Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science 288 (2012) 37–47

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

# Inferring Effective Types for Static Analysis of C Programs[1]

Bertrand Jeannet[2] Pascal Sotin[2]

*INRIA*

**Abstract**

The C language does not have a specific Boolean type: Boolean values are encoded with integers. This is also true for enumerated types, that may be freely and silently cast to and from integers. On the other hand, verification tools aiming at inferring the possible values of variables at each program point may benefit from the information that some (integer) variables are used solely as Boolean or enumerated type variables, or more generally as finite type variables with a small domain. Indeed, specialized and efficient symbolic representations such as BDDs may be used for representing properties on such variables, whereas approximated representations like intervals and octagons are better suited to larger domain integers and floating-points variables.
Driven by this motivation, this paper proposes a static analysis for inferring more precise types for the variables of a C program, corresponding to their effective use. The analysis addresses a subset of the C99 language, including pointers, structures and dynamic allocation.

*Keywords:* Static Analysis, Type Inference, C Programming Language, Boolean, Finite Types.

## 1 Introduction

### Verification of C programs.

The initial motivation for this work was to infer invariants on C programs with the tool ConcurInterproc [6]. There are two main issues when one wants to connect an academic analyser to the C language:

(i) The analyser might encounter features of the C language it was not designed to deal with. This leads in the best case to the use of imprecise fall-back treatments and in the worst case to a silently unsound analysis.

(ii) The analyser may not recognize in the C presentation features for which it was designed. This leads to a less precise treatment of the program.

This article address a problem belonging to Point (ii).

[1] This work was supported by the french ANR project ASOPT.

[2] Emails: `bertrand.jeannet@inria.fr` `pascal.sotin@inria.fr`

doi:10.1016/j.entcs.2012.10.006

```
int b,x;           ⇒   bool b; int x;
if (b) x++;            if (b) x++;
```

Fig. 1. Boolean typed `int`

**Boolean values encoded with integers variables.**

The verification tool ConcurInterproc distinguishes numerical, Boolean and finitely enumerated variables. We want to cast C programs as input of this analyser and to exploit its type system. Unfortunately, the C type system is too weak. For example, in Fig. 1, both `b` and `x` are declared as `int` but the analyser would gain precision by considering `b` as a boolean and `x` as a number (a disjunctive analysis, depending on the truth value of `b` would then be performed). Moreover, even if `b` was declared as a Boolean enumerated type {`false=0, true=1`}, this does not imply that it is not assigned somewhere else the value `2`.

**Contribution.**

We propose a static analysis for C programs which specializes in a sound way the generic integer type of some variables and structure fields into Booleans or inferred enumerated types. This analysis takes into account aliasing properties raised by procedure calls and pointers. This static analysis allows the initial weakly-typed C program to be transformed into a semantically equivalent, strongly typed program, which can be more efficiently analyzed by verification tools such as ConcurInterproc [6]. After a short presentation of the context and related work (Section 2), we first describe our analysis in a simple context involving only procedures and integer variables (Section 3), before extending it to pointers, structures and dynamic allocation (Section 4), and discussing remaining issues in the conclusion.

## 2 General Context and Related work

As already mentioned, our motivation is to connect the ConcurInterproc verification tool [6], and its extension to pointers PInterproc [10]. These tools can treat the integer variables of C programs as numerical variables, by representing their possible values using for instance octagons [8], but they can handle more precisely (*ie.,* in a disjunctive way) those integer variables that are actually manipulated as Boolean or enumerated variables, using Bdds.

A simple solution to avoid the confusion between Boolean and numerical variables is to use a strongly-typed form of C (eg. Cyclone [7]) offering types like `bool` and ensuring that the program respects the declared types, but then this does not address ordinary C programs.

The question of strengthening the typing of a program for analysis purpose has been tackled by [3] in the context of interpreted languages, like Javascript, with both weak and dynamic typing. The authors perform a flow-sensitive static analysis which collects the possible types of a variable at a given point. Similarly, for compilation purpose, many techniques have been proposed to infer the possible classes of objects at invocation sites in order to optimize dynamic call resolution

|  |  |  |  |
|---|---|---|---|
| $\langle\text{prog}\rangle$ | ::= | $\langle\text{decl}\rangle\langle\text{proc}\rangle^+$ | list of variable and procedure declarations |
| $\langle\text{proc}\rangle$ | ::= | $\langle\text{typ}\rangle\ f(\langle\text{decl}\rangle)\text{“\{”}\langle\text{decl}\rangle\ \langle\text{stm}\rangle^\star\text{“\}”}$ | contains declarations and statements |
| $\langle\text{decl}\rangle$ | ::= | $(\langle\text{typ}\rangle\ x)^\star$ | declaration of typed variables |
| $\langle\text{typ}\rangle$ | ::= | **int** |  |
| $\langle\text{stm}\rangle$ | ::= | $\langle\text{lv}\rangle = \langle\text{expr}\rangle$ | assignment |
|  | \| | $\langle\text{lv}\rangle = p(\langle\text{expr}\rangle, \ldots, \langle\text{expr}\rangle)$ | procedure call |
|  | \| | **return** $x$ | returnig the value of a variable |
| $\langle\text{lv}\rangle$ | ::= | $x$ |  |
| $\langle\text{expr}\rangle$ | ::= | $\langle\text{cst}\rangle \mid \langle\text{lv}\rangle$ | constant or left-value |
|  | \| | $\langle\text{boolexpr}\rangle \mid \langle\text{intexpr}\rangle$ |  |
|  | \| | $\langle\text{expr}\rangle\text{“?”}\langle\text{expr}\rangle\text{“:”}\langle\text{expr}\rangle$ | conditional expression |
| $\langle\text{boolexpr}\rangle$ | ::= | $\text{“!”}\langle\text{expr}\rangle \mid \langle\text{expr}\rangle\langle\text{bool\_binop}\rangle\langle\text{expr}\rangle$ |  |
|  |  |  | *Boolean* expressions evaluating to 0 or 1 according to C99 stdandard |
| $\langle\text{intexpr}\rangle$ | ::= | $\text{“}-\text{”}\langle\text{expr}\rangle \mid \langle\text{expr}\rangle\langle\text{int\_binop}\rangle\langle\text{expr}\rangle$ |  |
|  |  |  | “*Integer*” expressions potentially evaluating to any value |
| $\langle\text{cst}\rangle$ | ::= | $0, 1, 2, \ldots$ |  |
| $\langle\text{bool\_binop}\rangle$ | ::= | “&&” \| “\|\|” \| “==” \| “! =” \| “<” \| … |  |
| $\langle\text{int\_binop}\rangle$ | ::= | “+” \| “−” \| “*” \| “/” \| “&” \| “\|” \| … |  |

Fig. 2. General Syntax

into static calls. Compared to our analysis, these analyses infers sets of types in a flow-sensitive way while we are looking for a unique flow-insensitive type for each of our variables.

## 3 Programs with procedure calls and scalar variables

We first present our static analysis in the simple context of programs built from a number of procedures manipulating only scalar variables (we exclude pointers from the scalars). This allows to discuss our approach in a simple setting, before investigating the additional issues raised by pointers, casts and dynamic allocation.

### *3.1 The considered input language*

We consider a simple subset of C, the grammar [3] of which is depicted on Fig. 2. $f, g$ denote procedure names, $x, y$ variable names. As our analysis is flow-insensitive, we do not detail the statements related to control. In short, in this subset all variables are declared as integers, there are no pointers, no structured types, no dynamically allocated data. We assume that all procedures return a value, and that variables are uniquely identified by their name.

We do not consider explicit enumerated type declarations, unlike a tool like SPLint [1], which complains about casts from one enumerated type to another one. This is because our analysis is not intended as an help for programmers to discover potential problems due to weak typing.

[3] We ignore details about separators, etc.

$$\frac{x = \mathit{expr} \in \langle\text{stm}\rangle}{D(x) \supseteq D(\mathit{expr})} \quad \frac{\textbf{return}\ \mathit{expr} \in \langle\text{stm}\rangle(f)}{D(f) \supseteq D(\mathit{expr})} \quad \frac{\begin{array}{c} x = f(\mathit{expr}_1, \dots, \mathit{expr}_n) \in \langle\text{stm}\rangle \\ \mathit{typ}\ f(\mathit{typ}_1\ x_1, \dots, \mathit{typ}_n\ x_n) \in \langle\text{proc}\rangle \end{array}}{\begin{array}{c} D(x) \supseteq D(f) \\ \forall i : D(x_i) \supseteq D(\mathit{expr}_i) \end{array}} \tag{2}$$

$$\begin{aligned} D(\mathit{cst}) &= \{\mathit{cst}\} \\ D(\mathit{boolexpr}) &= \{0,1\} \\ D(\mathit{intexpr}) &= \mathbb{Z} \\ D(\mathit{expr}\,\text{“?”}\,\mathit{expr}_1\,\text{“:”}\,\mathit{expr}_2) &= D(\mathit{expr}_1) \cup D(\mathit{expr}_2) \end{aligned} \tag{3}$$

Fig. 3. Inferring possible values for variables in scalar programs

### *3.2 Inferring the possible values of variables*

In this simple setting, the philosophy of our analysis is not really to infer types, but just to discover the set of possible values for any variable in a given procedure. This means that we focus on an *attribute-independent*, *flow-* and *context-insensitive* static analysis, which computes a function

$$D : \mathit{Proc} \uplus \mathit{Var} \to \mathcal{P}(\mathbb{Z}) \tag{1}$$

where

- $\mathcal{P}(\mathbb{Z})$ is the complete lattice of subsets of integers; the least upper bound operator of this domain coincides with the set union;
- $D(f)$ denotes the possible return values of the procedure $f$ and $D(x)$ the possible values of the variable $x$.

The functional set $\mathcal{D} = \mathit{Proc} \uplus \mathit{Var} \to \mathcal{P}(\mathbb{Z})$ ordered pointwise is a complete lattice (the codomain of any $D \in \mathcal{D}$ is finite).

This inference analysis is formalized on Fig. 3. It is based on the inspection of assignments, procedure call and return statements contained in procedures. We implicitly extend the function $D$ to expressions using Eqn. (3). Observe that we do not exploit the context of expressions: having the subexpression “`x+3`” or “`x?1:0`” in a procedure does not allow to infer any information on the possible values contained in $x$ in the C language. This analysis is quite similar to a constant propagation analysis, in which the constant flat lattice is replaced by the lattice $\mathcal{P}(\mathbb{Z})$.

The approximation we perform in this analysis is to consider that the set of possible values of any *integer* expressions (as defined in Fig. 2) is the set of all integer values. For instance, if $x$ may take the values 1 or 3 (*ie.,* $D(x) = \{1,3\}$), our analysis considers that $x+1$ may take any value (*ie.,* $D(x+1) = \mathbb{Z}$), instead of just a value in the set $\{2,4\}$. Without this approximation our analysis is not computable[4], because the lattice $\mathcal{D}$ does not satisfy the finite ascending chain condition. An alternative could be not to perform this approximation, but instead to use a widening operator that replaces finite subsets of $\mathbb{Z}$ by $\mathbb{Z}$ when their cardinality is greater than a given threshold. This alternative corresponds to the disjunctive completion of constant propagation analysis, equipped with a widening operator to ensure convergence.

[4] or at least very costly, if one considers that all variables are finite machine integers

Given a specific program, the longest chains of elements in $\mathcal{P}(\mathbb{Z})$ appearing in the analysis is of length $H$, being at most the number of numerical constants appearing in the program, plus 3 (because of the "predefined" constants 0, 1 returned by Boolean operators, and the top element $\mathbb{Z}$). Hence the full analysis converges in at most $H^{|Proc|+|Var|}$ steps, where $|Proc|$, $|Var|$ denotes resp. the number of procedures and variables.

## 3.3 Typing the analyzed program

Once the function $D$ is computed by the previous analysis, we have to translate the weakly-typed C program into a strongly-typed variant of the C language, in which operators are typed as described on Fig. 4(a).

This transformation is based on the fact that a *finite* value $D(x) = \{v^1, \dots, v^n\}$ implicitly defines an enumerated type, denoted $typ_{D(x)}$ in formula. If $D(x) = \mathbb{Z}$, then by convention $typ_{D(x)} = \mathsf{int}$. The transformation consists in two operations:

(i) Adding enumerated type declarations:
- for each different finite value $D_k = \{v_k^1, \dots, v_k^{n_k}\}$ of $D$ we insert the C type declaration "`typedef enum { lk1=vk1,...,lkn=vkn } tk`";
- we implicitly add the predefined type "`typedef enum { false=0, true=1 } bool`";
- each variable declaration "`int x`" with $D(x) = D_k$ is then replaced by "`tk x`". The same holds for the return type of procedures.

(ii) Inserting casts between integers and finite types, to ensure proper typing. Expressions and assignments are translated as defined in Fig. 4, in which we use the following operation on types:

$$t \sqcup t' = \begin{cases} t \text{ if } t = t' \\ \mathsf{int} \text{ otherwise} \end{cases} \tag{4}$$

Fig. 5(b) shows the results of this transformation on the prog. of Fig. 5(a). Observe that the definition of the cast operators $cast_{\mathsf{bool} \leftarrow \mathsf{int}}$ and $cast_{\mathsf{int} \leftarrow \mathsf{bool}}$ does not follow exactly the same pattern as for ordinary enumerated type, as *any* non-zero integer values is associated to the Boolean `true`.

## 3.4 Discussion

The soundness criterium is that the new program should have the same operational semantics as the original program. It is easy to see that typing error will not occur, given the properties of the function $D$ computed by the analysis and the definition of functions *typ* and *cast*.

```
int main()
{
  int x,y;
  y = x ? 1 : 0;
  x = 1;
  return y;
}
```

There is however a problem if some variables are read before being initialized. Look at the program on the right. Our inference analysis assigns to `x` the type `enum { l1=1 }`. Hence, seen as a Boolean, `x` is always true and the function returns `1`. The C99 standard specifies on the

| | | |
|---|---|---|
| $!^\flat$ | : | $\mathsf{bool} \to \mathsf{bool}$ |
| $-^\flat$ | : | $\mathsf{int} \to \mathsf{int}$ |
| $\&\&^\flat, \|\|^\flat$ | : | $\mathsf{bool}\times\mathsf{bool} \to \mathsf{bool}$ |
| $+^\flat, *^\flat, <<^\flat, \&^\flat, \ldots$ | : | $\mathsf{int}\times\mathsf{int} \to \mathsf{int}$ |
| $<^\flat, >^\flat, \ldots$ | : | $\mathsf{int}\times\mathsf{int} \to \mathsf{bool}$ |
| $==^\flat, !=^\flat$ | : | $\alpha\times\alpha \to \mathsf{bool}$ |
| $\cdot ?^\flat \cdot :^\flat \cdot$ | : | $\mathsf{bool}\times\alpha\times\alpha \to \alpha$ |

(a) Strongly-typed versions of C99 operators. $\alpha$ is a type variable used for polymorphic operators.

$$\begin{aligned}
typ(cst) &= int\\
typ(x) &= typ_{D(x)}\\
typ(op_1^\flat\ e) &= t \text{ if } op_1^\flat : t_1 \to t\\
typ(e_1\ op_2^\flat\ e_2) &= t \text{ if } op_2^\flat : t_1 \times t_2 \to t\\
typ(e_1 ?^\flat e_2 :^\flat e_3) &= typ(e_2) \sqcup typ(e_3)\\
typ(cast_{t_2 \leftarrow t_1}(e)) &= t_2
\end{aligned}$$

(b) Typing expressions

$$\begin{aligned}
[\![cst]\!] &\mathrel{\hat=} cst\\
[\![lv]\!] &\mathrel{\hat=} lv\\
[\![op_1\ e]\!] &\mathrel{\hat=} op_1^\flat(cast_{t_1\leftarrow t'}([\![e]\!])) \text{ if } \begin{cases} op_1^\flat : t_1 \to t\\ t' = typ([\![e]\!])\end{cases}\\
[\![e_1\ op_2\ e_2]\!] &\mathrel{\hat=} cast_{t_1\leftarrow t_1'}([\![e_1]\!])\ op_2^\flat\ cast_{t_2\leftarrow t_2'}([\![e_2]\!]) \text{ if } \begin{cases} op_2^\flat : t_1\times t_2 \to t\\ t_i' = typ([\![e_i]\!])\end{cases}\\
[\![e_1\ op_2\ e_2]\!] &\mathrel{\hat=} cast_{t'\leftarrow t_1'}([\![e_1]\!])\ op_2^\flat\ cast_{t'\leftarrow t_2'}([\![e_2]\!]) \text{ if } \begin{cases} op_2^\flat : \alpha\times\alpha \to \mathsf{bool}\\ t' = t_1' \sqcup t_2'\\ t_i' = typ([\![e_i]\!])\end{cases}\\
[\![e_1\ ?\ e_2 : e_3]\!] &\mathrel{\hat=} cast_{\mathsf{bool}\leftarrow t_1'}([\![e_1]\!])\ ?^\flat\ cast_{t'\leftarrow t_2'}([\![e_2]\!]\ :^\flat\ cast_{t'\leftarrow t_3'}([\![e_3]\!]\\
&\qquad\qquad \text{if } \begin{cases} t' = t_2' \sqcup t_3'\\ t_i' = typ([\![e_i]\!])\end{cases}\\
[\![lv = e]\!] &\mathrel{\hat=} lv = cast_{t\leftarrow t'}([\![e]\!]) \text{ if } \begin{cases} t = typ(lv)\\ t' = typ([\![e]\!])\end{cases}\\
[\![lv = f(e_1,\ldots,e_n)]\!] &\mathrel{\hat=} lv = cast_{t'\leftarrow t}(f(cast_{t_1\leftarrow t_1'}([\![e_1]\!],\ldots,cast_{t_n\leftarrow t_n'}([\![e_n]\!]))\\
&\qquad\qquad \text{if } \begin{cases} f : t_1\times\ldots\times t_n \to t\\ t' = typ(lv)\\ t_i' = typ([\![e_i]\!])\end{cases}
\end{aligned}$$

(c) Translating expressions and assignments by inserting casts

$$cast_{t\leftarrow t}(e) = e$$

$$\begin{aligned}
cast_{t_k\leftarrow\mathsf{int}}(e) &= (e == v_k^1)\ ?\ l_k^1\ :\\
&\ \ \vdots\\
&\quad (e == v_k^{n_k-1})\ ?\ l_k^{n_k-1}\ :\ l_k^{n_k}\\
cast_{\mathsf{bool}\leftarrow\mathsf{int}}(e) &= (e == 0)\ ?\ \mathsf{false} : \mathsf{true}
\end{aligned}
\qquad
\begin{aligned}
cast_{\mathsf{int}\leftarrow t_k}(l) &= (l == l_k^1)\ ?\ v_k^1\ :\\
&\ \ \vdots\\
&\quad (l == l_k^{n_k-1})\ ?\ v_k^{n_k-1}\ :\ v_k^{n_k}\\
cast_{\mathsf{int}\leftarrow\mathsf{bool}}(e) &= e\ ?\ 1 : 0
\end{aligned}$$

$$cast_{t\leftarrow t'} = cast_{t\leftarrow\mathsf{int}} \circ cast_{\mathsf{int}\leftarrow t'} \text{ if } t \neq t'$$

(d) Definition of cast operators

Fig. 4. Generating a strongly typed version of the program

other hand that the value of `x` is undefined when `y` is assigned, which means that it can have any value. To deal with this aspect without complicating our framework, we choose to impose that *all variables are initialized before being read.* Checking this assumption can be done with the classical dataflow analysis implemented in most C compilers, and enforcing it can be done on the original program by replacing any non-parameter declaration "`int x`" by "`int x=0`".

A second important point is related to our motivation to exploit the ability of some tools to analyze more precisely finite-state variables. Because we insert casts from enumerated types to integers, we may loose at first glance the benefit of assigning enumerated types to some original integer variables of a program. This will not happen with the ConcurInterproc tool, thanks to the

```
int incrmod2(int x)
{
  if (x==0) x=1;
  else x=0;
  return x;
}
int main()
{
  int y = incrmod2(1);
  return y;
}
```

(a) Original C program

```
typedef enum { k0=0,k1=1 } t;
t incrmod2(t x)
{
  if (cast_int_t(x)==0) x=cast_t_int(1);
  else x=cast_t_int(0);
  return x;
}
t main()
{
  t y = incrmod2(cast_t_int(1));
  return y;
}
```

(b) Adding finite types

```
typedef enum { k0=0,k1=1 } t;
t incrmod2(t x)
{
  if ((x==k0 ? 0 : 1)==0) x=(1==0 ? k0 : k1);
  else x=(0==0 ? k0 : k1);
  return x;
}
int main()
{
  t y = incrmod2(1==0 ? k0 : k1);
  return y;
}
```

(c) Expanding casts

```
typedef enum { k0=0,k1=1 } t;
t incrmod2(t x)
{
  if (x==k0) x=k1;
  else x=k0;
  return x;
}
int main()
{
  t y = incrmod2(k1);
  return y;
}
```

(d) Propagating constants

Fig. 5. Inferring enumerated types and transforming the original program.

way it normalizes expressions by pushing operators in the branches of conditional expressions and simplifying trivial tests. For instance, it rewrites the expression `if ((x==k0 ? 0 : 1)==0) x=(1==0 ? k0 : k1)` in Fig. 5(c) as follows:

```
if ((x==k0 ? 0 : 1)==0) x=(1==0 ? k0 : k1)   ⇒
if (x==k0 ? 0==0 : 1==0) x=k1                ⇒   if (x==k0) x=k1
```

## 4 Adding pointers, structures and dynamic allocation

We now add pointers, structured types and dynamic allocation to our language. We extend the grammar of Fig. 2 as follows:

$$\begin{array}{rclrcl}
\langle\text{typ}\rangle & ::= & \langle\text{typ0}\rangle\text{``}*\text{''}^k & \langle\text{typ0}\rangle & ::= & \textbf{int} \mid \text{``}\texttt{typedef struct \{}\text{''}(\langle\text{typ}\rangle\ n)^*\text{``\}''}t \\
\langle\text{expr}\rangle & ::= & \ldots \mid \langle\text{pexpr}\rangle & \langle\text{pexpr}\rangle & ::= & \textbf{null} \mid \text{``\&''}x \mid \text{``\&''}(x \to n) \\
\langle\text{lv}\rangle & ::= & x \mid \text{``}*\text{''}x & \langle\text{stm}\rangle & ::= & \ldots \mid \langle\text{lv}\rangle = \textbf{alloc}(\langle\text{typ}\rangle)
\end{array} \tag{5}$$

We add in particular the operator `&` which creates a pointer value from a variable or a field of a structure (no function pointers). $n, m, \ldots$ denotes names of structures fields, assumed to be unique. We allow only one `*` operator in left-values (including the implicit `*` of `->`). Assignments like "`**x=**y`" should be decomposed as "`px=*x; py=*y; *px=*py`" and "`a->n = b->m`" as "`pa = &(a->n); pb = &(a->m); *pa = *pb;`".

The important assumption we do in this section is that there is no (implicit or explicit) cast between the types $t_1{*}^k$ and $t_2{*}^{k'}$ with $k \neq k' \vee t_1 \neq t_2$, and that the program is well-typed in this respect.

```
int main()
{
  int x = 0;
  int y = 1;
  int* p = NULL;
  p = &x; *p = 2;
  p = &y; *p = 3;
  return *p;
}
```

(a) Original program

```
typedef enum {
l0=0,l1=1,l2=2,l3=3
} t;
t main()
{
  t x = l0;
  t y = l1;
  t* p = NULL;
  p = &x; *p = l2;
  p = &y; *p = l3;
  return *p;
}
```

(b) Final program

Fig. 6. Program with pointers to scalars

```
typedef struct {
  int n;
} t;
int main()
{
  t x; t* y;
  int *p,*q;
  y = alloc(t);
  p = &(y->n);
  y = &x;
  q = &(y->n);
  *p = 1;
  *q = 2;
  *p = *p < 1;
  return *p;
}
```

(a) Original program

```
typedef enum {
  l0=0,l1=1,l2=2
} e;
typedef struct {
  e n;
} t;
int main()
{
  t x; t* y;
  e *p,*q;
  ...
  *p = l1;
  *q = l2;
  *p = (*p==l0)?l1:l0;
  return *p;
}
```

(b) Final program

Fig. 7. Program with structures

## 4.1 Purpose of our inference analysis

In Section 3 our finite type inference reduced to the analysis of possible values of scalar variables. In this new setting, the goal of our type inference is

(i) as before to detect the *scalar variables* that are manipulated as Boolean or enumerated types, and to infer the corresponding type;

(ii) but also to do so for the *fields of structured types*;

while taking into account typing and aliasing properties induced by pointers. Our analysis will return a unique type for a given field name, meaning that we renounced to capture distinct (boolean/integer) uses of the same structured type in different contexts.

Consider the program of Fig. 6(a). We want to infer that $p$ may point to $x$ or $y$. This allows to infer that $D(x) = \{0, 2, 3\}$ and $D(y) = \{1, 2, 3\}$. Now, as $x$ and $y$ may be pointed to by the same pointer $p$, they should have the same type. Hence we generate the program of Fig. 6(b).

Consider now the program of Fig. 7(a). We know that $y$ is a pointer to a structure of type $t$, by its type. *We do not need more information about pointers to structures*, as the field $n$ of all structures of a given type may be eventually specialized to a unique type. In other words, all the locations corresponding to the field $n$ are summarized into a single location named `.n`. We still need to infer that $p$ and $q$ may point to the scalar field $.n$ of an object of type $t$, and to deduce from this fact that the scalar field may contain a value in the set $D(.n) = \{0, 1, 2\}$. This results in the program of Fig. 7(b).

To conclude, we need a *weak* form of points-to analysis, in which we are only interested in points-to relation between pointers variables, integer variables and fields of structures.

$$\frac{x = \mathit{expr} \in \langle\text{stm}\rangle \quad \mathsf{int}{*}^{+}\ x \in \langle\text{decl}\rangle}{P(x) \supseteq P(\mathit{expr})} \qquad \frac{{*}x = \mathit{expr} \in \langle\text{stm}\rangle \quad \mathsf{int}{*}{*}^{+}\ x \in \langle\text{decl}\rangle}{\forall y \in P(x) : P(y) \supseteq P(\mathit{expr})}$$

$$\frac{\textbf{return}\ \mathit{expr} \in \langle\text{stm}\rangle(f) \quad \mathsf{int}{*}^{+}\ f(\ldots) \in \mathit{Proc}}{P(f) \supseteq P(\mathit{expr})} \qquad \frac{\begin{array}{c} x = f(\mathit{expr}_1, \ldots, \mathit{expr}_n) \in \langle\text{stm}\rangle \\ \mathit{typ}\ f(\mathit{typ}_1\ x_1, \ldots, \mathit{typ}_n\ x_n) \in \langle\text{proc}\rangle \end{array}}{\begin{array}{c} P(x) \supseteq P(f) \text{ if } \mathit{typ} = \mathsf{int}{*}^{+} \\ \forall i \mid \mathit{typ}_i = \mathsf{int}{*}^{+} : P(x_i) \supseteq P(\mathit{expr}_i) \end{array}}$$

$$\begin{aligned} P(\mathsf{null}) &= \emptyset \\ P(\&x) &= \{x\} \quad \text{(only applied to a var. of type } \mathsf{int}{*}^{*}) \\ P(\&(x \to n)) &= \{.n\} \quad \text{(only applied to a field of type } \mathsf{int}{*}^{*}) \\ P(\mathit{expr}\,\text{“?”}\,\mathit{pexpr}_1\,\text{“:”}\,\mathit{pexpr}_2) &= P(\mathit{pexpr}_1) \cup P(\mathit{pexpr}_2) \end{aligned} \tag{6}$$

Fig. 8. Points-to analysis

$$\frac{x = \mathit{expr} \in \langle\text{stm}\rangle \quad \mathsf{int}\ x \in \langle\text{decl}\rangle}{D(x) \supseteq D(\mathit{expr})} \qquad \frac{{*}x = \mathit{expr} \in \langle\text{stm}\rangle \quad \mathsf{int}{*}^{+}\ x \in \langle\text{decl}\rangle}{\forall y \in P(x)\ :\ D(y) \supseteq D(\mathit{expr})}$$

$$\frac{\textbf{return}\ \mathit{expr} \in \langle\text{stm}\rangle \quad \mathsf{int}\ f(\ldots) \in \langle\text{proc}\rangle}{D(f) \supseteq D(\mathit{expr})} \qquad \frac{\begin{array}{c} x = f(\mathit{expr}_1, \ldots, \mathit{expr}_n) \in \langle\text{stm}\rangle \\ \mathit{typ}\ f(\mathit{typ}_1\ x_1, \ldots, \mathit{typ}_n\ x_n) \in \langle\text{proc}\rangle \end{array}}{\begin{array}{c} D(x) \supseteq D(f) \text{ if } \mathit{typ} = \mathsf{int} \\ \forall i \mid \mathit{typ}_i = \mathsf{int} : D(x_i) \supseteq D(\mathit{expr}_i) \end{array}}$$

$$\begin{aligned} D(\mathit{cst}) &= \{\mathit{cst}\} \\ D({*}x) &= \textstyle\bigcup_{y \in P(x)} D(y) \\ D(\mathit{boolexpr}) &= \{0, 1\} \\ D(\mathit{intexpr}) &= \mathbb{Z} \\ D(\mathit{expr}\,\text{“?”}\,\mathit{expr}_1\,\text{“:”}\,\mathit{expr}_2) &= D(\mathit{expr}_1) \cup D(\mathit{expr}_2) \end{aligned} \tag{7}$$

Fig. 9. Inferring possible values for variables and fields

## 4.2 Formalization of the analysis

We still perform a weak form of flow and context-insensitive points-to analysis, that infers a function

$$P : \mathit{Proc} \uplus \mathit{Var} \uplus \mathit{Field} \to \mathcal{P}(\mathit{Var} \uplus \mathit{Field})$$

which maps procedure return values, variables and fields of pointer type to variables and fields. $P(x)$ (resp. $P(.n)$) will be an overapproximation of the set of variables and fields to which $x$ (resp. the field $.n$ of any object) may point to. This function is the smallest solution of the inference rules of Fig. 8, in which Eqn. (6) extends $P$ to expressions of type $\mathsf{int}{*}^{k}, k > 0$.

We then generalize the scalar value analysis of Section 3.2 by inferring a function

$$D : \mathit{Proc} \uplus \mathit{Var} \uplus \mathit{Field} \to \mathcal{P}(\mathbb{Z})$$

which maps integer variables and fields to possible values. This function is the smallest solution of the inference rules of Fig. 9, in which Eqn. (7) extends $D$ to expressions of type $\mathsf{int}$.

## 4.3 Typing the analyzed program

As mentioned in Section 4.1, assigning types to variables is a bit more complex than in the purely scalar case, because two variables pointed to by the same pointer should be given the same type. Otherwise, the need for a cast may depend on the value of the pointer. Therefore,

- If $x$ (or $.n$) is initially declared as an integer, $typ(x) = typ_{\bigcup\{D(y) \mid \exists p: P(p) \supseteq \{x,y\}\}}$;
- If $x$ (or $.n$) is initially declared as a pointer $\mathtt{int}*^k, k > 0$, $typ(x) = (typ_{\bigcup_{y \in P^k(x)} D(y)})*^k$, where $P^k$ denotes the $k$-th iterate of $P$.

The insertion of casts is done exactly as in Section 3.3. Observe that we do not need casts between pointers: we cannot have "`t* x; int* y;...; y=x`" in the final program, because such an assignment makes the variables and fields pointed to by $x$ and $y$ (hence, also $x$ and $y$) having the same type.

### 4.4 Discussion

In this section, we extended the proposition of Section 3 to a broader subset of C. However this proposal was done under some assumptions (absence of casts and pointer arithmetic) and should be seen as a demonstration of how the value analysis and points-to analysis interact. It is possible to relax these assumptions by using classical well-studied points-to analysis. In particular, the technique of Steensgarrd [11] seems well-suited, since it is interprocedural, flow-insensitive and it accepts the language of Equation 5. This technique infers the pointing relation and the effective structures manipulated by a C program with casts.

Handling arrays in addition to structures and pointers can be integrated to the points-to analysis by giving a unique type to the whole array and assuming that no out-of-range access occurs.

Note that the condition that variables must be initialized before being read, mentioned in 3.4, should also be satisfied for dynamically allocated memory, but this is more complex to check or to enforce.

## 5 Conclusion

We presented a way to determine the set of Boolean and enumerated variables among a set of variables of type `int` in a C program. This information, of little use for compilation, allows to improve the precision of program verification by assigning these variables to the adequate abstract domain.

The process takes as input a large subset of C (including functions, structures, pointers) and performs a simple points-to analysis followed by a value analysis. The results of these analyses allows to transform the program in a strongly-typed equivalent version by refining the types and by inserting explicit casts in the right place.

Note that this work would not be necessary if the abstract domains used by the analysers where able to dynamically switch the types of the variables they manipulate when the latter are escaping their capabilities. But the abstract domains proposed in the literature tend to be very specialized (eg. floating points [2], numerical arrays [4]), and taking more general cases into account would add a burden to their complexity.

Our work is complementary with the compilation of C program to intermediate language or to simpler subsets [9,5]. These proposals can be seen as frontends dedicated to verification by reducing the gap between C and the simpler analyser input language, thus answering Point (i) of the introduction.

An implementation has been developed for ConcurInterproc [6], having `c2newspeak` [5] as a frontend. The analyser only handles scalar types thus does not require the points-to version of the analysis (Section 4) but further developments for analysers with richer memory model will benefit from it.

## References

[1] Evans, D. and D. Larochelle, *Improving security using extensible lightweight static analysis*, IEEE Software **19** (2002), pp. 42–51.

[2] Goubault, E., M. Martel and S. Putot, *Asserting the precision of floating-point computations: A simple abstract interpreter*, in: D. L. Métayer, editor, *ESOP*, Lecture Notes in Computer Science **2305** (2002), pp. 209–212.

[3] Guha, A., C. Saftoiu and S. Krishnamurthi, *Typing local control and state using flow analysis*, in: G. Barthe, editor, *ESOP*, Lecture Notes in Computer Science **6602** (2011), pp. 256–275.

[4] Halbwachs, N. and M. Péron, *Discovering properties about arrays in simple programs*, in: R. Gupta and S. P. Amarasinghe, editors, *PLDI* (2008), pp. 339–348.

[5] Hymans, C. and O. Levillain, *Newspeak, Doubleplussimple Minilang for Goodthinkful Static Analysis of C*, Technical report, EADS (2008).

[6] Jeannet, B., *Relational interprocedural verification of concurrent programs*, in: *Software Engineering and Formal Methods, SEFM'09* (2009).

[7] Jim, T., J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney and Y. Wang, *Cyclone: A safe dialect of c*, in: C. S. Ellis, editor, *USENIX Annual Technical Conference, General Track* (2002), pp. 275–288.

[8] Miné, A., *The octagon abstract domain*, Higher-Order and Symbolic Computation **19** (2006).

[9] Necula, G. C., S. McPeak, S. P. Rahul and W. Weimer, *CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs*, in: R. N. Horspool, editor, *CC*, LNCS **2304** (2002), pp. 213–228.

[10] Sotin, P. and B. Jeannet, *Precise interprocedural analysis in the presence of pointers to the stack*, in: G. Barthe, editor, *ESOP*, Lecture Notes in Computer Science **6602** (2011), pp. 459–479.

[11] Steensgaard, B., *Points-to analysis by type inference of programs with structures and unions*, in: T. Gyimóthy, editor, *CC*, Lecture Notes in Computer Science **1060** (1996), pp. 136–150.