Retrofitting Symbolic Holes to LLVM IR

(Extended Abstract)

Bruce Collie
University of Edinburgh
bruce.collie@ed.ac.uk

Michael O'Boyle
University of Edinburgh
mob@inf.ed.ac.uk

Abstract

Symbolic holes are one of the fundamental building blocks of solver-aided and interactive programming. Unknown values can be soundly integrated into programs, and automated tools such as SAT solvers can be used to prove properties of programs containing them. However, supporting symbolic holes in a programming language is challenging; specifying interactions of holes with the type system and execution semantics requires careful design.

This paper motivates and introduces the implementation of untyped symbolic holes to LLVM IR, a strongly-typed compiler intermediate language. We describe how such holes can be implemented safely by abstracting unsound and type-unsafe details behind a new primitive IR manipulation. Our implementation co-operates well with existing features such as type and dependency checking. Finally, we highlight potentially fruitful areas for investigation using our implementation.

1 Context

Our work in this paper is motivated by efforts to extend a program synthesizer we implemented in previous work [1, 2]. This synthesizer produces programs in LLVM intermediate representation (IR) [4]; we do this to allow for synthesized programs to be inserted into existing applications, and to permit translation to a searchable representation [3].

In [1] we synthesize programs by first generating control-flow code based on a library of partial components, then stochastically inserting instructions to each generated basic block. This approach did not scale as the complexity of synthesized programs increased; there were too many possible locations that instructions could be added to, and we had no general way to express constraints on what instructions should be sampled.

The solution we arrived at was for components to contain *symbolic holes*. Instead of selecting both location and value for instructions, the synthesiser now only has to select a value. Our design constraints can therefore be summarised: we require a way to add symbolic holes to LLVM IR programs, and an interface by which values can be assigned to these holes. Some of our components are generic, so holes with no explicit type should be supported. Finally, the implementation should remain as compatible with existing LLVM tools for program manipulation (i.e. programs with holes should still be valid IR).

TyDe '20, August 23, 2020, Remote
2020.

2 Encoding Holes

Our encoding of holes uses "uninterpreted" functions with no definition to represent symbolic holes; one such function declaration is generated for each symbolic hole. For example, a hole of type `i32` is encoded as:

```
declare i32 @hole0()
%0 = call i32 @hole0()
```

When manipulating this program, the value `%0` can be used anywhere a concrete value of type `i32` could be:

```
%1 = add i32 %0, i32 1
```

For holes where the type is not known ahead of time, we create a special hole type:

```
%hole.t = type {}
declare %hole.t @hole1()
%2 = call %hole.t @hole1()
```

If a hole is known to depend on other values (hole or not), we encode this using function parameters. For example, if we know that `%3` should depend on `%1` and `%2`, but not specifically how it should be computed:

```
declare %hole.t @hole2(i32, %hole.t)
%3 = call %hole.t @hole2(i32 %1, %hole.t %2)
```

To allow for untyped hole values to appear in concrete operations (e.g. computing the sum of two hole values), we use a similar encoding. Actual LLVM opcodes such as `add` do not support custom types like `%hole.t`:

```
%4 = call %hole.t @add(%hole.t %2,
    ↪ %hole.t %3)
```

These functions are replaced by concrete opcodes once the type of both operands is known.

This encoding of holes and operations allows us to remain safely within the LLVM type system, and to take full advantage of safety mechanisms such as use-def checking. Programs using our encoding are valid LLVM and can be manipulated as such, but cannot yet be linked and executed.

3 Solver Interface

The programs we encode with symbolic holes are not yet complete. A concrete value must be assigned to each hole in order to produce an executable program. These assignments are the role of a domain-specific client (e.g. a solver or synthesizer). Our encoding of holes is independent of the decision procedure that assigns values to them.

LLVM makes frequent internal use of the "replace all uses with" (RAUW) primitive. RAUW replaces a value in a program with another of the same type; because LLVM IR is in SSA form this replacement is well-defined. If the type of all holes are known, then clients would simply be able to use RAUW to substitute each hole for an appropriate value. However, we allow untyped holes and so require a more powerful abstraction.

To allow clients to select a concrete value for *untyped* holes, we introduce a new IR manipulation primitive: RAUW-NT (New Type) that abstracts a set of unsound transformations that effectively change the type of values.

Implementing RAUW-NT When the original and replacement values have the same type, RAUW-NT behaves identically to RAUW. When they do not, RAUW-NT behaves as follows.

First, it checks that the original value has type `%hole.t`. Arbitrary changes of type are not supported; fixing a value for an untyped hole is the only supported case. Then, any use (e.g. a hole depending on it, or an operation on holes such as `@add`) of the original value is redeclared with the relevant parameter type changed. The call site is then replaced with one where the new value is passed. Finally, the original holes are deleted and the new values renamed where appropriate.

In the code below, consider assigning the constant `i32 5` to the hole `%0`:

```
declare %hole.t @hole()
declare i32 @hole1(%hole.t)
%0 = call %hole.t @hole()
%1 = call i32 @hole1(%hole.t %0)
```

The code produced by RAUW-NT will (after renaming) be:

```
declare i32 @hole1(i32)
%1 = call i32 @hole1(i32 5)
```

The original hole no longer exists, and its uses appear to have changed type to `i32`.

Backpropagating Types If a value is assigned to a hole that is used by an operation, then type information may flow backwards as well as forwards. For example, we know that both operands of an add operation must be the same type, and the result also has that type. We therefore replace untyped operands and operations with typed ones when partial information becomes available about them. It is also possible for RAUW-NT to fail at this point if the inferred types are incompatible.

Implementing RAUW-NT requires manipulating LLVM IR in a way not originally intended by its designers, and is fiddly to implement correctly. By abstracting it, clients can make high-level decisions on the values they assign without worrying about low-level IR manipulation.

4 Research Directions

As described, our method for embedding holes in LLVM IR programs is only the first implementation step towards full solver-aided programming integrated with the compiler. Some promising next steps towards this are:

Synthesis We are already using our implementation in the next version of our synthesizer; in a paper currently under review we are able to synthesize more complex functions than in previous work [1]. Being able to express constraints and partial type information in synthesis components independently of each other has proved to be a useful feature.

Solver Integration We hope to investigate further integration with solver-aided techniques beyond whole-program synthesis. Where our synthesis procedure uses whole-program behaviour to determine correctness, solver-aided techniques generally use local measures such as assertions; these can be added independently of our hole encoding. One use of these assertions is to implement *angelic execution*, one of the principal techniques highlighted in [6]

Another example is superoptimization over bit-vector programs; these problems can be stated easily within the LLVM type system. Expressing a length-N superoptimization constraint can be achieved simply by using N suitably-typed symbolic holes.

Source Language Support Another avenue for future work is the development of libraries in languages that target LLVM IR (e.g. C, C++ or Fortran) that expose symbolic hole primitives to programmers. Doing so would allow for programs written in these languages to take advantage of solver-aided programming without requiring specialised knowledge of the compiler.

Tool Support Languages with first-class hole support such as Hazel [5] emphasize the need for supporting tooling and programming environments. Extending this kind of support to LLVM IR programs would enable interactive programming for a wide class of existing applications.

Theory This paper focuses on the implementation of a system for manipulating LLVM IR programs in the presence of potentially untyped values; the implementation is driven by the constraints of the existing language tooling and manipulation utilities. Existing work [7] deals with formalising LLVM IR; grounding our system formally in similar terms alongside gradual typing and hole-enabled programming is interesting future work.

References

[1] B. Collie, P. Ginsbach, and M. F. P. O'Boyle. 2019. Type-Directed Program Synthesis and Constraint Generation for Library Portability. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 55–67. https://doi.org/10.1109/PACT.2019.00013

[2] Bruce Collie and Michael O'Boyle. 2019. Augmenting Type Signatures for Program Synthesis. *arXiv:1907.05649 [cs]* (July 2019). arXiv:cs/1907.05649

[3] Philip Ginsbach, Lewis Crawford, and Michael F. P. O'Boyle. 2018. CAnDL: A Domain Specific Language for Compiler Analysis. In *Proceedings of the 27th International Conference on Compiler Construction (CC 2018)*. ACM, New York, NY, USA, 151–162. https://doi.org/10.1145/3178372.3179515

[4] Chris Lattner. 2002. *LLVM: An Infrastructure for Multi-Stage Optimization*. Ph.D. Dissertation. Computer Science Dept., University of Illinois at Urbana-Champaign.

[5] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 14:1–14:32. https://doi.org/10.1145/3290327

[6] Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. ACM, Indianapolis, Indiana, USA, 135–152. https://doi.org/10.1145/2509578.2509586

[7] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. *ACM SIGPLAN Notices* 47, 1 (Jan. 2012), 427–440. https://doi.org/10.1145/2103621.2103709