Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 267 (2010) 139–150
www.elsevier.com/locate/entcs

Concrete Memory Models for Shape Analysis

Pascal Sotin[1,2] and Bertrand Jeannet[3]

INRIA Grenoble – Rhône-Alpes
655 avenue de l'Europe
38 334 Saint Ismier Cedex, France

Xavier Rival[4]

INRIA Rocquencourt and DI - Ecole Normale Supérieure
45, rue d'Ulm
75230 Paris Cedex 05 - France

Abstract

This paper discusses four store-based concrete memory models. We characterize memory models by the class of pointers they support and whether they use numerical or symbolic offsets to address values in a block. We give the semantics of a C-like language within each of these memory models to illustrate their differences. The language we consider is a fragment of Leroy's Clight, including arrays, pointer arithmetics but excluding casts. All along the paper, we link these concrete memory models with existing shape analyses.

Keywords: Memory models, language semantics, C-like orogramming languages, shape analysis.

1 Introduction

The purpose of shape analysis is to infer properties on the runtime structure of the memory heap. Shape analysis goes beyond alias and null-pointer analyses, in term of expressivity and precision. The applications of shape analyses include optimizing compilation, absence of runtime errors (dereference of dangling or null pointers), proof of programs, automatic parallelisation, ...

Memory models. Like most static analyses, shape analyses perform approximation. One has thus to distinguish the concrete memory model that a shape analysis

[1] Supported by ANR ASOPT.
[2] Email: pascal.sotin@inria.fr
[3] Email: bertrand.jeannet@inria.fr
[4] Email: xavier.rival@ens.fr

doi:10.1016/j.entcs.2010.09.012

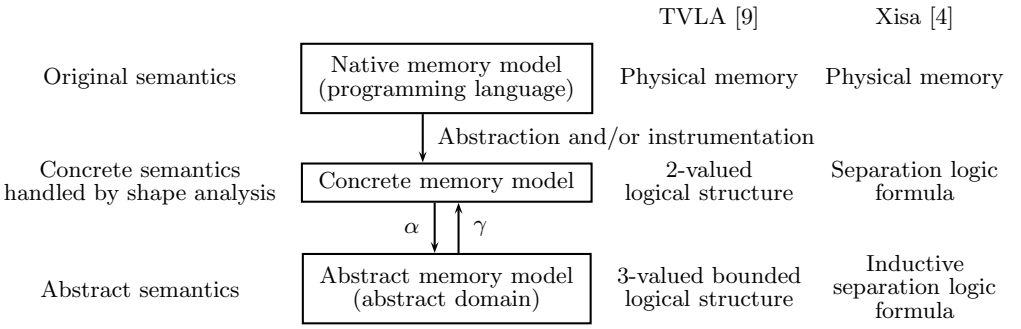


Fig. 1. Native, concrete and abstract memory models

tackles, and the abstract memory model/representation used by the analysis to express properties. For instance, in [9], the concrete memory model is an unbounded 2-valued logical structure, and the abstract memory representation is a bounded 3-valued logical structure. This paper focuses on the concrete memory models and the operations they allow.

However, these concrete models do actually abstract some properties, as they do not completely model the physical memory of a computer. For instance, the physical numerical addresses may be ignored, as is the case for [9] which cannot define the semantics of C pointer arithmetics.

Thus, concrete memory models can be classified by the operations they support and that can be analysed by a shape analysis based on it. Ultimately, the set of operations to be supported is defined by the programming language under consideration. Fig. 1 depicts the articulation between programming languages, concrete and abstract memory models, and instantiates these concepts with two very expressive shape analyses, TVLA and Xisa [9,4].

Classifying memory models. If one looks at the native memory models of the most common programming languages, one can distinguish two main models:

- Java, OCaml, and similar languages with garbage collection share a memory model in which pointers always point to the base address of objects.
- C, C++, and to a lesser extent ADA, in which pointers can also point inside an object, and where more operations are allowed, such as taking the address of a record field, either explicitly (C, C++) or implicitly (reference parameter passing in C++, ADA), pointer arithmetics, ...

This is naturally reflected in concrete memory models, which we classify as either *object*, or *standard*, the latter one supporting more operations on pointers.

An orthogonal consideration, rather independent from the language, is the way fields inside a memory location are addressed. This is usually done by combining a location identifier and an *offset* within it. Concrete memory models may use either symbolic offsets (e.g., $p.f[3].g$ in Java syntax) or numerical offsets (e.g., $\langle p, 8\rangle$). This provides a second classification criterion for memory models, as illustrated on Tab. 1.

Offset type \ Pointer use	standard	object
numerical	StdNum	ObjNum
symbolic	StdSym	ObjSym

Table 1
Classification of concrete memory models

$s \in State$	$= Env \times Store$	:	program state
$\epsilon \in Env$	$= Var \to Loc$	:	environment
$\sigma \in Store$	$= Addr \to Scalar \cup Ptr$	:	store mapping addresses to values
$Addr$	$= Loc \times Offset$	:	addresses
$l \in Loc$		:	location (atomic memory block)
Ptr	$= \left\langle \begin{array}{ll} Loc \times Offset & : \text{Standard} \\ Loc & : \text{Object} \end{array} \right.$		
$o \in Offset$	$= \left\langle \begin{array}{ll} \mathbb{Z} & : \text{Numerical} \\ Path = (Field \cup \mathbb{Z})^\star & : \text{Symbolic} \end{array} \right.$		

Table 2
Semantic domains

Motivations. Analyses, like TVLA, were developed for Java-like languages. Others, such as the separation-logic-based Xisa, target a subset of C. The memory models for shape analyses are often only described at the abstract level. A clear view of the concrete memory model is needed to understand the scope of these analyses and be able to reuse them in different context.

Contributions. We present the formal semantics of a C-like programming language within the four memory models mentioned in Tab. 1. We discuss the operations supported by each of them, emphasizing on differences and on shape analyses using them.

2 Semantic Domains and Clight Expressions

In this section, we present formally the four memory models, and introduce the considered language.

2.1 Memory Models

We consider the four memory models depicted in Tab. 1 leading to the store-based semantics domains of Tab. 2 (i.e., with explicit store). An environment ($\epsilon \in Env$) is a mapping from the program variables (Var) to the memory locations (Loc) where their content is stored. A store ($\sigma \in Store$) is a mapping from addresses ($Addr = Loc \times Offset$) to values $Values = Scalar \cup Ptr$. The scalar values ($Scalar$) are for example integer values. Memory models differ on the nature of pointers (Ptr)

statement ::=	*lexpr*=*expr*	assignment	
expr ::=	*lexpr*	left value	
|	&*lexpr*	address taking	
|	*expr* + *aexp*	pointer arithmetics	
lexpr ::=	id	variable	
|	*lexpr*.id	field selection	
|	∗*expr*	dereferencing	
aexp ::=	...	arithmetical expr.	

(a) Expressions.

τ ::= `int`
| `array`(τ, n)
| `pointer`(τ)
| `struct`$\{(\text{id}, \tau)^\star\}$
| *name*

(b) Types.

Fig. 2. Considered Clight fragment.

and the nature of offsets (*Offset*). Note that locations are not given a numerical address and are unordered. As a consequence, it is impossible, with an address $\langle l, o\rangle$, to refer to an element of another location $l' \neq l$, whatever the value of o. This is the case of some formal semantics of C [1] and most shape analyses [9,3,4].

The *standard* memory model is close to C-like low-level languages as everything that can be addressed can also be stored in a pointer. The *object* memory model is close to languages like ML or Java, where pointers are restricted to *references*, which can designate an object, but not a field or a cell.

Beyond the nature of pointers, another characteristic of a memory model is the nature of offsets. The *symbolic* memory model is higher-level and deals with sequences of labels, called paths. A label is either a field name or an index in an array. We write ε for the empty path and $\pi.f$ for the path π continued with label f. The *numerical* memory model is lower-level and deals with true offsets (in bytes) within locations. This memory model can be used only when the target architecture is known (size of types, layouts). An Application Binary Interface (ABI) should provide such information, allowing an architecture-based manipulation of the numerical memory model.

One goal of this paper is to discuss how to express the semantics of a fragment of Clight within the four memory models of Tab. 2.

2.2 Clight

We take a fragment of Clight [1] that excludes cast, union types and multidimensional arrays. Numerical expressions are included in this fragment, but we do not detail them as they do not involve memory issues. We thus consider the expressions defined by the grammar of Fig. 2(a).

A Clight program is statically typed. This allows us to know the type $\tau \in \mathit{Type}$ of any expression. When we need this information, we write expr^τ to bind the type of *expr* with τ. We consider the types defined by the grammar of Fig. 2(b). We assume that it is possible to name types (e.g. with a `typedef`), e.g., in order to handle recursive data structures.

In Clight, only numbers and pointers may be assigned. The assignment of a structure is not allowed and arrays (e.g. `int t[2]`) are used like pointers (e.g. `int *p`). We use the syntactic sugar `t[n]` for `*(t+n)`.

3 Semantics for Clight in Standard Memory Model

In this section, we give a semantics for our fragment of Clight within the standard memory model, considering either numerical or symbolic offsets. We recall that the store is of the form: $\mathit{Store} = \mathit{Loc} \times \mathit{Offset} \to \mathit{Scalar} \cup (\mathit{Loc} \times \mathit{Offset})$. This store allows pointers to be taken within a block. In [6], Laviron et al. present an analysis, inspired by separation logic, that implements this form of pointers in the abstract domain. The analysis of Calcagno et al. [2] also uses an instance of a similar model.

Parameters. We parametrize the standard semantics by three operators (., + and $\downarrow$) and a constant ($\emptyset$), so as to be generic for the symbolic and numerical variants of the model.

$$\begin{aligned} \emptyset &: \mathit{Offset} \\ \langle l, o\rangle.{}_\tau f &: (\mathit{Loc} \times \mathit{Offset}) \times \mathit{Type} \times \mathit{Field} \to \mathit{Loc} \times \mathit{Offset} \\ \langle l, o\rangle +_\tau k &: (\mathit{Loc} \times \mathit{Offset}) \times \mathit{Type} \times \mathbb{Z} \to \mathit{Loc} \times \mathit{Offset} \\ \downarrow \langle l, o\rangle &: (\mathit{Loc} \times \mathit{Offset}) \to \mathit{Loc} \times \mathit{Offset} \end{aligned}$$

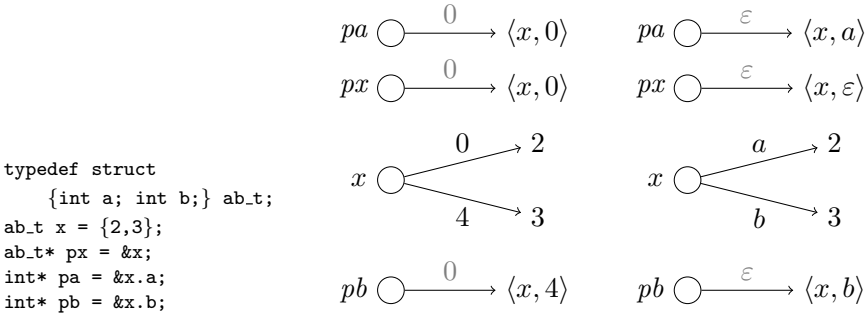
$\emptyset$ stands for the empty offset. $\langle l, o\rangle.{}_\tau f$ computes the address of a member f within a structure of type τ pointed to by $\langle l, o\rangle$. $\langle l, o\rangle +_\tau k$ computes the address resulting of pointer arithmetics on a pointer $\langle l, o\rangle$ to an object of type τ. $\downarrow \langle l, o\rangle$ returns a pointer on the first cell of an array pointed to by $\langle l, o\rangle$ (with symbolic offsets, there is the need to distinguish these two notions, in order to allow pointer arithmetics). These operators are defined by:

Numerical	Symbolic
$\emptyset = 0$	$\emptyset = \varepsilon$
$\langle l, n\rangle.{}_\tau f = \langle l, n + \texttt{offsetof}(f, \tau)\rangle$	$\langle l, \pi\rangle.f = \langle l, \pi.f\rangle$
$\langle l, n\rangle +_\tau k = \langle l, n + k \times \texttt{sizeof}(\tau)\rangle$	$\langle l, \pi.n\rangle + k = \langle l, \pi.(n + k)\rangle$
$\downarrow \langle l, n\rangle = \langle l, n\rangle$	$\downarrow \langle l, \pi\rangle = \langle l, \pi.0\rangle$

Numerical operators use types through the functions `offsetof` and `sizeof`, which are defined by the ABI.

Semantics. We define three semantic functions:

$$\begin{aligned} [\![]\!]^{\mathcal{A}} &: \mathit{Env} \times \mathit{Store} \to \mathit{Loc} \times \mathit{Offset} && \text{Address of a expression} \\ [\![]\!]^{\mathcal{V}} &: \mathit{Env} \times \mathit{Store} \to \mathit{Scalar} \cup (\mathit{Loc} \times \mathit{Offset}) && \text{Value of an expression} \\ [\![]\!]^{\mathcal{S}} &: \mathit{Env} \times \mathit{Store} \to \mathit{Env} \times \mathit{Store} && \text{Effect of a statement} \end{aligned}$$



(a) C code. (b) StdNum. (c) StdSym.

Fig. 3. Standard store with simple structure.

The semantic of statements is rather standard:

$$\begin{aligned}[\![lexpr\texttt{=}expr]\!]^{\mathcal{S}}(\epsilon,\sigma) \;=\; & \text{let } \langle l, o\rangle = [\![lexpr]\!]^{\mathcal{A}}(\epsilon,\sigma) \text{ in} \\ & \text{let } v = [\![expr]\!]^{\mathcal{V}}(\epsilon,\sigma) \text{ in} \\ & (\epsilon, \sigma[\langle l, o\rangle \mapsto v])\end{aligned}$$

For the two other functions, we elide the store (σ) and environment (ϵ) parameters as they are constant.

$$[\![\mathrm{id}]\!]^{\mathcal{A}} = \langle \epsilon(\mathrm{id}), \emptyset\rangle \qquad [\![lexpr^{\tau}.\mathrm{id}]\!]^{\mathcal{A}} = [\![lexpr]\!]^{\mathcal{A}}.{}_{\tau}\mathrm{id} \qquad [\![*expr]\!]^{\mathcal{A}} = [\![expr]\!]^{\mathcal{V}}$$

$$[\![lexpr^{\tau}]\!]^{\mathcal{V}} = \text{if } \texttt{is\_array}(\tau) \text{ then } \downarrow [\![lexpr]\!]^{\mathcal{A}} \text{ else } \sigma([\![lexpr]\!]^{\mathcal{A}})$$

$$[\![\&lexpr]\!]^{\mathcal{V}} = [\![lexpr]\!]^{\mathcal{A}}$$

$$[\![expr\ {}^{\mathtt{ptr}(\tau)} + aexp]\!]^{\mathcal{V}} = [\![expr]\!]^{\mathcal{V}} +_{\tau} [\![aexp]\!]^{\mathcal{V}}$$

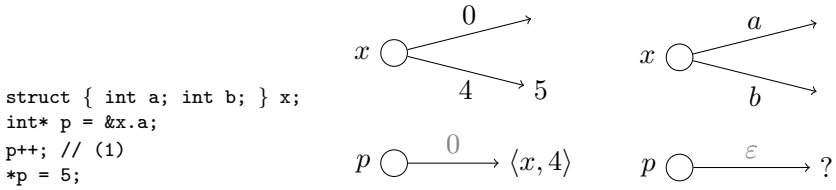
Both semantics evaluate left-values of array type using references. In addition, the symbolic variant transforms array left-value $p.\pi$ into the pointer to the first cell of the array $\downarrow p.\pi = p.\pi.0$. We refer to the numerical and symbolic variants of this semantics respectively by *StdNum* and *StdSym*.

Example. We consider a simple program and the memory states it generates. Program states are depicted with the following conventions:

- A location is depicted by a circle (distinct circles are distinct locations).
- When a location l is pointed to by a variable x (i.e., $\epsilon(x) = l$), the name x is written near the location.
- A binding $\langle l, o\rangle \mapsto v$ in the store is depicted by an arrow, starting at location l, labelled by o and pointing to v.

Figure 3 illustrates the two memories for a simple structure. Note that in the numerical model, the address of `x` points to the beginning of the structure, and is not distinct from the address of `x.a`.

Numerical more expressive than symbolic. As mentioned in Sect. 2.1, ad-



(a) C code. (b) StdNum. (c) StdSym.

Fig. 4. Unrestricted pointer arithmetics.

dresses belonging to different locations are incomparable. *Inside* a location, both models are able to deal with pointer arithmetics within an array. However, for the StdNum model in which offsets are numbers, pointer arithmetics in a structure may indeed lead to correct executions, as illustrated by Figure 4. On line `(1)` of Figure 4(a), with the numerical model, the value of pointer `p` coincides with the address of `x.b` and causes the final memory to be as depicted in Figure 4(b). The same phenomenon happens when an array is accessed outside of its bounds. The numerical domain can match the out-of-bound address with another value of the structure, while the symbolic domain cannot.

4 Semantics in Native Object Memory Model

Many shape analyses (e.g. [9,3]) are based on the object memory model, and thus are suitable mainly for Java-like languages and cannot handle full C. In this model, the store only allows pointers to be taken on the base address of a block: $\mathit{Store} = \mathit{Loc} \times \mathit{Offset} \to \mathit{Scalar} \cup \mathit{Loc}$ (see Tab. 2). For example, if `x` is a structure and `t` an array, expressions like `&x` and `&t[0]` can be stored in a pointer, while expressions like `&x.a` or `&t[2]` cannot.

This model can be used to give a semantics to the subset of Clight, where the "address of" operator can be applied only to variables, so that all pointer values have a null offset. In the following section however, we show how to model our full Clight fragment on an *instrumented* object memory model.

5 Semantics in Instrumented Object Memory Model

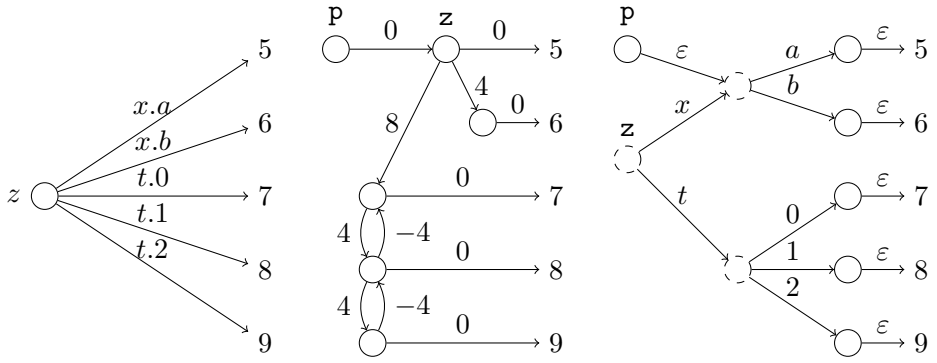
In this section, we instrument the object memory model in order to allow pointers within a block.

5.1 Instrumenting the store

We do not formalize the instrumentation; instead, we sketch the principle in Figure 5. First, locations now correspond to single memory cells and not to blocks; they are obtained by splitting the former locations of the standard memory. Original edges from Fig. 5(b) can still be found in Figures 5(d) and 5(c) with ø offset. Then, we add enough information to navigate within structures and arrays. Edges with offsets different from ø are instrumentation edges (we tried to keep it minimal).

```
struct {ab_t x; int[3] t;} z;
ab_t* p = &z.x;
```

(a) C code.



(b) Symbolic standard. (c) Numerical instr. (d) Symbolic instr.

Fig. 5. Instrumented object store.

In Fig. 5(d), the dashed locations are virtual, since they have no ø edge, hence no associated value.

In [5], Kreiker et al. adapt TVLA so as to handle pointers within structures. The concrete memory model of Fig. 5(d) corresponds to their fine-grain semantics.

5.2 Object Semantics in the instrumented store

We now present the numerical and symbolic instrumented object semantics. Both semantics will navigate through the store using instrumentation edges. They differ by the way they enter a structure and the way they perform pointer arithmetics. We parametrize our semantics with ø, ., + and ↓, like the standard semantics of Section 3.

$$\begin{aligned} \emptyset &: \mathit{Offset} \\ l._\tau f &: \mathit{Loc} \times \mathit{Type} \times \mathit{Field} \to \mathit{Loc} \\ l +_\tau k &: \mathit{Loc} \times \mathit{Type} \times \mathbb{Z} \to \mathit{Loc} \\ \downarrow l &: \mathit{Loc} \to \mathit{Loc} \end{aligned}$$

Semantics.

$$\begin{aligned} [\![]\!]^{\mathcal{A}} &: \mathit{Env} \times \mathit{Store} \to \mathit{Loc} && \text{Address of an expression} \\ [\![]\!]^{\mathcal{V}} &: \mathit{Env} \times \mathit{Store} \to \mathit{Scalar} \cup \mathit{Loc} && \text{Value of an expression} \\ [\![]\!]^{\mathcal{S}} &: \mathit{Env} \times \mathit{Store} \to \mathit{Env} \times \mathit{Store} && \text{Effect of a statement} \end{aligned}$$

$$[\![lexpr{=}expr]\!]^{\mathcal{S}}(\epsilon,\sigma) = \text{let } l = [\![lexpr]\!]^{\mathcal{A}}(\epsilon,\sigma) \text{ in}\\ \text{let } v = [\![expr]\!]^{\mathcal{V}}(\epsilon,\sigma) \text{ in}\\ (\epsilon, \sigma[\langle l, \emptyset\rangle \mapsto v])$$

$$[\![\mathrm{id}]\!]^{\mathcal{A}} = \epsilon(\mathrm{id}) \qquad [\![lexpr^{\tau}.\mathrm{id}]\!]^{\mathcal{A}} = [\![lexpr]\!]^{\mathcal{A}}.{}_{\tau}\mathrm{id} \qquad [\![{*}expr]\!]^{\mathcal{A}} = [\![expr]\!]^{\mathcal{V}}$$

$$[\![lexpr^{\tau}]\!]^{\mathcal{V}} = \text{if } \texttt{is\_array}(\tau) \text{ then } \downarrow [\![lexpr]\!]^{\mathcal{A}} \text{ else } \sigma([\![lexpr]\!]^{\mathcal{A}}, \emptyset)$$
$$[\![\&lexpr]\!]^{\mathcal{V}} = [\![lexpr]\!]^{\mathcal{A}}$$
$$[\![expr\ \texttt{ptr}(\tau) + aexp]\!]^{\mathcal{V}} = [\![expr]\!]^{\mathcal{V}} +_{\tau} [\![aexp]\!]^{\mathcal{V}}$$

We refer to the numerical and symbolic variants of this semantics respectively by *ObjNum* and *ObjSym*.

Numerical operators. We have $\mathit{Offset} = \mathbb{Z}$ and $\emptyset = 0$. The address of the first cell of an array is confounded with the address of the array itself, so $\downarrow l = l$. Pointer arithmetics (in an array) follows positive or negative instrumentation edges. Similarly, the address of the first field of a structure is also the address of the structure. For the other fields, the semantics follows an instrumentation link.

$$l.{}_{\tau}f = \text{let } n = \texttt{offsetof}(\tau, f) \text{ in}\\ \text{if } n = 0 \text{ then } l \text{ else } \sigma(l, n)$$

$$l +_{\tau} k = \begin{cases} \sigma(l, \texttt{sizeof}(\tau)) +_{\tau} (k-1) & \text{when } k > 0\\ l & \text{when } k = 0\\ \sigma(l, -\texttt{sizeof}(\tau)) +_{\tau} (k+1) & \text{when } k < 0 \end{cases}$$

Symbolic operators. We have $\mathit{Offset} = \mathit{Field} \cup \mathbb{Z} \cup \{\varepsilon\}$ and $\emptyset = \varepsilon$. The address of the first cell of an array is found following the 0 instrumentation edge, so $\downarrow l = \sigma(l, 0)$. Pointer arithmetics (in an array) requires the ability to navigate backward on the instrumentation edges, using the function $\sigma_{\mathbb{N}}^{-1}$. Fields of a structure are all accessed through instrumentation edges.

$$l.f = \sigma(l, f)$$

$$l_{\text{cell}} + k = \text{let } \langle l_{\text{array}}, n\rangle = \sigma_{\mathbb{N}}^{-1}(l_{\text{cell}}) \text{ in}\\ \sigma(l_{\text{array}}, n + k)$$

$$\text{where } \sigma_{\mathbb{N}}^{-1}(l_c) = \langle l_a, n_c\rangle \text{ such that } \begin{cases} \langle l_a, n_c\rangle \mapsto l_c \in \sigma\\ n_c \in \mathbb{N} \end{cases}$$

Standard more expressive than object. Figure 6 contains a code[5] that is defined in standard semantics, but not in our instrumented object semantics. The

[5] The code shown relies on pointer comparison, but the problem we point to also appears in the absence of this feature (discussed in Section 6).

problem comes from the computation or storage of the out of bound expression `&t[N]`, which has no associated location. Note that this slight restriction could be removed at the cost of a more complex instrumentation.

```
int t[N]; int *p;
for(p=&t[0]; p<&t[N]; p++){...}
```

Fig. 6. Out-of-instrumentation.

6 Discussion

More pointer arithmetics. We can add pointer arithmetics operations to our language – pointer difference, pointer equality and pointer comparison – without calling into question what has been said before, *as long as* it does not involve different locations. (Remember that locations are not given numerical addresses.)

The standard numerical semantics will easily handle these new operations. So will the standard symbolic semantics, with the restriction to operands in the same array. In the instrumented object semantics, these features will require a traversal of the array instrumentation edges.

On the other hand, full pointer arithmetics requires associating physical addresses to locations.

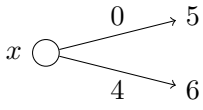
Unions. When we add union types to the set of types, we can consider two distinct semantics for them.

- *Layout-based.* Writing through a branch of a union invalidates the values which share bytes with the data written. These overlappings are architecture dependent and their precise resolution requires the offset information of the numerical memory model, as found in the ABI.
- *Path-based.* Writing through a branch of a union invalidates the values written through other branches. Branch information is natively kept by the symbolic memory model.

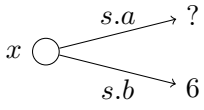
Fig. 7 illustrates these two semantics by showing that a numerical memory does not record the branch used for writing, and that a symbolic memory does not express the absence of conflict between $\langle x, r\rangle$ and $\langle x, s.a\rangle$.

```
union { int r;
  struct {int a; int b; } s;} x;
x.r = 5;
x.s.b = 6; // (1)
```

(a) C code.



(b) StdNum.



(c) StdSym.

Fig. 7. Union semantics in numerical and symbolic.

Casts. Union types can be used to perform casts, by writing through a branch and reading through another one. However the proposals sketched above do not allow this.

More generally, all the semantics we presented assume a static typing of expressions. Casts introduce a dynamic typing and are not handled by any of them. To handle casts, additional assumptions on the internal representation of types are needed (this could be provided by the ABI), and the memory model needs to keep the value of each byte, like in [7].

7 Conclusion

We classified the concrete store-based memory models using two criteria: the way they store pointers and the way they represent offsets. For each of the four memory models, we gave a compact semantics of a fragment of Clight, which includes arrays and pointer arithmetics. For this language, the usual semantics can be expressed with our standard memory model (Sect. 3). The object memory model, commonly considered in shape analyses, leads to strong semantic restrictions, that we overcome by instrumentation (Sects. 4 and 5).

Even if the semantics we presented covers most of our Clight fragment, we pinpoint minor differences which reflect strengths and weaknesses of the memory models (Figs. 4 and 6). Figure 8 depicts the ordering we obtain. Full and formal equivalences between semantics, by means of restriction and instrumentation are left for further work. We also discussed how the features we left aside (e.g. unions) would interact with the memory models (Sect. 6).

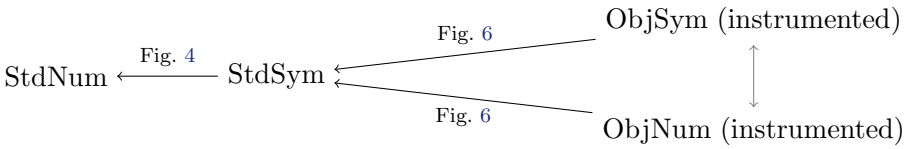


Fig. 8. Clight semantics expressiveness. Arrows go from less to more expressive models.

References

[1] Blazy, S. and X. Leroy, *Mechanized Semantics for the Clight Subset of the C Language*, J. Autom. Reasoning **43** (2009).

[2] Calcagno, C., D. Distefano, P. W. O'Hearn and H. Yang, *Beyond reachability: Shape abstraction in the presence of pointer arithmetic*, in: *Static Analysis Symposium, SAS'06*, LNCS **4134**, 2006.

[3] Calcagno, C., D. Distefano, P. W. O'Hearn and H. Yang, *Space invading systems code*, in: *Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR'08*, LNCS **5438**, 2008.

[4] Chang, B.-Y. E. and X. Rival, *Relational inductive shape analysis*, in: *Principles of Programming Languages, POPL'08* (2008).

[5] Kreiker, J., H. Seidl and V. Vojdani, *Shape Analysis of Low-Level C with Overlapping Structures*, in: *Verification, Model Checking, and Abstract Interpretation, VMCAI'10*, LNCS **5944**, 2010.

[6] Laviron, V., B.-Y. E. Chang and X. Rival, *Separating Shape Graphs*, in: *European Symposium on Programming, ESOP'10*, LNCS **6012**, 2010.

[7] Miné, A., *Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics*, in: *Languages, Compilers and Tools for Embedded Systems, LCTES'06*, 2006.

[8] Reynolds, J. C., *Separation Logic: A Logic for Shared Mutable Data Structures*, in: *Logic in Computer Science, LICS'02*, 2002, pp. 55–74.

[9] Sagiv, M., T. Reps and R. Wilhelm, *Parametric shape analysis via 3-valued logic*, ACM Transactions on Prog. Languages and Systems **24** (2002).