MonoProc; Monotone Data-Flow Analysis for a Simple Procedural Language

Pepijn Kokke

In this paper I will provide a brief overview of the MonoProc system, and its relation to the framework for data-flow analysis as described in (Flemming Nielson 1999).

Language

Syntax

A program in MonoProc consists of a number of statements and a number of procedure declarations. These are allowed to occur in any order. Statement declarations have the following form, with their usual interpretations.[1]

```
Stmt
  ::= "skip" ";"
    | Name "=" AExpr ";"          -- assignment
    | Name "(" AExpr* ")" ";"     -- call statement
    | "if" "(" BExpr ")" "then" Block
    | "if" "(" BExpr ")" "then" Block "else" Block
    | "while" "(" BExpr ")" Block
```

Aside from these syntactic constructs, there are two implemented forms of syntactic suger.

The syntactic construct

```
    | "return" Name ";"
```

is transformed to an assignment to a special variable called **return**, and

[1] While **BExpr** are syntactically not considered statements in their own right, they are allowed under the **Stmt** constructor in my implementation as it greatly simplifies the implementation.

```
    | Name "=" Name "(" AExpr* ")" ";"
```

is transformed to a call statement, followed by an assignment of the *value* of **return** to the variable in question.

Furthermore, **AExpr**s and **BExpr**s have the following form.

```
AExpr
  ::= Name            -- variable
    | Integer         -- primitive integer
    | "-" AExpr
    | AExpr "+" AExpr
    | AExpr "-" AExpr
    | AExpr "*" AExpr
    | AExpr "/" AExpr

BExpr
  ::= "true"
    | "false"
    | "~" BExpr
    | AExpr "<" AExpr
    | AExpr "<=" AExpr
    | AExpr ">" AExpr
    | AExpr ">=" AExpr
    | AExpr "==" AExpr
    | AExpr "!=" AExpr
```

Aside from the above , another constructor for **AExpr**s exists: *null* values. While *null*-assignments are not syntactically allowed, they are used to model function entry and exit in the analyses.

Finally, procedure declaclarations have the following form.

```
Decl
  ::= Name "(" Name* ")" Block
```

NB. A **Name** is any string of letters, digits and underscores that starts with a lowercase letter.

Semantics

Semantically, the MonoProc language does not stray far from the WHILE language as defined in (Flemming Nielson 1999, 3–5).

A notable exception is that in MonoProc the user cannot choose the name for a function's return value. This does not, however, limit the expressiveness of the language, as the user can assign to global values or user extra parameters to simulate a chosen return value.

Framework

My implementation of monotone frameworks is based on the description provided in (Flemming Nielson 1999, 33–97; Hage 2013). Therefore, I will refer the reader to these pages for a detailed description of data-flow analysis and monotone frameworks.

In the paragraphs below I will

1. provide a mapping from the descriptions in (Flemming Nielson 1999) to modules and functions in my implementation;
2. give a brief overview of the combinators that are used in the construction of framework instances;
3. discuss the deviations from (Flemming Nielson 1999) in my implementation.

From Nielson, Nielson and Hankin to MonoProc

Basic data-flow functions as described in §2.1 (*init*, *final*, *flow*, ...) can be found in the *Flowable*[2] module.

The **AExp**(_) function can be found in the *Available* module as *available*(_), and is defined on any term that can contain arithmetic expressions (including arithmetic expressions).

The $FV(\_)$ function can be found in the *FreeNames* module as *freeNames*(_). As with *available*(_), it is defined on any term that can contain variable names.[3]

The definition of a monotone framework and of a lattice can be found in the *Analysis* module, together with some basic functions for building monotone frameworks—a description of which will follow below under Constructing Monotone Frameworks.

The analyses (available expressions, live variables, ...) are defined in the *Analysis* module, under their usual abbreviations (AE, LV, ...). These will be discussed in Analyses.

[2] All modules in this paragraph can be found in the *PROC.MF* module, which is the module containing the implementation of the monotone framework.

[3] A new module, *UsedNames*, has also been implemented under *PROC.MF*. However, as this module is only used during evaluation—and not analysis—this does not seem an appropriate time to discuss it.

Several versions of the *MFP* algorithm have been implemented in the *Algorithm.MFP* module.

`mfp`: Computes a pointwise maximal fixed-point—using call strings[4] as a context—and collapses all pointwise results using the join operator ($\sqcup$).

`mfpk`: As *mfp*, but allows the user to provide a parameter k that is used to bound the length of the call strings.[5]

`mfp'`: As *mfp*, but it returns a pointwise analysis.

`mfpk'`: As *mfpk*, but it returns a pointwise analysis.

Furthermore, several versions of the *MOP* (actually *MVP*) algorithm have also been implemented. These can be found in the *Algorithm.MOP* module.

`mop`: Computes a *meet over all (valid) paths* analysis by computing all valid paths, and composing the transfer functions along these paths.

`mopk`: As *mop*, but allows the user to provide a parameter k that is used to bound the lengths of the paths *by bounding the number of times that a path is allowed to visit a single program point.*

NB. In general, *MOP* isn't guaranteed to terminate on recursive or looping programs. In my implementation, however, *MOP* is guaranteed to terminate on all programs, but is not guaranteed to return correct results. More precisely, with recursive calls and loops *mopk* will examine at most k recursive calls or iterations. In general this is not a safe extension, but for practical purposes it suffices. As *mop* is internally based on *mopk*, this loss of guarantee of correctness also affects my implementation of *mop*.

Constructing Monotone Frameworks

Below I will discuss the functions that can be used to construct instances of monotone frameworks, in order of relevance.

`framework`: The empty monotone framework, of which all properties are left undefined. Using this will result in error messages stating which properties you still have to provide and—if possible—what functions you can use to do this. *NB.* It is common to use the invocation of `framework` to provide an instance of a lattice.

[4] In the implementation, call strings are referred to as call *stacks*, because their implementation more closely resembles a stack (where the top element is the latest function call).

[5] In the implementation, plain `mfp` calls `mfpk` using `maxBound` as a value for k. While this is technically still a bounded analysis in terms of call string length, `maxBound` usually evaluates to around 2147483647, which should suffice for most practical purposes. The same holds for `mop`/`mopk`; however, since meeting over all paths with at most 2147483647 repetitions of the same program point can take... rather long, I'm assuming nobody will encounter this in practice.

`embelished`: Provides the instance with a table of declarations, and precomputes the blocks in the program.

`distributive`: Allows for an easy definition of the transfer functions of distributive analyses that return sets (*AE*, *LV*, ...). It defines the transfer function in terms of a *kill* and a *gen* function (see Flemming Nielson 1999, fig. 2.6).

`forwards, backwards`: Defines the direction of the analysis. Practically, it sets the *direction* value in the instance and precomputes the *flow*, *interflow* and *extremal values* for the analysis.

See below for a concrete definition of an instance, taken from *Analysis.AE*.

```
mfAE :: Prog -> MF (Set AExpr)
mfAE p
  = forwards p
  $ distributive killAE genAE
  $ embelished p
  $ framework
  { getI = empty
  , getL = Lattice
    { join    = intersection
    , refines = isProperSubsetOf
    , bottom  = available p
    }
  }
```

The final type of the "instance" is $Prog \to MF$, and it will in fact only compute the actual instance upon application to a program. This is internalized in the `analyse` functions, however, and therefore the user will rarely have to do this.

Deviations from Nielson, Nielson and Hankin

While my implementation is often true to the description in (Flemming Nielson 1999), there are a few exceptions. These will be briefly discussed below.

Limiting MOP to MOP-k

This has been discussed above, and is only restated here for completeness.

Procedure Entry and Exit Label

The most notable of these is that a procedure's entry and exit point do not receive any explicit labels. Instead the values for *init* and *final* for the procedure's body are used in interflow expressions.

This effectively means that the transfer functions for a procedure call and its entry are forced to be the same function—and likewise for exit and return—which limits the analytical capabilities of my implementation. However, the distinction between these pairs of transfer functions was not required for the simple analyses that were implemented.

Analyses

Below I will provide a brief overview of the implemented analyses.

Available Expressions (AE): Available Expression analysis computes, for every program point, what expressions are currently available, i.e. have been computed in previous execution steps but have not yet been overwritten.

This is useful for, for instance, removing duplication of expressions.

Live Variables (LV): Live Variable analysis computes, for every program point, which variables may be used in future computations.

This is useful for, for instance, removing variables and assignments that are never used.

Very Busy Expressions (VB): Very Busy Expression analysis computes, for every program point p, which expressions are guaranteed to be used in all paths from p up to the next assignment to any variable used in p.

This is useful for, for instance, eager evaluation of very busy expressions.

Reaching Definitions (RD): Reaching Definitions computes, for every program point, which assignments are known not to affect the outcome of the program. It does this by looking for the assignments that are not used before their next redefinition.

This is useful for, for instance, removing non-reaching definitions.

Constant Propagation (CP): Constant Propagations computes, for every program point, which variables are known to have a constant value in all paths leading up to that program point.

This is useful for, for instance, optimizing by converting variables to constant values.

References

Flemming Nielson, Hanne Riis Nielson, Chris Hankin. 1999. *Principles Of Program Analysis*. Springer-Verlag.

Hage, Jurriaan. 2013. "Automatic Program Analysis." University Lecture. http://www.cs.uu.nl/docs/vakken/apa.