or an outer block in which the statement occurs. However the goto statement that exists only in ANSI C allows arbitrary program execution jumps within a function.

4.4 Border-Issues

Volatile Program Variables (ANSI C and MISRA C). A program variable may change its value without being explicitly changed by the program that is symbolically executed. For instance, when a reference to a program variable is passed to an `external program` via an API function the external program may change the program variable concurrently, e.g. hardware drivers, system clock, etc. Handling of program variables that may be modified concurrently by an external program requires techniques for handling concurrency. These techniques are also required when full Java is supported, so that we don’t classify this as a special feature of C.

The modifier `volatile` in C is used to prevent an optimizing compiler from performing certain optimizations involving volatile program variables. In Java the modifier `volatile` ensures memory synchronization of threads that have shared usage of the volatile program variable.

Function pointers (ANSI C and MISRA C). Invocations of dereferenced function pointers can be handled by techniques for handling polymorphism in Java. A function call through a dereferenced function pointer can be replaced
by an if-cascade over the value of the function pointer and with concrete function calls in the then-part. The problem with dereferencing a function pointer obtained from an external API function is no harder than with polymorphism when an object is returned from an API method and it belongs to a subclass of the return class type of the method.

5 The KeY-System

The KeY-System [6] is a mixed interactive and automatic verification* and test-generation system for Java. It is based on an instance of Dynamic Logic [13] called JavaCardDL (JDL) and a sequent calculus that interleaves rules for theorem proving and symbolic execution of Java. The symbolic execution rules handle 100% of JavaCard and some additional features of Java. Dynamic Logic is an extension of First-order Logic where formulas \( \varphi \) can be prepended by the modal operators \( \langle p \rangle \) and \( [p] \) for every program \( p \). The formula \( \langle p \rangle \varphi \) means that \( \varphi \) holds after the execution of the program \( p \) so that the implication \( \phi \rightarrow \langle p \rangle \varphi \) corresponds to the Hoare triple \( \{ \phi \} p \{ \varphi \} \).

Different program states are realized by different first-order interpretations of function symbols (including constants). Therefore program variables are modeled in the logic as constants, an expression like \( o.a \) that accesses an object attribute, is modeled as the term \( a(o) \), and in case of an array access \( o.a[i] \) the corresponding term is \( \mathbb{I}(a(o), i) \).

JDL has the additional modal operator \( \{ \} \) called update. Updates are assignments between terms (not expressions) and are therefore free of side-effects. This allows a powerful calculus for simplifying and merging sequences of elementary updates like \( \{ t_1 := t_2 \} \{ t_3 := t_4 \} \) into parallel updates \( \{ t_1 := t_2 \} \| \{ t_3 := t_4 \} \). The latter can be seen as a table from locations to values where the aliasing problem is handled. A conditional update \( \{ i < c ; U \} \) has only the effect of the update \( U \) when \( c \) is true. An arbitrarily long sequence of elementary updates \( \{ t_{a_1} := s_{a_1} \} \ldots \{ t_{a_n} := s_{a_n} \} \) over an ordered universe with \( a_1 < \ldots < a_n \) can be written as the quantified update \( \{ \text{for} \ x ; t_x := s_x \} \). Formulas with the modal operators \( \{ \}, \langle \rangle, \text{ and } [\] \) can be in any case translated into pure first-order formulas. In symbolic execution rules we abbreviate \( \langle \varphi \rangle \) and \( \{ \varphi \} \) by \( \langle \rangle \) and \( \{ \} \) respectively. Thus the symbolic execution of a side-effect free assignment is handled by the rule:

\[ \text{Replace } \langle a := b \rangle \text{ by } \{ a := b \} \]

JDL uses "relativized" constant domain semantics: an infinitely large pool of objects that may be created at runtime exists at any time of program execution but quantification is implicitly restricted to objects that are marked as created by the non-rigid index terms \( \text{next}_T \) for every type \( T \). The semantic is that objects of type \( T \) identified by the object identifier terms \( \text{obj}_T(i) \) are created for \( i < \text{next}_T \) and not created for \( i \geq \text{next}_T \). Quantification over created objects of type \( T \) is therefore realized by \( \forall i . \forall o . ((i < \text{next}_T \land o = \text{obj}_T(i)) \rightarrow \Phi(o)) \) and is abbreviated as \( \forall o : T . \Phi(o) \). Dynamic object creation is therefore handled by the following rule:
Replace \((a = \text{new } T();)\) by
\(\{a \leftarrow \text{obj}_{T}(\text{next}_{T}) || \text{next}_{T} \leftarrow \text{next}_{T} + 1\}\)

6 Special Symbolic Execution Calculus Rules for C

We have identified features in the C dialects that require special treatment. Additionally, we have developed a symbolic execution calculus for C0 and for some features of ANSI C. Since this paper is about symbolic execution, we do not give advice on how to handle undefined behavior or abrupt program termination because the handling depends on the purpose of the symbolic execution framework.

Reference and Dereference Operators. The combined usage of the reference and dereference operators requires that program variables in C must be treated like objects in Java. Dereferencing a pointer to a program variable in C behaves like accessing the member of an object in Java. Therefore the dereference operator is a non-rigid function symbol \(\ast\) (for every elementary type) that holds the value of the program variable and the program variable itself is a constant in the logic that represents the address of the program variable (etp: expression to pointer). Implicit axioms are required that ensure that all program variables (i.e., their addresses) are distinct. Then after the address of a program variable is determined it is finally dereferenced (ett: expression to term). The translation from expressions consisting of program variables, access to attributes, access to array elements, reference operators, and dereference operators to terms is given by the rewrite rules:

\[
\begin{align*}
ett(X) & \rightarrow \ast(\text{etp}(X)) \\
etp(X,m) & \rightarrow \ast^{m}(\text{etp}(X)) \\
etp(\&X) & \rightarrow \text{etp}(X) \\
etp(\ast X) & \rightarrow \text{ett}(X) \\
ett(X[i]) & \rightarrow \text{etp}(i, \text{ett}(X)) \\
etp(a) & \rightarrow a
\end{align*}
\]

where \(\ast^{m}\) is the corresponding function symbol for each member \(m\), \(\text{etp}()\) is the binary function symbol that represents addresses of array elements, and \(a\) is a program variable identifier. Examples of applying the syntactic transformation ett on expressions are:

<table>
<thead>
<tr>
<th>Expression</th>
<th>(*a)</th>
<th>(\text{a.m})</th>
<th>(\ast a)</th>
<th>(\ast(a)).m</th>
<th>(\ast(a.m))</th>
<th>(\text{a[i]})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term</td>
<td>(\text{*a})</td>
<td>(\ast(m\text{a}))</td>
<td>(\ast(a))</td>
<td>(\ast(a)).m</td>
<td>(\ast(a.m))</td>
<td>(\text{a}[(i)])</td>
</tr>
</tbody>
</table>

Assignment of Structures. An assignment between variables of a structure type can be handled by unfolding (unf) the assignment into a finite set of assignments (here updates) between the respective members of the structures or array elements in case of an embedded constant size array in the structure. The unfolding of the assignment has to take place over the pointers of the members of the structure (resp. array elements). Therefore the final dereferenciation of the expression must be delayed until assignments between program variables of basic types are obtained.
Replace \( \langle x \cdot y \rangle \) by \( \set{ \text{unf}(X = Y) } \)

\[
\text{unf}(X = Y) \leadsto \\
\begin{cases}
\text{unf}(X,m_1 = Y,m_1), \ldots, \text{unf}(X,m_n = Y,m_n), \text{if } X \text{ has a struct type} \\
\text{unf}(X[0] = Y[0]), \ldots, \text{unf}(X[n] = Y[n]), \text{ if } X \text{ has an array type} \\
\text{ett}(X) := \text{ett}(Y), \text{ if } X \text{ has an elementary type}
\end{cases}
\]

*) The enumeration of elementary updates between array elements can be replaced by the single combined quantified and conditional update: \( \{ \text{for } i : 0 \leq i \leq n : \text{unf}(X[i] = Y[i]) \} \)

**Evaluation Order of Expressions and Function Arguments (ANSI C only).** KeY’s symbolic execution calculus for Java eliminates side-effects by the program transformation rule:

Replace \( \langle \text{stmt(expr}_1, \ldots, \text{expr}_n) \rangle \) by

\[
\langle \text{tmp}_1 = \text{expr}_1; \ldots; \text{tmp}_n = \text{expr}_n; \text{ stmt} \rangle(\text{tmp}_1, \ldots, \text{tmp}_n);
\]

where \( \text{stmt(expr}_1, \ldots, \text{expr}_n) \) is a (compound) statement with expressions \( \text{expr}_1, \ldots, \text{expr}_n \) that are replaced by the temporary program variables \( \text{tmp}_1, \ldots, \text{tmp}_n \).

In case of critical side-effects (see section 4.3) any permutation of the statements \( \text{tmp}_1 = \text{expr}_1; \ldots; \text{tmp}_n = \text{expr}_n; \) may or must be considered depending on the purpose and overall semantics of the symbolic execution framework.

**Type casts (ANSI C).** A commonly used feature in C is the cast between a pointer of type \( \ast \text{void} \) and a different pointer. A simple approach is to associate every pointer with a runtime type as it is done for Java and treat the type \( \ast \text{void} \) like the class type \( \text{Object} \) in Java regarding type casts.

A logic and a calculus for handling type changes of objects (that would change the relativized constant domain) or for handling mappings or interpretations between byte sequences and arbitrary types is very involved and not covered here. An idea for the first problem is however to extended the allocation and deallocation mechanism and add a reallocation mechanism. Changes of the runtime type could then be realized by delayed allocation, deallocation and reallocation.

**Explicit Object Allocation and Deallocation (ANSI C only).** In order to distinguish different kinds of pointers, the type system is extended by the disjunct types Static, Root, and NonRoot for an additional classification of the terms:

\[
\text{obj}_T(n) : \text{Root} \\
\ast^{m}(x), \; \Downarrow^{x}(x,i) : \text{NonRoot} \\
\; a, \; \ast^{m}(a), \; \Downarrow^{a}(a,i) : \text{Static}
\]

where \( n, i : \text{Nat}, x : \text{Root} \cup \text{NonRoot}, \) and \( a \) is a global or local program variable.
The logic is extended by the non-rigid function $\text{cr} : \text{Root} \rightarrow \{\text{tt, ff}\}$ that indicates whether the root object is created (valid pointer) and the rigid function $\text{root} : \text{Root} \cup \text{NonRoot} \rightarrow \text{Root}$ that returns the root object from a non-root object as defined by the axiom schema for every attribute $m$:

$$
\text{root}(m(X)) = \begin{cases} 
X, & \text{if } X : \text{Root} \\
\text{root}(X), & \text{if } X : \text{NonRoot}
\end{cases}
$$

$$
\text{root}([i](X, i)) = \text{root}(m(X))
$$

For a subset of C that excludes casting of runtime types allocation can be handled by the rule:

\textbf{Replace} $(a = (\ast T)\text{malloc}((\text{sizeof}(T))));$ by \\
$\{(*a) := \text{obj}_{T}(\text{next}_T)\mid\text{cr}(*a) := \text{true} \mid \text{next}_T := \text{next}_T + 1\}$

where $(\ast T)\text{malloc}((\text{sizeof}(T)))$ is regarded as a single expression. The quantified formula \\
$\forall o : T_1 \Phi(o)$ is now an abbreviation for:

$$
\forall i, o : T_1, ((\text{root}(o) = \text{obj}_{T_2}(i) \wedge i < \text{next}_{T_2} \wedge \text{cr}(\text{obj}_{T_2}(i)) = \text{tt}) \rightarrow \Phi(o))
$$

Deallocation of Root-pointers is handled by the rule:

\textbf{Replace} $(\text{free}(a)) ;$ by $\{\text{cr}(*a) := \text{ff}\}$

Deallocation is only allowed with dynamic pointers that were obtained by the allocation function and whose referenced objects are created. The handling of the dereference operator must be extended such that the createdness of the object that the pointer is dereferenced to is checked. The createdness of a non-root object is the same as of its root object that can be obtained through the function root (defined above). How the violation of these rules is handled depends on the purpose and semantics of the symbolic execution framework.

7 Related Work and Conclusion

The Rationals [2] and [5] contain much of the text of the ANSI C specifications and discussions about thoughts and intentions of the standardization committee. A formal specification of C89 is given in the PhD thesis [18] by N.Papaspyrou. The specified language “differs slightly from ANSI C”. The specification must leave some space for further instantiation of the language or leave some aspects unspecified because of what has been said above. A dynamic logic for C0 is developed in [23]. For the theorem prover Nqthm the C semantic has been specified and is documented in the technical reports [16, 8]. A derivation of verification rules for C from operational definitions can be found in [17]. Finally, a cross-reference between C and Java is provided by [9].

The situation when working with the semantics of C is different from the situation when working with Java or JavaCard because of the complicated specification and vague semantics. It may be necessary to choose a particular subset of ANSI C or allow the instantiation of the symbolic execution framework for particular semantics.
References


A Dynamic Logic for Deductive Verification of C Programs with KeY-C

Oleg Mührk, Daniel Larsson, and Reiner Hähnle

Chalmers University of Technology, Department of Computer Science and Engineering
S-412 96 Gothenburg, Sweden
oleg.myrk@gmail.com, danla@chalmers.se, reiner@chalmers.se

Abstract. We present KeY-C: a tool for deductive verification of C programs. KeY-C allows verification of C programs w.r.t. operation contracts and invariants. It is based on an earlier version of KeY that supports JAVA CARD. In this paper we outline syntax, semantics, and calculus of C Dynamic Logic (CDL) that were adapted from their JAVA CARD counterparts. Currently, the tool is in an early development stage. As a side-product of this work we expect to generalize KeY architecture for easily adding the support for new programming languages. This paper is a further development of our work described in [11].

1 Introduction

We present KeY-C, a variant of the software verification tool KeY [2], that supports a subset of C as its target language. KeY is an interactive theorem proving environment and allows to prove properties of imperative/object-oriented sequential programs. The central concept is an axiomatization of the operational semantics of the target language in the form of a sequent calculus for dynamic logic, i.e., a program logic. The rules of the calculus that axiomatize program formulas define a symbolic execution engine of programs. The system provides heuristics and proof strategies that automate large parts of proof construction, for example, first-order reasoning, arithmetic simplification, and symbolic execution of loop-free programs are performed mostly automatically. The remaining user input typically consists of occasional existential quantifier instantiations.

The main creative part is to define a program contract and supplying loop (in)variants. KeY was designed with ease of interactive proof construction in mind (see screenshot in Figure 1) and to lower the initial learning curve. In particular, the concepts and the syntax of the language specific dynamic logic are as close as possible to the target languages.

The existing KeY system can handle JAVA CARD and most of sequential JAVA, allowing verification of complex programs [2, Part IV]. Its calculus contains about 1000 rules of which about half are language-independent Dynamic Logic (DL) rules. We are working on adding gradual support for a portable type-safe subset of C, axiomatized in C Dynamic Logic (CDL). In the future we plan to
create specializations of this calculus for particular platforms such as MISRA-C [10] or IA-32 (C on Intel 32 bit processors). As a side-product of this work we expect to generalize KeY architecture for easily adding the support for new programming languages.

In Section 2 we outline the principles of the Dynamic Logic (DL), in Section 3 we present an example C program that is used in Sections 4-8 of this paper to illustrate the C Dynamic Logic (CDL) and its calculus. In Section 9 we describe the current status of the KeY-C system and further directions, Section 10 describes related work and we summarize this paper in Section 11.

2 Principles of Dynamic Logic in KeY

Dynamic Logic (DL) is based on First-Order Logic (FOL) extended with a type system — function and predicate arguments and function results are equipped with a type. In order to arrive at a reasonable calculus, the subtyping relationship ⊆ must form a lattice. In the formal semantics all elements of the semantic domain also receive a type. In order to represent different states during program execution, functions and predicate symbols are split into rigid and non-rigid symbols. Rigid symbols behave the same as in FOL, while non-rigid symbols can have different values in different execution states. Basic DL has two types — int with signature and semantics of mathematical integers \( \mathbb{Z} \) and boolean that models boolean values true and false.

DL instantiated for a particular programming language is a modal logic with two parametric modalities \([P]\) and \(<P>\) for every syntactically correct and type-checked program P. Its semantics is defined in terms of Kripke structures, where states (worlds) are determined by the values of non-rigid symbols and transitions are defined by the semantics of the programming language. A formula is valid iff it is true in all possible states.
The formula \([P]\phi\) is true in a state \(s\) iff \(\phi\) holds in the final state reached when \(P\) is started in \(s\) provided that \(P\) terminates at all. In other words, \([P]\phi\) asserts partial correctness of \(P\) w.r.t. postcondition \(\phi\). The formula \(\langle P\rangle \phi\) is true in a state \(s\) iff there exists a final state \(s'\) reached when \(P\) is started in \(s\), such that \(\phi\) holds in \(s'\). In other words, \(\langle P\rangle \phi\) asserts total correctness of \(P\) w.r.t. postcondition \(\phi\). Note that in the case of indeterministic programs there can be more than one final state and \(\phi\) must hold just in one of them. For this reason, in practice, working with non-deterministic programs in DL requires first converting them to deterministic counter-parts at the level of formal semantics, e.g. by feeding in indeterministic choices as an additional input.

A sequent calculus is used for performing deduction. A sequent is of the form \(\Gamma \vdash \Delta\), where \(\Gamma\) and \(\Delta\) are sets of formulas. The semantics of a sequent is the same as that of the formula \(\Lambda \Gamma \Rightarrow \bigvee \Delta\). The DL calculus builds upon standard FOL with equality and arithmetic. DL calculi rules that work on program modalities always modify the first active statement of the modalities.

Locations are special non-rigid functions that can have an arbitrary value in different states of the Kripke structure and are used to model modifiable memory locations of the program. Let there exists a state for every combination of the values of the locations, while the value of other non-rigid symbols in some state may, for instance, depend on the values of some locations in this state.

The main principle of DL is to reduce program modalities into so-called updates. An atomic update has the form \(U = \{loc := val\}\), where \(loc\) is a location expression and \(val\) is its new value term. Semantically, the validity of \(U \phi\) in state \(s\) is defined as the validity of \(\phi\) in state \(s'\), which is state \(s\) where the values of locations are modified according to the update \(U\). There are operations for sequential and parallel composition of updates as well as for quantification, where the update is quantified by a free variable satisfying some condition. Any composition of updates can be automatically transformed to a normal form consisting of a parallel composition of quantified atomic updates (the details are inessential for this paper). An update applied to a pure FOL formula can be automatically transformed into a pure FOL formula without an update. Such FOL formula has to be proven using the standard rules of FOL sequent calculus and the rules expressing the properties of the particular Kripke structure. Most loop-free sequences of program statements can be turned into updates fully automatically.

Typically, the contracts for the programs are represented in the form of proof obligations \(pre \Rightarrow [P]\text{post}\) (for partial correctness) and \(pre \Rightarrow \langle P\rangle \text{post}\) (for total correctness), where \(P\) defines the program being reasoned about (typically a function body), precondition \(pre\) specifies assumptions about the initial state and postcondition \(post\) specifies the requirements for the final state.

Of course, there are many more details, some of which will be handled later in this text. A full treatment of the subject can be found in [2, Part IV], although the constructions there are slightly over-specialized for JAVA.
3 Example C Program

To illustrate the details of C Dynamic Logic (CDL) we use the function list2arr on Figure 2 that converts a linked list pointed to by ifirst into an array and stores the computed length of the linked list into *ocount and the pointer to the first element of the created array into *oarr. The linked list itself is deleted. If the array length is zero or memory allocation failed then *oarr is set to the null pointer. In this paper we mostly work with the program code in the first column, but we also look at some interesting statements in the second column.

```c
struct Node {
    struct Node* next;
    int elem;
};

int* arr = (int*)0;
if (count > 0) {
    arr = (int*)
    alloc(count, sizeof(int));
    if (arr != (int*)0) {
        struct Node* ptr = ifirst;
        int index = 0;
        while (index < count) {
            arr[index] = ptr->elem;
            struct Node* oldPtr = ptr;
            ptr = ptr->next;
            free(oldPtr);
            index++;
        }
    }
    *oarr = arr;
}
```

Fig. 2. Example C program

4 CDL Syntax and Semantics

The type system and the signature of CDL reflect the peculiarities of the C language. To represent C integer values we introduce logic integer types CHAR for char, SINT for signed int, UINT for unsigned int, etc. The reason why we cannot use mathematical integers int for this purpose is because C allows multiple bit-representations of the same mathematical integer value (eg, so-called negative zero). In our example we use only logic type SINT as the C type signed int
is a synonym for C type int.

Further, we need to reason about objects. Objects are memory areas that hold values and can be referenced by pointers (and consequently are values). We introduce a supertype of all object types Void. Symbols representing pointer
rvalues will have a logic type which is either Void (in the case of C type void*) or one of its subtypes. To represent null pointer rvalues we introduce a subtype of all object types Null. The interpretation of this type consists of exactly one element represented by the constant null. All concrete object types are a subtype of Void and a supertype of Null.

In order to represent C scalar objects (arithmetic or pointer) we introduce a logic scalar object type T ⊂ Void for any type T. This object type is equipped with a location T::value : T ⊂ T, representing the value of the scalar object.

Further, for every C struct S we introduce a logic struct object type $S$ with rigid member accessor functions allowing to navigate from the object instance to its members. In the case of struct Node from our example these accessor functions are Node::next : Node → Node and Node::elem : Node → int ⊂.

Finally, in order to represent arrays we introduce for each logic object type T a logic array object type $T[] ⊂ Void$ with two rigid functions. The first function is array element accessor function $T[]::elem : T[] ⊂ int → T$. The second is rigid array size function $size(a)$ defined for all arrays.

Please note that both struct and array accessor functions are rigid, so their value is the same in all states and so is array size function. The only non-rigid symbols so far that can have different values in different states are the values of scalar objects. In the case when a C scalar variable (e.g. int i) cannot be referenced by any pointer we can avoid the extra level of indirection provided by the scalar type T ⊂ (e.g. SINT ⊂) and model such variables just as locations of the value type (e.g. of type SINT). For instance, most local variables in C programs are never referenced by pointers. In our system we mark such variables with C specifier register.

In order to model object allocation we introduce for each logic object type T a rigid object repository function T::(lookup) : int → T and a rigid array repository function $T[]::(lookup) : int, int → T[]$. Further we introduce an object counter location next : int that is used to allocate objects T::(lookup)(next) and arrays SINT[]::(lookup)(next, l) of size l and is incremented on each allocation { next := next + 1 }. The repository functions return distinct objects for distinct values of next.

In order to reason about programs we need to express the properties of our heap structure. It turns out that the heap objects form trees rooted in repository functions with parent relationship defined by struct member and array element accessor functions and scalar objects serving as the leaves of the tree. Note that this structure is rigid and does not change in different states. Consequently we need axiomatization of trees containing such rules as

\[
\$Node::elem(o_1) \neq \$Node::next(o_2),
\$Node::elem(o_1) = \$Node::elem(o_2) \Leftrightarrow o_1 = o_2,
\$Node::elem(o) \neq SINT[]::(lookup)(i),
\$Node::next(o) \neq \$Node[]::(lookup)(i).
\]

Further we need to define function rootObj : Void → Void that returns for given object the root of the tree this object belongs to. In order to work with
pointers we need to define predicate `isRealObj : Void` that expresses that the path from the object to the root does not exceed the bounds of the arrays on this path. Such objects represent valid C memory areas that can be accessed. C language allows pointers to array `tails` — one element past the last element of the array. The predicate `isValidObj : Void` is supposed to express this property by being true iff the object is real or it is the tail element of a real array.

Finally, we need to express whether the objects are currently allocated or not. This is achieved by associating with each tree of objects a `block` that can be accessed with a rigid function `objBlock : Void → Block` such that

\[
\text{rootObj} (o_1) = \text{rootObj} (o_2) \iff \text{objBlock} (o_1) = \text{objBlock} (o_2).
\]

Now we define a location function `storage : Block → StorageMode` that can have one of the four possible values `ST_AUTOMATIC`, `ST_ALLOCATED`, `ST_STATIC` and `ST_TRAP`, where the first three denote possible storage modes and the last denotes that the object is not allocated yet or has been already destroyed.

5 CDL Modality Calculus

Now let’s consider our example from Figure 2. In this example we assume that no local variables or formal parameters of the function `list2arr` can be referenced by a pointer, which means that we prefix their declarations with the specifier `register`, as described earlier. We introduce one exception to this rule by not prefixing the declaration of the local variable `count` with `register` in order to illustrate how things become more laborious when local variables can be referenced by pointers.

The proof obligation for total correctness of function `list2arr` has the form `pre ⇒ (P)post`, where `P` is the function body and `pre` and `post` are respectively pre- and postconditions expressed in terms of locations `ifirst : $Node`, `ccount : INT` and `carr : INT` that model the formal parameters of the function. Parts of the pre- and postconditions will be discussed in Section 8. Here we focus on how to reduce the program part of the proof obligation. According to FOL sequent calculus `pre ⇒ (P)post` can be reduced into `pre ⊢ (P)post`.

The main principle of CDL modality calculus is to process the first active statement of the modality and either reduce into an update or replace it with simpler expressions. Now let’s consider the beginning of our function body `P`:

```c
{  
    int count = 0;
    register struct Node *ptr = ifirst;
    ...
}
```

The variable `count` has to be allocated in the beginning of the compound statement and destroyed in the end of the compound statement. We achieve this by first introducing a `block frame` and then registering the local variable `count` to be destroyed at the end of the frame execution.
block-frame(count) {
    allocate count;
    count = 0;
    ...
}

Now statement allocate count; is reduced into a sequential update

\[
\begin{align*}
\{ & \text{count} := \text{SINT}::\text{(lookup)}(\text{next}); \\
& \text{storage(objBlock(count))} := \text{ST\_AUTOMATIC}; \\
& \text{next} := \text{next} + 1
\}.
\end{align*}
\]

Further, statement count = 0; is reduced into update

\[
\{ \text{SINT}::\text{value}(\text{count}) := \text{SINT}::\text{fromInt}(0) \} ,
\]

where \text{SINT}::\text{fromInt}: \text{int} \to \text{SINT} converts mathematical integers into CDL integer representation (if there are many possible representations, the the one chosen by the compiler for the constants is selected). Now, the declaration of ptr is reduced into

\[
\text{register struct Node *ptr;}
\]

ptr = ifirst;

that is reduced into sequential update composition

\[
\{ \text{ptr} := s_0 \} \{ \text{ptr} := \text{ifirst} \} .
\]

Here \(s_0\) is a fresh skolem symbol of type \$\text{Node}\) that can have any value. According to the C specification local variables are created in \textit{indeterminate} state that can be either an \textit{unspecified valid} value or a \textit{trap} value. Accessing trap values leads to \textit{undefined} behavior. We model pointer trap values by objects that have storage mode \text{ST\_TRAP}, while integer trap values must be representable by the CDL integer types (like \text{SINT}). We model indeterminate and unspecified values with fresh skolem symbols and our calculus is designed in such way that proofs about programs that can perform actions that lead to undefined behavior cannot be closed (we have skipped proving the validity of the values when reducing value assignments above).

The result of symbolically executing the first two statements of the function body has the form \(\mu \text{post}, \text{pre} \vdash u(P1)\), where \(P1\) is the rest of the program contained within the block frame

block-frame(count) {
    \text{while (ptr != (struct Node*)0) } \{ \\
    \hspace{1em} \text{count} = \text{count} + 1; \\
    \hspace{1em} \text{ptr} = (*\text{ptr})\cdot\text{next}; \\
    \} \\
    \text{*ocount} = \text{count}; \\
    \ldots
}
and update $\mathcal{U}$ is the sequential composition of the updates described above. After all the statements in the block frame are executed and its body becomes empty, the variable count can be destroyed

```c
block-frame() { 
    destroy count ;
}
```

Destruction statement results in an update

\[
\{ \text{storage(objBlock(count)) := ST\_TRAP} \}.
\]

An empty block frame with an empty variable list

```c
block-frame() {}
```

can be removed from the modality. An empty modality without any code inside can also be removed

\[
\mathcal{U}(\emptyset)^{post} = \mathcal{U}^{post}.
\]

6 CDL Selection and Iteration Calculus

Let’s first consider the rule for the while statement in the sequent $\textit{pre} \vdash \mathcal{U}(\textit{P1})\textit{post}$ that we derived in the previous section. This sequent is reduced into four new sequents.

- **Invariant Initially Valid:**
  \[
  \textit{pre} \vdash \mathcal{U}(\textit{inv} \land \textit{variant} \geq 0).
  \]

- **Body Preserves Invariant:**
  \[
  \textit{pre} \vdash \mathcal{U}^V(\textit{inv} \Rightarrow (\langle \text{int } i0 = C \mid= 0 \rangle (\text{SINT::toInt}(i0) = 1 \Rightarrow (\langle B \rangle (\textit{inv}))))).
  \]

- **Use Case:**
  \[
  \textit{pre} \vdash \mathcal{U}^V(\textit{inv} \Rightarrow (\langle \text{int } i0 = C \mid= 0 \rangle (\text{SINT::toInt}(i0) = 0 \Rightarrow (\langle \text{P2} \rangle \textit{post})))).
  \]

- **Termination:**
  \[
  \textit{pre} \vdash \mathcal{U}^V(\langle \text{int } i0 = C \mid= 0 \rangle (\text{SINT::toInt}(i0) = 1 \Rightarrow (\langle B \rangle (\text{variant < variantSaved}_0 \land \text{variant} \geq 0)))).
  \]

Here $C$ is the condition of the while statement, $B$ is the while statement body, $\text{P2}$ is $\text{P1}$ with while statement removed, $i0$ is a temporary variable, $\textit{inv}$ is a loop invariant, $\textit{variant}$ is a non-negative loop variant that decreases with each execution of the loop body, and $\text{variantSaved}_0$ is a fresh skolem symbol needed to save the value of the variant before executing the loop body. $\mathcal{V}$ is an anonymous
update that resets the locations, which can be modified by C and B, with the values of fresh skolem symbols

\[ \mathcal{V} = \{ \text{ptr} = s_1; \text{SINT::value(count)} = s_2 \} \]

In principle we should reset all locations as the loop body must preserve invariant \( I \) for any initial state satisfying \( I \). However, by resetting only those locations that are modified within the loop body, we get a much more user-friendly rule that is still sound. This rule is a variation on the work described in [3].

Leaving the example program for a moment, if statements

\[ (\ldots \text{if} \ (E) \ B_1 \ \text{else} \ B_2 \ \ldots) \phi \]

are reduced into

\[ \text{if} \ (\text{SINT::toInt}(E) \neq 0) \ \text{then} \ ((\ldots \ B_1 \ \ldots) \phi) \ \text{else} \ ((\ldots \ B_2 \ \ldots) \phi) \]

when \( E \) is a variable with register specifier in its declaration and into

\[ (\ldots \ \text{register int i0} = E; \ \text{if} \ (i0) \ B_1 \ \text{else} \ B_2 \ \ldots) \phi \]

otherwise. Notice the pattern \( (\ldots \ \ldots) \) to match the surrounding context (currently only the block frames). Here

\[ \text{if} \ (C) \ \text{then} \ (B_1) \ \text{else} \ (B_2) \]

is equivalent to

\[ (C \Rightarrow B_1) \ \land \ (\neg C \Rightarrow B_2) \]

Such formulae in the succedent of a sequent lead eventually to case distinctions and two branches of the proof, unless one can prove that one of the branches is never taken.

7 CDL Expression Calculus

In this section we discuss how C expressions are reduced into updates by looking at the while statement body \( B \).

Let's consider the first statement \( \text{count} = \text{count} + 1; \). We introduce a special \textit{value-of} operator \( @ \) that denotes the value of a scalar object, so the statement becomes \( @\text{count} = @\text{count} + 1; \). The main principle of expression calculus is first reducing expression statements into simpler ones

\begin{align*}
\text{register int i1} & = @\text{count}; \\
\text{register int i0} & = i1 + 1; \\
@\text{count} & = i0;
\end{align*}

Now \( i1 = @\text{count}; \) is reduced into update

\[ \{ \ i1 := \text{SINT::value(count)} \} \]
but before that we have to prove that the value assigned is valid
isValidVal(SINT::value(count)), where

\[ \text{isValidVal}(i) \equiv \text{SINT::MIN} \leq \text{SINT::toInt}(i) \leq \text{SINT::MAX} . \]

Here \( \text{SINT::toInt} : \text{SINT} \to \text{int} \) converts CDL integers into mathematical integers. Statement \( i0 = i1 + 1; \) is reduced into update

\[
\begin{array}{l}
\{ i0 \leftarrow \text{SINT::fromInt(SINT::toInt(i1) + 1) } \},
\end{array}
\]

which means we have to prove that the result does not overflow

\[ \text{SINT::MIN} \leq \text{SINT::toInt}(i1) + 1 \leq \text{SINT::MAX} . \]

Note that this rule works only if mathematical integers have at most one bit representation, because otherwise \( \text{SINT::toInt}^{-1}(\text{SINT::toInt}(i1) + 1) \) is not uniquely defined.

Creating a calculus for C integer expressions is complicated by the fact that mathematical integers can have different bit representations on different platforms and even during one run of a program. Moreover, in general it is undefined what happens in the case of arithmetic operation or conversion overflows. For now we have chosen CDL integer types to be isomorphic with mathematical integers. We have chosen CDL integer values to be valid iff they fit into the bounds of minimum and maximum values of the corresponding C type and only proofs about programs that do not produce overflows can be closed. This approach works in the case when mathematical integers have at most one bit representation (e.g., two’s complement on IA-32) or when we want to abstract away from bit representations and thus programs with bit-operations are not supported. At the same time it is not hard to modify the integer expression calculus for any particular specialization of the C specification.

Now let’s consider the second statement of the loop body
\( \text{ptr} = @((\text{strlen})\text{next}); \). First, this statement is reduced into simpler statements

\[
\begin{array}{l}
o1 \leftarrow *\text{ptr};
o0 \leftarrow o1.\text{next};
\text{ptr} = @o0;
\end{array}
\]

Here \( o1 \) is of C type \text{struct Node} (CDL type \$Node) and \( o0 \) is of C type \text{struct Nodes} (CDL type \$Node@). In order to navigate object trees we have introduced an additional object reference assignment operator \( o \leftarrow oexp \) similar to JAVA reference assignments. Statement \( o1 \leftarrow *\text{ptr}; \) is reduced into update

\[
\begin{array}{l}
\{ o1 \leftarrow \text{ptr} \},
\end{array}
\]

which implies we have to prove that we can dereference the pointer \text{objExists} (\text{ptr}), where

\[ \text{objExists}(o) \equiv \text{isRealObj}(o) \land \text{storage(objBlock}(o)) \neq \text{ST_TRAP} . \]
Statement \( o0 \leftarrow o1\text{.next} \) is reduced into update
\[
\{ \ o0 := \text{SNode::next}(o1) \ \}
\]
while statement \( \text{ptr} = \text{oo0} \) is reduced into update
\[
\{ \ \text{ptr} := \text{SNode@::value}(o0) \ \}
\]
but before that we have to prove that the pointer assigned is valid
\( \text{isValidPtr}(\text{SNode@::value}(o0)) \), where
\[
\text{isValidPtr}(o) \equiv o = \text{null} \lor \text{isValidObj}(o) \land \text{storage(objBlock}(o)) \neq \text{ST\_TRAP}.
\]
Recall that \( \text{isValidPtr}(o) \) allows references to array tails.

Creating a calculus for C pointer expressions has many technical challenges. Dynamic arrays can only be referenced by pointers to the first element of the array, so pointer indexing \( p0 = \text{arr} + \text{index} \); in our example program (\( \text{arr}[\text{index}] \) is reduced into \( *\text{(arr + index)} \)) results in a cumbersome sequential update
\[
\{ \ p0 := \text{SINT@[]}::\text{elem}(\text{SINT@[]}::\text{objParent}(\text{arr}), \text{objParentIdx}(\text{arr}) + \text{SINT::toInt}(\text{index})) \ \}\!
\]
Here \( \text{objParent} : \text{Object} \rightarrow \text{Object} \) gives the parent object of the argument (in this case the parent array) and \( \text{objParentIdx} : \text{Object} \rightarrow \text{int} \) gives the index of the argument within its siblings (in this case the array elements). When comparing pointers for equality, a pointer to the first child of a struct or an array is equal to the pointer to their parent. Moreover, a pointer to an array tail may be equal to a pointer to a root object of any allocated block. In the case when pointer equality is undecidable we use a fresh skolem symbol as the result of the pointer comparison.

Now consider statement \( p0 = \text{calloc}(0, \text{sizeof(int)}); \) in our example program. As we work with a type-safe subset of C then such statement allocates objects of logic array type \( \text{SINT@[]} \) and with array size equal to \( \text{SINT::toInt}(0) \), while it's return value is a pointer to the first element of this array cast into type \( \text{Void} \). The resulting sequential update is
\[
\{ \begin{align*}
\ o0 := & \text{SINT@[]}::\text{lookup}(\text{next}, \text{SINT::toInt}(0)); \\
\ p0 := & \text{SINT@[]}::\text{elem}(o0, 0); \\
\text{storage(objBlock}(o0)) := & \text{ST\_ALLOCATED}; \\
\text{next} := & \text{next} + 1
\end{align*}
\}
\]
As \( \text{calloc} \) can also fail we have also to prove that postcondition holds in the case of update \( \{ \ p0 := \text{null} \ \}\). Statement \( \text{free}(p1); \) results in an update
\[
\{ \ \text{storage(objBlock}(p1)) := \text{ST\_TRAP} \ \}
\]
which means we have to prove that
\[
\text{objExists}(p1) \land \\
\text{storage(objBlock}(p1)) = \text{ST\_ALLOCATED} \land \\
\text{objAliased}(\text{rootObj}(p1), p1),
\]
where \texttt{objAliased(o1,o2)} is a reflexive transitive closure of the relation of \(o_2\) being the first child of \(o_1\).

Finally, C supports \emph{deep} value assignments \(o_1 = o_2\) of objects when all member values of \(o_2\) are copied into \(o_1\). Such assignments can be modeled by just rewriting them into a sequence of scalar assignments, but we reduce them into a straightforward update.

The order of evaluating the subexpressions of C expressions is unspecified in many cases. For this reason we require checking by some \emph{external} means that the different evaluation orders give the same result. Surprisingly, it is impossible to do type-checking of expressions in a completely portable way as the types of integer constants, the resulting types of integer promotions and of the usual integer conversions depend on knowing the exact minimum and maximum representable values of the types. For this reason we require all type conversions to be made explicit via type casts, that can be inserted automatically by a preprocessor, if needed.

8 Example Proof

Unfortunately, due to space limitations we cannot discuss the full proof of our example on Figure 2, however we present the constructions needed to prove that the first loop does what is intended. In order to express the precondition that the function argument \texttt{ifirst} refers to a linked list we need to introduce two fresh rigid functions \(len: \text{int} \rightarrow \text{Node}\). Now

\[
\begin{aligned}
\text{pre} & \equiv L(\text{ifirst}) \land \text{objExists(count)}, \\
\text{post} & \equiv \text{SINT::value(count)} = len, \\
\text{inv} & \equiv 0 \leq \text{SINT::toInt(SINT::value(count))} \leq len \land \\
\text{ptr} & = \text{list(SINT::toInt(SINT::value(count)))}, \\
\text{variant} & = len - \text{SINT::toInt(SINT::value(count))},
\end{aligned}
\]

where \(L(\text{ifirst})\) expresses that \texttt{ifirst} points to the beginning of the linked list

\[
L(\text{ifirst}) \equiv \left( \text{list}(0) = \text{ifirst} \land len \geq 0 \land list(len) = \text{null} \land \\
\forall \text{int } i; ((0 \leq i) \land (i < len) \Rightarrow Q(i)) \right)
\]

and \(Q(i)\) expresses the properties of the linked list node \(i\)

\[
Q(i) \equiv \left( \text{objExists(list}(i)) \land \\
\text{SNode::value(SNode::next(list}(i)) = list}(i + 1) \land \\
\text{isValidVal(SINT::value(SNode::elem(list}(i)))) \right).
\]

The constructions for the second part of the function \texttt{list2arr} are much more involved because of the side-effects on the heap (allocating an array and deleting the linked list). For instance, \texttt{count} should not point to any of the elem fields of the nodes of the linked list. We have also completely avoided the problem of what is not modified during the function execution.
9 Current Status and Further Work

At the moment of writing this text the type system, the signature, and the calculus described above are implemented in the KeY-C and we are at the stage of debugging the calculus and improving its usability. In essence, we can work with a large subset of C variable declarations, expressions, and we support while and if statements. However, recall that we only work with type-safe C programs. The program in our example can be reduced into FOL sequents, but we didn’t have time to close all branches of the proof yet.

Supporting full C, of course, requires a lot more work to reason about numerous small and not-so-small features of the C language — for loops, const and volatile modifiers, string literals, typedefs, enumerations, const expressions, unions, bit-fields, varargs just to name a few. Of the more conceptual extensions we could name introducing modularity; translation units, extern and static variables, function calls, and function pointers. Luckily the C module system can be viewed as a special case of the JAVA module system, so taking over the calculus from the KeY for JAVA should be straightforward, although laborious. Calling function pointers can be implemented in the same way as polymorphic method calls in JAVA.

Another interesting direction is supporting C jump statements — continue, break, return and goto. Of those the first three are present in the KeY for JAVA and should be easy to take over. The calculus for the goto statement is yet to be developed, however, it seems that we can put together a reasonable calculus by combining the ideas of switch statements, exceptions and loop invariants, which after all are a goto statement in different disguises.

Further, an update calculus could be extended to support unions and deep value assignments in a more natural form. A deep update calculus would allow a surface syntax for deep object value updates of the form $\{ o_1 := o_2 \}$ that would be semantically equivalent to copying all the members of object $o_2$ into $o_1$, e.g.,

\[
\begin{align*}
\{ & \text{$\text{SNode}::$value($\text{SNode}::$next($o_1$)) := $\text{SNode}::$value($\text{SNode}::$next($o_2$));} \\
& \text{$\text{SINT}::$value($\text{SNode}::$elem($o_1$)) := $\text{SINT}::$value($\text{SNode}::$elem($o_2$))} \}
\end{align*}
\]

The former syntax is clearly more readable. The main challenge here is to develop rules for update simplification and application.

Similar ideas could be used to support reasoning about non-type safe raw memory access (as bytes). The idea is first to introduce a location representing individual bytes

\[
\text{byte : Block, int } \rightarrow \text{ UCHAR} ,
\]

Now, for instance, $((\text{unsigned char}*)o)[3] = 234$ could be reduced into an update

\[
\{ \text{byte(objBlock(o), objOffset(o) + 3) := UCHAR::fromInt(234)} \}
\]

where objOffset(o) is object’s offset with respect to the beginning of the block. Now, we could introduce a surface syntax for deep object value updates of the
form \{ o_1 := o_2 \} that would be semantically equivalent to copying the bytes of object \( o_2 \) into \( o_1 \), i.e.

\[
\begin{cases}
& \text{for } i; \ 0 \leq i < \text{sizeof}(T); \\
& \text{byte}(\text{objBlock}(o_1), \text{objOffset}(o_1) + i) := \\
& \text{byte}(\text{objBlock}(o_2), \text{objOffset}(o_2) + i)
\end{cases}
\]

What you see here is a quantified update over variable \( i \) satisfying the condition \( 0 \leq i < \text{sizeof}(T) \), where \( \text{sizeof}(T) \) tells the size of object type \( T \) in bytes. The idea is that a type-safe program would only require deep value updates of the former form, while in a non-type-safe program only those places that work explicitly with bytes would require the latter kind of updates. Creating a dynamic logic for raw memory access without a deep update calculus would mean using only the latter kind of updates, which, we surmise, would be unusable for non-trivial programs.

Finally, in C any pointer can point to any memory location and consequently significant portions of program contracts and proofs are devoted to proving that object references cannot be aliased. We surmise that a static analysis could be developed to compute non-aliasing properties of the object references which could then be used to simplify the proofs. One possible way to use this information is to modify the CDL type system so that unalised references to the same C type receive distinct types in CDL that have the same structure (i.e., member or value accessor functions). Another very simple static analysis could detect scalar variable declarations that are never taken address of and prepend a \texttt{register} specifier to them.

10 Related Work

ACE (Assertion Checking Environment) [13], Caduceus [6], and the separation logic Isabelle/HOL approach described in [14] are all based on Hoare-style deductive verification using interactive theorem proving. Both ACE and Caduceus translate the C program (ACE expects a MISRA-C program while Caduceus can handle type-safe C programs) annotated with formal specifications to an intermediate language. In ACE, this intermediate product is given as input to the Stanford Temporal Prover (STeP). Caduceus uses it to generate the needed verification conditions which are then to be proved using one of several supported theorem provers (e.g., Coq and Simplify). In [14], the authors present a formal memory model for C programs and describe the implementation of this framework in the theorem prover Isabelle/HOL. The idea is to support both a low-level view of the heap as an array of bytes and two high-level views—multiple typed heaps and separation logic—and to unify these views in a sound way. This way, the verification approach can accurately reflect the real semantics of C, while well-behaved programs (that use pointers in a type-safe way) are still easy to reason about. Comparing these three methods to our approach reveals that, in ACE, Caduceus, and the Isabelle/HOL approach, interleaving of symbolic execution and logical reasoning is not possible and interaction does not take place
at the level of C source code but on intermediate code. As part of the Verisof
t project (www.verisoft.de) a Hoare calculus and formal semantics of the C subset
C0 on top of Isabelle/HOL were developed [12], however the verification of C
programs is less automated than in KeY.

MAGIC [4], SLAM [1] and BLAST [7] are all verification approaches based on
the abstract-verify-refine paradigm. An abstract model of the program is created
using predicate abstraction and then model checking is used to check whether
the abstraction conforms to the specification. If the verification step fails due
to a too coarse abstraction, information gained from the failed verification step
is used to refine the model and then the verification step is repeated. MAGIC
verifies C programs with respect to finite state machine specifications and uses
simulation as the notion of conformance. Both SLAM and BLAST check safety
properties of C programs and use reachability analysis to check conformance.
All three mentioned approaches assume that the C program is type-safe. An
other approach based on abstraction is Atrée [9], a static analyzer based on
abstract interpretation [5]. Atrée focuses on subtle machine implementation as-
spects such as rounding errors introduced by floating-point arithmetics. Atrée
can automatically discover all potential run-time errors of a certain class using
a sound approximation of the C semantics. Compared to our approach, the
abstraction-based methods described above are more automated than ours but
less expressive. Verification of functional properties is not possible.

11 Summary

In this paper we briefly described the ongoing effort to develop KeY-C, the C
variant of the verification system KeY. As a side-effect of this work we
expect to be able to generalize KeY architecture for easily adding the support for new
programming languages. The implementation is done in JAVA and we are using
the Cetus framework [8] for parsing and analysing C programs. The JAVA target
version of KeY is available from www.key-project.org. KeY-C is available from
the authors on request. We would like to acknowledge the members of the KeY
project for fruitful discussions.

References

1. Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging system
software via static analysis. In Conference Record of POPL’02: The 29th ACM
SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages
2. Bernhard Beckert, Reiner Hilfner, and Peter Schmitt, editors. Verification of
Object-Oriented Software: The KeY Approach, volume 4334 of LNCS, Springer,
2006.
while loops in deductive program verification. In Kung-Kiu Lau, editor, Proc. 7th
International Conference on Formal Engineering Methods (ICFEM), Manchester,


Formal Methods in the Robin project: Specification and verification of the Nova microhypervisor

Hendrik Tews*
Radboud Universiteit Nijmegen, The Netherlands
http://www.cs.ru.nl/~tews
June 5, 2007

The objective of the Robin project is to develop an open robust computing infrastructure. The Nova micro hypervisor is currently being developed as a basis for this robust infrastructure. One workpackage of Robin concentrates on the application of formal methods to this newly developed micro hypervisor. The goals within Robin are (1) to verify some properties of a selected hypervisor module and (2) to develop a formal specification for the hypervisor interface.

This paper presents our approach for the verification of Nova. I will especially discuss the challenges and some solutions of operating-system kernel verification.

1 Introduction

The aim of the Robin project is to solve the following sort of dilemma: PDA’s are used for browsing the world-wide-web and for storing private data. For web-browsing one wants to install latest software however, this opens the door wide for every attacker.

The solution that is currently developed in the Robin project is depicted in Figure 1. It is usually called the Nizza security architecture. A micro-hypervisor exploits the virtualization support in the new x86 CPU’s to provide a basis for running dedicated applications and possibly multiple copies of legacy operating systems (like Linux and Windows). Pieces of software can be completely encapsulated such that they can only communicate to the outside through special channels. With such an architecture

*The author has been supported by the European Union through PASR grant 104600.
one can browse the world-wide-web in one operating system (OS) instance, while, at
the same time, one also writes a classified document in a different OS instance. The
hypervisor makes sure that the browser OS cannot see the classified document, even
if both OS instances have been taken over by an attacker. The Encapsulation ensures
that the classified document can only be sent through one channel to an encryption
module that runs directly on the hypervisor. Even if the installation media for the
legacy OS was compromised, an attacker cannot get access to the classified document.
For more information on the Nizza architecture see [Tew07, HHF+05, FH06, FH03,
HPHS04, Fes06].

The goal of work-package 4 of the Robin project is to develop a verification approach
such that some properties of one selected hypervisor module can be mechanically
proved. Further we plan to develop a formal specification of the hypervisor interface.
For the specification the main challenges at the moment are non-technical. Our aim
is to make the formal specification part of the hypervisor documentation. For that it
must be based on simple set theory and the public parts of the specification must be
free of fancy symbols such as $\in, \forall, \subseteq$.

In the remainder of the paper I concentrate on the verification goal. Section 2
describes the challenges, Section 3 the verification approach and Section 4 some goals
of the verification.

2 Challenges of low-level system-software verification

A verification of an OS kernel, even if it is only a small hypervisor is a very challenging project for a number of reasons.

**C++ source code**  Currently, there seems to be no convincing alternative to C/C++ for kernel programming, at least from the point of view of many OS groups. Consequently the Robin hypervisor is written in C++. C++ programs are difficult to formalise for a number of reasons:

- The C++ standard [Int98] is relatively vague in order to permit conforming C++ implementations on the widest platforms. For instance the signed integral types are not required to contain negative numbers. Further, casts between different pointer types might change the pointer (to satisfy alignment requirements), except for the case where one casts to `void *` and back to the original pointer.

  Because of the vagueness of the C++ standard almost every program relies in some way on platform or compiler specific properties. Consequently, a formalisation of C++ program must incorporate some properties of the specific C++ implementation that is used to compile the program.

- The template mechanism of C++ alone is Turing complete [Vel]. This means the compiler can be forced to do arbitrary computations at compile time. A formalisation of C++ templates is accordingly difficult.

  The micro hypervisor will only use few templates. If they are getting too difficult we will work with the template instantiations instead.

- Type casts and `goto`-jumps are features that are traditionally not handled in textbooks on program semantics. However, it is impossible to write a micro hypervisor without typecasts and to avoid unduly performance penalties one needs some kind of unstructured jump such as `setjmp/longjmp` [Lon] at a few places.

**Embedded assembly code and direct hardware manipulations**  For operations that are not supported in C++ (mostly direct hardware manipulations) the hypervisor sources will contain some assembly code, mostly in the form of inlined assembly. Assembly code is needed at least for the following operations:

- Access to hardware registers, such as those from the APIC (Advanced Programmable Interrupt Controller), but also special CPU registers, such as CR3 (page directory base register), EFLAGS (the flags register), the global descriptor table, the interrupt descriptor table, the task segment register and the feature control registers CR0 and CR4.
The hardware model and the semantics of data types provide the basic operations and properties for the verification of the hypervisor. For hardware data types the hardware model relies on the semantics of data types. Technically

$$\Phi_{\text{data-types}}, \Phi_{\text{hardware}} \vdash \varphi(\text{hypervisor})$$

where \( \varphi \) is one property from the hypervisor specification, such as termination without runtime type errors.

**Figure 2: Robin verification approach.**

- Embedding special instructions in the code, such as IRET (return from interrupt) and INVLPG (invalidate a TLB entry).

- Manipulating the stack frame to access and modify parameters of system calls or for programming non-local exists similar to `longjmp`.

**Nonstandard program environment** The hypervisor runs like usual programs in virtual memory. However, the hypervisor manipulates the virtual memory mapping itself. Some parts of the memory are visible multiple times at different virtual addresses. One can therefore have very subtle aliasing: A variable \( x \) at address \( a_1 \) can be changed by writing to the totally different address \( a_2 \).

The hardware manipulations that the hypervisor must perform bear the possibility of subtle errors. Certain bits in hardware data structures, such as the page directory entries, must be zero. A more subtle problem comes with the translation look-aside buffer (TLB). The TLB is a special kind of cache that caches page-directory traversals. As a cache the TLB is not transparent, which means, when changing a page-directory or page-table entry one must manually invalidate the TLB before using the new address mapping. Otherwise, depending on the execution history, the old mapping, still cached in the TLB, might be used.

## 3 A verification approach for a Micro Hypervisor

In this section I explain the approach that we are planning to use in the Robin project. The approach is depicted in Figure 2, it has already been worked out in the VFiasco project [HT05, HT]. Our approach heavily relies on the interactive theorem prover PVS [ORR+96]. The input language of PVS is higher-order logic enriched with predicate subtyping and some other forms of dependent types. For the verification we model
the x86 hardware and the hypervisor inside PVS and use later the prover component of PVS to establish theorems about it.

Our verification approach uses source code verification. That is, we translate the C++ source code into a set of specific functions that are defined in the PVS input language. Source code verification also means that we do not directly verify the object code that will really be running. However, source code verification lets us profit from the relatively high abstraction level present in the source code (which is lost in object code). A connection to the real object code is vaguely planned for the far future.

In our approach the translation of the C++ code into PVS depends on the hardware model and the semantics of data types. Both, the hardware model and the semantics of data types are PVS specifications that are currently developed. As expected the semantics of data types deals with C++ data types in PVS. Our semantics of C++ data types exploits underspecification to make it possible to detect erroneous type casts and wrong implicit type conversions (like, for instance, reading data from a union with the wrong type), see [HT03].

The hardware model formalises an abstract model of the x86 hardware inside PVS. It provides physical memory, virtual memory with address translation via page directories, some kind of TLB and much more. The hardware model does not blindly model the real hardware. Instead the hardware is modelled in such a way that certain subtle programming errors yield a specific error state instead of doing nonsense (like the real CPU). For instance the attempt to interpret a string as a page directory entry yields an abnormal result value. This kind of error checking works even for the hardware initiated page directory traversals done during address translation.

In order to translate C++ into PVS we use a denotational semantics for (a subset of) C++. This denotational semantics has been developed partly already in the VFiasco project. It correctly treats type casts, goto jumps and all the other complications that I pointed out in the preceding section. One can view the hardware model and the semantics of data types as providing the basic building blocks of our denotational C++ semantics. In our design the three components (C++ semantics, hardware model and data types) are relatively independent from each other. It is therefore possible

- to add additional axioms to the data types, for instance, to model a compiler specific assumption about the size of some data types or the precise behaviour of some type casts.
- to add new operations to the hardware model
- to use different versions of the hardware model for different pieces of the hypervisor. The boot code of the hypervisor can be verified against physical memory and the hardware independent parts can be verified against a traditional, untyped memory model.
- to adopt the semantics for new C++ features or compiler specific C++ constructs.

Figure 3 depicts the data flow of our verification approach. A semantics compiler translates the C++ source code into its semantics in higher-order logic and outputs
The figure shows the approach for source code verification. The process involves C++ sources with annotations, which are compiled using a semantics compiler (using Olmar) to produce semantics in HOL. This semantics is then translated into PVS as PVS source code. There will be two kinds of annotations in the source code. The first kind influences the syntactic form of the output. It will, for instance, be possible to place the semantics of a block or a group of statements into a separate function, in order to make it possible to modularise the verification. The second kind of annotations contains specifications for the code similar to JML [BCC+05]. The semantics compiler translates these specifications into PVS proof obligations.

The hardware model and the data type axiomatisation are directly developed in PVS. From the PVS point of view they provide declarations for all the function that appear in the output of the semantics compiler. With all the different pieces loaded in PVS one can start to prove hand-written, external specifications of the hypervisor.

The semantics compiler translates the sources of the program into semantic functions that precisely model the behaviour of the original source code. A semantic function is a state transformer of the following form:\footnote{I use \( \uplus \) for disjoint union. The notion \( \text{State} \) provides meaningful names for the injections. \( \text{1} \) is the unit or one-element set.}

\[
\text{State} \rightarrow \text{ok: State} \uplus \text{pagefault: State} \times \text{Page-fault info} \uplus \text{hang} \uplus \text{fatal} \uplus \ldots
\]

Here \( \text{State} \) is a set of machine states provided by the hardware model. Every state contains the contents of the physical memory and the contents of some important control registers (such as virtual address mapping or the stack pointer). The disjoint union on the right-hand side describes the possible results of a state transformer. If no abnormal condition occurs it yields a successor state tagged with \text{ok}. If a page fault occurs it yields a successor state plus some additional information. A result tagged with \text{hang} means that the program did not terminate (for instance because of a nonterminating while loop or a page fault that keeps occurring at the same instruction).
The result *fatal* is reserved for serious errors such as TLB inconsistency or reserved bit violations.

State transformers can be composed in the obvious way: If the first state transformer yields a result tagged with *ok* the result state is passed into the second state transformer. If any abnormal conditions occur the second state transformer is skipped and the result of the composition is the abnormal result of the first state transformer.

The hardware model provides the basic state transformers for reading to and writing from memory and for reading and writing the control registers. The semantics compiler composes the basic state transformers from the hardware model to build the semantics of its input program.

Program verification proceeds by reasoning in PVS over a nontrivial state transformer that represents the semantics of some source code. This is mostly done by applying a start state to the state transformer and proving properties of the result (for instance that the result is *not* tagged *fatal*). Such a verification could equivalently be performed by computing the weakest precondition of the verification goal with respect to the program.

A slightly different view on the verification is as follows: The hardware model defines a state machine. The basic state transformers of the hardware model describe the actions of the state machine. The program is symbolically executed on top of the state machine. Properties are derived from the state changes that one observes.

### 4 Verification goals for the Robin Micro Hypervisor

The preceding section made very clear that the precision of the verification hinges on the hardware model. With precision of the verification I refer to the amount and kind of errors whose absence is proved with a successful verification.

It is our aim to make the hardware model precise enough to let it catch the following kinds of errors:

- *Common errors*, such as dereferencing a null-pointer, nonterminating loops, wrong results (wrt. the functional specification).

- *Type errors*. A type error occurs when one attempts to read an instance of some type from a memory location where nothing or an instance of a different type has been stored. The hardware model will be precise enough to catch type errors for user code (for instance resulting from wrong pointer or address calculations) and for implicit hardware memory accesses (for instance reading page directories).

- *Virtual-memory aliasing errors*. Virtual-memory aliasing occurs when the virtual memory of two distinct variables is mapped to the same (or overlapping) physical memory region. A virtual-memory aliasing error happens if one has virtual-memory aliasing for two variables that are used at the same time.

- *TLB errors* (accessing a linear address\(^2\) for which the page directory or page table entry might be inconsistent with the translation look aside buffer)

\(^2\)On the IA32 architecture virtual addresses, which are page-wise mapped to physical addresses, are
• allocation errors (using the same memory for different variables at the same time)

Within Robin we are not targeting the following errors:

• any kind of hardware error

• errors that can only occur on systems with more than one logical processor (i.e., on systems with multiple CPUs or with active hyper-threading)

At the moment we have several candidates of proof obligations that we would like to verify for the hypervisor in the future.

Normal termination The hardware model contains a lot of checks (like for instance reserved bit conditions and TLB consistency) that enter an abnormal state if they are not fulfilled. In order to prove that the hypervisor does not contain this kind of errors it is therefore sufficient to prove normal termination.

Dynamic type correctness Because of the type casts and the internal memory management the type correctness of the hypervisor cannot be checked with a type system. Instead, type correctness has to be established during verification. The semantics of data types relies on underspecified functions for reading and writing data from and to memory. Axioms for these functions just cover the case of reading some data type from a memory location where data of the same data type has been written before. In case of a type error, where one tries to read from a memory location that contains garbage or data of a different type, none of these axioms apply. One can thus proof nothing, not even normal termination, about a program with a type error. In order to prove type correctness it is sufficient to prove normal termination of the hypervisor code. The consistency of the axioms will be shown using refinement of theories in PVS [OS01]. For more details see [HT03].

Only kernel code runs in kernel mode One of the most terrible programming errors of an operating system is to execute arbitrary user level code with kernel mode privileges. This can happen if the hypervisor forgets to reset the privilege level on return to user code. It can also happen if the user manages to exploit a buffer overflow inside the kernel. In order to prove that only kernel code runs in the highest privilege level one has to prove that nobody tampers with the return addresses on the stack and that all control path that leave the kernel reset the privilege level in the right way.

With security applications in mind it would also be very interesting to prove the following.
**Address space separation** If one address space (read process) has access to memory of another address space, then this memory has previously been explicitly mapped from one of the involved address spaces to the other one. This property ensures that a legacy operating system cannot see the memory with the cryptographic keys of the encryption module, unless there is a very stupid programming error in the encryption module.

However, high level properties like this are currently not in our scope. We first have to be successful with more basic properties.

In principle one would like to prove that, whatever code is running inside one of those legacy OS's, it is impossible to break the encryption of the encryption module. However, this requires an attacker model which has not been considered yet in cryptography. Typical attacker models in cryptography are such that the attacker has full access to the messages on the internet and can additionally control some hosts there. What we need here, is an attacker that is able to execute arbitrary code on the CPU that runs the cryptographic engine. Because covert channels can only be minimised but never completely avoided, the attacker can additionally observe the internal state of the cryptographic engine at a very low bit rate. We are not aware of any work with such an attacker model.

### 5 Conclusion

This paper presents the approach that is followed in Nijmegen to verify some properties of the micro hypervisor, which is currently developed as basis of the Nizza architecture. I also discuss the special challenges of operating system kernel verification and some interesting properties we would like to verify.

### References


[Fes06] N. Feske. TUDtOS Demo CD. Available at demo.tudos.org, March 2006.


[lon] longjmp, siglongjmp — non-local jump to a saved stack context. Linux manual page.


[Vel] T. L. Veldhuizen. C++ templates are turing complete. Available at cite-seer.ist.psu.edu/581150.html.
Analysing Embedded System Software
—Extended Abstract—

Ansgar Fehnker, Ralf Huuck, Felix Rauch, and Sean Seecfried

National ICT Australia Ltd. (NICTA)* and University of New South Wales,
Locked Bag 6016, Sydney NSW 1466, Australia

Abstract. The verification of real-life C/C++ code is inherently hard. Not only are there numerous challenging language constructs, but the precise semantics is often elusive or at best vague. This is even more true for systems software where non-ANSI compliant constructs are used, hardware is addressed directly and assembly code is embedded. In this work we present a lightweight solution to detect software bugs in C/C++ code. Our approach performs static checking on C/C++ code by means of model checking. While it cannot guarantee full functional correctness, it can be a valuable tool to increase the reliability and trustworthiness of real-life system code. This paper explains the general concepts of our approach, discusses its implementation in our C/C++ checking tool Gomma, and presents some performance results on large software packages.

1 Introduction

Showing the full functional correctness of system software written, e.g., in C/C++, is a major challenge. It requires a precise understanding of the underlying semantics, typically needs to include an abstract hardware model, and has to give a full functional proof. There are a number of projects currently undertaking this task [1–3] supported by interactive theorem provers. While this is the only way to guarantee the full correctness of a program, it requires substantial resources both in time as well as in the number of highly qualified people.

On the other hand, commercial system software has a high pressure to market, needs to run on various platforms and is rewritten frequently, making the above approach even more challenging. There are a number of lightweight analysis approaches that seek to complement full verification by detecting software bugs at the coding stage and, thus, increasing the reliability and trustworthiness of the code. These tools make a limited but practical contribution to program correctness and can support full verification by reducing property violations in early stages.

* National ICT Australia is funded by the Australian Government’s Department of Communications, Information Technology and the Arts and the Australian Research Council through Backing Australia’s Ability and the ICT Research Centre of Excellence programs.
The model-checking community has made significant advances in recent years to cover realistic C/C++ programs and produced a number of powerful tools [4–7]. However, they are not yet well-suited for real-life embedded system code [8, 9]. On the other hand, commercial static analysis tools [10–13] cope well with most C/C++ code and make a valuable contribution to software correctness. In contrast to model checking tools, static analysers typically do not allow for any user-defined specifications, but rather implement a set of independent analysis heuristics or allow specification which are less expressive than the temporal logics used by model checkers.

In this work we present a static analysis approach based on model checking. While we retain the flexibility and power of temporal logics specifications, we are able to handle any parsed C/C++ code in a uniform manner. In particular, we present the underlying idea of translating C/C++ checks into model checking properties, which can then be checked by one single analyser, instead of a set of static analysis heuristics. In our case we use the NuSMV [14] model checker as back end. Moreover, we present some implementation details of our checker *Goanna* and its performance on the source of large, real-life open-source software packages.

Section 2 describes our underlying framework, while Section 3 presents some of our performance results and Section 4 discusses the current state of our research as well as ongoing and future work.

2 Static Analysis by Means of Model Checking

In this section we describe how to statically check properties of C/C++ source code by means of model checking. This approach has been inspired by [15, 16] and is also followed by [17, 18].

Using a model checker for solving static analysis problems has a number of advantages. All properties can be expressed in a single, flexible analysis engine. This means that it is easy to add new checks by adding new checking properties. In addition, the analysis scales well with increasing number of properties. The details of our path-sensitive, intra-procedural analysis can be found in [19].

The basic idea is to annotate the control flow graph (CFG) of a program with atomic propositions of interest. In order to check, e.g., for uninitialised variables, we can identify atomic propositions $\text{decl}_q$, $\text{read}_q$ and $\text{write}_q$, representing program locations where a variable $q$ is declared but not initialised, where it is read from or written to, respectively, and mark those locations in the CFG accordingly. The atomic propositions are identified by purely syntactic criteria on the abstract syntax tree (AST) of the program by means of a pattern language. We define patterns for each proposition, e.g., a variable is written to if it occurs on the left hand side of an assignment statement and so on. Once identified, the proposition is placed on the node in the CFG most closely corresponding to the nodes in the AST where it was identified.

An example of the resulting annotated CFG can be found in Figure 1. This representation is already very close to a Kripke structure and we can model check
that structure for properties of interest. For instance, checking for uninitialised variables can be expressed in CTL as:

\[ AG \text{ decl}_q \Rightarrow (A \neg \text{read}_q \ W \ \text{write}_q) \]

This means we require that on all program paths if a variable \( q \) is declared it must not be read until it has been written or it will not be written at all. We use the \textit{weak until} operator \( W \) here to include the second possibility. The latter can also point to unused variables, which is checked separately.

In the same style we can express other properties on correct pointer handling, variable usage or memory allocation and deallocation. Moreover, it allows specifying application specific properties to handle general programming guidelines, API-specific rules or even hardware/software interface rules for device drivers.

Once the patterns relevant for matching atomic propositions have been defined and the CFG has been annotated, it is straightforward to translate the annotated graph automatically into the input format of a model checker. Adding new checks only requires one to define the property to be checked and the patterns representing atomic propositions. All other steps can be fully automated.

Although this framework was developed in first instance for C/C++ it can be also extended to deal with embedded assembly code. This is important for the embedded systems space, since interaction with the hardware is frequently implemented as embedded assembly code. In particular, we take C/C++ and ARMv6 assembly interface information for our analysis into account, check for compliance of embedded assembly code with its C/C++ interface, and perform various checks on the pure assembly level. The combined analysis of C/C++ code with embedded assembly code enhances, in addition, the precision of the analysis.
3 Implementation and Evaluation

The aforementioned approach has been implemented in our program analyser Goanna, using the open source model checker NuSMV [14] as a generic back-end analysis engine. The surrounding code for pattern matching structures of interest, property definitions, CFG generation, translation into NuSMV, and representation of analysis results is written in OCaml. Moreover, Goanna can be invoked just like the gcc/g++ compiler and, therefore, integrates seamlessly into standard development environments such as Eclipse (cf. Figure 2).

![Eclipse integration of Goanna](image)

**Fig. 2.** Eclipse integration of Goanna

We evaluated Goanna on a number of open source packages ranging from highly optimized system software such as the L4 microkernel\(^1\) to large application code bases such as the 260 kLoC\(^2\) OpenSSL package.

For an unoptimized version of Goanna some run-time results for OpenSSL are shown in Figure 3. It shows that over 80\% of all files are analyzed within 1 second and that 99\% of all files are analyzed within 5 seconds. The whole analysis takes less than 15 minutes. Proportionally, the time spent purely in NuSMV is mostly negligible with 98.7\% of all files being analyzed in less than 2 seconds.

The run-times of Figure 3 are based on checking for 15 properties ranging from simple uninitialized variables, over potential null-pointer dereferences, to memory leaks. It is worth to mention that increasing the number of properties typically scales well in our framework as it only increases the number of labels and property specifications in the same NuSMV model, which is handled well by

---

\(^1\) [http://l4hq.org/](http://l4hq.org/)

\(^2\) LoC = Lines of Code
the model checker. For instance, increasing the number of properties from one to 15 only doubled the overall analysis time.

Moreover, we found that the analysis time is not only well within the same order of magnitude as the compile time, but that the memory requirements of the analysis fit easily in the RAM of current developer machines.

The analysis of C/C++ code with embedded assembly code was evaluated for Pistachio 0.4 implementation\(^3\) of L4, compiling for an ARM SA1100 architecture. It contains 54 C++ files, two of which have embedded assembly blocks (3.7%), and they include a total of 72 header files, of which 10 have embedded assembly blocks (13.8%). The additional assembly analysis lead to a modest increase from 75.9 seconds to 77.3 seconds, which is an increase of only 1.4 seconds or 1.8%.

4 Conclusion

**Summary.** In this work we presented our framework and results on model checking system software by means of static analysis. We showed how to easily encode static checks as model checking properties, providing the basis for an extendable and flexible checker. Moreover, we implemented our analysis framework in Goanna, the first static checker using NuSMV as its analysis engine, and presented some run-time and scalability results. We showed that this is a viable solution that can be integrated well in the software development process.

**Ongoing and Future Work.** Currently, we are working on improving the precision of the analysis. Future work will focus on further increasing the performance of Goanna, integrating a full inter-procedural analysis and defining a user interface for property specification.

\(^3\) [http://l4hq.org/](http://l4hq.org/)
Acknowledgements We thank Bernard Blackham, Jörg Brauer, Patrick Jayet and Michel Lusenberg for their implementation efforts and general contributions.

References

Verification of C Programs Via Natural Semantics and Abstract Interpretation
(Extended Abstract)

Roberto Bagnara\textsuperscript{1}, Patricia M. Hill\textsuperscript{2}, Andrea Pesco\textsuperscript{t}t, and Enea Zaffanella\textsuperscript{1}

\textsuperscript{1} Department of Mathematics, University of Panna, Italy
\{bagnara, pesco\textsuperscript{t}t, zaffanella\}@cs.unipr.it
\textsuperscript{2} School of Computing, University of Leeds, UK
hill@comp.leeds.ac.uk

We are witnessing a substantial lack of available tools able to verify the absence of relevant classes of run-time errors in code written in (reasonably rich fragments of) C and C++. This is despite the progress made in recent years in the fields of program analysis and verification, and despite the huge impact such tools could have on the quality of a good portion of our software universe. It is interesting to observe that, among the dozens of freely available software development tools, hardly any, by analyzing the program semantics, are able to certify the absence of important classes of run-time hazards such as, say, the widely known \textit{buffer overflows} in C code. The reason is, of course, that C and C++ are complex languages and the techniques that can be used to dominate this complexity still do not reduce tool development to simple, manageable tasks. Our overall aim for this research is to investigate how known techniques based on natural semantics and abstract interpretation can be extended so as to conveniently formalize and implement a range of analysis and verification tools for modern imperative languages such as C and C++.

\textit{The language.} With this aim in mind, in [2] we define a core language —called CPM— that has much in common with C and includes several features that are problematic from the point of view of the semantic analysis of C and C++ code: recursive functions, run-time system and user-defined exceptions, realistic data and memory models, pointer types to both data objects and functions, and non-structured control flow mechanisms. Note that the contradiction between targeting “real” imperative programming languages and choosing CPM, an unreal one, is only apparent. As C misses exceptions and C++ is too hard as a starter, choosing any one of these would not have allowed us to assess the adequacy of the methodology described below with respect to the above goals.

\textit{Static analysis.} Verification of many program properties using static program analysis via abstract interpretation [9,11] is now a well-researched area. Static analysis is conducted by mimicking the concrete execution of the programs on an abstract domain. This is a set of computable representations of program properties equipped with all the operations required to mirror, in an approximate though correct way, the real, concrete executions of the program. Therefore, a formal \textit{concrete semantics} for the language to be analyzed that models all the
aspects of executions that are relevant to the properties of interest must be provided. Of course, we need to work on a language that is completely defined. For the C language, for instance, this can be achieved by converting the source programs to some more constrained language —like CIL, the C Intermediate Language described in [17]— where all ambiguities have been removed and by fixing an ABI (Application Binary Interface) so as to conform to the C implementation of interest. The problem is then to define a concrete semantics for the fully specified language so as to ensure that:

(i) this semantics is recognizable as a sound characterization of the language at the intended level of abstraction;
(ii) this semantics observes the properties that are the subject of the verification problems of interest;
(iii) this semantics allows for the computation of precise abstractions.

We now review the first two points; we will come back to the third point after introducing the abstract semantics.

Concrete semantics. We need a formal semantics that can be recognized (without requiring a strong mathematical background) as corresponding to the intuitive, often involved and incomplete explanations provided by standardization documents. For this purpose, we adopted the G∞SOS approach of Cousot and Cousot [12] which generalizes with infinite computations the natural semantics approach by Kahn [16] which, in turn, is a “big-step” operational semantics defined by structural induction on program structures in the style of Plotkin [18]. A semantics for CPM is then expressed by means of a concise set of rather simple rules that are quite readable and, most importantly, directly correspond to executable Prolog clauses. What was not clear to us when we started this work is whether the approach “scales” when applied to languages like C: for example, how can run-time errors and non-structured control flow mechanisms be modeled in this framework? We now know that the natural semantics is fit for the purpose:

- the formal semantics of CPM \(^3\) has been successfully explained to undergraduate students in their 3rd year of Computer Science;
- the Prolog implementation of this formal semantics in the ECLAIR system\(^4\) (with the help of a C++ implementation of memory structures) is efficient enough to allow the execution of non-trivial C programs, something that enables everyone to build confidence on the fact that the concrete semantics is faithful to the intuitive, informal semantics.

Concerning point (ii) above, we can only formally reason about properties if they are observable in the chosen concrete semantics. For example, if we want to

\(^3\) Which includes CIL as a sublanguage in addition to the exception handling features of C++ and Java.

\(^4\) The ‘Extended CLAIR’ system targets the analysis of mainstream programming languages by building upon CLAIR, the ‘Combined Language and Abstract Interpretation Resource’, which was initially developed and used only in a teaching context (see http://www.cs.unipr.it/clair/).
prove that a program uses pointer arithmetic in a safe way, we need a concrete semantics that allows us to observe all the unsafe uses. Such a concrete semantics cannot simply model pointers as plain addresses, as more information is required than that to detect the violations. In the concrete semantics for CPM, these violations are optionally reported as run-time exceptions so that, proving that such an exception can never be thrown, amounts to proving the desired property. All exceptional and undefined behaviors (such as divisions by zero, overflows of signed integer variables and dangerous uses of the shift operators) are modeled by exceptions. This, besides the need to deal with user-defined exceptions as found in C++, Java and Python, is the reason for the inclusion in CPM of exception propagation and handling mechanisms. Note however that accommodating exceptions impacts on the specification of the other components of the semantic construction. For example, short-circuit evaluation of Boolean expressions cannot be normalized as proposed in [8], because such a normalization process, by influencing the order of evaluation of subexpressions, is unable to preserve the concrete semantics as far as exceptional computation paths are concerned.

Abstract semantics. Following the abstract interpretation approach, we also require an abstract semantics that has a correlation with the concrete semantics. In addition, we require that appropriate abstract domains are available that can provide correct approximations for the values and all the operations that are involved in the concrete computation [9-12]. For CPM we have formally defined in [2] an abstract semantics framework that follows and extends the approach outlined in the works by Schmidt [19-21]. We have proved that any semantics within this framework will have a safe correlation with the concrete semantics (a summarized version of this proof is available in [2]). Moreover, this framework has been designed to be both modular and generic. It is modular because the overall static analyzer is naturally partitioned into components with clearly identified responsibilities and interfaces, something that greatly simplifies both the proof of correctness and the implementation. It is generic, since it is designed to be completely parametric on the analysis domains. In particular, and here we come to point (iii) above, it provides —differently from all published proposals we know of— full support for relational domains (i.e., abstract domains that can capture the relationships between different data objects). Achieving this goal constrains the design of both the concrete and the abstract semantics. As was the case for the concrete semantics, the abstract semantics rules for CPM are almost directly translated to generic Prolog code that can be interfaced with specialized libraries implementing several abstract domains, including accurate ones such as those provided by the Parma Polyhedra Library [3-5]. So this working prototype, which is currently being extended with the pointer analysis described in [13-15], demonstrates that the proposal of Schmidt can play a crucial role in the development of reliable and precise analyzers for real imperative languages including C, Java and, we believe, C++ and RPython (http://pypy.org/).

Further work. Although our framework is only fully specified for the core CPM language, and this encompasses C but not C++, we do not have a definite answer
concerning the appropriateness of our proposal for the verification of C++ programs. That said, we do not see what, in the current design, would prevent the extension of the core language together with its concrete and abstract semantics so as to handle any other features of mainstream, single-threaded imperative programming languages.

Our proposed analysis framework is parametric on abstract memory structures. While the literature seems to provide all that is necessary to realize very sophisticated ones, we can confidently predict that, among all the code out there waiting to be analyzed, some will greatly exacerbate the complexity/precision trade-off. The ability to analyze C programs will confront us with a huge variety of inputs and it is hardly likely that the same compromises will be able to accommodate programs as diverse as the huge, pointer-free, synthesized loops handled by *ASTRÉE*\(^5\) and, say, libraries for manipulation of strings. However, these are research problems for the future — now we have a formal design on which analyzers can be built, our next goal is to complete the build and make this technology truly available and deployable.

References


\(^5\) The *ASTRÉE* analyzer can automatically verify the absence of some kinds of runtime errors in large safety-critical embedded control/command codes [6, 7].
DeHydra Source Analysis Tool

Taras Glek *

April 10, 2007

Abstract

Static analysis tools are a mature technology. They can be applied to large codebases and can find certain classes of errors in large code bases. However even if the tools have an internal AST-level view of the code analyzed, they do not allow custom queries of the AST. Instead, they are largely used to check certain “ canned” properties and either do not support user-extensible checks or provide limited domain-specific languages specialized for specifying automatons.

This paper describes DeHydra, a code analysis tool that can specify automatons in addition to allowing other code queries and checks. It uses JavaScript which is a general purpose programming language as opposed to a specialized DSL. Additionally, the tool allows some inter-procedural analyses instead of just intra-procedural ones.

1 Introduction

Mozilla[2] is a large project which in the default configuration produces over 2400 object files. While working on a source-to-source transformation tool for the Mozilla project it became clear that an AST alone is not sufficient and one needs to use static analysis in order to check that certain functions can be rewritten correctly. Additionally, when fixing a bug, optimizing code or re-factorin it is useful to find certain patterns in the source code. Some examples are:

- Finding functions that always return the same error code. These can be rewritten to not return an error code.
- Verifying that fields in structs are ordered efficiently.
- Finding code that interacts with garbage collection in an unsafe manner. This is similar to finding incorrect malloc/free uses.
- Building a callgraph.
- Finding dead functions, classes or even modules.
- When fixing a bug it is useful to be able to find all other similar cases in the codebase.
- Proving that certain functions are never called from a particular function. This is useful for security and performance audits.

*The author is employed by the Mozilla Corporation, which supported this work.
Since these are typical problems for any large-scale C++ project, existing tools were evaluated. These tasks require a tool that can do user-defined control flow sensitive inter-procedural checks on C and C++ source code. Additionally the tool needs to be scale to large codebases.

Most static analysis tools do not address C++ due to the complexities of parsing the language. Coverity Prevent[4] and CQUAL[5] are exceptions. However, Coverity is proprietary so it is unclear whether it does control flow analyses. Additionally, it does not support user-defined analyses. CQUAL checks data flow instead of control flow so one can not express queries depending on code flow.

There are two open source tools that fit the user-extensibility criteria and support control flow checking.

1.1 UNO

UNO[1] is a lightweight static analysis tool with a DSL for checking user-defined properties. The C-like imperative DSL allows one to pattern match paths in control flow graph in order to find invalid ones. The DSL is limited to inter-function analysis. Unfortunately, UNO is based on a primitive C parser which does not support all C code present in Mozilla or parse C++. Due to limitation in the DSL, UNO also flattens the AST before pattern matching, losing information in the process.

Technical limitations aside, UNO is easy to learn and use, the DSL has room for future extensibility and serves as a good model for other tools to follow.

1.2 mygcc

Mygcc[3] is a static analysis backend that integrates into GCC and thus would be easy to integrate into most build systems in order to perform analyses. Additional benefit of using Mygcc is that it can parse anything that GCC can compile.

Mygcc is similar to UNO in that it is designed for intra-function analysis. It also does pattern matching over a control flow graph, but unlike UNO uses an even more restrictive declarative DSL. Like UNO, it also does not pattern match on types. It is unclear if it supports C++ or if the DSL could be extended to allow user analyses other than path-based error checking.

2 DeHydra

Due to limitations in existing tools, it was decided to implement DeHydra. It is a tool similar in spirit to UNO except that it supports user-defined analyses written in JavaScript, works with C and C++ and is capable of inter-procedural analyses.

The design of DeHydra is based on the following guidelines:

- Avoid building a direct competitor to Coverity, UNO or other existing tools. Instead DeHydra is to focus on niches not covered by other analysis tools. So no bounds checking, points-to or any other defect-finding analyses are implemented in the C++ or are part of the supplied JavaScript library.

- The tool should be equally good as a bug-finder and a query tool for source code

- Keep the full AST on the C++ side and only pass a simplified version to the JavaScript automaton. This way the CFG is easier to match on, and keeps the scripts from being overly C++ specific. This will allow the tool to be extended to also analyze JavaScript.
• Do not require the JavaScript scripts to implement all possible callbacks. This way the scripts end up shorter and easier to understand.

• Support persistence for incremental analyses.

2.1 JavaScript for Analysis

In order to shorten implementation time and provide more features JavaScript was chosen instead of a specialized DSL. This saved a lot of debugging time and provided DeHydra with a language that has a large developer base. Since JavaScript is a general purpose language, it allows more sophisticated analyses than possible with a limited DSL.

By leveraging the SpiderMonkey[10] JavaScript runtime DeHydra was able to rely on a widely deployed and efficient language runtime instead of introducing and optimizing yet another DSL. JavaScript is sufficiently similar to the C-like DSL used in UNO that it was trivial to implement an UNO compatibility layer written in pure JavaScript such that UNO analyses could be ported with minimal changes.

JavaScript is also appealing in terms of features. It provides objects, closures and higher order functions which allow for concise and powerful pattern matching via functional programming.

Another benefit of using a scripting language is that it enables fast turn-around time. There is no need to recompile analyses for every change, resulting in faster incremental development. JavaScript’s ability to print out any data structure to console is also a great debugging aid. Additionally, JavaScript is familiar to a lot of developers so the learning curve should be gentler than that for other static analysis tools’ DSLs.

2.2 Control Flow Graph

DeHydra contains a control flow graph builder module that translates the Elsa AST into a control flow graph. It supports goto and return statements, and branch and iteration structures.

The graph vertices are basic code blocks connected with transition edges. Edges are labeled with the condition statements that must evaluate to true in order for the edge to be traversed. Figure 6 illustrates how a CFG is represented in DeHydra.

2.3 Control Flow Graph Traversal

To find errors such as the simple mismatched malloc/free check, DeHydra scripts are executed iteratively in depth-first traversal over every path between the entry and the exit blocks in the graph to prove that every malloc is always followed by a free. Scripts are not made aware of conditionals. From the script’s perspective, functions are sequential basic blocks.

This is modeled by having two notions of state. There is global JS state, which corresponds to the global variables and there is a local state. For every basic block entered, local state is passed as the second parameter to the process(vars, state) function and the return value becomes an input state for subsequent basic blocks.

DeHydra attempts to avoid the exponential complexity of traversing every path in the graph. An edge is only visited if the same edge has not been previously visited with an equal input state. This requires scripts to make local state transitions independent of side-effects through the global JavaScript state.
int main(int argc, char **argv) {
char *c;
if(argc)
    c = malloc(1);
if(!argc)
    return;
free(c);
}

Figure 1: Code with an unfeasible path

2.4 Symbolic Evaluator

Following every path in the CFG is computationally expensive, but also causes false positives because not all paths are feasible. Figure 1 shows the source code with an example of this problem. Figure 6 has the corresponding CFG. In the graph, there exists a path from malloc.c:7 to malloc.c:9, but the path will never be taken at runtime.

Initially it was planned to use value numbering to detect “equal” condition statements but that proved unsuitable since one still needs to evaluate expressions to check that their values are identical. Simple abstract interpretation is turned out to be good solution to the problem.

DeHydra infers variable values based on known values and checks guard conditions in order to determine if a conditional expression may evaluate to true. If an expression may evaluate to true, values of variables used in the expression are inferred to be one of ZERO, NONZERO or UNKNOWN.

Paths are deemed to be unfeasible if, in order for a conditional expression on an edge to evaluate to true, a variable along the CFG path is required to be both ZERO and NONZERO.

3 Running DeHydra

3.1 API

JavaScript sees a small and simplified subset of the abstract syntax tree so it will be possible to add another engine in addition to the current C/C++ parser to add support for analyzing other languages such as JavaScript, Java, etc.

When traversing a function's CFG, JavaScript is provided with information on variable name, type and id. Additionally JavaScript is aware of structs, classes, virtual functions, class hierarchy information and bitfields. New types and fields are easy to add and are added as the need arises.

Table 2 lists callbacks that can be defined in user scripts and Table 3 lists the types passed as arguments. Note that both class structure and function bodies are exposed to the scripting API, allowing for conservative global analyses. For a complete list of built in functions, see Table 1.

4 Processing Source Code

DeHydra processes a source code unit at a time. For analyzing Mozilla there is a frontend script that produces a CPP-expanded .a file for every object .o file produced during a Mozilla build. Then DeHydra is applied to every one of those files.
### Table 1: Built-in Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>error(string)</code></td>
<td>Terminates DeHydra with an error message.</td>
</tr>
<tr>
<td><code>print(string)</code></td>
<td>Prints out a message, execution continues.</td>
</tr>
<tr>
<td><code>read_file(string)</code></td>
<td>Returns the contents of a file specified by the string.</td>
</tr>
<tr>
<td><code>write_file(string, string)</code></td>
<td>Writes the second parameter to a file specified by the first.</td>
</tr>
<tr>
<td><code>is_global(id)</code></td>
<td>Returns true when passed an id of a var object corresponding to a C++ global variable.</td>
</tr>
<tr>
<td><code>is_zero(id)</code></td>
<td>Returns true if the variable is definitely zero.</td>
</tr>
<tr>
<td><code>is_nonzero(id)</code></td>
<td>Returns true if the variable is definitely not zero.</td>
</tr>
<tr>
<td><code>graph(string)</code></td>
<td>Writes out the CFG graph in .dot format to a the specified file.</td>
</tr>
<tr>
<td><code>set_block_color(string)</code></td>
<td>Colors the current basic block with the specified color.</td>
</tr>
</tbody>
</table>

### Table 2: JavaScript Callbacks

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>process(vars, state)</code></td>
<td>Called sequentially for every statement using variables in a CFG path.</td>
</tr>
<tr>
<td></td>
<td>All path-related state is passed in via the state variable and out as a return value.</td>
</tr>
<tr>
<td><code>path_end(state)</code></td>
<td>Called once a path has been traversed.</td>
</tr>
<tr>
<td><code>graph_end()</code></td>
<td>Called once the CFG for a function has been traversed.</td>
</tr>
<tr>
<td><code>process_class(class)</code></td>
<td>Called for every struct and class declaration.</td>
</tr>
<tr>
<td><code>input_end()</code></td>
<td>Called once all source files have been processed. Usually used for serialization.</td>
</tr>
</tbody>
</table>

Scripts can save, reload and extend a persistent analysis state, to incrementally analyze a whole program. However, care must be taken when analyzing a large codebase like that of Mozilla to avoid running out of address space on a 32-bit system. DeHydra leaves such space optimizations to the script writer. It is simply a matter of being prudent and only retaining information required for the analysis and possibly spreading it across multiple files. Alternatively, one can switch to a 64-bit system.

## 5 Analyses Implemented

### 5.1 Bitfield Packing Checker

DeHydra can be used to avoid compiler bugs. For example, the struct in Figure 2 fits into 1 word on most compilers, but the Microsoft compiler will make that 2 words if the types of bitfields in the struct are not identical. Changing the `char` to an `int` fixes this problem. Figure 3 shows a DeHydra script that found all the remaining packing problems in Mozilla.
struct {
    char ch:1;
    int i:1;
};

Figure 2: This packs into 2 words in Microsoft C++ compilers.

function process_class(c) {
    var bitf
    for each (var t in c.members) {
        if(!t.bitfieldBits)
            continue
        else if(!bitf)
            bitf = t.type
        else if(bitf != t.type)
            error("Bitfield types in "+c.name+" disagree: "+bitf + " != "+t.type)
    }
}

Figure 3: A DeHydra script to find packing problems like in Figure 2

5.2 Ported malloc/free Checker

DeHydra contains an UNO-compatibility layer implemented in JavaScript. Figure 7 contains the JavaScript version. UNO version can be found in the tarball on the UNO website[1].

6 Persistence and Inter-Procedural Analyses

Without having any user-extensible tools for inter-procedural analysis to draw inspiration from it was hard to come up with a scalable design. Generally, tools have two modes of operation for inter-procedural analysis: inlining and a conservative mode. Inlining builds a CFG by inlining every function called from main(), whereas the conservative approach annotates functions and then uses callgraphs to propagate and verify inter-procedural properties.

Table 3: JavaScript Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vars</td>
<td>An array of var objects.</td>
</tr>
<tr>
<td>state</td>
<td>An object that keeps path-specific state. It starts out as a JavaScript undefined value until it is defined by the script.</td>
</tr>
<tr>
<td>var</td>
<td>An object describing a variable-like expression. See Table 4.</td>
</tr>
<tr>
<td>class</td>
<td>An object describing a class. See Table 5.</td>
</tr>
<tr>
<td>member</td>
<td>An object describing a class member. See Table 6.</td>
</tr>
</tbody>
</table>
const GRAPH_FILE = "/tmp/inheritance.js"

var graph = eval(read_file(GRAPH_FILE)) || new Object()

function input_end() {
    write_file(GRAPH_FILE, uneval(graph))
}

function process_class(c) {
    if(graph[c.name]) return
    graph[c.name] = c.bases.map(function(x) {return graph[x]})
}

Figure 4: A DeHydra script to generate an inheritance graph

Inlining the CFG graphs is not realistic for a project the size of Mozilla, so the conservative approach was implemented. Only write_file and read_file functions had to be written in C++ so scripts could save/restore their state with JavaScript uneval() and eval() when analyzing multiple files. The rest of the conservative analysis can then be done in pure JavaScript. However, at the time of writing there aren’t any completed inter-procedural analyses.

6.1 Completing JavaScript Analyses in Web Applications

JavaScript was chosen for DeHydra for its easy embedding, efficient runtime and functional programming features. However, serialized DeHydra state produced with uneval() can also be loaded by a JavaScript script in a web page. Then the second part of the analysis can then be done in a web browser.

Modern Web browsers such as Mozilla Firefox support interactive vector graphics and can be used to build sophisticated visualization tools. DeHydra can be used as a backend to generate graphs for these web applications to visualize. One example is a class browser[7] implemented from a graph generated by code in Figure 4. Most IDEs struggle with the seemingly simple task of displaying a class hierarchy for a large project and either make the browsing too slow to be usable or run out of memory. On the other hand, a carefully written incremental class browser front-end could be easily implemented by relying on DeHydra as a backend.

Existing text tools that rely on regular expression such as LXR[8] could be complemented by AST-aware graphs produced by DeHydra.

7 Future Work

7.1 Nightly Analyses

When working on a large codebase like Mozilla it is easy to get lost in the source. Questions such as “what types of object hold references to objects of type A and its derived classes?” or “what functions call function B?” can take a long time to obtain an answer to manually. DeHydra can speed that up by generating graphs describing certain aspects of the source code. However, depending on the analysis these graphs can take a few hours to a day to produce for Mozilla.
Thus DeHydra could run a few generic analyses nightly, then the graphs would be made available for downloading.

With prepackaged nightly analyses, the developer would navigate to a page with an interactive JavaScript console, express these questions in JavaScript code and get the answer immediately.

### 7.2 Web Visualization

Displaying textual results of a query can be meaningless if there are thousands of matches returned; in fact it won’t be much different than the result of a refined query with only hundreds of matches returned. An interactive application-specific code browser could be built with web technologies such as SVG and hyperlinking. That could be an efficient and aesthetically pleasing way to query a large codebase.

### 7.3 Dead Code Detection

Over time, a large project like Mozilla accumulates dead code. While dead code elimination is a well-known compiler technique, it is not widespread at code level. There are no open source dead code detection tools for C++. PC Lint[9] is a commercial package that claims to do dead code detection, but it is originally a C tool, so it is unclear how it deals with C++ features such as virtual methods or polymorphic methods. It is unclear if it can detect dead exported functions. Additionally, lint-type tools generally have a high signal to noise ratio, so it may be hard to find dead code warnings.

On the other hand, one can write a specialized script in DeHydra that would resolve all virtual methods, analyze all components in the repository to even be able to suggest removal of exported methods. An alternative analysis could use the class hierarchy and compare it against all invocations of constructors to detect unused classes.

### 7.4 Source-to-Source Transformation Assistance

Mozilla code typically uses out-parameters to return values and return statement to indicate if an error has occurred. Additionally, at the call-site the return value is checked for errors: if an error occurs, the caller typically returns with that error. Thus, there are at least two values “returned” from every getter and there is a large amount of code within Mozilla that just checks for error return values only to propagate them further.

In the long term the solution is to use the return statement to return values and C++ exceptions to propagate errors. However, switching to exceptions is a big task. It requires doing a Mozilla-wide exception safety analysis; converting unsafe code to be exception-safe and may take a long time to write a tool that can do both the analysis and source transformation correctly.

A simpler rewrite task would be to convert the really simple getters to return values on success and NULL on error. Callers would have to be rewritten accordingly, but it would be a much smaller task that switching to exceptions. Figure 5 contains an example getter. DeHydra can detect these, by checking parameter types and checking the CFG to make sure that there are only two possible return values: NS_OK and some error. Then the serialized callgraph described in Section 7.1 would be referenced to provide all of the call sites. The names of the getter and callsites would then be written to a config file to be used as input to the source transformation tool.
nsresult nsBidiPresUtils::GetBidiEngine(nsBidi** aBidiEngine)
{
    nsresult rv = NS_ERROR_FAILURE;
    if (mBidiEngine)
    {
        *aBidiEngine = mBidiEngine;
        rv = NS_OK;
    }
    return rv;
}

Figure 5: An example of typical function that would be rewritten

8 Conclusion

Static analysis is a mature technology for finding defects in software, and can also be used for querying code. This requires the static analysis tools to allow the user to specify queries. However, most tools do not support user-customizable analyses, shy away from C++, or are exclusively focused on defect finding. There are two existing tools that come close to offering code querying abilities. Both Mygcc and UNO allow user-scripting, but apart from other limitations, their scripting languages do not allow for general queries against a code-base.

DeHydra is a new general static analysis tool for C++. While most tools provide " canned" and hard to customize hard-coded analyses, DeHydra only allows analyses in JavaScript—drawing a clear line between parsing code, CFG building, abstract interpretation and source analysis scripts. By reusing the Elsa C++ parser and the SpiderMonkey JavaScript runtime, DeHydra development mostly focused on the static analysis aspect. Most of the development effort went into the control flow graph builder, abstract interpreter and API design.

DeHydra has already found bugs in the source code and shown potential for more complicated future analyses.

List of Figures

1 Code with an unfeasible path ............................................. 84
2 This packs into 2 words in Microsoft C++ compilers ................ 86
3 A DeHydra script to find packing problems like in Figure 2 ........ 86
4 A DeHydra script to generate an inheritance graph .................. 87
5 An example of typical function that would be rewritten ............. 89
6 CFG for code in Table 1 ....................................................... 90
7Malloc checker implemented in JavaScript .................................. 92

List of Tables

1 Built-in Functions .......................................................... 85
2 JavaScript Callbacks ....................................................... 85
3 JavaScript Types ........................................................... 86
4 Fields in the var object ................................................... 91
Figure 6: CFG for code in Table 1
Table 4: Fields in the var object

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Symbol name</td>
</tr>
<tr>
<td>type</td>
<td>C/C++ type.</td>
</tr>
<tr>
<td>id</td>
<td>Unique id (for a single dehydra run) used for tracking variable use throughout the source because names in C++ refer to different things depending on the scope.</td>
</tr>
<tr>
<td>assign</td>
<td>An array of var objects used in assignment: symbol = (exp). Note, in the future assign will point directly at another var object instead of using an intermediate array.</td>
</tr>
<tr>
<td>decl</td>
<td>String describing location of symbol declaration: “example.c23/symbol”</td>
</tr>
<tr>
<td>fieldOf</td>
<td>Used for field access: (expr).symbol. Points to the var object that defines (expr).</td>
</tr>
<tr>
<td>isCallable</td>
<td>Used to call a function: symbol(x).</td>
</tr>
<tr>
<td>isUse</td>
<td>Evaluated to derive value: x = symbol.</td>
</tr>
<tr>
<td>isAlias</td>
<td>Address taken: &amp;symbol.</td>
</tr>
<tr>
<td>isDecl</td>
<td>Symbol appears in a declaration: Foo symbol.</td>
</tr>
<tr>
<td>isDeref</td>
<td>Dereferenced *symbol.</td>
</tr>
<tr>
<td>isParam</td>
<td>parameter: function(Foo symbol).</td>
</tr>
<tr>
<td>isReturn</td>
<td>Used in a return value: return symbol.</td>
</tr>
</tbody>
</table>

Table 5: Fields in the class object

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Class name</td>
</tr>
<tr>
<td>bases</td>
<td>Array of strings of names of base classes.</td>
</tr>
<tr>
<td>members</td>
<td>Array of member objects.</td>
</tr>
</tbody>
</table>

Table 6: Fields in the member object

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Member name</td>
</tr>
<tr>
<td>type</td>
<td>C/C++ type.</td>
</tr>
<tr>
<td>decl</td>
<td>String describing location of member declaration, same format as the decl field in the var object.</td>
</tr>
<tr>
<td>bitfieldBits</td>
<td>Array of member objects.</td>
</tr>
</tbody>
</table>
function uno_check(vars) {
    if (select("malloc", [CALL]))  // unmarked symbols of type function call
    {
        if (select("", [DEF]))) // unmarked symbols DEFined in those stmts
        {
            if (match(1, [DEF])) // are there matching symbols with mark 1?
                error("malloc follows malloc");
            else
                mark(1); // mark 1
        }
    }
    else
        error("result of malloc unused");
}
else if (select("free", [CALL]))
{
    if (select("", [USE]))
    {
        if (match(1)) {
            unmark(); // remove mark
        }
        else
        {
            error("free without malloc");
        }
    }
    else
        error("no argument to free");
}
}

function path_end(state) {
    if (marked(1))
    {
        if (!known_zero())
            error("malloc without free");
    }
}

Figure 7: Malloc checker implemented in JavaScript
References


Experiments in validating formal semantics for C

Sandrine Blazy
ENSHE and INRIA Rocquencourt
Sandrine.Blazy@ensshe.fr

Abstract. This paper reports on the design of adequate on-machine formal semantics for a certified C compiler. This compiler is an optimizing compiler that targets critical embedded software. It is written and formally verified using the Coq proof assistant. The main structure of the compiler is very strongly conditioned by the choice of the languages of the compiler, and also by the kind of semantics of these languages.

1 Introduction

C is still widely used in industry, especially for developing embedded software. The main reason is the control by the C programmer of all the resources that are required for program execution (e.g., memory layout, memory allocation) and that affect performance. C programs can therefore be very efficient, but the price to pay is a programming effort. For instance, using C pointer arithmetic may be required in order to compute the address of a memory cell.

However, a consequence of this freedom given to the C programmer is the presence of run-time errors such as buffer overflows and dangling pointers. Such errors may be difficult to detect in programs, because of C type casts and more generally because the C type system is unsafe.

Many static analysis tools attempt to detect such errors in order to ensure safety properties that may be ensured by more recent languages such as Java. For instance, CCured is a program transformation system that adds memory safety guarantees to C programs by verifying statically that memory errors cannot occur and by inserting run-time checks where static verification is insufficient [1]. Another example is Cyclone, a type-safe dialect of C that makes programs invulnerable to errors such as those detected by CCured [2].

More generally, formal program verification ensures that a program does what is stated in its specification, written as assertions using logical formulas. Assertions express expected properties of the program. Formal program verification generates proof obligations that are usually discharged towards external proof assistants or decision procedures (e.g., [3, 4]).

Recently, separation logic has been defined as an extension of Hoare logic for reasoning about programs that manipulate pointer structures [5]. Separation logic aims at proving fine-grained properties about pointers and memory footprints, such as non-overlapping (i.e., separation) between memory regions. Contrary to Hoare logic, separation logic allows local reasoning on the memory footprint of a statement. Such a local reasoning facilitates program proofs.
Some decision procedures have been defined for separation logic, but they are
dedicated to toy languages (e.g., [6]). Shallow embeddings of separation logic in
theorem provers have also been defined (e.g., [7–9], where [9] handles a large
subset of C).

To sum up, there are several ways to avoid errors in C programs. But, once
a property has been formally verified in a C program, how can we ensure that
it is also verified by the machine code generated by a C compiler? Bugs in the
compiler may invalidate the verification of the source code. It is then necessary
to formally verify a property expressing the equivalence between a source code
and its corresponding machine code. Several kinds of equivalence may be defined;
they may be more or less hard to formally verify. The equivalence may be the
transcription in the target language of the property that has already been for-
mally verified on the source program. For instance, the equivalence can express
memory safety of the target program. Another way to formally verify that two
programs are equivalent consists in verifying that they are computing the same
observable results. A program execution is thus abstracted in the computation
of input-output and final values of the program.

A way to avoid errors in C programs is to formally verify the C compiler itself.
This yields a certified compiler, that is a compiler equipped with a machine-
checked proof that the semantics of the source code is preserved along the
compilation process [10, 11]. The comparison between this approach and other
well-known approaches such as translation validation and proof-carrying code is
detailed in [10].

We are currently developing a realistic, certified compiler called CompCert
that targets critical embedded software [10, 12, 13]. [10] details the back-end of
this compiler, [12] explains its memory model and [13] presents its first front-
end. Since then, the front-end has been rebuilt in order to handle a larger subset
of C and to facilitate the proof of semantic preservation. The formal semantics
have also become more precise. The rest of this paper reports on the design of
formal semantics for the CompCert compiler. It discusses the different kinds of
semantics that have been studied for the languages of the compiler.

2 Formal semantics for certified compilation

This section describes the CompCert compiler, and details the design and the
validation of its formal semantics.

2.1 The CompCert certified compiler

The source language of the CompCert compiler is a large subset of C. Only goto
and long jump statements are not handled by the CompCert compiler. The target
language is the assembly language for the PowerPC architecture. The CompCert
compiler is an optimizing compiler. It accomplishes a series of program trans-
fomation phases. A program transformation is either a translation to a lower
level language or a program optimization. The formal verification of the compiler consists in formally verifying each of its phases. All the formal verifications are developed and machine-checked using the Coq proof assistant. The main optimizations of CompCert are constant propagation, common subexpression elimination and instruction scheduling. Thus, the CompCert compiler generates compact and reasonably efficient target code.

There are 6 intermediate languages in the CompCert compiler. Thus, we have defined a formal semantics for each of the 8 languages of the CompCert compiler. Each formal semantics relies on a memory model that is common to all the languages of the compiler. The CompCert compiler ensures memory safety, mainly concerning reads and writes in memory [12].

The formal verification of the CompCert compiler is the proof of the following semantic preservation theorem (that is written in Coq in Fig. 1): for all source code $S$, if the compiler transforms $S$ into machine code $C$ without reporting error, and if $S$ has well defined semantics, then $C$ has the same semantics as $S$, modulo observational equivalence. Thus, $S$ and $C$ are considered as semantically equivalent if there is no observable difference between their executions. Let us note that the successful compilation of a program does not necessarily imply that this program has well defined semantics. An erroneous program is not conform to some formal semantics or it is not transformed successfully during the compilation process, thus simplifying the definition of both the formal semantics and program transformations.

Generally speaking, during the formal verification of a C compiler, only some events are observed. Usually, these are the final results of the C programs. Other events may also be observed, according to the precision of the semantics for C. For instance, the call graph, or the trace of read and write accesses may also be observed, together with the final results. The trust in the certified compiler increases in proportion to the number and variety of observable events. However, there is a tradeoff between the trust gained by a higher amount of observable events, and the admissible optimizations that the compiler may perform.

2.2 On-machine formal semantics

The kind of formal semantics has a strong impact on the semantic preservation properties that can be verified. There are mainly two kinds of formal semantics that are adapted to formal reasoning on program equivalence. These are operational semantics.

Big-step operational semantics (a.k.a. natural semantics) relate formally a program to its final result, and lend themselves well to the proof of compiler-like program transformations. However, these semantics apply only to terminating programs, and do not allow observing intermediate states during program execution. These two features are distinct disadvantages for the intended application domain of the CompCert compiler: embedded software is typically reactive in nature, meaning that programs do not normally terminate and their interactions with the outside world is what matters, not their final result.
Small-step operational semantics based on (finite or infinite) sequences of elementary reductions of the program source allow precise observations of the program execution and also the observation of non-terminating programs. However, such reduction semantics are difficult to exploit when proving the correctness of compiler transformations such as the generation of a control-flow graph from a structured program.

As big-step semantics are simpler than small-step semantics, they are usually preferred in order to define and reason on languages such as C, with non-local constructs (e.g., return, continue and break statements) mixed with structured programming (e.g., loops). A recurring issue in the formal verification of a C compiler is the development of appropriate operational semantics for the source, intermediate and target languages. Designing adequate on-machine operational semantics for C is not a trivial task. Adequate on-machine semantics are such that their associated induction principles are quite easy to formulate and the corresponding induction steps are provable without too much difficulty.

The validation of the formal semantics is another recurrent issue. The best way to validate the formal semantics for a language such as C is to prove a great deal of properties about the semantics. The formal verification of a C compiler provides an indirect but original way to validate the semantics of the C language. It is relatively straightforward to formalize operational semantics, but much harder to make sure that these semantics are correct and capture the intended meaning of programs. In our experience, proving the correctness of a translation to a lower-level language has detected many small errors in the semantics of the source and target languages, and therefore has generated additional confidence in both.

An interesting result is that the main structure of the CompCert compiler is not conditioned by the program transformations, but very strongly by the choice of the languages of the compiler, and also by the kind of semantics of these languages. Thus, the intermediate languages of the CompCert compiler have been designed in order to facilitate the proofs of translation between languages. When a proof of translation from a language $L_1$ to a simpler, lower-level language $L_2$ required to specify different concepts (e.g., correspondences between memory states) that made the reasoning more complex, an intermediate language $L_i$ between $L_1$ and $L_2$ has been defined. The formal verification of both translations from and to $L_i$ happened to be much easier to achieve than the formal verification of the translation from $L_1$ to $L_2$. This is why there are 6 intermediate languages in the CompCert compiler.

Having so many intermediate languages is not common for a compiler. From the programming point of view, it seems to be easier to define as few intermediate languages as possible and to write as few program translations as possible. From the proof point of view, it is easier to define several intermediate languages and elementary translations between slightly different languages.

Turning to the kind of formal semantics we adopted in CompCert, most of the semantics were initially of the big-step operational kind. These big-step semantics capture the final result of program execution, as well as traces of calls
to input and output functions. Thus, the formal verification of the CompCert compiler proves that a target program computes the same result as its corresponding source program, and also that the trace of all input-output activities of the program is preserved by all the phases of the compiler. This addition of traces leads to a significantly stronger observational equivalence between source and machine code.

Fig. 1 shows this semantics preservation theorem written in Coq. As explained previously in Section 2.1, if a program called \texttt{prog} is compiled into a PowerPC program called \texttt{tprog} without reporting error (first hypothesis of the theorem), and if the execution (in C) of \texttt{prog} terminates and yields a trace and a final integer value (i.e. the return value of the \texttt{main} function of \texttt{prog}), then the execution of \texttt{tprog} (in PowerPC assembly) yields the same results (i.e. the same trace and integer value).

\textbf{Theorem transf\_c\_program\_correct:}

\begin{verbatim}
forall prog tprog trace n,
  transf\_c\_program prog = Some tprog =>
  Csem\_exec\_program progr trace (Vint n) =>
  PP\_Csem\_exec\_program tprog trace (Vint n).
\end{verbatim}

\textbf{Fig. 1. Main theorem: semantic preservation of the CompCert compiler}

We also investigated other forms of on-machine semantics in order to observe non-terminating programs. We have defined several kinds of small-step semantics for some intermediate languages, and proved the equivalence between small-step and big-step semantics for terminating programs, thus giving the opportunity to validate differently the semantics. Small-step semantics have been defined for the languages of the compiler back-end and the semantic preservation proofs have been adapted rather easily.

Another direction that we are currently investigating is coinductive big-step semantics, where evaluation rules still relate programs and program fragments to their final outcome, but some of the rules are interpreted coinductively (infinite derivation trees) instead of inductively (finite derivation trees) as usual. The coinductive interpretation enables these semantics to describe the evaluation of non-terminating programs. In coinductive semantics, inference rules are similar to those of natural semantics, provided that these rules are interpreted coinductively instead of the usual inductive interpretation, or in other terms provided that infinite evaluation derivations are considered in addition to the usual finite evaluation derivations. Such coinductive big-step semantics have been defined for smaller languages than those of the CompCert compiler [14]. These results are encouraging and in the near future, we intend to define coinductive semantics for the languages of the compiler front-end.

Lastly, we also defined a deep embedding of a separation logic for an intermediate language of the CompCert that is close to C [15]. The separation logic
consists of an assertion language (with operators that are specific to separation logic) and an axiomatic semantics. We have proved the soundness of this axiomatic semantics with respect to the natural semantics used in the verification of the CompCert compiler, thus giving another opportunity to validate differently the semantics. This experiment is a first bridge between, on the one hand, program proof in the style of Hoare, and on the other hand the CompCert compiler verification effort.

2.3 Quantitative results

Most of the CompCert compiler is written directly in the Coq specification language, as pure functions. Using the extraction facility of Coq (which translates Coq specifications to Caml code), as well as the CIL library that provides an industrial-strength parser and type-checker for the C language [16], and adding a PowerPC printer written directly in Caml, we obtain Caml source code for the whole compiler, making it directly executable. To our knowledge, this is the biggest Caml code that has been automatically extracted from a Coq development. Figure 2 details the architecture of the CompCert compiler. The box in the middle of the figure represents the Coq development and shows the different translations between the languages of the compiler.

![Fig. 2. Architecture of the CompCert certified compiler](image)

Benchmarking on a set of realistic examples of C programs of a few thousand lines shows that the performance of the generated PowerPC code is entirely acceptable: performance is much better than that of the gcc compiler at optimization level 0, and only slightly inferior to that of gcc at optimization level 1.

The certified compiler represents about 45 000 lines of Coq. The formal semantics represent 8% of this code. The transformations performed by the compiler represent 13% of the code. The rest of the code correspond to the correctness proofs (22% for the lemmas and 50% for the proofs in Coq of these lemmas) and libraries (7% of the code).

3 Conclusion

In conclusion, several solutions exist for producing trusted C software. Formal program verification ensures properties written as assertions in programs. Formal
compiler verification (i.e., certified compilation) ensures that target code behaves as prescribed by the semantics of the corresponding source code. The design of adequate on-machine semantics is crucial for program verification and certified compilation.

This paper has reported on the design of formal semantics for a certified compiler, the CompCert compiler. These formal semantics have been validated by two kinds of machine-checked proofs: correctness of translations, and semantic equivalence between different kinds of semantics. Designing a certified C optimizing compiler gives a good opportunity to validate its formal semantics for C. The future of C program verification is to connect machine-verified source programs to machine-verified compilers, and run the object code on machine-verified hardware.

References


Olmar: manipulating C and C++ abstract syntax trees in Ocaml

Hendrik Tews*
Radboud Universiteit Nijmegen, The Netherlands
http://www.cs.ru.nl/~tews

June 5, 2007

Olmar is a branch of the Elsa C++ parser that adds the ability to marshal the abstract syntax tree of any C or C++ input as Ocaml variant type to the disk. The marshaled abstract syntax tree can later be processed with a pure Ocaml program. Alternatively additional Ocaml code could also be linked directly into the else parser. This paper describes Olmar, the technology used inside it and the first Olmar application: ast_graph for visualizing C and C++ abstract syntax trees.

1 Introduction

For applying formal methods to C++ programs one needs sooner or later access to the abstract syntax tree (Ast) of C++ programs. For applications in formal methods one mainly needs to inspect the Ast and walk over it. It is my strong believe that this sort of programming is best done with pattern matching over an ML-like variant type. However, freely available C++ parsers are written in C or C++ and there is no pattern matching facility for the internally used data structures.

C++ is terribly difficult to parse. The grammar is not LALR(1), but even with general LR-parsing techniques there remain ambiguities in the grammar that can only be resolved with some semantic analysis. Developing a new C++ parser in a functional programming language is therefore no practical option.

In order to solve the need for a pattern-matchable Ast of C++ programs I decided to mount an Ocaml back-end to an existing, free C++ parser. The Ocaml back-end would rebuild the internal data structures of the C++ parser as an Ocaml variant type. I decided to use Ocaml as pattern-matching language because I am very familiar with Ocaml and its foreign language interface.

*The author has been supported by the European Union through PASR grant 104600.
In the search for a free C++ parser I came across Elsa [McP], which is now part of the Oink static analysis tools [WCM]. Elsa is a C++ parser that includes type-checking, semantic disambiguation and even template instantiation. Scott McPeak, the author of Elsa, already noticed that ML-style variant types are the right data structures for abstract syntax trees. Unfortunately this knowledge did not let him abandon his most beloved C++. Instead he wrote a preprocessor, called Astgen, that translates ML-style variant type descriptions into a C++ class hierarchy. Elsa's internal abstract syntax tree is to a large extend described in the Astgen input language. One can therefore easily add functionality in the Elsa Ast by doing some Astgen meta-programming.

The rich functionality and the possibility of saving work with Astgen meta-programming led to the decision to reuse Elsa as C++ parser. The Elsa and Oink maintainers are very suspicious of Ocaml. Additionally, they were very ineffectively organised before the resignation of Scott McPeak in March 2007. I was therefore unable to contribute any of the new functionality back to the Elsa distribution. The new Ocaml back-end of Elsa is therefore available as a branch of the Elsa development under the name Olmar at http://www.cs.runl/~tews/olmar.

The functionality of Olmar is mostly complete. All Ast data that I am aware of is translated into Ocaml. There exist one real Olmar application: ast_graph for visualising Asts, see Figure 1 for an example. Additionally, there is checkast for consistency checking of the generated Ocaml data and countast, a simple demo application. In the Robin project [Tew07] we plan to use Olmar to develop a semantics compiler, that translates C++ into its semantics in Pvs.

In the following I talk a lot about abstract syntax trees of C++ programs. They will appear as C++ data structures inside the Elsa parser or as Ocaml data structures. I will refer to these different incarnations as the Elsa Ast and the Ocaml Ast, respectively. I will refer to the process of translating an Elsa Ast into an Ocaml Ast as Ast reflection into Ocaml.

Section 2 describes the internals of Olmar, Section 3 tells how to use Olmar, Section 4 is about ast_graph, the visualisation program and Section 5 describes the Olmar API.

2 Architecture and Technology

The original Elsa distribution is split into 4 subdirectories.

smbase Library with basic data and container types. Although Elsa is of course written in C++, it does not use the standard template library. It only relies on the stuff provided from smbase.

ast The Astgen tool, which generates a C++ class hierarchy from an ML-style variant type description. The code generated by Astgen includes many utility functions and various visitors (if requested in the Astgen source). The data type for the abstract syntax tree of Astgen input files is also generated by Astgen.

elkhound A parser generator for generalised LR parsing. The Elsa parser is generated by Elkhound. Elkhound can generate parsers in C++ or Ocaml.
Figure 1: Part of the abstract syntax tree of the hello world program from page 111.
elsa The Elsa parser and type checker. Here are the Astgen sources for the Elsa syntax tree and the C++ grammar for Elkhound. Further there are more than 1000 test cases in the subdirectory in.

Olmar contains a fifth subdirectory.

asttools Ocaml sources for ast_graph, check_cast and the countAst example. There are some general purpose Olmar modules that probably every Olmar application will reuse. The Ocaml code for the Ast reflection into Ocaml mostly resides in the els and in the ast subdirectories.

I considered three approaches for obtaining abstract syntax trees of C++ programs in Ocaml. The first two have important deficiencies. Olmar uses the third approach.

Ocaml back-end of Elkhound With the Ocaml back-end of Elkhound one could use the Elsa C++ grammar in order to let Elkhound generate a C++ parser in Ocaml. One would however loose Elsas disambiguation, the type checking and the template instantiation.

XML Elsa can dump the internal Ast in XML format. A promising idea was to let Ocaml read such XML files. However, PXP, the Ocaml XML library from Gerd Stolpmann [Sto], generates an Ocaml object hierarchy from the XML input. Transforming such an object hierarchy into a variant type requires, precisely like an event based PXP pull parser, some kind of high-level reparsing of the Ast, something that I wanted to avoid by using an XML library. What is missing is a tool that reads an XML document type definition and generates from it both a suitable variant type and a XML parser that reads conforming XML files into the variant type.

Ocaml reflection capabilities for Ast nodes The Elsa Ast class structure is equipped with an additional Ast traversal function that translates every Elsa Ast object into a corresponding Ocaml value. The construction of Ocaml values works bottom-up. Astgen generates various visitors, but the visitor methods have always return type void. Therefore these visitors cannot be used because the Ocaml values of the child nodes must be returned to the parent.

The data in the Elsa Ast can be divided into three parts:

Syntax nodes The syntax nodes directly represent the input file. The corresponding C++ classes are almost completely generated by Astgen. There are altogether 32 different types of syntax nodes.

Type nodes Type nodes are built by the type checker. They do not represent concrete syntax. The class hierarchy for type nodes is defined manually (there exist plans to refactor the the data of the type nodes into Astgen controlled classes). There are 10 different type nodes.

1The exception is $\texttt{attribute}$ that is manually defined as a special case of $\texttt{grouping}$ for the $\texttt{Declarator}$ nodes.
Plain data The remaining primitive data in the Ast, mostly enumerations, but also strings and integers. There are 16 enumeration types with together 219 enum constants.

The types of the syntax and type nodes can again be distinguished into

Structured node types where the node type is split into several subtypes. For instance the node type of statements has 22 subclasses, for every statement one. The 20 structured node types split into 142 variants.

The structure of the syntax nodes has always height two, that is, there is one general node type, such as statement, that is split into several node subtypes, such as S_if, S_switch and so on. However, these node subtypes are not again split into different variants. These types are best mapped to an Ocaml variant type, where every node subtype gives rise to one constructor.

The type nodes are handwritten, not bound to Astgen limitations, and have sometimes a more complicated structure. I nevertheless map every node super type to an Ocaml variant. I thereby loose some structure. For instance AtomicType has a NamedAtomicType variant, which is split again into five variants. In the Ocaml Ast the type NamedAtomicType does not exist, it corresponds to five of the six constructors of AtomicType.

Flat node types such as the node type for whole translation units (consisting of just a list of TopForm’s and optionally a scope) that are not split into different cases. There are altogether 22 flat node types. In the Ocaml Ast, flat node types are represented as tuples or records.

As said before, reflection into Ocaml works in principle bottom-up, combining the Ocaml values of the child nodes into a new Ocaml value for the current node. However, there are the following complications.

Null pointers In C++ a pointer to a child node can of course be NULL. In comments in the source code the child links of the Elsa Ast are relatively well classified into nullable and non-nullable ones. In the Ocaml Ast nullable child node links are wrapped in an option type. To minimise the number of options I started with all child nodes non-nullable and added option wrappers for all the segmentation faults that the Elsa regression tests produced. Currently there are 118 option wrappers among the 320 links to child nodes in the Ocaml Ast type.

Sharing To save memory and for resolving cycles (see next point) Ocaml reflection must preserve sharing of nodes in the Elsa Ast. Therefore every Ast node holds an Ocaml value that is initialised with the corresponding Ocaml Ast node when it has been computed. This value is returned when the node is visited a second time.

These pointers from Elsa to Ocaml Ast nodes must be registered with the Ocaml garbage collector as global roots. For big Asts they are the source of a serious performance problem. On every minor collection all global roots are scanned by
the garbage collector, even if the pointed value is already in the old generation. To improve performance Olmar changes the minor heap size from 32KB to 8MB.

Cycles The syntax nodes alone form a real tree, that is, there are no cycles in the Elsa Ast that consist of syntax nodes only. The type nodes, however, add lots of cycles. Strictly speaking the Elsa Ast is only a connected directed graph. There are simple cycles between the type nodes themselves, for instance every declared entity contains a pointer back to its scope and the scope, of course contains a pointer to all its declarations. There are also cycles between type and syntax nodes, because certain syntax nodes have a link to their type and certain type nodes have a link to the syntax from which they stem.

One could have avoided the problem with cycles by only reflecting the syntax nodes to Ocaml. Then, at least for some applications, one would have to derive type information a second time on the Ocaml side. Alternatively one could try to reflect the type nodes to Ocaml but somehow without the cycles. However, the fields in the Elsa Ast cannot be clearly divided into cyclic and non-cyclic ones. A field that closes a cycle in one situation does not participate in any cycle in different situations. To avoid cycles it is necessary to at least partly revise the type nodes. This is far beyond my current level of understanding of the Elsa Ast.

When constructing the Ocaml Ast cycles are resolved as follows: From every possible cycle I select one edge as the cycle closing edge. Those cycle closing edges are wrapped in a option ref type in the Ocaml Ast. The tree traversal that constructs the Ocaml Ast never follows a cycle closing edge. Instead, the corresponding Ocaml field is initialised with ref None. Additionally this reference cell and the target node of the cycle closing edge are registered in the list of broken cycles. When the tree traversal finishes I pick a reference cell and an Elsa Ast node from the list of broken cycles. I restart the traversal on the node (which might add more to the list of broken cycles) and update the reference cell after it has been finished. The Ocaml Ast generation is complete when the list of broken cycles becomes empty.

To determine the cycle closing edges I also used the Elsa regression tests. I added code for cycle detection and started with no cycle closing edges. Every time a cycle was detected by the regression tests I investigated the situation and determined a new cycle closing edge.

Correctness and type safety of the Ocaml Ast Although an Ocaml call-back is used for value construction there are plenty possibilities to construct an incorrect Ocaml value. The Ocaml marshalling functions that are used to read and write the Ast to disk are not safe. An incorrect Ocaml Ast value could therefore lead to crashes of pure Ocaml programs.

I first used a branch of the Ocaml system for type safe unmarshalling [HMC] for testing. However, this code suffers from serious performance problems. It takes more than 4 hours to check 7MB of Ocaml Ast data. I therefore eventually
wrote my own Memcheck module [Tew06], which only needs 50 seconds to check the 7MB. The check-cast utility distributed with Olmar relies on Memcheck to type check Ocaml Asts. The functionality can easily be integrated in any Olmar application.

The tree traversal for the reflection into Ocaml performs the following operations:

1. Register local variables with the Ocaml garbage collector.

2. Return immediately if the Ocaml Ast value has been constructed before.

3. If not done yet, obtain a handle to the Ocaml value construction function for the type or constructor corresponding to this node.

4. Cycle check: test that the current node is not on the stack of currently visited nodes and add it to the stack.

5. Compute the Ocaml Ast nodes for all children. Children that are on a cycle closing edge are treated as described above.

6. Construct the Ocaml value for this node with the function from point 3.

7. Remove this node from the stack of currently visited nodes and return.

Using an Ocaml call-back to allocate and fill the value in Point 6 has the advantage that the Ocaml reflection code does not need to worry about the encoding of variant type constructors in Ocaml (which depends on the order in the Ocaml type declaration). The treatment of cycles could be simplified a lot if the Ocaml value would be allocated before visiting the children and filled afterwards without an Ocaml call-back. A future version of Olmar might adopt this approach. However, initially I was too much afraid of incorrect Ocaml Asts.

3 Using Olmar

Olmar is available with a BSD license. For download and installation see the Olmar website http://www.cs.rutgers.edu/tews/olmar/.

To use Olmar there are two options:

1. Let the Elsa parser ccparse save the Ocaml Ast into a file and have a stand-alone application that later reads it. The latter application can be a pure Ocaml program, enjoying all the nice properties of statically checked typosafe programs.

2. Link the application into the Elsa parser ccparse and call it from ccparse’s main function.

I prefer Option 1 because of the easier development, the type-safety bonus and because the additional overhead is negligible. Therefore I will only provide some hints for Option 2 at the end of this section.

The following steps are necessary to use the abstract syntax tree of a file input.cc in a stand-alone Ocaml application:
1. The Elsa parser does not contain a pre-processor. Therefore one needs to run the preprocessor from a standard C++ compiler, for instance for GCC:

\[ \texttt{g++ -E -o input.ii input.cc} \]

This saves the pre-processed code in file \texttt{input.ii}.

2. Run the Elsa parser with Ocaml Ast generation enabled. Ocaml Ast generation can be enabled either via the option \texttt{-oc file}, which expects a file name for saving the Ocaml Ast. Alternatively one can supply the tracing Option \texttt{-tr marshalToOcaml}. Then the Ocaml Ast filename is derived from the input by appending "\texttt{oast}". For instance

\[ \texttt{ccp parse -oc input.oast input.ii} \]

will save the Ocaml Ast in \texttt{input.oast}.

3. Optionally type-check the generated Ocaml Ast with \texttt{check.oast}. This will check if the Ocaml Ast is a valid image of the expected type of Ocaml Asts.

\[ \texttt{check.oast input.oast} \]

4. Load the Ocaml Ast into the target application. This can be done with the function \texttt{OastHeader.unmarshalOast}. Ocaml Ast files contain a string with version information on the first line. It follows data written by the marshalling function from the Ocaml library. First the number of syntax tree nodes as an Ocaml integer and then the real Ast as marshaled value of type \texttt{annotated translation Unit_Type}.

**Stand-alone applications with integrated Elsa parser** Technically the \texttt{ccp parse} from Ocaml is a combined C++/Ocaml application with C++ main function, where Ocaml code is only called as call back from C++. The easiest way to add more Ocaml code is to add the relevant source files to the \texttt{CCPARSE.ML} makefile variable in the \texttt{elsa} subdirectory. One only has to register the new entry points with Ocaml's call back facility during module initialisation (or put a hook into the \texttt{camLcallbacks} module) and add code to \texttt{elsa/main.cc} to call the new facilities. The Ocaml Ast is build in function \texttt{marshalToOcaml} in \texttt{main.cc}.

**4 ast_graph: visualising C++ syntax trees**

\texttt{ast_graph} is currently the only real Ocaml application. To use it one follows steps 1 and 2 of the preceding section to generate the Ocaml Ast file and applies \texttt{ast_graph} to the Ocaml Ast file. \texttt{ast_graph} will pretty print the syntax tree in the dot file format, which can then be further processed with the graphviz utilities [Gra].

Visualisations of complete Asts are unwieldy huge for the following reasons:
• For typical C++ files only a fraction of the code belongs to the file itself. The overwhelming part comes from included files. For instance with gcc 4.1.2 including `iostream` adds about 30,000 lines of which about one half is real code containing more than just white space or a single brace.

• There are about 400 built-in functions. Therefore even the empty input file contains more than 1000 nodes for built-in functions.

I have not yet found suitable software to display graphs in the order of 100,000 nodes. All of zgrviewer, ghostview, xfig fail for huge graphs or do not provide efficient functionality to scroll and zoom in such large displays. `ast_graph` provides therefore options to select the nodes that are pretty printed.

As example consider the following simple hello world program.

```c
#include <iostream>
using namespace std;

int main() {
    cout << "Hello World!\n";
    return 0;
}
```

It abstract syntax tree contains 306,768 nodes. Figure 1 was generated with the following commands.

```
g++ -E -o hello-world.ii hello-world.cc
ccparse -oc hello.oast hello-world.ii
ast_graph -loc hello-world.cc -dia 2 hello.oast
dot -Tps nodes.dot -o hello.eps
```

The two options for `ast_graph` have the following effect: First all nodes with location information from the file “hello-world.cc” are selected. Then all nodes reachable with two steps (either both forward or both backward) are added. Using `-dia 3` gives already a huge graph with about 700 nodes, because it adds the successor nodes of the node “Scope 2650”. There is one such successor node for every element in the name space `std`.

## 5 Olmar programming interface

The Ocaml Ast has type `annotated translationUnit.type`. The Ocaml type definition is distributed over the files `elsa/ast_annotation.ml` (type `annotated`), `elsa/cc.mLtypes.ml` (enumerations for syntax nodes), `mLtypes.ml` (enumerations for type nodes) and `cast_gen_type.ml` (syntax and type nodes). All Ocaml node types are polymorphic in order to simplify extensions of the Ast; One simply has to replace the type `annotated` with something more complex. The type `annotated` currently contains an integer that uniquely identifies the Ast node and (the pretty much useless) memory address of the corresponding Elsa Ast node.
The unique node integers are guaranteed to be in the interval \([1 \ldots \text{node\_count}]\), which implies further, that all integers in this interval are actually identifying some node. The dense numbering of Ast nodes can be used to store the Ast nodes in an array. The module \texttt{asttools/superast.mli} provides an Ast node super-type and functions to store a syntax tree into an array.

Olmars applications can either be written as recursive function over the syntax tree or as iteration over the Ast array. The \texttt{count-ast} demo application is available in both version: as recursive function (\texttt{asttools/count-ast.mli}) and as iteration (\texttt{asttools/count-ast-new.mli}).

The recursive function approach is complicated because of the cycles in the Ast. All children links of an \texttt{option ref} type might lead into a cycle. However, some parts of the Ast might only be accessible via such an \texttt{option ref} link. The module \texttt{asttools/dense-set.mli} provides the type of sets of positive integers, based on bitmap implementation. It is used in \texttt{count-ast} and \texttt{ast\_graph} to avoid cycles.

Last but not least, the module \texttt{asttools/ast\_util} provides utility functions to extract the node annotation and location information (when present).

6 Conclusion

This paper presents the internal and the programming interface of Olmar, a branch of the Elsa C++ parser that adds functionality to export a C++ abstract syntax tree into Ocaml as a variant type.

An interesting idea to enhance Olmar is to apply the here described Ocaml reflection to the abstract syntax tree of Astgen files and to use it to export the abstract syntax tree of the Elsa Astgen sources to Ocaml. This would allow to write Ocaml programs that generate most of the tedious Ast traversal code inside \texttt{ast\_graph} and \texttt{count-ast}.

Regrettably, a merge of Olmar back into the official Elsa distribution seems not very likely at the moment because of technical but also social issues. See the \texttt{ocaml-devel} mailing list for lengthy discussions.

References


Author Index

Bagnara, Roberto, 75
Blazy, Sandrine, 95
Daum, Matthias, 17
Fehnker, Ansgar, 69
Gladisch, Christoph D., 29
Gleki, Taras, 81
Hähnle, Reiner, 43
Hill, Patricia M., 75
Huttick, Ralf, 69

Larsson, Daniel, 43
Mähr, Oleg, 43
Moy, Yannick, 1
Pescetti, Andrea, 75
Rauch, Felix, 69
Sceifried, Sean, 69
Tews, Hendrik, 59, 103
Zaffanella, Enna, 75