

An Examination of Local Definitions

Jason Kirschenbaum
The Ohio State University,
Columbus OH 43210, USA

Email:kirschen@cse.ohio-state.edu

Abstract—This paper presents an exploration of the issues involved in using local mathematical definitions within Resolve component contracts and realizations. Several issues are highlighted along with possible solutions.

I. INTRODUCTION

The verifying compiler grand challenge involves work on many fronts. Currently, advances in programming languages, automated proof tools, and mathematical theories may be needed to create a verifying compiler. Indeed because of the scope of the challenge, as interpreted by the Resolve community [1], practices that have been accepted as suitable for use in both the specification of and reasoning about software components have to be re-examined in light of the verifying compiler vision.

One such practice is the use of local mathematical definitions, that is a definition introduced in a particular software component, be it contract or realization. These local definitions present several challenges in automated verification of software component implementations. In particular,

- 1) local definitions may require new lemmas for automated verification
- 2) those lemmas will need to be proved, and
- 3) local definitions may need to be connected to other mathematical results.

Given these challenges, we advocate strict restrictions on the use of local definitions along with characterizing the possible maintenance issues that arise with both local definitions and mathematical theories.

As a motivating example for the paper, we will use the `NaturalFacility` component along with several extensions. The contract is shown in Fig. 1 and is written in the OSU dialect of Resolve. In this dialect, there are no new mathematical types, only new subtypes of a fixed number of base mathematical types; the definition of `NATURALMODEL` is needed to achieve the effect of a Natural Number type. In this dialect, `control` is a special type for use in control structures (what would be called a Boolean expression in C), e.g., in an if-condition or while-condition. The component supports several very simple operations; while extremely simple, it is enough to illustrate the potential issues in using local definitions.

```
contract UnboundedNaturalFacility

  math subtype NATURALMODEL is integer
  exemplar n
  constraint
    n >= 0

  type Natural is modeled by NATURALMODEL
  exemplar n
  initialization ensures
    n = 0

  procedure Increment (updates n: Natural)
  ensures
    n = #n + 1

  procedure Decrement (updates n: Natural)
  requires
    n > 0
  ensures
    n = #n - 1

  function AreEqual (restores m: Natural,
                    restores n: Natural): control
  ensures
    AreEqual = (m = n)

  function IsGreater (restores m: Natural,
                    restores n: Natural): control
  ensures
    IsGreater = (m > n)

  function Replica (restores n: Natural): Natural
  ensures
    Replica = n

end UnboundedNaturalFacility
```

Fig. 1. `UnboundedNaturalFacility` contract

Figures 2 and 3 shows extensions to the `NaturalFacility` component, `IsOdd` and `Halve` respectively. The `IsOdd` function checks if a number is odd. Since we are interested in *automated* verification and automated proof tools have difficulties with quantifiers, the code includes a local mathematical definition `IS_ODD`. Thus, the quantifiers are encapsulated within the mathematical definition. The `Halve` extension ensures clause requires no quantifiers, and is a mathematical assertion; no requires clause is needed because the `NATURALMODEL` ensures that all integers that satisfy the `NATURALMODEL` constraints are non-negative.

Finally, we include a contract for an implementation of a `Multiply` extension using the famous Egyptian¹ multiplication algorithm. We omit the contract for brevity. The

¹Or Ethiopian

```

contract IsOdd enhances UnboundedNaturalFacility

  definition IS_ODD (n: NATURALMODEL) : boolean
  is
    there exists k: NATURALMODEL (n = 2 * k + 1)

  function IsOdd (restores n: Natural): control
  ensures
    IsOdd = IS_ODD (n)
end IsOdd

```

Fig. 2. IsOdd extension

```

contract Halve enhances UnboundedNaturalFacility

  procedure Halve (updates n: Natural)
  ensures
    #n = n + n + 1 or #n = n + n
end Halve

```

Fig. 3. Halve extension

corresponding implementation is shown in fig. 4.

II. LOCAL DEFINITIONS, LEMMAS AND PROOFS

The first question that we are compelled to examine is the requisite lemmas and proofs regarding the new local mathematical definition, `IS_ODD`. First, these definitions and lemmas may be simple enough that a software engineer could, in principle, come up with both the lemmas and proofs required, however in the vision of verified software, mathematicians are employed to develop theories—they will be needed here. In the most general case, the lemmas required for these local definitions could be just as subtle as some of those in the full-blown mathematical theory; therefore, the

```

realization Egyptian implements Multiply
  for UnboundedNaturalFacility

  uses IsOdd for UnboundedNaturalFacility
  uses IsPositive for UnboundedNaturalFacility
  uses Double for UnboundedNaturalFacility
  uses Halve for UnboundedNaturalFacility
  uses Add for UnboundedNaturalFacility

  procedure Multiply (updates n: Natural,
                    restores m: Natural)
  variable b, p: Natural
  b := Replica (m)
  loop
    maintains #n * #m = n * m + p
      and m >= 0 and b = #b
    decreases m
  while IsPositive (m) do
    if IsOdd (m) then
      Add (p, n)
    end if
    Double (n)
    Halve (m)
  end loop
  n := p
  m := b
end Multiply
end Egyptian

```

Fig. 4. Egyptian implementation of Multiply

mathematicians must be, at least, involved in the creation of the local definitions and corresponding results. This is similar to new mathematical theories.

We also note that, because the local definitions are in the contract for an operation, local definitions become a part of *every* use of that contract. Thus, the Egyptian implementation must have access to the mathematical definition of `IS_ODD`. The automated tools must have rules for manipulating any new mathematical symbols, either via simplification or via deduction rules. Therefore, in this case, `IS_ODD` must have supporting lemmas to aid the automated tools. Moreover, each of the newly created lemmas introduce proof obligations that must be proved.

An issue with these required pieces is determining where they should reside. Clearly proofs should not be included in the source code; this violates the separation of concerns—only the lemma statements are needed in the VC proofs. Moreover, those lemmas may need to include automated prover directives. For example, certain simplifications may only ever need to be used in one direction. These directives, while helpful to the prover, simply do not belong in source code. Therefore, we argue that the lemmas themselves should be factored off into a “local theory” file. Since the lemmas are already in the local theory file and must be used by the prover anyway, we can move the local definition to the local theory. Hence, local definitions in this context are a small mathematical theory!

Based on this argument, we argue that *any* current local definition in a contract should be moved to a mathematical theory; users gain little by leaving it locally and already have the same maintenance issues of a general mathematical theory. Local theories for implementations, however, may still be used; the fact that nothing outside of the implementation can depend on that theory (because of the modular reasoning property) we still may gain some net benefit.

This reasoning process assumed the “worst” case, namely that we have (relatively) deep mathematical results that require non-trivial lemmas. The “best” case would be local definitions used only for naming; a device used by people to manage complexity. For example, one may introduce a predicate in some contract or realization `IS_GOOD (n)` whose definition is simply that `n` is between zero and ten. Clearly, such use does not carry the possible baggage of the general case. New syntax to distinguish between the two extremes, along with rules for how the proof system and automated provers should treat the definitions can be created to mitigate this issue.

III. LOCAL DEFINITIONS AND EARLIER MATHEMATICAL RESULTS

The final issue with local definitions addressed in this paper is how the “new” local definitions connect with existing operations contracts and mathematical theories. Contracts are always written with the effects of any operations denoted by the effects on a particular mathematical model of the component in question (mathematical strings for a `Stack` component, mathematical sets for a `Set` component, etc.).

Moreover, the contracts are also written using a particular version of the mathematical theory. As the theories are revised, new definitions, lemmas, etc. are added.

This process can happen on a very small scale with local definitions. The contracts for `Half` and `IsOdd` show how this can occur. While the ensures clause of `Half` is exactly what is desired and is strong enough for people to reason through any VCs that may arise, automated tools would have a harder time with the VCs. Instead of an ensures clause of `#n = n + n + 1` or `#n = n + n`, with the `IS_ODD` predicate we can more precisely characterize the results, i.e., `if IS_ODD(#n) then #n = n + n + 1 else #n = n + n`.

By connecting operation contracts to new mathematical functions and predicates, the VCs can become more “obvious”. Instead of the automated tools needing the meaning of `IS_ODD` to prove the VCs, by using the second formulation of the ensures clause, most of the VCs instead only require simple logical manipulations.

Even though this issue occurs naturally when examining local definitions it also affects general mathematical theory development. Certain components may depend on one mathematical theory, while others depend on an enhanced version. Information that allows the automated tools/VC generator to automatically resolve many of these incompatibilities will increase the “obviousness” of VCs and must be addressed.

IV. CONCLUSION

In conclusion, we’ve presented some issues with local definitions and concluded that, for automated verification, in most cases local definitions simply should not be used. The overhead required to make the definitions usable by the automated provers along with the other maintenance and other issues identified in this paper would best be spent creating a new mathematical theory or augmenting an existing one with help from the mathematical specialists. Moreover, we’ve examined the effect of mathematical theory extensions and their ramifications on generated VCs.

V. ACKNOWLEDGMENTS

The author is grateful for the original code implementations created by Paolo Bucci, and Bruce Weide. This work was supported in part by the National Science Foundation under grants DMS-0701187, DMS-0701260, and CCF-0811737.

REFERENCES

- [1] W. OGDEN, J. HOLLINGSWORTH, J. KRONE, M. SITARAMAN, and B. W. WEIDE, “The resolve software verification vision,” in *Proceedings of Resolve Workshop*, Clemson, SC, 2007, pp. 1–3. [Online]. Available: http://dsrg.cs.clemson.edu/resolve_2007/proceedings.html