Project 2: Speculation

Goal

Add type speculation over function arguments to SDyn.

Time

This project is due by 12PM (**noon**, not midnight), Friday, March 1st, 2019. NOTE: This document previously specified that the due date was February 22nd, but as that is during reading week, it has been extended.

Requirements

- Your implementation must be based on SDyn 1.2 or newer.
- You must not break any existing functionality of SDyn. In particular, calling a function with a variety of types, even if those types change, should still work. Note that the speculation scaffolding patch includes a longer-running test, `tests/binsearch2.sdyn`, the behavior of which must be correct whether you're using the scaffolding or not.
- Your code should compile and run on a standard 64-bit x86_64 Linux environment. It will be tested on `linux.student.cs.uwaterloo.ca`.
- Project 1 must be retained: Accesses—both reads and writes—to fields of an SDyn object must use inline caching. That is, after some number of executions of any such function, if objects are provided with the same shape as used in previous executions, their fields should be accessed in constant time, without performing any hash-table lookup.
- If a function is called some fixed number of times—you may choose the number—with the same types for *all* of its arguments (including `this`), and if all further calls *also* use the same types, then for each argument,
  - type-checking must be performed exactly once,
  - the repeated type should be the first type checked, and
  - boxed ints and bools must be unboxed exactly once.

Options

- It is presumed that you will use the existing inline cache and speculation scaffolding provided on the web site, but not required. Note that the speculation scaffolding *requires* the inline cache scaffolding as written, but only mild modification should be necessary to make it work without. Note that there was a bug in the implementation of `SPECULATE` prior to the scaffolding patch, so if you don't use it, you ought to at least get the relevant fix.
- If you use the patch: It implements some of the work for the RECORD operation, namely the JIT part, which calls `sdyn_record` in value.c. `sdyn_record` as written only sets up the array and the function self-destruct; it's your job to implement counting (trivial) and speculation based on those counts (nontrivial). Most, if not all, of your changes should be in the `FUNDECL` and `PARAMS` cases of ir.c.
- You are only required to specialize if calls are "perfect", i.e., they use the same type every time. You are of course free to implement more sophisticated specialization.
- You are not required to perform any unboxing.
- The number of calls to trigger speculation is up to you.

- It is not necessary to modify the JIT per se to implement speculation: The provided template allows for an implementation just by modifying the IR generator.
- While you are still required to implement inline caching, it's not necessary to retain inline caches over the recompilations invoked by speculation (and the scaffolding provides no easy way of doing this anyway).

Hints

(These hints only apply to the provided scaffolding)

`RECORD` will record its values into the *parse* node in its `immp`. This is simply because a parse node is a helpful place to hang this information, as everything else—the IR and the machine code—will self-destruct. The `imm` field of the `RECORD` is used to indicate the number of records to perform before destroying (and thus recompiling) the function. Only the first `RECORD` in any function actually performs counting, so you only need to place the counter there. Because of how `PARAM` and `RECORD` both interact with function arguments, the first `RECORD` must come after the *last* `PARAM`; that is, you must have a block of `PARAM`s and then a block of `RECORD`s. You can determine whether the recording phase has occurred simply by checking whether `seenTypes` has been set on a relevant node!

In the second compilation, you should not generate `RECORD`, but should instead generate `SPECULATE` for each parameter, and `SPECULATE_FAIL` for each `SPECULATE`. The `left` field of `SPECULATE_FAIL` is the associated `SPECULATE`. For the same reason as `RECORD`, as well as to assure that each `PARAM` has occurred, every `SPECULATE` must come after the last `PARAM`. `SPECULATE` should have the parameter to speculate over as its `left`, and the type to speculate as its `rtype`. Note that you *must* unbox ints and bools (that is, use `SDYN_TYPE_INT` in place of `SDYN_TYPE_BOXED_INT` and `SDYN_TYPE_BOOL` in place of `SDYN_TYPE_BOXED_BOOL`), as the JIT isn't built to handle boxed ints or bools except as fully generic boxes.

Because speculation can fail, it is necessary to duplicate each function, for the possibility that speculation did or did not succeed. You are not expected to do the complete power-set: That is, if *any* speculation fails, it's OK to go to a perfectly generic version of the function. The code for `PARAMS` in `irCompileNode` has been inlined into the code for `FUNDECL`, as this will make it easier to keep all this information. In the case of speculation (not recording), your code generation will probably need to look like so:

1. Generate each `PARAM` operation.
2. Clone the symbol table into a speculation-passed and speculation-failed version.
3. Generate each `SPECULATE` operation, remembering their offsets (look at how arrays work!) and updating the speculation-passed symbol table.
4. Generate the entire function body with the speculation-passed symbol table.
5. Generate each `SPECULATE_FAIL` and associate it with the appropriate `SPECULATE`.
6. Generate the entire function body with the speculation-failed symbol table.

Marks

Your submission will be graded partially on "black-box" tests, which test its correctness with no examination of the code, and partially on "white-box" tests, which involve direct inspection:

- 33.34%: All tests included in the SDyn template, plus private grading tests, run correctly, give correct output, and appear to correctly use speculation. If the submitted SDyn does not compile, none of these points are available. Although no performance requirements are set in this project, they are graded by a human (me), so execution times which test human patience will be given no points.
- 33.33%: Types are recorded correctly, and every parameter
- 33.33%: Speculation is implemented correctly.

Notes

- SDyn is released under the ISC license. Your changes are your own.
- If your code exposes any existing bugs in SDyn (quite likely, frankly), that's fine. Please tell me so, so I know to expect it and can fix it.
- You may implement speculation in any way you wish. That being said, while I will certainly smile upon an implementation that actually modifies the JIT to do speculation at a low level—this is the "right" way to do it—there are no bonus points for such an implementation.
- There are very few software components more difficult to debug than just-in-time compilers. `gdb` will catch bugs long after the root cause. Print statements are crucial but not sufficient. Accesses to GC-freed objects don't raise segmentation faults. If you stick to the runtime code, and not the JIT per se, `gdb` should mostly work as expected, but don't expect an easy time of debugging.
- Do not be afraid of rewriting. Rewriting a portion of this project, or even the entire project, may take considerably fewer hours than tracking down an obscure bug.

Submission

Project code must be submitted by email to the instructor. I have many email addresses, all of which go to the same inbox, but they get tagged differently; if you send your submissions to `cs842-2019@gregor.im`, I will appreciate it. As email isn't always the most reliable mechanism, I will respond to verify that I have received each submission. If you don't see a response from me, it's possible that something has gone wrong, so please tell me.

Late submissions are not accepted without prior agreement from the instructor. Extensions will rarely be granted. Requests for extensions sent within the 24 hours prior to the due date, or sent after the due date, will be ignored, except where existing university regulations on extensions, e.g. due to illness, are relevant.

Exceptions and help

For reasons of fairness, uniform projects are required. However, if you need to deviate from this document for exceptional reasons, or if you want any clarification or help, feel free to email the instructor.