

A modular abstract interpreter for Julia

MPRI internship report

Oscar Blumberg, supervised by Prof. Alan Edelman, MIT

2015

General context

Julia[BEK14, 1] is a dynamically typed, JIT compiled, programming language implementation with an emphasis on generic programming through a powerful and ubiquitous symmetric, dynamic multiple dispatch mechanism. It achieves superior run-time performance by relying heavily on static analysis and aggressive code specialization. It is meant to provide a tool for scientific computations that is both familiar and fast enough for this demanding usage. The Julia compiler also provides a range of metaprogramming and code generation techniques to the advanced user.

Efforts into optimizing dynamic languages date back to Common Lisp tradition[BAP86], and more recently the sizable effort into making various JavaScript engines high performance[GAE09]. Julia has the interesting advantage of having been designed from the ground up to be a good analysis target and, while being dynamic, avoids pathological behavior patterns that are very hard to infer statically.

Studied problem

Most optimizations in the Julia compiler are done by the LLVM[Lat02] library in the backend. LLVM is a widely used, production-grade compiler and generally generates excellent machine code. However, its intermediate representation operates at an abstraction level comparable to that of C.

Some program optimizations would be better performed on the much higher level semantic that Julia code exhibits earlier in the compilation pipeline. The backend has to deal with additional complexity due to the integration between the generated code and the various run-time systems: dynamic memory allocation, garbage collection, dynamic dispatch call-sites, ...

At the Julia level, all those operations are implicit and the allowed behaviors are much more restricted: it has no interior pointers, no stack escape, memory operations cannot trap, type-punning is forbidden, the list goes on. It makes some analysis much easier, alias and shape analysis being good examples.

I like this list... maybe can expand on how this is also easier for abstract interpretation later in this report

Proposed contributions

As the main project of this internship, an extensible static analysis library for Julia was started. Dubbed the *Green Fairy*, it is written itself in Julia. The exposed interface is that of an abstract interpreter, providing an intuitive model for program behavior inference.

It uses a spatially sparse representation of the interpreter state enabling scalability across large programs. The main value domain is an extensible generic implementation of the reduced product construction, making it reasonably easy to extend it with new models.

It also features a work-in-progress interprocedural flow-sensitive shape analysis, based on static shape graphs[SMR98]. The shape analysis uses a sparse representation as well, and an optimistic but conservative model of aliasing information for the unknown parts of the heap that allows it to treat gracefully the usual problem of the cost of context sensitivity. It is able to track bounded allocation lifetimes across loop iterations precisely, opening up the possibility of interesting static garbage collection optimizations.

Arguments supporting their validity

The implementation is available as an ongoing work at [2]. It is already capable of applying both value-level (currently constant propagation and type inference) and shape analysis to a significant body of Julia code. It succeeds in demonstrating our ideas on a *non-toy* real world language, including its more exotic and idiosyncratic features. We hope to integrate our work into the main compiler in the coming year.

Future work

Besides tying up the various technical loose ends and improving the non-algorithmic bottlenecks of the implementation, we have several goals for the future of this project.

A consequent amount of technical work has to be done to make the later stages of the compiler modular enough that the friction in taking advantage of new information from the analysis becomes much lower. In particular, we plan to implement optimizations using the shape analysis to provide precise no-escape information that will allow memory reuse and stack allocation, greatly reducing the load on the garbage collector and the dynamic allocator.

In the longer term, the natural extension to this work is to devise an intuitive interface that makes it possible for users to extend the abstract state of the interpreter with domain-specific informations. We hypothesize that there is a large class of optimizations that are hard-to-impossible to teach a compiler using only knowledge of the language's semantic, but are fairly easy to formulate once more information is given about specific data structures and idioms in use.

What optimizations do you think belong in this class?

Those points are developed further along in the report.

Wordy

Julia

Julia has a dynamically-typed semantic. The execution model is that of a typical imperative language : impure and call by value. It is entirely built around the concept of generic functions.

Type system dedede

Generic functions A function in Julia is in fact a collection of methods, partially ordered by their signature types as a sub-lattice of the type lattice[BEK14]. The dispatch mechanism is dynamic – albeit statically inferred in many cases – and is symmetric in all the arguments: the most specific method will be called at run-time depending on the argument types. It is ensured that for any argument type there exists a well defined least upper bound in the sub-lattice. More details about Julia’s generic functions can be found in [BEK14, Bez15].

Memory model At the specification level, Julia objects are tagged memory cells living in an infinite heap. Each has a finite set of numbered mutable fields, possibly pointing to other objects. The language relies on automatic memory management to fit in a physical finite heap. The current implementation uses a tracing garbage collector.

Compiler The main analysis performed on Julia code by the compiler is an interprocedural flow-sensitive type inference. It is followed by an inlining pass and several ad-hoc peephole optimization. The last step is to generate LLVM intermediate representation while using the inferred types to remove as much of the dynamic semantic as possible. This project addresses the limitation that the only static information computed on high level Julia code today is in the type lattice.

Say more

Green Fairy

The Green Fairy provides a generic framework that computes the fixed point of user-provided abstractions over a Julia program. The library handles the control flow semantics and presents itself as an extensible interpreter : users enrich the abstract machine’s state with custom abstractions.

We provide a set of generic basic blocks to construct those state extensions. The most common one is a non-relational variable analysis over any value domain. It can be used jointly with the supplied parametric reduced product[CCM11] implementation, allowing for the implementation of cooperative behavior between different domains. We will start by giving a short and simple example of its usage.

Interval example

Follows an implementation of the classical example of abstract domain: the (closed) numeric interval.

```

immutable Interval{T} <: Lattice
    lo :: T
    hi :: T
end

bot{T}(::Type{Interval{T}}) = Interval(typemax(T),typemin(T))
top{T}(::Type{Interval{T}}) = Interval(typemin(T),typemax(T))
isbot(x::Interval) = x.hi < x.lo

<=(x::Interval,y::Interval) = isbot(x) || y.lo <= x.lo && x.hi <= y.hi
join{T}(x::Interval{T},y::Interval{T}) = Interval(min(x.lo,y.lo), max(x.hi,y.hi))

function meet_ext{T}(x::Interval{T}, c::Const)
    !istop(c) && isa(c.v,T) || return x
    x.lo <= c.v <= x.hi && return Interval(c.v, c.v)
    bot(Intervall{T})
end

```

This domain is parametric over any ordered numeric types having an upper and lower bound. Defining a method for `isbot` is necessary since there are multiple representation of \perp .

The `meet_ext` function is the product domain reduction function. In that case we express that the knowledge that a value is constant can reduce an interval to a single point or \perp depending on the particular value. We are free to add other method definitions to this function to increase precision using information from other domains than `Const`. This modelization of the reduced product relies on multiple dispatch to easily provide extension points : the user can seamlessly provide reduction functions for any pair of domains, possibly defined by someone else.

Describing transfer functions is also straightforward, as demonstrated by this simple example implementing the transfer for the addition on the interval lattice:

```

function eval_call{T,V}(::Type{Interval{T}}, f::V, args::Vector{V})
    if f <= Const(Base.(+))
        intervals = map(v -> convert{Interval{T}}, v), args)
        lbs, ub = map(i -> i.lo, intervals), map(i -> i.hi, intervals)
        return Interval(sum(lbs), sum(ubs))
    end
    top(Intervall{T})
end

```

We can see that the arguments can be provided in different domain than that of intervals, requiring a `convert` call. This gives access to the full product to a transfer function, making it possible to use information from other domains in the computation.

Due to space constraints, we relegate to the appendix another example that demonstrates the construction of a *finite disjunction* domain that is parametric over any other domain and can thus serve as a composable building block. It can be used to enhance precision while keeping the implementation simple and decoupled. Those two examples illustrate the benefits of the pervasive use of ad-hoc polymorphism and multiple dispatch in Julia: it allows for the construction of reusable abstractions.

Several important opt-in features for value domains and their transfer functions were not described here for the sake of brevity. This includes fused join-comparisons, widening, participation in the dispatch machinery, the ability to control the propagation of values over function call boundaries, as well as state-altering actions such as exception raising or heap operations.

Insisit more on modularity, maybe make a new section

IR

Our interpreter operates on a low-level dialect of Julia source code. We will give here a simplified description of this internal representation.

A function is a control graph composed of extended basic blocks (EBB). Their defining property is to have a single control flow entry point as their head. Each one contains a linear succession of branches and variable assignments to literals and function calls.

Our implementation deals with most of the current Julia semantic. We will leave out some of those features because while they require additional work, they are not directly relevant to this report. This includes

- Exception handling. Even though special care was taken to handle it with complete precision (assuming of course precise throw-site information).
- Closure capture.
- Function calls with variable number of arguments, or “splatting”.

where put this

Generic functions have interesting repercussions on analysis strategies. Since virtually every function call is indirect, not knowing the argument types can lead to a combinatorial blowup of code to explore in an interprocedural analysis, leading to widening and more precision loss. It is therefore often worth to trade performance for precision as long as it can keep types fully known[Bez15].

Analysis sparsity

The Green Fairy is oriented towards building flow-sensitive analysis, that is, it provides information dependent on the program point. The advantage of flow sensitivity is in the precision gain, although it comes with a performance price compared to the more approximative flow-insensitive analysis. There are different techniques to lower the cost of flow sensitivity.

A simple way to implement a flow-sensitive computation is to store a dense mapping of program points to interpreter state. It has the benefit of being oblivious to the structure of said state. However, as the state or the analyzed program gets larger, there is a growing cost associated to the copying and comparison

of those state atoms. In the vast majority of cases, the states at two nearby program points are highly redundant and taking advantage of this redundancy can lead to consequent improvements in analysis performance. Exploiting this so-called *spatial* sparsity of the state space is an important design point of our implementation.

There is another related kind of sparsity, that is sometimes called *temporal*, where the analyzer benefits from knowledge of state dependencies to avoid recomputing transfer operators that only use unchanged subset of the state. It allows the interpreter to make results from computations flow directly into use sites. In the current implementation we do not use temporal sparsity, mainly due to a combination of time constraints and soundness concerns, although care was taken to ensure the feasibility of this optimization in the future. In the literature, the word *sparse* without qualification usually refers to temporal sparsity but we will mostly discuss spatial sparsity here.

Non-relational

We implemented a sparse representation of non-relational state that accommodates any user-defined domain for forward analysis in a transparent manner. This representation *does not* require use/def sites to be identified ahead of time, unlike the Static Single Assignment (SSA) transform or other techniques relying on pre-analysis[OHL12]. Furthermore, we present an extension of this idea to a relational domain, namely, the static shape graph heap model as introduced in [SMR98].

Static single assignment form The classical example of sparsity is the SSA transform. If a program is in SSA form, any forward variable analysis can easily be both spatially and temporally sparse. Taking advantage of the SSA transform is a two phase process. A pre-analysis identifies dataflow dependencies of variable uses over definitions. Once done, those dependencies are made explicit in the source code by renaming variables and inserting explicit join operations (φ instructions). It enables the use of a flow-insensitive analysis while keeping the precision of a flow-sensitive one.

Dominator tree An important notion in the SSA transform is that of domination: we say that a program point a *dominates* another point b if any path in the control flow graph from the root to b has to go through a . The domination relation is a partial order on program points and is moreover a tree (every initial segment is well ordered). It is usually called the dominator tree. In our case, since every path to a program point has to go through the head of its containing EBB – and all the points between the head and itself – we will say that a program point dominates an EBB if it dominates its head. Given an EBB, the program point being its parent in the dominator tree is called the *immediate dominator*.

The dominance frontier of a program point a , noted $DF(a)$, is the set of EBB heads that, while not being dominated by a , have predecessors that are. $DF(a)$ is the set of nodes that one has to pass through in order to, starting from a , leave its domination. Following the usual convention, we note $IDF(a) = \text{lfp}_a DF$ the iterated dominance frontier of a .

If a variable v has two definitions at a and b respectively, the SSA form disambiguates between them by inserting φ nodes in a subset of $\text{IDF}(a) \cap \text{IDF}(b)$. This set is an upper bound, because some of those φ may end up without any use. Computing the minimal φ placement is harder and requires a form of liveness analysis. The advantage of the iterated dominance frontier is that $\text{IDF}(a)$ only depends on a and can be cached independently of the variable considered.

Sparse map We take a slightly different approach from the usual SSA transform: we compute the same information but *on the fly*. We assume that use sites of variables cannot be statically determined in advance without computing transfer functions. It allows new use and definition sites to be discovered at an already processed program point in later iterations. The main reason for this choice is to allow for other quantities than variable values to profit from the sparse infrastructure. [examples]

Essentially, we are providing a generic sparse map implementation from an arbitrary set of names to any value domain. It can be used to represent any non-relational internal interpreter state of the user's choice.

It is hence inefficient in our case to maintain the renaming property. Instead, when encountering an use, the interpreter walks the dominator tree upward to find the nearest definition. This lookup is done following the tree spine, giving it an $O(\text{dominator tree depth})$ worst case complexity. In practice, the dominator tree is shallow and wide : in the Julia standard library (around 4MB of sources) the deepest dominator tree is of depth [?].

As for the implementation, we do not materialize φ instructions: they are kept out of band as a single compound summary information for each EBB head. They can be efficiently dynamically updated when required by the discovery of new definitions. We compute the dominator tree and frontiers using the algorithm described in [CDH01].

We have not yet witnessed this process to be a bottleneck. Would it become the case, there are several ideas that could be implemented to lower its cost. The most obvious one being use-forwarding. As an aside, the choice of EBBs for the program representation offsets the time spent doing this lookup in practice by making common dominator trees shallower. It is not a worst-case improvement but a welcomed average constant factor gain.

Relational

This structure cannot be readily used for relational domains, however we can extract an important insight from the non-relational case. We can think of the φ node living at an EBB's head as a summary of the state difference on any path between its immediate dominator and itself. It is less apparent in the case of a non-relational domain, since this difference is usually represented as an element of the domain itself, i.e., a small name-value mapping that only refers to changed names.

[good example + explanation of a generic translation] [trash a bit [OHL12] for saying relational in their title and only handling packed domains]

Shape analysis

[todo some platitudes about shape analysis]

Local variables

For a moment, we will ignore object fields and summary objects to concentrate on variable aliasing and its sparse structure. We will introduce those back later. One of the fundamental insight of [SMR98] is in the scheme used to describe abstract heap locations. The abstract address of an object is given by the exact set of stack variables that refers to it. It means that we can represent the abstract heap as a collection of set of variable names. The main advantage of this approach compared to the other usual choices, such as allocation site segregation, is that transfer functions operating on objects referenced by variables can *always* perform strong updates. It is due to the fact that each one of those abstract addresses can refer to at most a single object in any given concrete heap. Another very important point about this choice is that it provides a canonical way to compare two abstract heaps : there is no ambiguity on the mapping between the nodes, avoiding any kind of graph isomorphism problems.

In this framework there is no explicit representation of the may/must alias sets. We can construct them in the following way :

$$\begin{aligned} p(H, x) &= \{o \in H \mid x \in o\} \\ \text{may-alias}(H, x) &= \{y \mid p(y) \cap p(x) \neq \emptyset\} \\ \text{must-alias}(H, x) &= \{y \mid p(x) \subseteq p(y)\} \end{aligned}$$

We refer to [SMR98] for the formal deduction rules and computation recipes in this abstract domain. We will focus here on exposing a way to implement a similar state domain with a sparse representation. The main inspiration comes from [NL09]: they develop a sparse alias analysis for local variables that we use as a basis for our shape domain. The authors make the following observation.

Assuming that the analyzed code is in SSA form, we can order the set of program variable under the domination relation of their definition. That being done, it is clear that when transferring the state over a non-dominating edge, we can trim all heap objects of variables whose definition stopped dominating the current control point. In fact, we can always remove any dead variables from our abstract heap, and non-dominating definitions are always dead thanks to the fundamental property of SSA.

To be slightly more precise, here is a sketch of the transfer function for variable assignment and φ

nodes with two predecessors.

$$\begin{aligned}
T[x = y](o) &= \begin{cases} o \cup \{x\} & \text{if } y \in o \\ o & \text{otherwise} \end{cases} & T[x = y](H) &= \left\{ T[x = y](o) \mid o \in H \right\} \\
T[\tilde{x} = \tilde{y}] &= T[x_1 = y_1] \circ \dots \circ T[x_n = y_n] \\
T[\varphi(\tilde{x} = \tilde{y}, \tilde{z})](H_l, H_r) &= (T[\tilde{x} = \tilde{y}](H_l) \cup T[\tilde{x} = \tilde{z}](H_r)) \cap 2^{\text{dom}(\tilde{x})}
\end{aligned}$$

where $\text{dom}(\tilde{x})$ is the set of variable having their definition in a dominating position over the φ node, including itself. It is important to notice that we don't need to remove x from any object when assigning to x since the definition was non-dominating before applying T .

As we alluded to in the previous section, one can look at the restriction to dominating definitions as a *summarization* of the state between the φ 's EBB head and its immediate dominator.

From this formulation, we can deduce a sparse format for this simplified shape graph : there is a consequent amount of sharing between subsequent states. In fact, we can see from the transfer functions that they only need to manipulate the “most recently” defined variables in each object. In [NL09], this sharing is exploited by representing heap objects as sorted singly-linked list with hash-consing. The elimination of redundancy is thus implicitly handled by the data structure.

It has the benefit of keeping the implementation close to the mathematical structure, but forces the algorithm to keep the linked list heads of *all* live heap objects at all control points, defeating some of the sparsity. We take an alternate route of representing the tree explicitly with pointers in both directions. At the cost of additional lookups along its spine – much like in the non-relational case – we can cut down the storage at each control point to the set of modified objects. Doing this, we also avoid the need for hash-consing.

Alias tree Our encoding of variable aliasing takes the form of a forest of doubly-linked trees. Each node is labeled by a variable definition and several nodes can have the same label. Every tree respects the dominance ordering of variable definitions. An initial segment $x_1 \dots x_n$ of this forest represents the heap object $\{x_1, \dots, x_n\}$ at the control point where the definition x_n is.

Given a program point a and a variable definition x dominating a , we can characterize all heap objects x points to when the interpreter is at a . They are initial segments of the forest, they contain x , all their elements dominate a and they are maximal for those properties.

We can see from this description that additional work is needed to materialize the point-to set of a definition x at a control point further away from the definition point.

Algorithm 1 High level description of some transfer functions for sparse variable alias analysis

```
function MATERIALIZE( $H, a, x$ )
   $d = \text{FINDDEF}(x, H, a)$ 
   $R = \{\text{nodes of } H \text{ labeled by } d\}$ 
  while  $R$  changed
    for  $o \in R$ 
       $C = \text{children}(o \text{ in } H) \cap \text{dom}(a)$ 
      if  $C \neq \emptyset$ 
         $R = (R - \{o\}) \cup C$ 
      end
    end
  end
  return  $R$ 
end
```

```
function TRANSFER[ $x = y$ ]( $H, a$ )
   $R = \text{MATERIALIZE}(H, a, y)$ 
  for  $r \in R$ 
     $H, o = \text{ADDNODE}(H, \text{def}[a : x])$ 
     $H = \text{ADDEDGE}(H, o \rightarrow r)$ 
  end
  return  $H$ 
end
```

```
function RESTRICT( $H, \tilde{y}, n, a, b$ )
  for  $i \in [1, n]$ 
     $R_i = \text{MATERIALIZE}(H, a, y_i)$ 
  end
  while  $R$  changed
    for  $i \in [1, n]$ 
       $U = R_i \cap (\text{dom}(b) \cup \text{roots}(H))$ 
       $V = \{\text{parent}(d) \mid d \in R_i - U\}$ 
       $R_i = U \cup V$ 
    end
  end
  return  $R$ 
end
```

```
function TRANSFER[ $\varphi(\tilde{x} = \tilde{y}, \tilde{z})$ ]( $H_{\text{idom}}, a, H_l, a_l, H_r, a_r, n$ )
   $H = H_{\text{idom}}$ 
   $R^l = \text{RESTRICT}(H_l, \tilde{y}, n, a_l, \text{idom}(a))$ 
   $R^r = \text{RESTRICT}(H_r, \tilde{z}, n, a_r, \text{idom}(a))$ 
  for  $d \in \cup_i R_i^l \cup R_i^r$ 
     $V = \{i \in [1, n] \mid d \in R_i^r \cup R_i^l\}$ 
     $H, n = \text{ADDNODE}(H, \varphi[a : \tilde{x}_V])$ 
    if  $d \in \text{dom}(\text{idom}(a))$ 
       $H = \text{ADDEDGE}(H, d \rightarrow n)$ 
    end
  end
  return  $H$ 
end
```

Generation liveness

A traditional escape analysis is not capable of proving memory reuse opportunities in a loop with cross iteration dependencies. For example, in the following loop, every object allocated in the loop body survives for exactly 3 iterations.

```
while ?
  use(x, y, z)
  z = y
  y = x
  x = new()
end
```

We can take advantage of this information to allocate 3 memory cells and reuse them in a round robin fashion, instead of relying on the garbage collector at runtime.

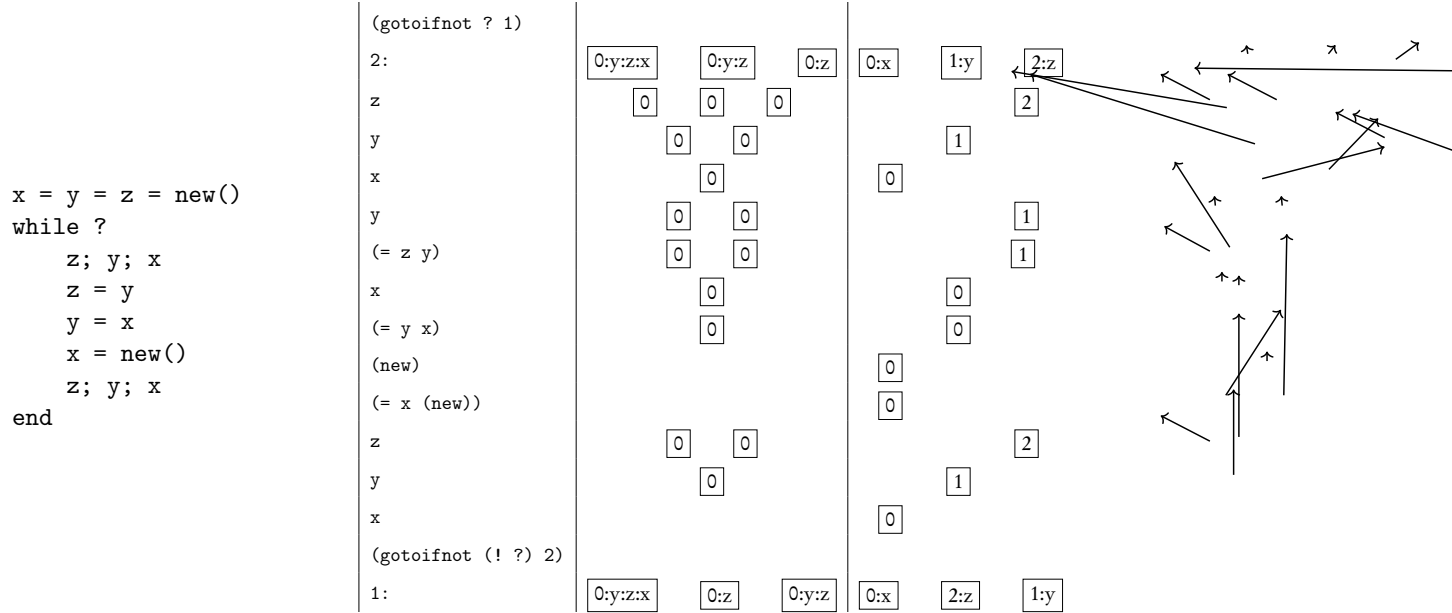


Figure 1: Example of alias trees with generation counting

On the leftmost column is a simple loop allocating one new object per iteration and keeping it alive for 3 consecutive iterations. It is followed by the lowered representation of the body of the loop, lightly edited for clarity. We expanded the 3 variables two times to make the aliasing and the generation increases more evident.

On the right are the two alias trees corresponding to the two allocation sites. They are automatically generated from our analysis results. The numbers labeling each node are generation numbers. Nodes with explicit variable names are part of an EBB’s head summary (our “ φ ” nodes).

To compute this information, we add to each heap object a “generation number” that represents the maximum number of younger and live objects from the same allocation site. This counter lives in the lattice $\{0, \leq 1, \leq 2, \dots, \leq \infty\}$ with widening to avoid infinite increasing chains. When an alias tree edge crosses its own allocation site, this counter is increased.

In order to preserve the property that the heap can always be materialized by looking up the dominator tree, we also compute a generation delta between an EBB’s head and its immediate dominator for each allocation site. We apply this delta when we restrict alias edges across dominating control edges.

Object fields

The full abstract heap model with object field is a graph where nodes are labelled with sets of variable definitions and edges with field indices. We will describe most of the abstract semantic assuming a single field name for simplicity, but the extension to a finite number field names is straightforward. Adding object fields also implies the presence of summary nodes. They are necessary to avoid infinite increasing chain in the lattice of shape graphs. A summary node is a node that no local variable points to, and may represent several concrete objects. There are a lot of different techniques and heuristics on when to collapse empty

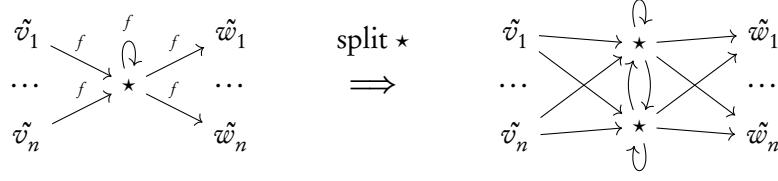


Figure 2: Summary node splitting

nodes into summary nodes. To preserve a fast implementation, we enforce the property that there can be no edge between two summary nodes: we always collapse them together. This limits the language of heap path we can describe outside our local stack frame to regular expressions of the form $(f_1 + \dots + f_n)^*$ where the f_i are field names.

We can define the ordering on this lattice in the following way. We will say that $H \leq H'$ iff H can be obtained from H' by a succession of moves among those possibilities:

- Remove a node and all its inbound and outbound edges
- Remove an edge
- Split a summary node into two summary nodes and duplicate all edges accordingly
- Demote a summary node to a normal node

Most transfer operators on the abstract heap are straightforward. Variable assignments do not change from the simpler model exposed in the last section. Field assignments are always strong except when the source is a summary node.

We can improve on the naive field access transfer by taking into account incompatible objects. Assuming no summary nodes, we can describe field access by

$$\begin{aligned}
 E(x, f) &= \{x\tilde{u} \rightarrow \tilde{v} \in H_{\text{edges}}\} & N(x, f) &= \{\text{dest}(e) \mid e \in E(x, f)\} \\
 P(y, x, f, \tilde{u}) &= \{\tilde{v} \in N(x, f) \mid \tilde{v} \cap \tilde{u} = \emptyset\} \\
 T[y = x.f](H)_{\text{nodes}} &= (H_{\text{nodes}} - N(x, f)) \cup \bigcup_{\tilde{v} \in N(x, f)} P(y, x, f, \tilde{v}) \cup \{\tilde{v}y\} \\
 T[y = x.f](H)_{\text{edges}} &= (H_{\text{edges}} - E(x, f)) \cup \{\text{src}(e) \rightarrow \text{dest}(e)y \mid e \in E(x, f)\}
 \end{aligned}$$

Implementation Object fields are additional edges in the alias tree, labelled with a field index. Given an initial segment $x_1 \dots x_n$ of the forest, we define the possible values of its k th field to be the set of destination of k -edges originating at x_p where p is the largest index such that x_p has at least one k -edge. Given this description, translating the abstract transfer into an implementation over the forest is straightforward.

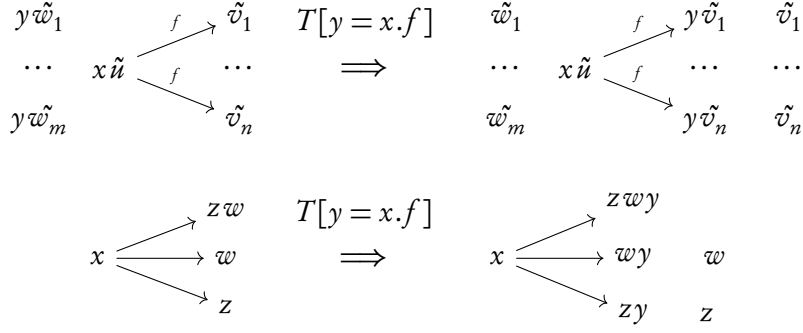


Figure 3: Transfer function for field access

The first graph shows the general case. The second is an example demonstrating incompatible nodes.

When summarizing the alias tree, we collect fields set during the summarized segment of code and copy those in the EBB head. This is also the only operation that can introduce new empty nodes. We promote them into summary nodes and merge connected ones.

Context sensitivity

All the preceding discussion only describes an intraprocedural setting. Making a heap analysis scale well interprocedurally is challenging. The most straightforward way of introducing context sensitivity is through *heap cloning*. Although it has been demonstrated that heap cloning can scale very well[LLA07], this result is only in the context of a flow-insensitive analysis.

We make the choice not to introduce any context sensitive information while computing the fixpoint of the abstract heap over a function. In the setting we described until now, we would have to treat function arguments – and objects reachable from them – conservatively as a single summary node. We instead introduce a new type of node that tries to keep more information by being less pessimistic about aliasing. Of course, inside the callee, we still can’t answer the question whether those outside objects alias or not. However, by keeping them separated we allow the caller to regain some information by eliminating impossible nodes and perform a form of “post-facto context sensitivity” using call site information.

This new type of node, that we will call *context* nodes, do not have the guarantee not to alias each other. Unlike summary nodes however, each is still mapped to at most a single concrete object. We will note context nodes by prefixing their content with γ . We include context nodes in the abstract semantic by adding the following rule to the set of $H \leq H'$ moves previously described.

One can contract (κ) two context nodes. The label of the resulting node is the union of the two labels. Edges from and to the nodes must be duplicated for every element of the cartesian product of the source and destination nodes respectively. More precisely, still assuming a single field name for clarity, the contraction of two γ nodes can be expressed as

$$* \rightarrow \gamma \tilde{u} = \left\{ e \in H_{\text{edges}} \mid \text{dest}(e) = \gamma \tilde{u} \right\} \quad \gamma \tilde{u} \rightarrow * = \left\{ e \in H_{\text{edges}} \mid \text{src}(e) = \gamma \tilde{u} \right\}$$

$$\begin{aligned}
\kappa[\gamma\tilde{u}, \gamma\tilde{v}](H)_{\text{nodes}} = & (H_{\text{nodes}} - \{\gamma\tilde{u}, \gamma\tilde{v}\}) \cup \{\gamma\tilde{u}\tilde{v}\} \\
& - (\text{src}(* \rightarrow \gamma\tilde{u}) \cup \text{src}(* \rightarrow \gamma\tilde{v}) \cup \text{dest}(\gamma\tilde{u} \rightarrow *) \cup \text{dest}(\gamma\tilde{v} \rightarrow *)) \\
& \cup \{\text{src}(e) \text{src}(e') \mid e \in * \rightarrow \gamma\tilde{u}, e' \in * \rightarrow \gamma\tilde{v}\} \\
& \cup \{\text{dest}(e) \text{dest}(e') \mid e \in \gamma\tilde{u} \rightarrow *, e' \in \gamma\tilde{v} \rightarrow *\} \\
\kappa[\gamma\tilde{u}, \gamma\tilde{v}](H)_{\text{edges}} = & H_{\text{edges}} - (* \rightarrow \gamma\tilde{u} \cup * \rightarrow \gamma\tilde{v} \cup \gamma\tilde{u} \rightarrow * \cup \gamma\tilde{v} \rightarrow *) \\
& \cup \{\text{src}(e) \text{src}(e') \rightarrow \gamma\tilde{u}\tilde{v} \mid e \in * \rightarrow \gamma\tilde{u}, e' \in * \rightarrow \gamma\tilde{v}\} \\
& \cup \{\gamma\tilde{u}\tilde{v} \rightarrow \text{dest}(e) \text{dest}(e') \mid e \in \gamma\tilde{u} \rightarrow *, e' \in \gamma\tilde{v} \rightarrow *\}
\end{aligned}$$

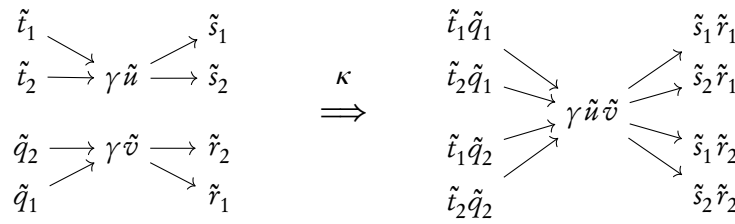


Figure 4: Contraction of two context nodes

An interesting consequence of this choice of semantic is that we can still perform strong field updates on γ nodes. They are weaker than local nodes in the two following ways:

- Other part of the interpreter cannot assume that two different γ nodes don't alias.
- The field access of a γ node is weaker.

The only change we need to make to $T[y = x.f]$ when x is part of a γ node is that we have to keep edges between $\gamma x \dots$ and the dangling nodes \tilde{v}_i introduced.

Checking

I can see the future

[summary of contributions] [technical/impl things] [backward analysis?]

Generic sparsification We hypothesize that a general transform from dense to sparse state structure underlies the particular one presented in this report and included in our implementation. Although we do not hope for a fully mechanized way to do such a conversion, it is likely that we could provide a set of generic reusable component to make the implementation as well as the soundness proof of those state domains easier. As more use cases appear we will try to develop a general framework, both theoretical and implementation-wise, for this process.

User provided optimizations We briefly hinted at domain-specific optimizations in the introduction. Our long term plan is to provide hooks in the compiler and the analyzer to allow for user defined analysis and optimization. Moreover, those should be able to easily build upon the existing infrastructure. Although the abstract interpreter model is a good fit in that regard, we have not yet found a fully satisfactory equivalent for the optimization side.

As a good example of the envisioned usefulness of this kind of modularity, consider the development of a high-level interface to a BLAS library. This was not chosen at random since the linear algebra routines are in fact a sizable chunk of the Julia standard library, and BLAS is used as their back end in the common case floating point numeric types.

It is easy to use under a simple model where the output of the computation is freshly allocated memory containing the result. This avoids aliasing bugs, however it is inefficient because it does not take advantage of the specialized BLAS functions that can output the result in place. Reasoning that the two are equivalent if one of the argument (and all its aliases) are dead after the BLAS call is almost impossible to the compiler because it requires understanding of the routine, of which the core is often hand written assembly. However, the rules of BLAS aliasing are very regular at a high level and someone writing a wrapper for the library could easily, given the right interface, integrate those rules directly as a compiler extension.

As we work towards making the interpreter more accessible, the need for good tooling becomes unavoidable.

Abstract debugging We would like to provide a common infrastructure aimed at debugging abstract domains. In its simplest form, this system would open the possibility for a domain to provide small run-time guards checking that the state conforms to what was inferred. The analyzer would then, in “verification” mode, insert those guards in the code before running it, along with the necessary information to trace back the exact transfer function that computed an unsound result. This would obviously not replace a formal proof of soundness but would help weed out implementation errors by large scale testing on the consequent amount of freely available Julia source.

Visualization As a way to help implementation work, a graphical visualization of the interpreter was hastily put together. The usefulness of this tool exceeded our – admittedly low – expectations. [maybe screenshot in annex] We believe that interactive feedback is a key aspect of allowing users with no particular background in program analysis to develop an understanding of the – in fact quite natural – underlying model.

References

- [Lat02] Chris Lattner – *LLVM: An Infrastructure for Multi-Stage Optimization* (M.S. Thesis, University of Illinois, 2002)
- [GAE09] Gal, Andreas, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan et al – *Trace-based just-in-time type specialization for dynamic languages* (ACM 2009)
- [BEK14] Jeff Bezanson, Alan Edelman, Stefan Karpinski, Viral B. Shah – *Julia: A fresh approach to numerical computing* (2014) arXiv:1411.1607
- [BAP86] Brooks, Rodney A., David B. Posner, James L. McDonald, Jon L. White, Eric Benson, and Richard P. Gabriel – *Design of an optimizing, dynamically retargetable compiler for common Lisp* (ACM 1986)
- [SMR98] Sagiv, Mooly, Thomas Reps, Reinhard Wilhelm – *Solving shape-analysis problems in languages with destructive updating* (TOPLAS 1998)
- [CC77] Patrick Cousot, Radhia Cousot – *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints* (SIGPLAN 1977)
- [CCM11] Patrick Cousot, Radhia Cousot, Laurent Mauborgne – *The reduced product of abstract domains and the combination of decision procedures* (Foundations of Software Science and Computational Structures, 2011)
- [Bez15] Jeff Bezanson – *Abstraction in Technical Computing* (2015)
- [LoA15] “Lots of authors” – *Static Single Assignment Book* (2015)
- [NL09] N. A. Naeem, O. Lhotak – *Efficient alias set analysis using SSA form* (ISMM 2009)
- [OHL12] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, Kwangkeun Yi – *Design and Implementation of Sparse Global Analyses for C-like Languages* (PLDI 2012)
- [CDH01] Cooper, Keith D., Timothy J. Harvey, Ken Kennedy – *A simple, fast dominance algorithm* (2001)
- [LLA07] Chris Lattner, Andrew Lenharth, Vikram Adve – *Making context-sensitive points-to analysis with heap cloning practical for the real world* (SIGPLAN 2007)
- [1] <http://julialang.org>
- [2] <https://github.com/carnaval/green-fairy>