Safe Low-Level Languages
Justin Slepak

1 Cyclone: Regions and affine types for memory management

1.1 Review: Tofte and Talpin Regions

Two hazards commonly associated with manual memory management are allocating a block of memory without ever freeing it (memory leak) and attempting to access a block of memory which has already been freed (dangling pointer access).

Tofte and Talpin described a "region type" system, in which data in the store is associated with a named region. The expression (e `at` ρ) allocates store space for the result of e in the region ρ. (`letregion` ρ `in` e) allocates a new region named ρ, which remains in scope in e. Once the body of the `letregion` expression is evaluated, all store objects in the region it created are freed. This means regions are managed in stack-based manner. A region allocated for the entire program can be treated as a heap, but data remains in this heap until program termination as the only way to deallocate a store entry is for its region to go out of scope. Including a tag identifying the regions a function may access in the function's type allows checking that function calls do not access regions which have already been freed.

Cyclone offers this region-based memory management in a C-like language. Every pointer type includes a region annotation. Each function automatically introduce a new region, which is freed when the function returns. The programmer can also wrap a block of code (within a function) with a region declaration, giving slightly finer control over memory allocation. There is also a "heap" region which is always in scope and is garbage-collected. "Region polymorphism" allows variables in a the region tag of a function or struct. For example `strcpy` can be used on arguments from arbitrary regions, and those regions are the ones `strcpy` will access: `char?`ρ `strcpy<`ρ`, `ρ_2`>(char?`ρ `d, const char?`ρ_2 `s)`.

1.2 Extending Tofte and Talpin

The lexical regions described above impede certain programming idioms. For example, the programmer must specify at the time of allocation when a block of memory will be deallocated, which prevents resources from being released early if the program determines (at runtime) that they are not needed. A function also cannot free memory allocated by its caller. This makes it impossible to write a tail-recursive function which deallocates memory before making the tail call. Any looping construct that preserves state from one iteration to the next is guaranteed to leak memory.

Cyclone includes additional capabilities to work around the limited expressiveness of Tofte and Talpin's memory management. The "dynamic region" construct behaves like a memory region except that the compiler inserts dynamic checks for availability of the region when it is accessed (many of Cyclone's safety features revolve around inserting dynamic checks which the programmer can avoid by writing code more amenable to static analysis). This requires a runtime structure which tracks what dynamic regions are available for use.

If the overhead associated with region management cannot be amortized over a large enough number of allocated objects, unique pointers can be used instead. These pointers resemble affine references in λ^{URAL}. Creating an alias to a unique pointer is prevented: copying a unique pointer consumes the original copy. Passing the pointer to the function is considered to consume the pointer, with the function expected to either deallocate it or return it back to the caller (possibly inside some structure). A unique pointer may be wrapped in a shared object, in which case the unique pointer can only be accessed via a swap operation. Because there are no aliases to a unique pointer, it is easy to check that it is never accessed after it is freed. The extension includes a construct which allows a temporary alias of a unique pointer inside a code block. Within the scope of that block, the alias can be used as if it were a non-unique pointer. Associating a freshly-named lexical region with the block ensures that the alias itself cannot escape.

Cyclone also provides reference-counted pointers, which are treated like unique pointers except that they can be aliased more freely. A special aliasing operation increments the count associated with the pointer, and a "drop" operation decreases the count. The analysis which checks pointer safety looks to see whether it is possible to reach a join point point in the program and have multiple different counts associated with a reference-counted pointer.

2 Vault: Linear types for protocol specification

2.1 Type guards

Vault uses a notion of "type guards," which are used to describe conditions on how data of that type must be used. Such a type is specified as `tracked(`R`)` T, where T is the underlying type, and R is the guard, expressed as a set of "keys." If a value is of a guarded type, accessing it requires that the keys named in the guard all be held at that point in the program. A global state called the "held-key set" tracks what keys are true at each point in the program. It is also possible to assign symbolic values to a key, and a type guard may specify what value must be assigned to a key. Thus a key can be used to track both availability of a resource and its state. Each particular guarded type is considered unique, e.g. every `tracked(logfile) FileHandle` must be an alias of the one `FileHandle` guarded by the `logfile` key. However multiple different underlying types may be used with the same guard. Statically associating each key with a runtime object requires some restriction on aliasing. Function calls which affect a tracked object must note those changes via the guard key.

Function types include an effect tag specifying pre- and post-conditions in terms of guard keys. The possible effects on a key are:

- Acquisition: `[+`K`@`a`]` specifies that key K must be not held before the call and will be held in state a when the function returns.
- Release: `[-`K`@`a`]` specifies that key K must be held in state a before the call and will not be held when the function returns.
- State change: `[`K`@`a`->`b`]` specifies that key K must be held in state a before the call and will be held in state b when the function returns.
- Creation: `[new `K`@`a`]` specifies that a fresh key K will be held in state a when the function returns.
- Preservation: `[`K`@`a`]` specifies that a fresh key K will be held in state a both before the call and when the function returns. This is effectively an abbreviation of `[`K`@`a`->`a`]`. The key state can also be ignored with `[`K`]`.

Guarded types can be parameterized over keys, allowing constructs like `type guarded_int<key K> = K:int;`. Then a function can be specified with the signature `void foo(guarded_int<F> x) [F];`, indicating that it takes a guarded integer as a parameter, and its effect is a preserving use of the guarded integer's key.

Guarded types can be used to describe a memory region construct:

```
interface Region {
  type region;
  tracked(R) region create() [new R];
  void delete(tracked(R) region) [-R];
}
```

The only way to gain access to a region is via `create`, which generates a fresh key. An object can be associated with that region by having it use the key generated from the `create` call. Delete frees the region by removing the region's key from the held-key set.

```
tracked(R) region rgn = Region.create();
R:point pt = new(rgn) point {x=1; y=2;};
...
Region.delete(rgn);
```

The first line creates a region (n.b. the resulting key is never bound and can only be accessed via `rgn`). The second line uses a tagged `new` to allocate a `point` object with the same guard as `rgn`. Operations can be performed on `pt` until the last line is reached. Then the region's key is released by `delete`. An attempt to use `pt` or allocate anything new in `rgn` past this point in the program is a type error because `rgn`'s key is no longer held.

2.2 Extensions to linear typing

As mentioned above, static protocol enforcement requires knowing the aliases of a given tracked object. The authors introduce a new model in which every object allocated on the heap is linear. This means that an object must be freed at some point and that no aliases are available to form a dangling pointer. Programming with only linear data is awkward, so the model permits nonlinear data to have linear components and includes two new operators: `adopt` and `focus`.

`adopt` e_1 `by` e_2 operates on two linear objects and creates a nonlinear reference to the result of e_1, termed the adoptee. The operational semantics associates a list of adoptees with each linear object; this list is updated by reducing an `adopt` expression to a reference. Only linear objects can be involved in adoption, so it is impossible for a single object to have multiple adopters. The reference produced by adoption is effectively guarded by a key associated with the adopter, which is released when the adopter is deallocated. Adoption also releases the key associated with the adoptee. The authors describe multiple ways to handle adoptees of a freed object. In one version, the adoptees are also freed. Alternatively, when an adopter is deallocated, the keys of its adoptees are reacquired, and `free` returns them in a linear list.

Wrapping a linear object in a nonlinear structure is convenient, but allowing access to it through that structure is unsafe because of the potential for aliasing. To allow temporary access to a guarded nonlinear object's linear components, `let` x `= focus` e_1 `in` e_2 binds the result of e_1 to x in e_2, with x treated as linear. This requires creating a fresh key to guard x and releasing keys associated with e_1 while evaluating e_2. This way, it is guaranteed that x is the only name which can be used to refer to the result of e_1. Also, x's key must be held at the end of e_2.

2.3 Typing rules for Core Vault

$$\begin{array}{rcll}
e & ::= & x \mid n \mid e.n \mid e.n := e \mid e(e) \mid e[c] \mid \texttt{new}\langle n\rangle \mid \texttt{free}\; e & \textit{(expression)} \\
 & & \mid \texttt{adopt}\; e : h\; \texttt{by}\; e \mid \texttt{let}\; x = e\; \texttt{in}\; e \mid \texttt{let}\; x = \texttt{focus}\; e\; \texttt{in}\; e & \\
 & & \mid \texttt{fun}\; f[\Delta](x : \sigma) : \sigma\; \texttt{pre}\; C\; \texttt{post}\; C\; \{e\} & \\
c & ::= & \rho \mid C \mid G & \textit{(type arguments)} \\
\tau & ::= & \texttt{int} \mid \texttt{tr(}\rho\texttt{)} \mid G \triangleright h \mid \forall[\Delta].(C, \sigma) \to (C, \sigma) & \textit{(types)} \\
\sigma & ::= & \exists[\rho, \{\rho \mapsto h\}].\texttt{tr(}\rho\texttt{)} \mid \tau & \textit{(linear types)} \\
h & ::= & \langle \sigma, \ldots \rangle \mid \tau[] & \textit{(heap types)} \\
G & ::= & \{\rho, \ldots\} & \textit{(guards)} \\
C & ::= & \bullet \mid \{\rho \mapsto h\} \uplus C & \textit{(capabilities)} \\
\Delta & ::= & \bullet \mid \rho, \Delta & \textit{(type contexts)}
\end{array}$$

Operations for allocating and freeing an array and accessing individual elements of an array are available but left out of the description above. A function defintion includes a type context Δ which gives the guard variables over which the function is parameterized. It also includes pre- and post-conditions which describe what set of keys (i.e. what capability) must be held at the beginning and end of the function. The capability of accessing a linear object and the handle to the object are considered separate in this language; the existential type is used to package the two together. The $\texttt{tr}(\rho)$ type indicates a heap reference; a capability must map the ρ to a particular h. This means that the key uniqueness rule is expanded to require all heap objects to be associated with distinct keys (even if the heap objects themselves have different types).

The type judgment for Core Vault takes the form, $\Delta; \Gamma; C \vdash e : \tau; C$. A type context (the keys currently in existence), a type environment, and a capability (the held-key set) are needed to derive that e has type τ. Deriving this type generates a new "output" capability, which represents what will be the held-key set after evaluating e (this is similar to the "output environment" strategy used for algorithmic typing in λ^{UAL}).

The typing rules rely on three auxiliary judgments: $\Delta \vdash C \leq G$ for showing that a capability satisfies a guard; $C \vdash C$ for rewriting capabilities; and $C; \sigma \vdash C; \sigma$ for converting between existentials and tracked types. The guard satisfaction judgment is straightforward. Base cases are given by

$$\frac{}{\Delta \vdash C \leq \bullet}(\text{GS-Empty}) \qquad \frac{}{\Delta \vdash \{\rho \mapsto h\} \leq \{\rho\}}(\text{GS-Single})$$

The other rules allow a capability to satisfy the intersection of two guards it satisfies individually and allows a disjoint union of capabilities to satisfy a guard that either side of the union would satisfy.

$$\frac{\Delta \vdash C \leq G_1 \qquad \Delta \vdash C \leq G_2}{\Delta \vdash C \leq G_1 \wedge G_2}(\text{GS-Intersect})$$

$$\frac{\Delta \vdash C_1 \leq G}{\Delta \vdash C_1 \uplus C_2 \leq G}(\text{GS-Union1}) \qquad \frac{\Delta \vdash C_2 \leq G}{\Delta \vdash C_1 \uplus C_2 \leq G}(\text{GS-Union2})$$

The existential conversion judgment uses a pair of simple "pack" and "unpack" rules.

$$\frac{}{C; \sigma \vdash C; \sigma}(\text{EC-Base})$$

$$\frac{}{C_1; \exists[\rho, \{\rho \mapsto h\}].\texttt{tr}(\rho) \vdash C_1 \uplus \{\rho \mapsto h\}; \texttt{tr}(\rho)}(\text{EC-Unpack})$$

$$\frac{}{C_1 \uplus \{\rho \mapsto h\}; \texttt{tr}(\rho) \vdash C_1; \exists[\rho, \{\rho \mapsto h\}].\texttt{tr}(\rho)}(\text{EC-Pack})$$

The capability rewriting judgment allows heap types inside a capability to be rewritten according to the existential conversion rule.

$$\frac{C_1 \vdash C_2}{\{\rho \mapsto h\} \uplus C_1 \vdash \{\rho \mapsto h\} \uplus C_2}(\text{CR-Base})$$

$$\frac{C_1; \sigma_i \vdash C_2 \sigma_i'}{\{\rho \mapsto \langle \sigma_1 \dots \sigma_i \dots \sigma_n \rangle\} \uplus C_1 \vdash \{\rho \mapsto \langle \sigma_1 \dots \sigma_i' \dots \sigma_n \rangle\} \uplus C_2}\,(\textsc{CR-Rewrite})$$

Several of the more interesting rules for Core Vault type judgments are described here; the full typing rules (extended to allow abstraction over capabilities) are given in [3].

$$\frac{\begin{array}{c} C_{in}; \sigma_{in} \vdash C_1; \tau_{in} \\ \Delta, \Delta'; \Gamma, [f : \tau_f][x : \tau_{in}]; C_1 \vdash e : \tau_2; C_2 \\ C_2; \tau_2 \vdash C_{out}; \sigma_{out} \end{array}}{\begin{array}{c} \Delta; \Gamma; C \vdash \texttt{fun}\ f[\Delta'](x : \sigma_{in}) : \sigma_{out}\ \texttt{pre}\ C_{pre}\ \texttt{post}\ C_{post}\ \{e\} \\ : \forall[\Delta'](C_{pre}, \sigma_{in}) \to (C_{post}, \sigma_{out}); C \end{array}}\,(\textsc{T-Fun})$$

Checking a function begins by converting the input type to a nonlinear version: This is done by unpacking the existential representation of the linear type to get a tracked type and an extended capability. The unpacked type and extended capability are used to check the function body, and the result is repackaged (if necessary) into an existential. The possibly repacked type must match the declared output type and postcondition.

$$\frac{\begin{array}{c} \Delta; \Gamma; C \vdash e_1 : G \rhd h; C_1 \uplus C_2 \\ \Delta; \vdash C_1 \leq G \qquad \rho \text{ fresh} \\ \Delta; \Gamma[x : \texttt{tr}(\rho)]; C_2 \uplus \{\rho \mapsto h\} \vdash e_2 : \tau_2; C_3 \uplus \{\rho \mapsto h\} \end{array}}{\Delta; \Gamma; C \vdash \texttt{let}\ x = \texttt{focus}\ e_1\ \texttt{in}\ e_2 : \tau_2; C_1 \uplus C_3}\,\text{T-Focus}$$

In order to focus e_1, we must be able to give it a guarded type satisfiable by a subset of the current capability. The remainder of the current capability augmented with a capability to access x must be sufficient type e_2. The capability to access x must also be included in the output capability of e_2. The output capability for the entire `focus` expression is the disjoint union of the part of the input capability that was used to satisfy e_1's guard and the output capability of e_2 minus the capability to access x. Thus all that the `focus` block comsumes is the set of capabilities which are released in evaluating its components, and the capability to access the result of e_1 is unavailable during e_2.

$$\frac{\begin{array}{c} \Delta; \Gamma; C \vdash e_1 : \texttt{tr}(\rho_1); C_1 \\ \Delta; \Gamma; C_1 \vdash e_2 : \texttt{tr}(\rho_2); \{\rho_1 \mapsto h\} \uplus C_2 \\ \Delta \vdash C_2 \leq \{\rho_2\} \end{array}}{\Delta; \Gamma; C \vdash (\texttt{adopt}\ e_1 : h\ \texttt{by}\ e_2) : \rho_2 \rhd h; C_2}\,\text{T-Adopt}$$

The capabilities held after evaluating the adoptee must suffice to evaluate the adopter. After evaluating the adopter, the program must still be capable of accessing the adoptee; that access is revoked in the final output capability.

Several possible semantics are described for `free`, but the type rules presented here assume that an object being deallocated also deallocates its adoptees. The other possibilities mentioned are having adoptees register a callback which should be used for freeing them and having `free` return a linear list of the deallocated object's adoptees.

3 Sing#: Multiparty protocols

Research on the Vault project eventually led to Singularity, an operating system implemented in Sing#, a Vault-like extension to C#. Sing# adds message-passing over channels which follow contracts, Vault-like state-machine descriptions which specify what send and receive operations may be performed at a given time. Memory safety for a single program is less of a concern because Sing# (like C#) uses a garbage-collected heap. However, the message passing is based on an "exchange heap" in addition to the private (garbage-collected) heap associated with each individual process. The exchange heap holds data which may be passed between processes. This is where the potential for unsafe memory accesses arises, so a system like Vault's guarded typing enforces a single-owner rule for exchange heap objects.

Channel contracts themselves are subject to some restrictions. Each cycle in the contract's state transition graph must include at least one send and one receive operation. This prevents cases like having a single process flood the channel with data, causing what amounts to a memory leak on the exchange heap. A message can only carry data of an "exchangeable type," i.e. a scalar or a tracked heap type, so that messaging cannot be used to gain untracked aliases. During implementation of this system, the authors decided that sending a channel endpoint in a message should only be allowed if the state of the channel carrying the endpoint message requires that the next action be a send operation. This avoids a race condition in which aliased channel endpoints caused an ill-timed send to go to the wrong receiver. This contract restriction avoids the use of a locking mechanism to handle such cases.

References

[1] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 59–69, New York, NY, USA, 2001. ACM.

This paper describes how Vault is used to enforce protocols regarding resource usage, e.g. that a file must be opened, then read/written, then closed in that order. Vault uses "type guards" to express these restrictions: A heap object can be statically associated with a particular set of keys which must be held in order to access the object. A key can be treated as a boolean value or an enumeration of states, in which case a type guard may specify that a key must be in a particular state. Functions can use guards to specify pre- and post-conditions and also abstract over key names. This allows, for example, a `fclose` function's type signature to say that it must be given a filehandle guarded by some abstract key, which must be held at the beginning of the function and will not be held at the end of the function.

Vault's contribution is to include this sort of protocol checking in the type system, and it does so essentially by extending C. The paper explains the static analysis used to check that type guards are satisfied (though not in terms of typing rules). The result is a low-level language with the opportunity for far more static verification. The authors demonstrate that this protocol enforcement can be applied to "real-world" systems without excessive annotation overhead.

[2] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity os. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 177–190, New York, NY, USA, 2006. ACM.

Singularity OS is an operating system built using Sing#, a descendant of C# with Vault-like protocol specification. The protocol specification system is extended to support message passing between separate processes. The channels used for message passing have associated contracts which describe what types of messages may be sent at what time. The contract is given in terms of a state machine, similar to associating state with a guard key in Vault. The resource management introduced by the Vault project is used to ensure that processes only access their own memory, do not leak memory, and do not perform an illegal send or receive operation on a channel.

The Singularity project shows how the advantages of high-level/memory safe languages can be leveraged for a low-level task. This paper itself focuses on the implementation of message-passing channels and how the static checking Vault introduced for a single-process setting can be applied to multiparty communication.

[3] Manuel Fahndrich and Robert DeLine. Adoption and focus: practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 13–24, New York, NY, USA, 2002. ACM.

This paper gives a type system for a stateful language in which all heap objects must be allocated and deallocated at linear type. The system follows a core language based on Vault, with two new operations. The `adopt` construct allows nonlinear references to a linear object. The `focus` construct allows a nonlinear object to be temporarily treated as linear. The authors describe use cases for these constructs in a C-like setting and describe how adoption and focusing might be inferred by the compiler. Full typing rules are included in an appendix. Several possible extensions and modifications to the core language are described.

The paper presents extensions to traditional linear typing which facilitate programming with linear types by allowing controlled aliasing. The authors describe an example of using Vault with these extensions to describe a vertex buffer interface for graphics programming.

[4] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Safe and flexible memory management in cyclone. Technical report, 2003.

This tech report focuses on the extensions to Tofte and Talpin's region type system. Regions like those described by Tofte and Talpin are refered to in Cyclone as "lexical" regions. Each lexical region is associated with a particular scope (including each individual function and brace-delimited block). When a lexical region's scope ends, data in that region is deallocated. The authors describe limitations of the Tofte and Talpin system and how they have extended Cyclone to handle these limitations.

The paper introduces dynamic regions, which are not attached to a particular scope. This allows data to escape into another scope, and the programmer can deallocate the region at any time. The associated drawback is the possibility of a compiler-generated runtime check on use of data in a dynamic region.