CTy: a Haskell DSL for Specifying and Generating Combinatoric Test-cases

I.S.W.B. Prasetya

J. Amorim

T.E.J. Vos

A. Baars

Technical Report UU-CS-2011-005.
March 2011

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

CTy: a Haskell DSL for Specifying and Generating Combinatoric Test-cases

I.S.W.B. Prasetya* J. Amorim T.E.J. Vos† A. Baars‡

Abstract.
The Classification Tree Method (CTM) is a popular approach in functional testing. It allows testers to systematically partition the input domains of a target program, and specifies the combinations they want. This paper presents an implementation of CTM as a domain specific language (DSL) embedded in a functional language Haskell. Such an implementation is lean, but very powerful. It furthermore gives the testers first class access to all features of Haskell, e.g. clean syntax, lazy evaluation, and higher order functions.

1 Introduction

The Classification Tree Method (CTM) is a combinatoric testing approach proposed by Grochtmann and Grimm [3]. Imagine we have a program P to test. To help in deciding which test-cases we will need, we first construct a 'classification tree'. Such a tree specifies how the domain of each parameter of P is abstractly and hierarchically split into partitions. The lowest level partitions are called 'classes'; each is represented by a single concrete test-value. Each combination of classes (where each parameter of P is represented exactly once) abstractly represents a test-case. So, the classification tree defines the maximum space of test-cases that we are interested in. However, if the input domains of P are complex then this space is quite large. So, it is not practical to just do all test-cases in the space. There are tools like EXTRACT [9], ADDICT[1], or CTE-XL [5] that implement CTM with visual editors like in Figure 1 to allow testers to visually create a classification tree, and select the class-combinations they want to use as test-cases. A 'concretization' phase then turns these combinations into concrete test-cases (that will actually test P). In principle we can just

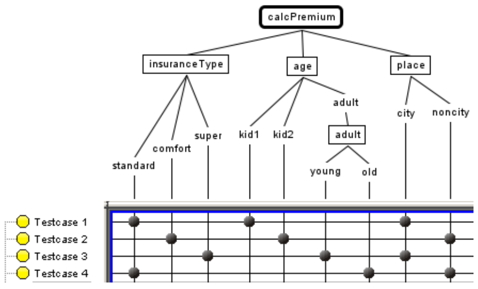

Figure 1. *A classification tree, graphically depicted in the tool CTE-XL. Below it is a so-called combination table, depicting various test-cases, and how each test-case is constructed by combining classes from the tree.*

manually list down the combinations we want. But this is not very reliable, since we may miss some important combinations. So, in e.g. CTE-XL people use combination rules to declaratively specify the combinations they want. The combinations are then generated. The CTM approach can be used in both white box or black box testing, and has been successfully applied in various industrial projects [5].

We will presents an implementation of CTM as a small Domain Specific Language (DSL). The DSL is called CTy, *embedded* in the functional language Haskell. We will have to drop the luxury of a visual editor, but we get something else in return. Haskell is a very good host language for embedding DSLs. Successful examples include DSLs for writing financial contracts [4], parsers [8], and animation [6]. DSLs in Haskell get native access to its features, e.g. clean syntax, lazy evaluation, polymorphism, and higher order functions. For us it means that we can write powerful expressions to specify and process the combinations of the classes that we want to include in our test-suite. As a concrete example, we will also discuss how CTy can be used to combinatorically test a web service.

*Dept. of Inf. and Comp. Sciences, Utrecht Univ. Email: wishnu@cs.uu.nl

†Dept. de Sistemas Informáticos y Comp., Univ. Polit. de Valencia. Email: tvos@dsic.upv.es

‡Univ. de Valencia, as above. Email: abaars@pros.upv.es

```
ctree1 = tree [insuranceType,age,place]
   where
   insuranceType = "insType" <== [
     "standard" %%= (frag Standard, (0.5::Float)),
     "comfort"  %%= (frag Comfort, 0.35) ,
     "super"    %%= (frag Super,  0.15) ]

   age = "age" <== [
     "kid1" %%= (Int32 4, (0.2::Float)),
     "kid2" %%= (Int32 16, 0.2),
     adult ]
     where
     adult = "adult" <== [
       "young" %%= (Int32 20, (0.4::Float)),
       "old"   %%= (Int32 66, 0.2) ]

   place = "place" <== [
       "city"    %%= (StringN 100 "Delft",  (0.6::Float)),
       "noncity" %%= (StringN 100 "Achterhoek", 0.4)  ]
```

Figure 2. *A classification tree in Haskell/CTy.*

2 Specifying input domain

Imagine as an example the function below for calculating the premium's price of some insurance; suppose we want to test it.

`calcPremium(itype, age, city)`

where `itype` is the type of insurace (e.g. standard,comfort,super).

Rather than just trying arbitrary test-cases, recall that in the CTM approach we first divide the input domains of `calcPremium` into logical partitions. This partitioning is specified with a tree structure called *classification tree*. This tree defines the maximum space of various test-cases that we are going to try (but note that we do not typically want to do them all). Figure 2 shows how such a tree is expressed in CTy. E.g. it divides the parameter `age` into three partitions: `kid1`, `kid2`, and `adult`. The last one is partitioned further into `young` and `old`.

Partitions of the lowest level (which are not further partitioned) are special, and are called *classes*. Each class logically specifies a subdomain. However, for testing purpose all values in this subdomain are assumed to be equivalent. Under this assumption we can represent a class with just one test-value. This is the value we will use when the class is later used to form a test-case. We construct a class using the following notations:

"`kid1`" %= `Int32` 4

This constructs a class named `kid1`, with the value 4 as the concrete test-value representing it. Each class is assumed to get a unique name. Test-values are represented by a separate type `TestVal` that supports a number of constructors. E.g. `Int32` in the above example represents a 32-bits two's complement integer. In Haskell itself we can have integers of any size, but because we may target a different language then it is necessary to specify the specific range that we want. The constructor `StringN` n s is used to construct a string (s) whose length is at most n.

With a slightly different notation, we can also decorate a class with additional information:

"`city`" %%= (`StringN` 100 "`Delft`", 0.4)

This constructs a class named `city`, with "`Delft`" as the test-value representing it, and decorates the class with the value 0.4, representing the importance/weigth of the class. In principle, we can put a decoration of any type though.

Non-class and non-top-level partitions are represented by the type `Partition`, and are constructed using the following notation:

"`adult`" $\Longleftarrow$ [$p_1, ..., p_n$]

This constructs a partition named `adult`, with all those p_i's as subpartitions.

A top-level partition specifies the whole domain of *a parameter* of our target function. In the tradition of CTM this is called *category*. So, any classification tree of `calcPremium` will consist of three categories: one for the parameter `itype`, one for `age`, and one for `city`. We will represent a category with its own type `Category`, but use the same $\Longleftarrow$ notation as above to construct it.

Finally, a classification tree itself is represented by the type $\texttt{CTree}_\alpha$, and is constructed with notation like this:

`tree` $[C_1, ..., C_n]$

where C_i is a category. See also the example in Figure 2.

The α in the type $\texttt{CTree}_\alpha$ is a type parameter representing the the type of test-values (that are sitting in the classes in the tree) and their decorations, if they have those. E.g. if we have no decoration then α is just `TestVal`. If we use a number as a 'weight' decoration then α is (`TestVal`, `Float`).

The types `Partition` and `Category` we mentioned before should actually be parameterized by α. The same applies to the types `TestCase` and `Suite` introduced later. However, in favor for readability, in the sequel we will however just omit this α.

3 Test-case and test-suite

A *test-case* should specify two components: (1) an input for the target program f that we want to test, and (2) the expected result, also called *oracle*. In our work, we will only address the problem of generating the inputs. Oracles will be left as place holders. If an executable specification for f exists, these place holders can be replaced by calls to this specification. Without a specification oracles cannot in principle be automatically generated.

Consider again the `calcPremium` example. Since each class contains a test-value, a test-case can be represented by a list t of classes where every category/parameter of `calcPremium` is represented in t by exactly one class. E.g. the list $[city, standard, kid1]$ would represent a test-case for `calcPremium`. Concretely, this corresponds a test-case whose input is this:

```
calcPremium(Standard, 4, "Delft")
```

We will however call any list of classes a 'test-case'. Those that satisfy the above criterion will be called 'complete' test-cases. We introduce these types to represent test-case and the corresponding notion of test suite:

```
type TestCase  =  [Class]
type Suite     =  [TestCase]
```

Below we will define a number of auxiliary notations for the purpose of explaining our concepts. Since they are however not exposed to CTy's users, we refrain from presenting them in terms of their actual Haskell implementation, and instead do so in plain mathematical notation.

If p is a class or a partition let $cat(p)$ denote the category to which p belongs. If P is a list or a set of classes or partitions, we define $cat(P) = \{cat(p) \mid p \in P\}$. Since a test-case is also a list of classes, we can use the same notation on it. If ϕ is a classification tree, we will write $cat(\phi)$ to denote the set of categories in ϕ.

A test-case t is *well-formed* if for any distinct $c, d \in t$, $cat(c) \neq cat(d)$. With respect to a given classification tree ϕ, t is *complete* if it is well-formed and furthermore $cat(t) = cat(\phi)$. A well-formed test-case that is not complete is called *partial*.

If p is a partition or a category, we write $class(p)$ to denote the classes that are descendants of p (as a tree). In other words, these are the classes that make up p. If P is a list or set of partitions, $class(P) = \cup\{class(p) \mid p \in P\}$. We will also write $c \in p$ to mean $c \in class(p)$. Similarly, we also write $c \in P$.

If t is a test-case, and γ be a set of categories. We write $t{\downarrow}\gamma = [c \mid c \in t \ \wedge \ cat(c) \in \gamma]$ to mean the subset of t restricted to γ. Analogously, $t{\downarrow}\bar{\gamma}$ is simply the complement of ${\downarrow}\gamma$, which is: $[c \mid c \in t \ \wedge \ c \notin t{\downarrow}\gamma]$.

Let s and t be test-cases. We write $s \supseteq t$ if as sets s subsumes t. They are equivalent, written as $s \equiv t$, if they subsume each other. The union and intersection of two suites are defined as follows:

$$\begin{array}{lcl} S \cup T & = & S +\!\!+ \; [\, t \mid t \in T,\ (\forall s \in S :: t \not\equiv s)\,] \\ S \cap T & = & [\, t \mid t \in T,\ (\exists s \in S :: t \equiv s)\,] \end{array}$$

4 Specifying combinations

The simplest way to generate a test suite from a classification tree is by using one of these functions:

```
allf  ::  CTree → Suite
allm  ::  CTree → Suite
```

The first produces the *full suite*, consisting of *all* possible test-cases induced by a given classification tree ϕ. The second one gives the *minimalistic suite*, which is the smallest suite such that every class in ϕ is covered/used at least once.

Recall that a test-case is essentially a combination over classes from different categories. The total number of combinations can explode very fast. So, for a function with complex input domains, we can expect its full suite to be too large, e.g. containing thousands of test-cases which in many situations is not feasible to be fully explored. On the other hand, the minimalistic suite is often too small to be adequate.

Inspired by CTE-XL [5] below we provide a simple language to write 'combination rules', so that we can declaratively specify the combinations that we want to cover. We will however use a deviated set of operators (than that of CTE-XL —see also Section 8).

The combination rules are implemented 'symbolic'. It means that they are represented as data in Haskell and have to be interpreted first to actually produce the corresponding suites. This allows us to check them first for their well-formedness, before interpreting them. This also allows us to more efficiently implement the complement operator. For further expressiveness, the generated test-suites can be further combined at Haskell's native level —we will return to this later.

A combination rule will be represented by the type `Rule`, which can be constructed in this way:

$$R \ ::= \ \texttt{incl}\, P \ \mid \ R \ op \ R \ \mid \ \texttt{neg}\, R$$

where P is a so-called 'monocat'. It is either single a category name or a list of partition names, such that $|cat(P)| = 1$. So, all partitions in P should belong to the same category. The *op* above is an operator from the following set:

$$|{+}|,\ |\&|,\ \&\&{*},\ \&\&{-}$$

Semantically, a combination rule R specifies a suite over the categories mentioned in R. We will define a function:

$$\texttt{rule} :: \texttt{Rule} \rightarrow \texttt{Suite}$$

that interprets a rule and produces the suite it specifies. This function is essentially our test-cases generator. R may indeed produce test-cases which are still partial —we will deal with this later.

A suite S is *well-formed* if it consists of well-formed test-cases, and for all $t, u \in S$, $cat(t) = cat(u)$ (all test-cases in S cover exactly the same set of categories). Two well-formed suites can be combined more efficiently since we do not have to keep checking whether two test-cases from the suites are actually compatible to be combined. Combination rules will be restricted (and checked) so that they only produce well-formed suites.

The simplest rule is of the form `incl` P. Since P is a monocat, all classes under P belong to the same category. Each is a candidate test-value for the category. Each is then lifted to a partial test-case, and then collected in a suite. So:

$$\texttt{rule}\,(\texttt{incl}\ P) \ =\ [\,[c] \mid c \in class(P)]$$

We discuss the definition of `rule`(`neg` R) later.

Next, this operator provides the union over the generated suites:

$$\texttt{rule}\,(R_1\ |{+}|\ R_2) \ =\ \texttt{rule}\ R_1\ \cup\ \texttt{rule}\ R_2$$

Similarly, we define intersection $|\&|$. To be well-formed, R_1 and R_2 above should specify the same set of categories.

If R_1 and R_2 are rules with no common categories, this operator construct the Cartesian product of their test-cases:

$$\texttt{rule}\,(R_1\ \&\&{*}\ R_2) \ =\ \texttt{rule}\ R_1\ \times\ \texttt{rule}\ R_2$$

$$S_2 \times S_2 \ =\ [\ s_1{+\!+}s_2 \mid s_1 \in S_1,\ s_2 \in S_2]$$

Notice that this operator is the binary version of `allf`.

The 'minus'-variant will combine R_1 and R_2 in a minimalistic way, analogous to `allm`:

$$\texttt{rule}\,(R_1\ \&\&{-}\ R_2) \ =\ \texttt{rule}\ R_1\ +\ \texttt{rule}\ R_2$$

$S_1 + S_2$ is defined as follows. Assuming $|S_2| < |S_1|$, we fill S_2 by duplicating some of its elements to make its size equal to S_1. We do the oppsite if $|S_2| > |S_1|$. Then the definition looks as below[1]:

$$S_1 + S_2 \ =\ [s_1{+\!+}t_1,\ \ldots\ , s_n{+\!+}t_n]$$

where $s_1, ..., s_n$ are elements of S_1, and $t_1, ..., t_n$ are elements of S_2.

Let N_1 and N_2 be the size of `rule` R_1 and `rule` R_2. Whereas $\&\&{*}$ produces a suite of size $N_1{*}N_2$, that of $\&\&{-}$ has the size of $max(N_1, N_2)$.

Notice that we can write nested rules and mix the full and the minimalistic variants of the operators. For example:

$$\texttt{r1} \ =\ \begin{cases} \texttt{incl}\ \texttt{"age"}\ \ \&\&{-}\ \ \texttt{incl}\ \texttt{"insType"} \\ |{+}| \\ \texttt{incl}\ [\texttt{"kid1"}]\ \ \&\&{*}\ \ \texttt{incl}\ \texttt{"insType"} \end{cases}$$

With respect to the classification tree in Figure 2, it specifies a suite of partial test-cases over the categories insurance-type and age. The first line minimally combines the two categories. The second line fully combines the category insurance-type and a single class `kid1` from the category age. The two suites are then merged. This gives us in total 6 test-cases, whereas simply fully combining the two categories will give 12 test-cases.

4.1 Derived operators

The above set of operators is complete. That is, we will be able to specify any subset of `allf`. Actually, we only need the `incl`, $|{+}|$, and $\&\&{*}$ operators, but just using these is not going to be convenient.

Boolean-like operators

For further convenience we can define a whole array of derived operators, e.g. as below, based on the full combinator $\&\&{*}$. Let P be a monocat, and R_1, R_2 be rules that have no common category:

$$\begin{array}{lcl}
\texttt{excl}\ P & = & \texttt{neg}(\texttt{incl}\ P) \\
R_1\ \texttt{or}_\texttt{f}\ R_2 & = & \ (R_1\ \&\&{*}\ R_2) \\
 & & \quad |{+}|\ (\texttt{neg}\ R_1\ \ \&\&{*}\ \ R_2) \\
 & & \quad |{+}|\ (R_1\ \ \&\&{*}\ \ \texttt{neg}\ R_2) \\
R_1\ \texttt{xor}_\texttt{f}\ R_2 & = & (R_1\ \ \&\&{*}\ \ \texttt{neg}\ R_2)\ |{+}|\ (\texttt{neg}\ R_1\ \&\&{*}\ R_2) \\
R_1\ \texttt{equ}_\texttt{f}\ R_2 & = & (R_1\ \&\&{*}\ R_2)\ |{+}|\ (\texttt{neg}\ R_1\ \&\&{*}\ \texttt{neg}\ R_2) \\
R_1\ \texttt{imp}_\texttt{f}\ R_2 & = & \texttt{neg}\ R_1\ \texttt{or}_\texttt{f}\ R_2
\end{array}$$

Analogously we can have the minimalistic version of those operators, which are based on the combinator $\&\&{-}$.

We use the names of the Boolean operators because intuitively our operators have analogous meaning. But

[1]It is essetially a zip; and is implemented lazily.

note that they do *not* form a Boolean algebra. E.g. &&∗ and $\texttt{or}_\texttt{f}$, as well as their minimalistic counterparts, are still commutative and associative, but they are not idempotent. E.g. R &&∗ R is not allowed in our algebra, since it does not produce a well-formed test suite. We also do not have distributivity. E.g.:

$$R_1 \;\&\&\!* \;(R_2 \;\texttt{or}_\texttt{f}\; R_3) \quad\neq\quad (R_1 \;\&\&\!*\; R_2) \;\texttt{or}_\texttt{f}\; (R_1 \;\&\&\!*\; R_3)$$

The right formula will not produce a well-formed suite.

k-wise combination

We can also define an operator $\texttt{wise}_2\ Z$ where Z is a list of rules with no common category. This constructs a new rule that combines the rules in Z, such that every possible pair from Z is fully combined, and then minimally combined with the rest. More precisely, for every pair of rules $R_i, R_j \in Z$, it first constructs the full combination $T_{i,j} = R_i$ &&∗ R_j, and then $T_{i,j}$ is minimally combined with the rules from $Z/\{R_i, R_j\}$ to obtain $U_{i,j}$. Then we take the union over all these $U_{i,j}$'s.

For a Z consisting of just three rules, this will give us:

$$\begin{array}{lcl} \texttt{wise}_2\,[R_1,R_2,R_3] & = & ((R_1 \;\&\&\!*\; R_2)\;\&\&\!-\; R_3) \\ & & |{+}|\;((R_2 \;\&\&\!*\; R_3)\;\&\&\!-\; R_1) \\ & & |{+}|\;((R_3 \;\&\&\!*\; R_1)\;\&\&\!-\; R_2) \end{array}$$

The definition can be further generalized to $\texttt{wise}_k$. This corresponds to a generalization of the k-wise-combination operator of CTE-XL [5]; the above is more general as it operates on rules rather than just on partitions as in CTE-XL.

4.2 Implementing Complement

The complement operator `neg` R should give us the 'complement' of R's suite. Consider `neg` (`incl` P_1 &&∗ `incl` P_2). Let C_i be the catagory of P_i. In this case the 'complement' is just the full space $C_1 \times C_2$ substracted with the suite of `incl` P_1 &&∗ `incl` P_2. Now, what is the complement of the one below?

$$(\texttt{incl}\ P_1 \;\&\&\!*\; \texttt{incl}\ P_2)\;\&\&\!-\;\texttt{incl}\ P_3$$

We decide that it is not sensical to get it by just substracting from the fully combined space $C_1 \times C_2 \times C_3$. Instead, we will susbtract it from $(C_1 \times C_2) + C_3$.

Calculating complement by substracting from a certain maximum space implies that we first have to construct this maximum space. Due to combinatorical nature of &&∗, this space can be quite large. The effort is wasted if we turn out to throw away most combinations from the space.

Because of the above two issues we calculate complement indirectly by first applying some symbolic rewriting to our rule.

Let us now impose that when interpreted over `Suite`, the algebras $(\texttt{neg}, |{+}|, |\&|)$ and $(\texttt{neg}, \&\&*, \&\&-)$ are to satisfy de Morgan equations. The equations will allow us to normalize a rule to its a negative normal form, obtained by pushing occurrences of `neg` as far as possible toward the 'atoms' of the formula.

The full procedure is shown below, by the function `nnf`. The various cases should be interpreted in the cascade-mode. No definition is provided for $\texttt{or}_\texttt{f}$ and $\texttt{or}_\texttt{m}$ since they are defined in terms of &&∗ and &&−.

$$\begin{array}{lcl}
\texttt{nnf}\ (\texttt{neg}(\texttt{neg}\ R)) & = & \texttt{nnf}\ R \\
\texttt{nnf}\ (\texttt{neg}(R_1 \;\&\&\!*\; R_2)) & = & \texttt{nnf}\ (\texttt{neg}\ R_1\ \texttt{or}_\texttt{f}\ \texttt{neg}\ R_2) \\
\texttt{nnf}\ (\texttt{neg}(R_1 \;\&\&\!-\; R_2)) & = & \texttt{nnf}\ (\texttt{neg}\ R_1\ \texttt{or}_\texttt{m}\ \texttt{neg}\ R_2) \\
\texttt{nnf}\ (\texttt{neg}(R_1\ |\&|\ R_2)) & = & \texttt{nnf}(\texttt{neg}\ R_1)\ |{+}|\ \texttt{nnf}(\texttt{neg}\ R_2) \\
\texttt{nnf}\ (\texttt{neg}(R_1\ |{+}|\ R_2)) & = & \texttt{nnf}(\texttt{neg}\ R_1)\ |\&|\ \texttt{nnf}(\texttt{neg}\ R_2) \\
\texttt{nnf}\ (\texttt{neg}\ a) & = & \texttt{neg}\ a \\
\texttt{nnf}\ (R_1 \;\&\&\!*\; R_2) & = & \texttt{nnf}\ R_1\ \;\&\&\!*\;\ \texttt{nnf}\ R_2 \\
\texttt{nnf}\ (R_1 \;\&\&\!-\; R_2) & = & \texttt{nnf}\ R_1\ \;\&\&\!-\;\ \texttt{nnf}\ R_2 \\
\texttt{nnf}\ (R_1\ |{+}|\ R_2) & = & \texttt{nnf}\ R_1\ \ |{+}|\ \ \texttt{nnf}\ R_2 \\
\texttt{nnf}\ (R_1\ |\&|\ R_2) & = & \texttt{nnf}\ R_1\ \ |\&|\ \ \texttt{nnf}\ R_2 \\
\texttt{nnf}\ a & = & a
\end{array}$$

To produce the suite of `neg` R we first normalize the formula. Basically, the normalization does some symbolical pre-calculation on the parts of the maximum space that we need to construct, so that we do not have to do the more expensive set complement. The only thing left is to define `neg` at the atomic level. Let C be a single category, and P be a (monocat) list of partitions:

$$\begin{array}{lcl}
\texttt{rule}\ (\texttt{neg}(\texttt{incl}\ C)) & = & [\,] \\
\texttt{rule}\ (\texttt{neg}\ (\texttt{incl}\ P)) & = & [\,[c] \mid c \in \mathit{class}(D)/\mathit{class}(P)\,]
\end{array}$$

where $D = \mathit{cat}(P)$.

5 Combining test-suites

Applying the function `rule` interprets a given combination rule R to produce the corresponding test-suite. Since suites are just lists, we can further combine and process them with any Haskell function of a compatible type. This can give us a lot of expressiveness. However, because we now directly combine suites (rather than rules), we will not check their well-formedness during the combinations. Doing so would be inefficient. We will just filter the final suite to throw away ill-formed test-cases. Since such filtering can make the meaning of suite operators less predictable, the user is now responsible to make sure that that does not happen (by not producing intermediate ill-formed test-cases).

We can combine two suites as follows:

$$S_1 \; op \; S_2$$

where *op* is one of the following operators:

$$|{+}|,\; |\&|,\; \text{`}\texttt{suchthat}\text{`},\; \text{`}\texttt{except}\text{`},\; |{\ast}|$$

The first two are just the union and intersection over suites (the $\cup$ and $\cap$ we defined before). The arguments of these two operators should be suites over the same categories.

S_1 ‘`suchthat`‘ S_2 specifies a subset of S_1 consisting of those test-cases t that subsumes some test-case in S_2:

$$S_1 \text{ `}\texttt{suchthat}\text{` } S_2 \;=\; [\; s \mid s \in S_1,\; (\exists t \in S_2 :: s \supseteq t) \;]$$

Then we can define the negative counterpart of this:

$$S_1 \text{ `}\texttt{except}\text{` } S_2 \;=\; S_1 \;/\; (S_1 \text{ `}\texttt{suchthat}\text{` } S_2)$$

$S_1 \; |{\ast}| \; S_2$ constructs the 'relational-joint' on S_1 and S_2. Let C_1 and C_2 be the set of categories of S_1 respectively S_2, $C = C_1 \cup C_2$, and $D = C_1 \cap C_2$ (so D is the set of their common of categories). The operator is defined as below:

$$\begin{array}{l} S_1 \; |{\ast}| \; S_2 \\ = \\ [\; s \mathbin{+\!\!+} t{\downarrow}(C/D) \mid s \in S_1,\; t \in S_2,\; s{\downarrow}D = t{\downarrow}D \;] \end{array}$$

For example, we can now write like this:

$$\begin{array}{l} \texttt{rule}\;(\texttt{incl}\;[\text{"}\texttt{kid1}\text{"}]\;\&\&{\ast}\;\texttt{incl}\;\text{"}\texttt{insType}\text{"}) \\ |{\ast}| \\ \texttt{rule}\;(\texttt{incl}\;\text{"}\texttt{place}\text{"}\;\&\&{\ast}\;\texttt{excl}\;[\text{"}\texttt{standard}\text{"}]) \end{array}$$

The first line interprets a combination rule to produce a suite that combines all possible insurance types with the class `kid1`. The second line gives a suite that combines all possible `place` with non-standard insurance-type. The operator $|{\ast}|$ will join these two suites, by picking the combinations of the test-cases from both suites that have a common insurance type.

Padding test-suite

Let f be the target function to test, and τ is the classification tree we want to use. A suite expression such as discussed above may not produce complete test-cases if the underlying combination rules in the expression do not mention all categories of τ. Before we can turn them to concrete test-cases for f, we need to pad/extend these test-cases to make them complete.

Let S be a test-suite. Suppose $\Delta = [D_1, ..., D_N]$ is the set of categories which are still missing from the test-cases in S. We can pad these test-cases by either fully or minimally combining them with Δ. This gives the following padding functions:

$$\begin{array}{lcl} \texttt{paddindf}_\tau\; S & = & S \times A_1 \times ... \times A_N \\ \texttt{paddingm}_\tau\; S & = & S + A_1 + ... + A_N \end{array}$$

where $A_k = \texttt{rule}\;(\texttt{incl}\;D_k)$; *times* and $+$ are operators on test-suites as defined before.

5.1 Priority-based selection

We can also sort a suite and then take e.g. just the first k test-cases. This is useful if after specifying the combination rule as explained above we still end up with a very large suite. Haskell can provide us with an easy and flexible way to program this.

Remember that a class can be decorated. We can use this to express the 'weight' of the class. Let W be a function that can retrieve this weight info from a class. We can generically define the corresponding notion of the weight of a test-case as below. Let $t = [c_1, ..., c_n]$ be a test-case.

$$\texttt{weight}_{(\oplus,W)}\; t \;=\; W\, c_1 \;\oplus\; ... \;\oplus\; W\, c_n$$

Let D be the co-domain of W. If we also provide a total ordering $\prec$ over D, we can define a function in Haskell to ascendingly sort any suite based on the value of $\texttt{weight}_{(\oplus,W)}$ and $\prec$ as ordering:

$$\texttt{ssort}_{(\prec,\oplus,W)}\; S \;=\; S'$$

where S' is the sorted version of S as explained above. Now we can for example define these:

$$\begin{array}{lcl} \texttt{wsort} & = & \texttt{ssort}_{(\geq,+,\texttt{snd})} \\ \texttt{psort} & = & \texttt{ssort}_{(\leq,*,\texttt{snd})} \end{array}$$

The $\geq, \leq, +, *$ above refer to the corresponding numerical relations and operators.

The first function above induces the concept of test-case weight that sums the weight of its classes. If the weight of a class indicates its importance, `wsort` will make important test-cases to float to the top.

If the weight is used to express the estimated likelihood that a class occurs in practice, `psort` induces the concept of test-case weight that corresponds to the estimated likelihood that the combination of the classes that occur in the test-case will actually occur in practice. It will sort a suite so that rare cases will float to the top.

We can now write this expression in Haskell:

```
suite = tau $$ rule (incl ["kid1"] &&* incl "insType")
              >>= paddingm
              >>= psort
              >>= lift_ . take 3
```

This first line produces the base suite (that combines `kid1` with all types of insurance). Then we pad the suite, then we sort it according to its likelihood, then we take the three rarest ones.

Functions like `paddingm` and `psort` have in principle the type `Suite`→`Suite`. But in some cases, e.g. as in `paddingm` it also needs access to the used classification tree. We can indeed just explicitly pass the tree, but then we will have to pass it again every time we need it. To make the notation nicer we implement these functions, and also the suite operators from the previous section, on top a so-called reader monad. In a functional language this allows us to store a value to a context. The functions that need them can implicitly inspect this context, which leads to a cleaner notation. As the reader can see above, we only pass the tree `tau` once to the whole expression.

Because we use monad, pipe-line processing such as what we do in the above example is not done using the usual function composition, but with the $\gg\!=$ operator, which is the monadic pipe-lining operator.

We can also apply an ordinary list function to process a suite (since suite is also a list); for example as in the application of `take` k in the above example (to take the first k elements from a list). Lifting is needed to pull it to the monad level.

6 Generating concrete suite

Let f be the target function to test. The test-cases as we discuss so far cannot be directly used for testing. They must first be passed to a *concretization* phase, to be turned into concrete test-cases. E.g. they have to be expressed in the programming language that we will use to actually execute the test-cases. Currently we only provide concretization targeting Haskell itself. In principle it is not too hard to customize it to target other languages.

An example of the result of concretization is shown below. It comes from a CTy suite of two test-cases:

```
module CTy.Example0Test1 where

import Test.HUnit
import CTy.Example0

-- start generated code

tc1  =  TestCase (
   assertEqual  "** Case standard, kid1, city"
   (calcPremium (Standard) (4) ("Amsterdam"))
   undefined )

tc2  =  TestCase (
   assertEqual  "** Case standard, kid1, noncity"
   (calcPremium  (Standard)  (4)  ("Achterhoek"))
   undefined )

-- to run the whole suite do: runTestTT suite

suite  =  TestList [tc1, tc2]

-- end generated code
```

This generated code is syntax-correct, but it is still incomplete. The reader can see that there are some `undefined`s in the code. These are 'place holders' for the expected values (the test-oracles) for each test-case. As pointed out before, we cannot generate them. So the tester will have to manually fill these in. After that, the suite is ready to run.

Performance

CTyis quite fast. It can generate a suite in a fraction of a second, or a bit longer if the suite is large. See the table below. The 2nd row gives the time needed to generate the suites of various sizes. In practice we seldom want to generate a suite of e.g. 30 thousands test-cases. So, next we sort them and select just few test-cases. The third row gives the total time of generating a suite, then sorting it, then selecting the first 100 test-cases.

$\lvert suite \rvert$	1K	30K	290K
time-1	0.0	2.0	33.8
time-2 (sort)	0.2	9.4	90.0

We use a 32-bits PC with AMD dual-core, 1.9 GHz, 3GB RAM, and Win-Vista. We run the examples in the interpreter-mode using GHCi.

In comparison, CTE-XL takes over 1.5 hour to produce a selection of 12K test-cases. It probably spends most of this time in producing the GUI elements needed for showing the test-cases, which we do not have to do here. So this comparison should not be taken as fair.

7 A bigger example: regressing Google Geocoding API

Google Geocoding API is a web-service provided by Google to calculate the geographic coordinate of a place. The place is described by its address, e.g. "padualaan 14, utrecht, netherlands". A coordinate is given in latitude and longitude e.g. $(52.08503, 5.1704884)$. It is a RESTful service; we can query it by sending a HTTP request e.g. like this (for the above example):

```
http://maps.googleapis.com/maps/api/geocode/xml?
address=padualaan+14+utrecht+netherlands
```

If the address is recognized, an XML containing its coordinate is returned. It it matches multiple places, the XML will contain multiple coordinates as well.

The service can handle different ordering of the components of an address (e.g. if we specify the town before the street), or if we mangle some component a little bit (e.g. 'utrecth' instead of 'utrecht'), or even omit the component. Note: there are cases where it seems to have problems.

Suppose we have a set P of important places/addresses (e.g. hospitals in your town). We want to make sure that over time Geocoding API always give consistent coordinates of those places. We will do this by regularly performing a regression test on the service. We use CTy to specify the test suite to use. We will construct one generic suite which can be instantiated for each address in P.

Let a be a full and correct address. We will construct a test-suite S_a where each test-case is obtained by permuting and mangling a's components in various ways. S_a is then used for regression with respect to a. This is done in two stages.

The first stage is a *manual approval stage*. For every test-case in S_a we verify if Geocoding API's answer is 'correct'. Every test-case is basically a query using some variant of a, sent to the API. The answer is correct if either the API says it doesn't recognize the query, or else if the coordinates returned by the API contains a correct one.

Once all test-cases in S_a are approved, we will store the API's answer of each test-case t and uses it as the oracle for t. Then we can proceed to the *regression stage* where we regularly re-run the S_a; it will report an error if one query gives a different answer than the stored answer (the one we approved earlier). We will do this for every address a in our set P.

The API is tested through the test-interface below. The interface takes care e.g. the HTTP connection with the API, formatting a query before it is sent to the API, and parsing the API's XML answer. Additionally it also: (1) mangles and permutes the given address according to the given mangling and order specifiers, and (2) checks the answer against the oracle.

ggTI *address coM ciM stM nrM o*

The function returns true if the result is equal to the oracle; else false. The parameter *address* is specified by a tuple $(country, city, street, streetnr)$. The next four parameters are mangling-specifiers for respectively country, city, street, and street-number. Finally o specifies the order/permutation of the address' components in the concrete query to the Geocoding API (e.g. padualaan+14+utrecht, or utrecht+padualaan+14).

```
ct a = tree [address,countryMode,cityMode,
             streetMode,numberMode,order]
  where
  address     = "Address" <== ["Address" %= addr_ a]

  countryMode = "CountryMode" <==
      [ "CoCorrect"  %= mode_ Correct,
        "CoSwapped2" %= mode_ Swapped2,
        "CoDropLast" %= mode_ DropLast,
        "CoEmpty"    %= mode_ Empty ]

  cityMode = "CityMode" <==
      [ "CiCorrect"  %= mode_ Correct,
        "CiSwapped2" %= mode_ Swapped2,
        "CiDropLast" %= mode_ DropLast,
        "CiEmpty"    %= mode_ Empty ]

  streetMode = "StreetMode" <==
      [ "StCorrect"  %= mode_ Correct,
        "StSwapped2" %= mode_ Swapped2,
        "StDropLast" %= mode_ DropLast ]

  numberMode = "NrMode" <==
      [ "NrCorrect" %= mode_ NCorrect,
        "NrEmpty"   %= mode_ NEmpty ]

  order = "ArgsOrder" <==
      [ "CommonOrder" %= mode_ CommonO,
        "CoCiSN"      %= mode_ CoCiSN,
        "NSCiCo"      %= mode_ NSCiCo,
        "CoNSCi"      %= mode_ CoNSCi ]
```

Figure 3. The classification tree for Google Geocoding API

We use a classification tree to express various ways to mangle each address component, and various permutations we want to consider; this is shown in Figure 3. Roughly for each address component we have four mangling modes: no-mangling, swapping the last two letters, dropping the last letter, and deleting the whole component. For street numbers, only the first and the last modes are used.

The tree is specified with respect to a specific address a. Given such an address, the tree induces in total 384 test-cases. This is too many for the manual approval stage. On the other hand, the minimal suite will just contain 4 test-case, which does not feel to be adequate. So, we use a rule to select the combinations that we think more important to cover:

```
r1 =  incl ["StCorrect"]
           &&* incl "ArgsOrder" &&* incl "CityMode"
      |+|
      neg (incl ["StCorrect"])
           &&- incl "ArgsOrder" &&- incl "CityMode"
```

Below we show how we construct the final test-suite. Basically it turns the rule above to a test-suite, then we apply minimalistic padding. The we do few other format-related things, e.g. fixing the order of the classes in each test-case.

```
ggSuite a = ct a $$ rule r1
                    >>= paddingm
                    >>= fixorder
                    >>= lift_ . strip
```

E.g. this suite will *fully* combine an un-mangled street-name with the four permutations included in the classification tree and with all mangled variations of the city-name. The resulting combinations are then *minmally* combined with the other parameters.

The above suite gives 20 test-cases. It is then given to a concretization function to generate the corresponding concrete test-suite.

8 Related work

Other tools implementing CTM are EXTRACT [9], ADDICT [1], and CTE-XL. The last one, by Lehmann and Wegener [5], is probably the most prominent one. CTE-XL also provides a set of operators to declaratively select a suite. The selection is expressed by two kinds of rules: 'combination rules' and 'dependency rules'. The first are used to specify which base-suite to generate, the second specify constraints on the suite and are used to filter the base-suite. Just as in CTy, any suite can be generated with CTE-XL. However the used operators are different. CTE-XL does not have the `neg`, $|+|$, and $|\&|$ in its combination rules. Having these operators allow us to provide the derived Boolean-like operators $\texttt{imp}_\texttt{f}$, $\texttt{imp}_\texttt{m}$, $\texttt{xor}_\texttt{f}$, $\texttt{xor}_\texttt{m}$, etc. We can also define the k-wise combination operator as a derived operator. CTE-XL provide Boolean operators to express dependency rules. CTy does not distinguish between the two kinds of rules. It uses the suite-level operators `suchthat` and `except` to express constraints, where any rule or suite can be used as a constraint.

Yu et al [9] proposed an extension to CTM where classes can be annotated with tags (also called 'properties'). Selector expressions are then written to specify that e.g. a class c can only occur in a test case t if t contains (or does not contain) a certain set of tags. The expresiveness of tags is equal to CTE-XL's dependency rules. A class can also be annotated with weight reflecting its importance. The generated test-cases are then sorted so that 'heavy' test-cases will be exercised first. We have seen that in CTy we can also sort and prioritize test-cases. But we also we more flexibility, e.g.:

$$\begin{array}{l}(suite_1 \ggg\!= sort_1)\\ \texttt{`suchthat`}\\ (\texttt{rule}\ R \ggg\!= sort_2 \ggg\!= lift_{\_}\ .\ \texttt{take}\ 3)\end{array}$$

This combines two suites sorted with different criteria. We are also not limited to sorting according to additive weight; we can basically define any sorting criterion.

Conrad proposed CTM_{EMB} as an extension of CTM for continuous or hybrid embedded systems; he later improved it together with Krupp in [2]. When testing an embedded system the sequences with which the test-cases are exercised may matter. That is, SUT's reaction to a test-case t depends not only on t, but also on the preceding test-cases. So, a test-case here is actually a *sequence* of ordinary CTM's test-cases (called here *test-steps*). Since the SUT is also continuous, we may have to specify how SUT's inputs evolve between two subsequent test-steps (e.g. they step, ramp, or follow a sinuous curve). This can be expressed in CTM_{EMB}. We did not explore this direction. It is conceivable that in this area the sequences tend to be very specific and delicate that visualization is important; then CTy is not an appropriate tool to generate them.

Singh et al proposed to associate formal predicates expressed in Z to the classes in a classification tree [7]. So, test-case t induces a formula f_t that logically identifies it, which is just the conjunction of the formulas associated to t's classes. Modulo decidability, this allows non-sensical test-cases to be filtered out by checking if the conjunction of f_t and SUT's pre-condition is satisfiable. We can also use a formula to specify a partition, and implicitly use e.g. its DNF-clauses as its classes. We did not explore into this direction. But in principle people can do this in Haskell. CTy itself allows any type to be used to represent a test-value, including a formula. Haskell has a number of theorem prover libraries. Some provide satisfiability checking. Moreover, they provide a symbolic representation of formulas, which means that we can analyze them. E.g. to apply an algorithm to calculate boundary test-values. Admittedly, for convenient use high level operators must be provided, which we have not done.

9 Conclusion

CTy is an implementation of the CTM approach. However, it is not graphical; it is less suitable for testers who are less proficient in programming. It is provided as a DSL embedded in Haskell. This gives us first class access to Haskell's features. E.g. we get type checking, higher order function, lazy evaluation, and libraries for free. The embedding approach gives us much expressiveness and flexibility, while the syntax is still quite clean. Testers do have to be familiar with some degree of Haskell concepts and syntax.

References

[1] A. Cain, T.Y. Chen, D. Grant, P.L. Poon, S.F. Tang, and T.H. Tse. An automatic test data generation system based on the integrated classification-tree methodology. In *Software Engineering Research and Applications*, volume 3026 of *LNCS*, pages 225–238. Springer, 2004.

[2] M. Conrad and A. Krupp. An extension of the classification-tree method for embedded systems for the description of events. In *2md Workshop on Model Based Testing*, volume 164 of *ENTCS*, pages 3–11, 2006.

[3] M. Grochtmann and K. Grimm. Classification trees for partition testing. *Software Testing, Verification & Reliability*, 3(2):63–82, 1993.

[4] S.L.P. Jones. Composing contracts: An adventure in financial engineering. In *Int. Symp. of Formal Methods Europe (FME)*, volume 2021 of *LNCS*, page 435. Springer, 2001.

[5] E. Lehmann and J. Wegener. Test case design by means of the CTE XL. In *Proc. of 8th European Int. Conf. on Software Testing, Analysis & Review (EuroSTAR)*, 2000.

[6] J. Peterson, P. Hudak, and C. Elliott. Lambda in motion: Controlling robots with Haskell. *LNCS*, 1551:91–105, 1999.

[7] H. Singh, M. Conrad, and S. Sadeghipour. Test case design based on Z and the classification-tree method. In *IEEE Int. Conf. on Formal Engineering Methods (ICFEM)*, pages 81–90, 1997.

[8] S.D. Swierstra. Combinator parsers - from toys to tools. *Electr. Notes Theor. Comput. Sci*, 41(1), 2000.

[9] Y.Y. Yu, S.P. Ng, and E.Y.K. Chan. Generating, selecting and prioritizing test cases from specifications with tool support. In *Int. Conf. on Quality Software (QSIC)*, pages 83–90, 2003.