Proceedings of the 8th International Conference on Applied Informatics
Eger, Hungary, January 27–30, 2010. Vol. 2. pp. 429–436.

Measuring Software Complexity by Types*

Gábor Páli[a], Tamás Kozsik[b]

Department of Programming Languages and Compilers,
Eötvös Loránd University
[a] pgj@elte.hu [b] kto@elte.hu

Abstract

In typed programming languages, types play an important role in many aspects of a program. This paper discusses a potential approach to software measurement techniques for software written in strongly typed, especially functional programming languages. As number and size of software systems implemented in such languages are continuously growing nowadays, existence of valuable software metrics is getting important. We are confident that it is worth focusing on types themselves when trying to say something about the quality of a software product. We examine the possibilities of describing program quality and complexity through a static analysis of types and deriving metrics from them.

Keywords: Functional Programming, Software Measurement, Software Quality, Metrics, Programming with Types, Haskell

MSC: 68N30 (Mathematical aspects of software engineering)

1. Introduction

With the rise of strongly typed programming languages, types are present in most programming languages and they are already exploited in many ways by the recent compiler systems. There are many active research projects in the field, and the most sophisticated languages tend to be based on exhaustive type theoretical work. Typing, and programming with types are virtually present from the birth of computer science, and they have proven to be a useful tool both in formulating and verifying programs for correctness.

In this paper we present a theoretical basis on how to measure software complexity by types, and define some simpler metrics based on it for demonstration. Main contributions of our work can be highlighted as follows.

*Supported by POSDRU/6/1.5/S/2008, TÉT AT-10/2008

– By skimming through the ways as types are already used (abstraction, safety, optimization, documentation), one might have the intuition that they are able to capture the intentions behind a program, therefore we can be describe how it was constructed. We convert this intuition into concrete mathematical concepts (Section 2.1).

– When analyzing the different relations between the employed types, we can draw several conclusions on the expected quality of the software. This helps us to determine whether the given software was developed by the right principles and to spot potential problems with the implementation, or even to discover further opportunities for improving the continuity of abstraction in the employed software architecture. We define some of the possible type-based measurement methods and explain their usage through an example (Section 2.2, Section 3).

– We add some notes on how to implement these methods and how the concepts should be represented in the real world. Although here we primarily focused on functional programming languages, we believe that it can be adapted to other programming languages or paradigms as well with a few modifications (Section 2.3).

2. Types as Measures of Complexity

Different kinds of software metrics on types can be defined to describe complexity of software products. Although only simple programs are discussed in the paper, we belive that the proposed solutions can be extended to programs incorporating several modules and software projects built upon them with a minimal effort. The main idea behind this approach is based on a claim for measuring, or at least giving an estimate on, the usage of an important abstraction, the concept of *types*.

Because metrics should be easy to compute yet exhaustive enough to provide quick and useful feedback for the programmer on her work, technically we describe statistical heuristics on types appearing in programs, and a method on how to turn type information into numbers. Metrics values are computed statically, purely based on the source code not on run-time profiling. It is based on the assumption that developers need to maintain all the source code, which costs money, and an important goal of using metrics is to estimate cost of the software development process from the beginning. Testing usually takes more time and covers only a part of all possible uses.

2.1. Basic Concepts

To explain the computations required for determining the value of the metrics, it is assumed that the analysis is run on a program P. During the analysis, we will need the set of types of P as

$$\tau_P = \tau_0 \cup \tau_1 \cup \left(\bigcup_{i=1}^{n} \tau_{M_i} \right) \tag{2.1}$$

where τ_0 refers to *base types* offered by the given language implementation, which are taken already available. Note that elements of τ_0 might not appear in P explicitly while they can be applied there. Types imported from a module M_i in P are referred as τ_{M_i}. Finally, τ_1 consists of types defined in P itself.

Similarly to the set of types, names in P can be also given by the same scheme:

$$\nu_P = \nu_0 \cup \nu_1 \cup \left(\bigcup_{i=1}^{n} \nu_{M_i} \right) \tag{2.2}$$

where all components are declared in the same way as in Eq. 2.1. The only difference that all the names must be fully-qualified in order to be unique.

Via τ_P and ν_P, a set of identifiers with their assigned types can be constructed for program P, called ι_P:

$$\iota_P : \mathcal{P}(\nu_P \times \tau_P) \tag{2.3}$$

Supported by the definitions introduced here, we can specify a function o (Eq. 2.4) which assigns a value to a name-type pair by counting the repetitions of a name with a given type. This function determines the *occurrences* of a name in a program P, i.e. the number of its references.

$$o \ : \ \mathbf{Prg} \times \mathbf{Id} \to \mathbf{N} \tag{2.4}$$

where **Prg** is the set of all well-formed programs, and **Id** is the set of all well-formed, fully-qualified identifiers according to the grammar of the language.

Another relevant operation is classification of elements of set ι_P by their types: Names with the same type are put in the same class as

$$[a]_P = \{ \ i \in \iota_P \ \mid \ i \sim a \ \} \tag{2.5}$$

where $\sim$ is a binary equivalence relation over ι_P:

$$\sim : \ \iota_P \times \iota_P \to \{ \ 0, \ 1 \ \} \tag{2.6}$$

Assuming that definition of $\sim$ is given, ι_P can be partitioned into finite number of disjoint subsets by types, where $\kappa_{P\sigma}$ is the set of identifiers in program P with type σ, and n is the number of distinct types (Eq. 2.7), i.e. the number of equivalence classes over ι_P by $\sim$. This corresponds to the number of types assigned to names in P.

$$\iota_P = \bigcup_{i=1}^{n} \kappa_{P\sigma_i} \tag{2.7}$$

By summing all the occurrences of each identifier with the same σ type, an accumulated value of $O_{P\sigma}$ can be given which tells how many times each type is referenced in the program P. This can be interpreted as a weight for σ:

$$O_{P\sigma_i} = \sum_{(\nu,\sigma)\in\kappa_{P\sigma_i}} o(\nu) \tag{2.8}$$

2.2. Type-Based Metrics

Different measurements can be formulated based on definitions of the o function (Eq. 2.4) and the derived $O_{P\sigma_i}$ values (Eq. 2.8). Among other things, the extracted statistical information would be suitable for describing the following.

- *Complexity.* The number of types and type schemes used in P, see Eq. 2.5.

  It helps to judge whether the program is needed to be split into smaller sections. A program with too many types might be "over-abstracted" in some sense.

- *Distribution of Types.* Set of normalized $O_{P\sigma_i}$ values together with their σ_i types which could be given as (for $1,\ldots,n$)

  $$D_{\sigma_i} = \frac{O_{P\sigma_i}}{\sum_{j=1}^{n} O_{P\sigma_j}} \tag{2.9}$$

  where

  $$\sum_{i=1}^{n} D_{\sigma_i} = 1 \tag{2.10}$$

  implicitly holds. Then D can be constructed from them as

  $$D = \{\ (\sigma_i, D_{\sigma_i}) \mid i \in [1..n]\ \} \tag{2.11}$$

  The resulting D set consists of type-distribution pairs, where each type is associated with its probability. These pairs tell us which types would be considered the "backbone" for the implementation. This might help program architects to identify and collect commonly used programmer-defined types and design a program component based on them.

- *Important Types.* Distribution of types in program P, sorted by their $\kappa_{P\sigma}$ values in descending order (Eq. 2.12). Therefore the most used types will appear in the beginning of the enumeration.

  $$(\sigma_i, x) \le (\sigma_j, y) \equiv x \le y \tag{2.12}$$

where $(\sigma_i, x), (\sigma_j, y) \in D$, $x, y \in \mathbb{R}$, $\sigma_i, \sigma_j \in \tau_P$, and $i \neq j$.

This presents us the correct usage of the types offered by the language, whether there are user-defined types used. After fixing the problems arisen from a poor strategy of working with abstractions, it can be also verified whether the rewritten version is a right step towards a better design.

2.3. Notes on Implementation

Now we are sketching simple algorithms on how to implement the sets and functions defined previously. First, consider the generation of names of P: Read and parse the given P, while collecting all the identifiers in a list. Construction of the set of types (Eq. 2.1) is performed by the collection of names: For each name in the finite set of names (Eq. 2.2), look up or infer the type of the given name, and construct a set of ordered name-type pairs (Eq. 2.3).

The implementation of the $\sim$ function (Eq. 2.6) is not straightforward and it may have several effects on the results. Checking equivalence of elements in ι_P is based on their type component. Therefore computing the result of $\sim$ is merely about deciding whether the given types can be considered the same. As a trivial approach, every type with a different name could be treated as an independent entity. But we think this might lead to very biased results, because there might be type synonyms defined in the program to help the reader which should not be considered a mistake at all, thus they should not be distinguished. For Hindley-Milner type systems where existence of principal types is unique, it could be a good compromise to say that two types are considered equal if their most general type or principal type is the same. Using this principle, we could say that $\sigma_a \sim \sigma_b$ holds if there exists a common general type. However, in other type systems one might need to use different methods, e.g. in structural type systems, type equivalence should be determined by a structural comparison.

Note that since all the identifiers found in program P are collected, it must contain names of functions (or subroutines) as well which types are more complex to handle. A possible way to work with types of functions is to consider only the value of what they return. This recommendation is based on the intuition that functions have arbitrary combination and number of parameters which would also result in unnecessarily fragmented statistics. According to these observations, we can say that type of a function should be always split into components. Here we represent functions as they appear in their curried form in λ-calculus:

$$f \;::\; \sigma_{f_0} \;\rightarrow\; \ldots \;\rightarrow\; \sigma_{f_n} \;\rightarrow\; \sigma_f \tag{2.13}$$

where σ_f corresponds to the type of the value returned by f, $f_0, \ldots, f_n$ are the arguments of f, and $\sigma_{f_0}, \ldots, \sigma_{f_n}$ are the types of arguments.

In this way f in ι can be resolved into a set (Eq. 2.14) which can be added to ι_P.

$$\{\; (f, \sigma_f), (\nu_{f_0}, \sigma_{f_0}), \ldots, (\nu_{f_n}, \sigma_{f_n}) \;\} \tag{2.14}$$

where $\nu_{f_i} \in \mathbf{Id}$, $\sigma_{f_i} \in \tau_P$.

3. An Example

To demonstrate how metrics could be calculated by following the formulas and suggestions we proposed above, let us take a real-life example (see Figure 1). The sample program solves the problem of A* path finding for an ASCII map and it is implemented in Haskell in 71 lines of code (ELOC[1]: 64). It uses a custom implementation for priority queues but we omitted it and its source code from the measurement because of space constraints. [2]

```
import Control.Monad (guard, liftM2)
import Data.List (elemIndex)
import qualified Data.Set as S
import qualified Data.Map as M
import qualified PriorityQueue as Q

type Point = (Int, Int)
type Map = [[Char]]

main :: IO ()
main = interact doit

heuristic :: Point -> Point -> Int
heuristic (x, y) (u, v) = abs (x - u) `max` abs (y - v)

astar :: (Ord a, Ord b, Num b) =>
  a -> (a -> [a]) -> (a -> Bool)
    -> (a -> b) -> (a -> b) -> [a]
astar s succ end cost heur
 = astar' (S.singleton s) (Q.singleton (heur s) [s])
 where
   astar' seen q
     | Q.null q  = error "No Solution."
     | end n     = next
     | otherwise = astar' seen' q'
    where
     ((c,next), dq) = Q.deleteFindMin q
     n     = head next
     succs = filter (`S.notMember` seen) $ succ n
     calc  = (+ c) . (subtract $ heur n) .
             liftM2 (+) cost heur
     costs = map calc succs
     nexts = map (: next) succs
     czs   = Q.fromList (zip costs nexts)
     q'    = dq `Q.union` czs
     seen' = seen `S.union` S.fromList succs

find :: Char -> Map -> Point
find c m = find' 0 m
 where find' _ [] = error "Cannot find tile."
       find' y (h:t)
        | Just x <- elemIndex c h = (y, x)
        | otherwise = find' (y + 1) t

successor :: Map -> Point -> [Point]
successor m (x,y)
 = do u <- [x + 1, x, x - 1]
      v <- [y + 1, y, y - 1]
      guard (0 <= u && u < length m)
      guard (0 <= v && v < length (head m))
      guard (u /= x || y /= v)
      guard (m !! u !! v /= '~')
      return (u, v)

path :: Map -> [Point] -> Map
path m l = iterY m l 0
 where
  iterY [] _ _      = []
  iterY (h:t) l n   = iterX h l n 0 : iterY t l (n + 1)
  iterX [] _ _ _    = []
  iterX (h:t) l n m = pick : iterX t l n (m + 1)
    where pick = if (n,m) `elem` l then '#' else h)

doit :: String -> String
doit s = unlines . path m $ astar start succ (== end) cost h
 where
  m           = lines s
  start       = find '@' m
  end         = find 'X' m
  succ        = successor m
  h           = heuristic end
  cost (x, y) = costsM M.! (m !! x !! y)
  costsM      = M.fromList [('@',1),('x',1),('X',1)
                           ,('.',1),('*',2),('~',3)]
```

Figure 1: A* path finding for an ASCII map written in Haskell

During the analysis of the sources, an additional refinement was made in the calculation. Haskell programs may contain patterns to match against for splitting the expressions. We did not follow these matches and count references only for the original terms, not for their subexpressions. We identified 25 different types and type schemes which implies that the *complexity* of this program is 25. Compared to the normalized lines of code (64), their ratio is ab. 0.39 types introduced per line, which is reasonable since Haskell programs are rather compact.

[1] Effective Lines of Code.

[2] For the complete sources see `http://www.haskell.org/haskellwiki/Haskell_Quiz/Astar`.

The *distribution* (Eq. 2.11) of the types is shown in Figure 2. It also shows that the *important types* are `Point`, `Map`, `Bool`, and α with different constraints. Polymorphic types like α refer to the number of names with generic type employed in the program, their values might be considered uncertainty in the measurement. However, it is enough to determine how the different polymorphic data types (`Data.Map.Map`, `Data.Set.Set`, lists, and monads) are used. High probability values for `Point` and `Map` clearly support their existence. Type `Bool` is also important because of the frequent use of Boolean conditions. In summary, we can say that the program contains generic components (the `astar` function) which can be used in other similar programs, while it uses lists with many types, `Map`s, `Set`s, and its own `PriorityQueue`.

σ	O_σ	D_σ
α	17	0.104
Point*	16	0.098
Map*	15	0.092
$\alpha^{Num,Eq,Show}$	13	0.079
Bool	11	0.067
[α]	11	0.067
[Int]	10	0.061
α^{Num}	10	0.061
[Point*]	7	0.043
Set α	6	0.037
PriorityQueue* α^{Ord} β	6	0.037
[String]	5	0.030
Int	5	0.030

σ	O_σ	D_σ
$\alpha^{MonadPlus}$	4	0.026
α^{Ord}	4	0.026
$\alpha^{Ord,Num}$	4	0.026
((α^{Ord},[β]),PriorityQueue* α^{Ord} [β])	4	0.026
String	3	0.018
Maybe α	2	0.012
[α^{Ord}]	2	0.012
Map α	2	0.012
α^{Monad}	2	0.012
$\alpha^{Eq,Ord}$	1	0.006
Char	1	0.006
[(α,β)]	1	0.006
IO α	1	0.006

Figure 2: Detailed type statistics for the sample program. Names defined in the program are superscripted by stars, all the others are pre-defined. Constraints (type classes) also appear as superscripts.

4. Related Work

To our knowledge, there has been only a few researches on the subject of measuring complexity of programs written in a functional programming language. In general, we believe there is no other known software measurement method based on types, which implies we might have managed to add a new factor for evaluating software quality.

Thesis of Klaas van de Berg [1] can be considered the first publication which takes functional programming languages into account in software metrication, using Miranda[3] which is similar to today's popular language in the area, Haskell. Research of Chris Ryder on software measurement for functional programming [2] [3] aims to define metrics for software engineering purposes to support refactoring together with visualization techniques to explore further relationships between the metric values and source code. His work is an extensive study on evaluation and validation of many possible ways for metrics on functional, especially Haskell programs. Király and Kitlei have a recent proposal [4] for Erlang on how to measure

[3]Miranda is trademark of Software Research, Ltd.

structural complexity of functional programs. Compared to their approach, we ignore details regarding style and size of the syntax, since we believe that types can describe these factors. Erlang uses dynamic typing which would make hard for us to adapt our approach, dependability on static type information is a limitation of our current model.

5. Conclusions

In contrast to the common interests and techniques, importing concepts from statistics in the definition of metrics helps to abstract away from the pure syntactical details. It can give us a deeper insight on how a program is constructed rather than spotting out the typical programming mistakes, like using too many parameters for functions. Chasing the flow of types via generating distributions can quickly and easily present a potential evaluation of programs on a higher level. This paper is only an initial discussion of this presumably novel approach, and many details are still left for future work: Implement the described measurement methods, fine-tune the implementation details and evaluate the results to fit well with intuitions of experienced programmers. It should be also investigated how it can be used in conjunction with refactoring.

Finally we would like to thank Emil Axelsson and Josef Svenningsson from Chalmers University of Technology, and Mátyás Barczy from University of Debrecen who contributed to the development of this paper a lot with their invaluable comments, and Dan Doel for his nice Haskell example. We would like to acknowledge the financial support of Programul Operaţional Sectorial Dezvoltarea Resurselor Umane 2007-2013 (POSDRU/6/1.5/S/3/2008, Romania) and TÉT AT-10/2008: Verified and Certified Software Components (Hungarian-Austrian Bilateral Scientific and Technological Cooperation).

References

[1] Van Den Berg, K. *Software Measurement and Functional Programming*, PhD thesis, Cip-data Koninklijke Bibliotheek, University of Twente, Enschede (June 1995)

[2] Ryder, C. *Software Measurement for Functional Programming*, PhD thesis, Computing Lab, University of Kent, Canterbury, UK (August 2004)

[3] Ryder, C., Thompson, S. *Software Metrics: Measuring Haskell*, In Marko van Eekelen and Kevin Hammond (eds.), Trends in Functional Programming, pp. 31–46 (September 2005)

[4] Király, R., Kitlei, R., Horváth, Z. *Structural Complexity Metrics for Functional Programming Code*, 8^{th} International Conference on Applied Informatics (January 2010)