Typed Multi-Language Strategy Combinators

James Koppel
MIT, Cambridge, MA, US

Abstract

Strategy combinators (also called *strategic programming*) are a technique for modular program transformation construction invented by Bas Luttik and Eelco Visser, best known for their instantiation in the Stratego language. Traditional implementations are dynamically typed, and struggle to represent transformations that can be usefully applied to some types, but not all.

We present the design of our strategy-combinator library COMPSTRAT, a library for type-safe strategy combinators which run on Patrick Bahr's *compositional datatypes*. We show how strategy combinators and compositional datatypes fuse elegantly, allowing the creation of type-preserving program transformations which operate only on datatypes satisfying certain properties. With this technique, it becomes possible to compactly define program transformations that operate on multiple programming languages. COMPSTRAT is part of the Cubix framework and has been used to build four program transformations, each of which operates on at least three languages.

2012 ACM Subject Classification Software and its engineering → Translator writing systems and compiler generators; Software and its engineering → General programming languages

Keywords and phrases program transformation, strategic programming

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.16

Supplementary Material *Software (Source Code)*: `https://github.com/cubix-framework/cubix/tree/master/compstrat`; archived at `swh:1:dir:710191ca2e67e6f922bccc5a9bbc1088699d618f`

Funding This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1122374.

1 Introduction

If a program transformation can be written, it can be written using rewrite rules. Rewrite rules become usable when paired with strategies, functions that apply them. Building strategies is made economical by *strategy combinators* [8, 16, 15].

Typed functional programming has shown many advantages for building language tools. Unsurprisingly, there have been many attempts to bring strategy combinators to typed functional languages, including STRAFUNSKI [7], KURE [2, 11], and Scrap Your Boilerplate (SYB) [5] and its extension RECLIB [10]. All of these grapple with the same challenge: how to build transformations that can run on many types, while preserving the guardrails of strong static typing. All known solutions to this problem use some form of dynamic typing. And in that, we argue, all of them go too far.

Figure 1 illustrates the problem of functions which are "too" dynamically-typed. Each framework defines its own type for generic rewrites, which we call `Rewrite x`; Table 1 gives the encoding for each framework. The first three – STRAFUNSKI, SYB, and RECLIB – are maximally dynamically typed, in that the type system system allows `Rewrite x` to be applied to nearly any type, requiring runtime type-casing to constrain it. Each of these are dynamically typed. Now consider the code in Figure 1, which can be thought of as a fragment of an application that invokes some refactoring transformation, storing the new program and a message in a `Result` data structure. Although the implementation of `doRefactoring` may look innocuous to someone encountering it without exact memory of the `Request` type, it contains a deadly bug. And yet this code typechecks verbatim in SYB! STRAFUNSKI and RECLIB

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 16; pp. 16:1–16:9
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Table 1 Previous Haskell libraries for strategy combinators.

Library	Year	Encoding of `Rewrite x`
STRAFUNSKI	2003	(**Monad** m, Data x) ⇒x →m x
SYB	2003	(**Monad** m, Data x) ⇒x →m x
RECLIB	2006	(Data x) ⇒ a → x → **Maybe** (a, x) (and many variations)
KURE	2007	(**Monad** m) ⇒c →G →m G where G = GExp Exp | GDecl Decl |...
COMPSTRAT	2013[3]	(**Monad** m, HTraversable f) ⇒Term f l → m (Term f l)

```
myRefactoring :: Rewrite JavaProgram
 ...

data RefactorRequest = Request String JavaProgram
data RefactorResult = Result JavaProgram String

doRefactoring :: RefactorRequest →RefactorResult
doRefactoring (Request p s) = Result p (myRefactoring s) -- Oops
```

Figure 1 Example pitfall of dynamically-typed strategy combinators.

would give an error, but only to point out that the rewrite runs in the **Maybe** monad. After fixing the type errors, the true problem remains: myRefactoring is being run on a string rather than on a program, with no effect. When the transformation being applied is one step in a larger sequence, such a bug can linger undetected. KURE[1] is less dynamically typed, permitting a Rewrite x to only be applied to terms of a custom program type. But in exchange it allows a more elementary class of dynamic typing errors: if a KURE rewrite attempts to replace all identifiers in a program with statements, the malformed output will be detected only at runtime.[2] These contrast the COMPSTRAT representation, which, while also internally allowing dynamic type dispatch, is able to place arbitrary constraints on both the set of nodes and set of sorts supported by a rewrite.

In previous work [4], we created CUBIX[4], the "One Tool, Many Languages" framework which makes it possible to build source-to-source program transformations where the entire target programming language is a type parameter. For example, Figure 2 shows a slightly-simplified[5] version of the top-level code for the test-coverage instrumentation transformation, which inserts extra output for test coverage. Despite supporting five languages, its implementation totals only 202 lines.

In Figure 2, the careful reader familiar with strategy combinators may recognize the allbuR (all subterms, bottom-up Rewrite) combinator from STRATEGO's allbu combinator. Indeed, all serious CUBIX transformations are built using a custom strategy combinator library, COMPSTRAT. And it turns out that the special representation of programs which lies at the core of CUBIX incremental parametric syntax, realized using Patrick Bahr's compositional data types [1] synergizes with strategy combinators in a way that allows both more type

[1] Older solutions in monotyped or untyped languages, namely Stratego itself and JJForester [14], are similar to KURE in that they only express rewrites over arbitrary terms.
[2] In fairness, the recommended APIs of KURE do not permit constructing such a malformed rewrite, although the types permit it.
[3] COMPSTRAT was first implemented in 2013, released on Hackage in 2015, mentioned in the 2018 CUBIX paper, and never explained in an academic paper until now
[4] `http://cubix-framework.com/`
[5] The original contains extra code related to management of label indices.

```
-- | Inserts "blocksCovered[n] = true" printouts for test-coverage
--   at every basic block
--
-- Runs on C, Java, JavaScript, Lua, and Python
instrumentTestCoverage :: forall fs l. (CanInstrument fs)
                       => ProgInfo fs -> TermLab fs l -> Annotater (TermLab fs l)
instrumentTestCoverage progInfo t = performCfgInsertions @(StatSort fs) progInfo
                                  $ allbuR (addCoverageStatement progInfo) t
```

Figure 2 Slightly-simplified top-level code of test-coverage transformation, screenshotted from `www.cubix-framework.com`.

safety and an easier implementation compared to all previous attempts. Using COMPSTRAT, it is natural to define a rewrite where e.g.: the compiler guarantees that expressions will only be transformed into other expressions, and guarantees that the transformation may be run on Python and Java programs but not C.

Previous CUBIX papers [9, 4, 3] shied away from discussion of strategy combinators. This paper is the first presentation of the COMPSTRAT library. We shall not present most of its elements – they are built from the basic combinators the same way as in every other strategy combinator library. We shall dwell briefly on its multi-language ability – much of that comes from the representation, and would be the same for any other paradigm built atop CUBIX. But we shall present the key ideas of what makes COMPSTRAT different.

2 Background

In this section, we give an abbreviated summary of incremental parametric syntax and compositional data types, the program representation underlying CUBIX. The rest of this paper will assume familiarity with strategy combinators.

Incremental parametric syntax is a technique for modularly constructing the datatypes of terms in different languages, in a way where an off-the-shelf representation for a single language can be incrementally refined into one built from modular components, gradually reducing the amount of language-specific code that needs to be written to build a tool for multiple languages. Though it can be presented abstractly, in terms of operators for combining and modifying the signatures of languages, its only known instantiation, in CUBIX, is built as an extension of compositional data types [1], which are in turn an extension of data types à la carte [12]. We now present extremely compressed explanations of each of these.

First, the idea of data types à la carte is to make a datatype modular by removing explicit recursion. Instead of defining terms as a recursive datatype such as **data** Exp = Add Exp Exp | Val **Int**, the datatype is defined in *unfixed* form, with the recursive occurrences of Exp replaced by a parameter to be filled in later, i.e.: **data** Exp e = Add e e | Val **Int**. With the recursion removed, the cases of this datatype can be decomposed into fragments which can be recombined in exponentially many variations. Likewise, operations on nodes are defined on individual fragments, and combined into operations on any datatype built out of these fragments. See Figure 3 for a full example.

The encoding of datatypes à la carte produces only unsorted terms. Compositional data types extend this further by allowing multi-sorted terms. The encoding is modified so that everything takes an extra parameter indicating the sort; datatypes are now defined as

```
data Add e = Add e e
data Val e = Val Int
data (f :+: g) e = Inl (f e) | Inr (g e)
data Term f = Term (f (Term f))

type ExpSig = Add :+: Val
type Exp = Term ExpSig

addExample :: Exp
addExample = Term (Inl (Add (Term (Inr (Val 118))) (Term (Inr (Val 1219)))))
```

Figure 3 Using data types à la carte to present the expression 118+1219, with addition and constant nodes defined in separate fragments.

```
data ArithL; data AtomL; data LitL

data Arith t l where
  Add :: t AtomL → t AtomL
                 → Arith t ArithL
data Atom t l where
  Var    :: String   → Atom t AtomL
  Const  :: t LitL   → Atom t AtomL
  Parens :: t ArithL → Atom t AtomL

data Lit (t :: * → *) l where
  Lit :: Int → Lit t LitL
```

Figure 4 Example language fragments.

GADTs so that each constructor may only have one sort. Terms in compositional data types have types which look like `Term LangSig ExpL`, which is read "terms of sort *Exp* in language Lang." See Figures 4 and 5 for a full example.

Incremental parametric syntax is enabled by taking compositional data types and adding a small idea with a huge impact. The key new ingredient in incremental parametric syntax is to control the subsorting relationship by including *sort injection nodes*. These nodes allow the tree to transition between language-specific and generic nodes in a controlled fashion, so that terms are now represented as a combination of language-specific and generic nodes. This both allows incremental development to support a new language, deferring the upfront cost to replace all nodes in a language with generic nodes, and also allows for language-specific customization of generic nodes, by e.g.: making it possible to independently specific which nodes in a specific language may be used as the LHS of a generic assignment node. In total, this approach makes it feasible to scale the datatypes à la carte approach to multiple real languages. Indeed, Cubix is the first known framework[6] to do so, with support for C, Java, JavaScript, Lua, and Python [4]. Figure 6 gives an example of such a tree, and Table 2 gives examples of ways to refer to different classes of terms.

[6] GitHub's semantic framework is the second, although they later abandoned this approach, citing difficulties stemming from their monosorted approach [13]

```
data (:+:) f g t l = Inl (f t l) | Inr (g t l)
data Term f l = Term (f (Term f)) l
type LangSig = Arith :+: Atom :+: Lit
type LangTerm = Term LangSig
```

Figure 5 Combining the fragments of Figure 4.

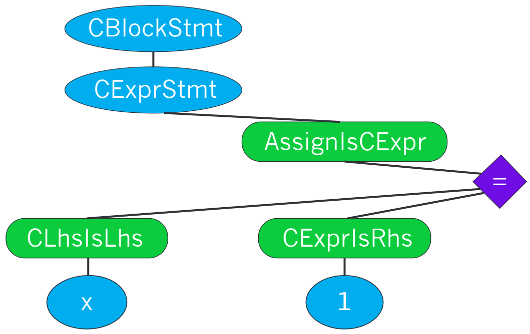

Figure 6 A term in the incremental parametric syntax for C. The ellipses (light blue) represent language-specific nodes; rhombi (purple) represent generic nodes; rounded rectangles (green) represent sort injection nodes.

Table 2 Various term types in Cubix.

Type signature	Description
Term f AssignL	Assignments in any language
Term MJavaSig l	Java terms of any sort
(Assign :<: f) ⇒ Term f IdentL	An identifier in any language that contains generic assignments
Term f (StatSort f)	A statement in any language. The statement sort is language-specific.
(InjF f IdentL PositionalArgExpL , CallAnalysis f) ⇒ Term f IdentL	An identifier in any language which supports a call analysis, and where identifiers may be used as ordinary arguments to functions

One result of this approach is that a single sort may have significance across multiple languages. This means that testing for the sort of a node can be quite useful in strategy combinators, a fact which compstrat exploits.

3 compstrat: Strategy Combinators on Compositional Datatypes

Strategy combinators are an expressive way to build complicated traversal patterns from small building blocks, exemplified by the famous definition of a bottom-up traversal combinator, `allbu(t) = all(allbu(t)); t`. Built on these primitives, implementations of strategy combinator libraries tend to be short, elegant – and identical. Indeed, compstrat provides many identically named combinators to KURE. We thus focus our presentation of compstrat on these building blocks.

Lämmel, Visser, and Visser[6] name a handful of primitives that any strategy combinator library must support. Their list consists of: basic identity and failure strategies, sequential composition, left-biased choice, type-based dispatch, and one-layer traversals applying a strategy to either all or any child of the present node. We shall first define the type of rewrites, from which the basic rewrites and sequencing combinators follow trivially, including the sequencing of rewrites with failure (which is achieved by use of the **Maybe** monad, same as in other Haskell strategy combinator libraries). From there, we turn to discussion of the two other main primitives: one-layer traversal (**all** in Stratego), and type-specific rewriting.

We do simplify some aspects not relevant to showing the unique aspects of compstrat's approach. We present definitions on monads rather than applicatives, and ignore the support for terms with holes. We also present only type-preserving rewrites, ignoring type-altering transformations such as the `crush` operator.

```
type RewriteM m f l = f l → m (f l)
type Rewrite f l = RewriteM Identity f l

type GRewriteM m f = forall l. RewriteM m f l
type GRewrite f = GRewriteM Identity f
```

Figure 7 Rewrite type.

3.1 The basic encoding

Figure 7 gives the type of rewrites in COMPSTRAT.

While straightforward, this is already delivering most of the novel value of COMPSTRAT. Consider a rewrite of type Rewrite (Term f) ExpL, which is statically guaranteed to rewrite expressions to expressions in any language. KURE's encoding does not have a type for multi-language rewrites at all. Meanwhile, the encodings of SYB, STRAFUNSKI, and RECLIB at best permit a rewrite that can run on any type whatsoever. Meanwhile, COMPSTRAT's encoding allows adding arbitrary constraints to the languages and sorts supported by a rewriting, e.g.: (CanTransform f) ⇒ Rewrite (Term f) ExpL, allowing high levels of control on where a rewrite may be applied.

Note that the traditional presentation of strategy combinators assumes all rewrites may fail. But here, rewrites can be defined which are statically known not to fail. Only some combinators deal in failable rewrites, indicated by a **MonadPlus** constraint on the m variable.

3.2 One-layer traversal

Applying a rewrite to every child of a node should be simple, and it is: the one-layer traversal primitive is effectively identical to a method of HTraversable from [1], an analogue of the standard Traversable typeclass for higher-kinded terms.

```
allR :: (Monad m, HTraversable f) ⇒GRewriteM m (Term f) →GRewriteM m (Term f)
allR f (Term t) = fmap Term (hmapM f t)
```

Yet this definition, by using a typeclass which generalizes over all tree nodes but not other types, has advantages over the equivalent combinators for the other libraries. The authors of KURE [11] criticized the inflexibility of approaches such as SYB based on Data.Data, where the only way to define a custom traversal pattern is to provide a custom Data instance – but such an instance would violate the laws that Data is supposed to follow![7] Like KURE, COMPSTRAT makes it possible to define a custom traversal pattern, as custom HTraversable instances are simpler to write than Data and have fewer constraints. But unlike KURE, users can still obtain the default traversal automatically.

3.3 Specialization and dispatch

Like previous approaches, the use of compositional data types makes it straightforward to define a rewrite which fails except on a single constructor. Unlike previous approaches, COMPSTRAT's Rewrite type makes it easy to give a type signature for a rewrite which only runs on a single sort – or a single language, or a family of sorts. For example, here's a rewrite that is statically known to run only on languages which have generic identifiers.

[7] An alternative is to add extra typecases to every rewrite application over every tree that may contain a node where the default traversal is insufficient.

```
-- compstrat
tryR :: (Monad m) => RewriteM (MaybeT m) f l -> RewriteM m f l

-- KURE
tryR :: MonadCatch m => Rewrite c m a -> Rewrite c m a

-- Closest equivalents in Strafunski
succeed :: Maybe x -> x
ifM :: MonadPlus m => m a -> (a -> m c) -> m c -> m c
```

Figure 8 Support for failure.

```
vandalize' :: (Ident :<: f) => Rewrite (Term f) IdentL
vandalize' (project -> (Just (Ident s))) = return (iIdent (s ++ "_foo"))
```

It uses the open-sum projection operator, available in all implementations of datatypes à la carte.

COMPSTRAT uses its DynCase class to allow specialization based on sort, as distinguished from the more typical specialized based on specific constructors is based on the DynCase class. It takes a term of an unknown sort b, and possibly returns a proof that b is equal to some known sort a. This makes it possible to lift single-sorted rewrites to multisorted, as in the dynamicR and promoteR combinators later in this section.

```
class DynCase f a where
  -- | Determines whether a node has sort @a@
  dyncase :: f b -> Maybe (b :~: a)
```

The DynCase typeclass may look like the built-in Typeable typeclass relied on by SYB, STRAFUNSKI, and RECLIB. But it's different in an important way. A generated instance of this class for the language in Figure 4 looks like this:

```
instance (Arith :<: f) => DynCase (Term f) ArithL where
  dyncase x = case project x of
                Just (Add _ _) -> Just Refl
                _              -> Nothing
```

As we can see, dyncase is implementable just as a mundane case match, without terms needing to carry extra runtime-type information, as is required by Typeable. It is used to write isSortR, which fails except at nodes of the desired sort, and dynamicR, which makes a sort-specific rewrite run at all sorts, failing at all save the desired sort.

```
isSortR   :: (DynCase f l, MonadPlus m) => Proxy l -> RewriteM m f l'
dynamicR :: (DynCase f l, MonadPlus m) => RewriteM m f l -> GRewriteM m f
```

3.4 Flexible Monad Stack

An idea which could be easily added to existing strategy combinator libraries, but strangely isn't, is the flexible monad stack. To explain this, let us look at the type signature of COMPSTRAT's tryR combinator in Figure 8, which takes a failable rewrite and makes it always succeed, and contrast it with its equivalents in other frameworks.

(Note that we found no equivalent in either SYB or RECLIB.)

The COMPSTRAT version takes a rewrite which *may* fail, and returns one which is statically known not to. The other versions input and output rewrites in the same monad. As a result, code running rewrites in the other frameworks must frequently check for failure or call fromJust on code which is known to never fail; clients of COMPSTRAT need not.

tryR is used to define promoteR, which escalates a sort-specific rewrite to run on all sorts, doing nothing at nodes of the wrong sort.

```
promoteR :: (DynCase f l, Monad m) ⇒RewriteM (MaybeT m) f l →GRewriteM m f
promoteR = tryR . dynamicR
```

Other combinators in COMPSTRAT using this idea include allStateR, which accumulates results in a local state monad, and prunetdR, which runs a failable transformation on all nodes except the descendants of transformed nodes.

3.5 Putting it together

We can now show a simple transformation built using COMPSTRAT: using only two lines of code and two lines of type signatures, the vandalize transformation modifies all identifiers in a term of any language that uses CUBIX's generic identifier node. Note that it (1) is guaranteed to never fail (2) can only be run on terms, and only terms in applicable languages, and (3) is guaranteed to never change the sort of what it runs on (e.g.: when run on an expression, returns an expression; when run on a declaration, returns a declaration). It uses vandalize' defined above; here is the new part:

```
vandalize :: (Ident :<: f, HTraversable f) ⇒ GRewrite (Term f)
vandalize = allbuR (promoteR (addFail vandalize'))
```

4 Examples and Applications

The original CUBIX paper presented four program transformations, all of which use COMPSTRAT, although two of them only use it in a small way. We defer to the original paper [4] for a description of these transformations. Here we present code snippets taken from these transformations showing interesting uses of COMPSTRAT.

Specifically, we show pieces of the Hoist transformation, which lifts variable declarations to the top of their scope. It uses the custom combinator transformOuterScope f g, which runs g on all nodes outside a scope-delimiting block, and f on all nodes inside it. New combinators used here include guardBoolT, which takes an operation returning a boolean and converts it to a failable rewrite; guardedT, essentially an if-statement in strategy-combinator form; and addFail :: Rewrite m f l → Rewrite (MaybeT m) f l, which treats a non-failable rewrite as failable. Another thing to note here is the use of the CanHoist constraint, which ensures this combinator (and others in the same file) may only be run on applicable languages.

```
transformOuterScope :: ( MonadHoist f m, CanHoist f)
                        ⇒ GRewriteM m (Term f) →GRewriteM (MaybeT m) (Term f)
                                                  → GRewriteM m (Term f)
transformOuterScope f g = tryR (
   guardedT (guardBoolT (isSortT (Proxy :: Proxy BlockL))) (addFail (alltdR f))
                                                           (addFail g)
   >=> allR (transformOuterScope f g))
```

Along with the sort-specific rewrites addIdents, transformBlockItems, and transformStatSorts, this combinator is now used to define the core of the Hoist transformation.

```
hoist = transformOuterScope
          (promoteR addIdents)
          ((dynamicR transformBlockItems)
            >+> (dynamicR transformStatSorts))
          items
```

5 Conclusion

We have presented a library for type-safe strategy combinators. Unlike all previous approaches, COMPSTRAT allows programmers to build transformations that are statically guaranteed not to fail, statically guaranteed to preserve sorts, and restricted at the type level to only run on (certain classes) of terms rather than on arbitrary datatypes. COMPSTRAT is available from `https://github.com/cubix-framework/cubix/tree/master/compstrat`, a part of the CUBIX framework.

References

1 Patrick Bahr and Tom Hvitved. Compositional Data Types. In *Proceedings of the Seventh ACM SIGPLAN Workshop on Generic programming, WGP@ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 83–94, 2011.

2 Andy Gill. A Haskell Hosted DSL for Writing Transformation Systems. In *Domain-Specific Languages*, pages 285–309. Springer, 2009.

3 James Koppel. One CFG-Generator to Rule Them All. `https://www.jameskoppel.com/files/papers/cubix_cfg.pdf`. Accessed: 2022-10-28.

4 James Koppel, Varot Premtoon, and Armando Solar-Lezama. One Tool, Many Languages: Language-Parametric Transformation with Incremental Parametric Syntax. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):122, 2018.

5 Ralf Lämmel and Simon Peyton Jones. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. *ACM SIGPLAN Notices*, 38(3):26–37, 2003.

6 Ralf Lämmel, Eelco Visser, and Joost Visser. The Essence of Strategic Programming. `https://www.researchgate.net/publication/277289331_The_Essence_of_Strategic_Programming`, 2002.

7 Ralf Lämmel and Joost Visser. A Strafunski Application Letter. In *International Symposium on Practical Aspects of Declarative Languages*, pages 357–375. Springer, 2003.

8 Bas Luttik and Eelco Visser. Specification of Rewriting Strategies. In *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications*, pages 1–16, 1997.

9 Varot Premtoon, James Koppel, and Armando Solar-Lezama. Semantic Code Search via Equational Reasoning. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 1066–1082, 2020. `doi:10.1145/3385412.3386001`.

10 Deling Ren and Martin Erwig. A Generic Recursion Toolbox for Haskell, Or: Scrap Your Boilerplate Systematically. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, pages 13–24, 2006.

11 Neil Sculthorpe, Nicolas Frisby, and Andy Gill. The Kansas University Rewrite Engine: A Haskell-Embedded Strategic Programming Language with Custom Closed Universes. *Journal of Functional Programming*, 24(4):434–473, 2014.

12 Wouter Swierstra. Data Types à la Carte. *Journal of Functional Programming*, 18(04):423–436, 2008.

13 Patrick Thomson, Rob Rix, Nicolas Wu, and Tom Schrijvers. Fusing Industry and Academia at GitHub (experience report). *arXiv preprint*, 2022. `arXiv:2206.09206`.

14 Arie van Deursen and Joost Visser. Building Program Understanding Tools Using Visitor Combinators. In *Proceedings 10th International Workshop on Program Comprehension*, pages 137–146. IEEE, 2002.

15 Eelco Visser. Stratego: A Language for Program Transformation Based on Rewriting Strategies: System Description of Stratego 0.5. In *International Conference on Rewriting Techniques and Applications*, pages 357–361. Springer, 2001.

16 Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building Program Optimizers with Rewriting Strategies. *ACM Sigplan Notices*, 34(1):13–26, 1998.