Gradually Typing Strategies

Jeff Smits
Delft University of Technology
The Netherlands
j.smits-1@tudelft.nl

Eelco Visser
Delft University of Technology
The Netherlands
e.visser@tudelft.nl

Abstract

The Stratego language supports program transformation by means of term rewriting with programmable rewriting strategies. Stratego's traversal primitives support concise definition of generic tree traversals. Stratego is a dynamically typed language because its features cannot be captured fully by a static type system. While dynamic typing makes for a flexible programming model, it also leads to unintended type errors, code that is harder to maintain, and missed opportunities for optimization.

In this paper, we introduce a gradual type system for Stratego that combines the flexibility of dynamically typed generic programming, where needed, with the safety of statically declared and enforced types, where possible. To make sure that statically typed code cannot go wrong, all access to statically typed code from dynamically typed code is protected by dynamic type checks (casts). The type system is backwards compatible such that types can be introduced incrementally to existing Stratego programs. We formally define a type system for Core Gradual Stratego, discuss its implementation in a new type checker for Stratego, and present an evaluation of its impact on Stratego programs.

CCS Concepts: • **Software and its engineering** → **Semantics**; *Polymorphism*; *Extensible languages.*

Keywords: gradual types, strategy, generic programming, type preserving

ACM Reference Format:
Jeff Smits and Eelco Visser. 2020. Gradually Typing Strategies. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering (SLE '20), November 16–17, 2020, Virtual, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3426425.3426928

1 Introduction

The Stratego language supports program transformation by means of term rewriting with programmable rewriting strategies [30]. Stratego's traversal primitives support concise definition of generic tree traversals. For example, the definition of `bottomup(s)` in Figure 4 defines in one line a generic bottom-up traversal that can be instantiated with a selection of rewrite rules to be applied in a particular transformation, without needing to define a traversal for each constructor in the abstract syntax. Stratego is used in the Stratego/XT program transformation tool suite [2] and the Spoofax language workbench [11] and used in production in research, education, and industry [6, 12].

Stratego is a dynamically typed language, because its language features cannot be captured fully by a static type system. While dynamic typing makes for a flexible programming model, it also exposes Stratego programmers to unintended type errors. Static typing of strategies has been considered before by Lämmel and Visser [16], Lämmel [14], and others. Lämmel and Jones [15] adopted Stratego's strategic programming in the SYB Haskell design pattern. These efforts focus on the statically typable fragment of strategies, making them unsuitable, as is, as a type system for Stratego. Furthermore, there is a considerable base of existing Stratego code, and having to convert that, at once, to statically typed code would preclude adoption of a type system.

In this paper, we introduce a gradual type system for Stratego that combines the flexibility of dynamically typed generic programming, where needed, with the safety of statically declared and enforced types, where possible. We integrate ideas for statically typing strategies by Lämmel [14] with ideas from the gradual typing literature [24, 25]. In particular, we extend conventional static types with the special type for type preserving transformations [14]. And we introduce a dynamic type in the tradition of gradual type systems to account for, as yet, untyped code. At the interface of statically and dynamically typed code, the type checker inserts dynamic type checks (through casts and proxies) to guarantee the assumptions of static code. This ensures that the type system is backwards compatible such that existing code can pass the type checker as is, and such that types can be introduced incrementally to existing code. At the intersection of typed strategies and gradual types, we find an interesting dynamic types for strategies. For example, the type unifying strategies of Lämmel [14] do not need a special type, but can be modeled with a dynamic input type and a specific result type.

SLE '20, November 16–17, 2020, Virtual, USA
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8176-5/20/11.
https://doi.org/10.1145/3426425.3426928

The contributions of this paper are:

- We motivate and validate the gradual type system for Stratego with idiomatic examples (Sections 2 and 3).
- We formally define a type system for Core Gradual Stratego, which combines static types for generic traversals with gradual types for partially typed programs (Section 4). The combination supports the (partially) dynamic typing of strategic programming patterns that are inherently not (completely) statically typable. This constitutes the first static type system for the Stratego language.
- We have implemented a type checker based on the type system, which reports violations in the IDE and which generates code with casts and proxies for type checking at run time at the interface of statically and dynamically checked code (Section 4).
- We evaluate the application of the type checker on an existing Stratego codebase to which we added type annotations (Section 5).

2 Rewriting Strategies and Types

Stratego is a language for the definition of program transformations with rewrite rules and programmable rewriting strategies [30]. In this section we give a (non-exhaustive) introduction to Stratego by means of examples as motivation for a type system. We give two encodings of the same transformation. The first encoding is completely statically typable. The second encoding requires dynamic typing. Furthermore, we analyze typical type errors.

2.1 Program Transformation with Stratego

The main ingredients of Stratego programs are algebraic signatures, rewrite rules, and strategies.

Algebraic Signatures. A Stratego program defines transformations on abstract syntax trees represented using first-order terms. The structure of terms is defined by means of an *algebraic signature*, which introduces sorts (types) and constructors on those sorts. A constructor declaration $c : t_1 * \ldots * t_n \longrightarrow t_0$, defines a constructor c with the sorts of its arguments and result. A constructor declaration $: t_1 \longrightarrow t_0$ is an *injection* of t_1 into t_0. This means a value of t_1 can be used directly as a value of t_0 without a constructor. Figure 2 defines the signature of a small imperative language with expressions and statements. An example of a well-formed term of sort Stat would be:

```
Seq(Assign("x", Add(Var("x"), Int("1"))),
    Lt(Var("x"), Int("3")))
```

Note that sorts and constructors are separate namespaces. In Spoofax, signatures are generated from a syntax definition in SDF3 [5], directly describing the structure of abstract syntax trees produced by parsers.

```
rules
  desugar : // Exp to Exp
    Min(e) → Sub(Int("0"), e)

  desugar : // Stat to Stat
    For(x, e1, e2, s) →
    Seq(Assign(x, e1),
        While(Lt(x, e2),
              Seq(s, Assign(x, Add(x, Int("1"))))))

  desugar : // Exp to Exp
    Inc(x) → Stat(Assign(x, Add(x, Int("1"))), x)

  desugar : // Stat to Stat; lift Stat from Exp
    stat@<is-simple-stat> → Seq(s1, s2)
    where <oncetd-hd((Stat(s1, e) → e))> stat ⇒ s2
```

Figure 1. Rewrite rules

```
signature
  sorts Var Exp constructors
    Var  : string → Var
         : Var → Exp
    Int  : string → Exp
    Add  : Exp * Exp → Exp
    Sub  : Exp * Exp → Exp
    Lt   : Exp * Exp → Exp
    Min  : Exp → Exp
    Inc  : Var → Exp
    Stat : Stat * Exp → Exp
  sorts Stat constructors
    Exp    : Exp → Stat
    Skip   : Stat
    Assign : Var * Exp → Stat
    Seq    : Stat * Stat → Stat
    While  : Exp * Stat → Stat
    For    : Var * Exp * Exp * Stat → Stat
```

Figure 2. Signature

Rewrite Rules. Basic transformations are defined using *term rewrite rules*. A rewrite rule has the form $l : t_1 \longrightarrow t_2$ with label l, left-hand side term t_1 and right-hand side term t_2. Applying a rewrite rule with label l to a term t means matching the term against the left-hand side term t_1, binding the variables in that term to sub-terms of t, and then replacing term t with the instantiation of the right-hand side t_2. If the match fails, the rule fails to apply. If the specification has multiple rules with the same name, they are tried in order.

A rule `l : t_1 → t_2 where s` is a conditional rule, where `s` is a strategy expression. When applying a conditional rule, the condition `s` is applied to the subject term, using variables bound in the match of the left-hand side, and possibly binding variables that are used in the right-hand side. When the condition fails, applying the rule fails.

Figure 1 shows examples of rewrite rules for a desugaring transformation on expressions and statements of the language of Figure 2. The first two rules define `Min` and `For` in terms of other constructs. The third rule defines increment (think C-style post increment `x++`) in terms of assignment to the variable incremented. Since `Inc(_)` is an expression and `Assign` is a statement, replacing one by the other would not be well-formed. The `Stat` constructor defines an expresssion form that allows embedding a statement within an expression. The fourth rule defines lifting of statements embedded in expressions to the statement level. The match pattern uses a guard that checks that the term is a simple statement (defined in Figure 3), and the condition of the rule applies a left-most depth-first ('once-top-down') traversal that finds an embedded occurrence of a term `Stat(s1, e)`, replacing that occurrence with the expression, and binding the statement to variable `s1`. The syntax for the match is `n @ t` to bind variable `n` to the term `t` and `< s >` to apply strategy `s` as a guard to the term, i.e. it only matches if the strategy succeeds on the given term. The argument of `oncetd` is an *anonymous* rewrite rule `(Stat(s1, e) → e)` that does not scope its variables. Thus, bindings are available in the context of the traversal. (Such contextual binders were introduced by Visser et al. [30] and later generalized to *dynamic rewrite rules* by [3].) For example, the rules give rise to a sequence of transformations such as the following:

```
    Assign(Var("x"), Min(Inc(Var("x"))))
--> Assign(Var("x"), Min(Stat(Assign(Var("x"),
                                 Add(Var("x"), Int("1"))),
                                 Var("x"))))
--> Assign(Var("x"), Sub(Int("0"),
                         Stat(Assign(Var("x"),
                                 Add(Var("x"), Int("1"))),
                                 Var("x"))))
--> Seq(Assign(Var("x"), Add(Var("x"), Int("1"))),
        Assign(Var("x"), Sub(Int("0"), Var("x"))))
```

Rules are not applied automatically, but their application (order) is determined by a strategy.

Strategies. Rewrite rules transform a term into another term. Traditional rewrite systems apply such rules exhaustively throughout a term. Stratego provides *programmable strategies* for ordering the application of rewrite rules to a term. For example, Figure 3 defines the `transform` strategy to apply the `desugar` rules of Figure 1 using a bottom-up strategy that tries to apply the rules at each node. Strategies such as `bottomup` and `try` are not built-in, but generic, parametric strategies defined in terms of basic *strategy combinators*.

Stratego's built-in strategy combinators include identity **`id`**, failure **`fail`**, sequential composition `s1; s2`, and ordered choice `s1 <+ s2`. Matching (and returning) a term pattern

```
strategies
  transform = bottomup(try(desugar))

  is-simple-stat = ?Assign(_,_) <+ ?Exp(_)
    <+ ?While(_,_) <+ ?For(_,_,_,_)

  oncetd-hd(s) =
    While(oncetd(s),id) <+ For(id,oncetd(s),id,id)
    <+ For(id,id,oncetd(s),id) <+ Exp(oncetd(s))
    <+ Assign(id, oncetd(s))
```

Figure 3. Strategies

```
strategies
  try(s)      = s <+ id
  topdown(s)  = s; all(topdown(s))
  bottomup(s) = all(bottomup(s)); s
  oncetd(s)   = s <+ one(oncetd(s))
  alltd(s)    = s <+ all(alltd(s))
```

Figure 4. Generic strategies

(**`?`**`t`) and instantiating (aka building) a term pattern (**`!`**`t`) are first-class citizens. The expression `<s>t` applies strategy `s` to term `t` and the expression `s ⇒ t` matches the result of `s` against term `t`. The generic traversal combinators **`all`**`(s)` and **`one`**`(s)` apply a transformation to all, respectively, one, of the sub-terms of a term. Given a constructor $c : t_1 * \ldots * t_n \longrightarrow t_0$, a corresponding *congruence traversal* strategy $c(s_1, \ldots, s_n)$ transforms c-terms, applying the corresponding strategies to the sub-terms.

Figures 3 and 4 use these combinators to define strategies[1]. Strategy `is-simple-stat` determines whether a term is a simple statement through pattern matching. The traversal strategy `oncetd-hd` uses congruence traversal to apply a `oncetd(s)` traversal to selected arguments of constructors (the 'heads'). Figure 4 defines several generic strategies. The strategy `try(s)` applies `s` and when that fails succeeds with the original term. The strategy `bottomup(s)` first visits the direct sub-terms of the subject term with a recursive call and then applies `s` to the result. Strategy `oncetd(s)` transforms the first term for which `s` succeeds in left-most depth-first traversal. Strategy `alltd(s)` applies `s` to all outermost terms for which `s` succeeds.

Non-Well-Formed IR. The rules defined in Figure 1 take care to only replace terms with terms of the same sort. The `Stat` constructor is used to embed a statement within an expression. While this is good practice, it is not required. Figure 5 shows an alternative approach to the transformation. The transformation is defined in two stages. In the first stage, the `desugar-inc` rule replaces `Inc` terms with assignments, creating non-well-formed intermediate terms. In the second

[1]Note that there is no strict syntactic separation between rules and strategies. The section headers are used to indicate intention.

```
rules
  desugar-inc : // Exp to Stat
    Inc(x) → Assign(x, Add(x, Int("1")))

  lift-assign : // lift Stat from Exp
    s1@<is-simple-stat> → Seq(Assign(x, e), s2)
    where <oncetd-hd((Assign(x, e) → x))> s1 ⇒ s2
strategies
  transform   = desugar-all; lift-all
  desugar-all = bottomup(try(desugar-inc))
  lift-all    = alltd(lift-assign; lift-all)
```

Figure 5. Alternative rules and strategies

```
rules // errors caught by current compiler
  desugar :
    Inc(Vaz(x)) → // unknown constructor
    Stat(Assign(y, // unbound variable
                Add(Var(x), Int("1"))))
  // constructor Stat used with wrong arity
  desugar-some =
    top-down(desugar) // unknown strategy
rules // errors not caught by current compiler

  desugar : // expression replaced with statement
    Stat(stat, e) → stat

  desugar : // lifting also applied to expressions
    stat → Seq(s1, s2)
    where <oncetd((Stat(s1, e) → e))> s ⇒ s2

  desugar :
    Inc(x) → Stat(
      Assign(Var(x),         // Var instead of string
             Add(x,Int(1))), // int instead of string
             x)
```

Figure 6. Rules and strategies with errors

stage, the `lift-assign` rule lifts assignments embedded in expressions to assignment level. While intermediate terms are not well-formed with respect to the signature, after applying `transform`, terms are well-formed again.

2.2 Type Errors

Stratego is a memory-safe, dynamically typed language. The Stratego runtime, in collaboration with the code generator or interpreter, ensures that a program that passes the static checks, does not crash. A program may terminate with a transformed term, with a (pattern match) failure, or with an exception. (An alternative version of conditional rules requires that the condition succeeds and raises an exception if it does not.)

The front-end of the compiler applies some static checks. Constructors need to be declared and used with the arity corresponding to the declaration. Rules and strategies need to be defined when called. Variables should be bound when used in a build pattern. Otherwise, the compiler is fairly permissive, allowing programs such as in Figure 5. In Figure 6 we give examples of errors that are caught by the Stratego compiler and errors that are not caught by the compiler nor by the runtime.

A typical consequence of the lack of static typing is that a transformation is applied successfully, but constructs a non-well-formed term. Subsequently, a pretty-printer (e.g., generated from a syntax definition [5]) transforming terms to Box expressions, fails because the term does not match the expected abstract syntax schema. This leads to expensive debugging sessions to track down the origin of the non-well-formedness. A type system for Stratego that identifies such errors statically will make Stratego programming much more productive at micro scale. At macro scale, having types for interfaces will make code more maintainable. Furthermore, the compiler could benefit from the guarantees of static types in optimizations. At the same time, such a type system should not prevent the usage of existing code.

3 Gradually Typing Strategies

We introduce a type system for Stratego that addresses the lack of static type checking discussed in the previous section. In this section, we first discuss the requirements for a type system, and then we give a high-level overview of the type system building on the examples of the previous section. In the next section, we formalize the type system.

3.1 Requirements

The design and implementation of a type system for Stratego should satisfy the following requirements. It should be backward compatible such that existing programs are accepted as is, with the same run time semantics. It should impose minimal type annotation requirements on programs to preserve the concise style of Stratego programs. It should support generic traversal primitives, providing static typing where possible, but also support dynamic usage. It should support a simple migration path from untyped to typed code. It should be modularly checkable for integration into the incremental compiler of Smits et al. [26]. It should have limited negative impact on performance. In the rest of this paper, we describe a type system that mostly meets these criteria; we have not evaluated performance yet.

3.2 Types for Stratego

We first discuss complete static checking. We will write strategy to indicate both rules and strategies.

Top-Level Type Annotations. To force checking the type of a strategy, one explicitly declares its type using a type annotation. For example, we can declare the types of `transform` and `desugar-inc` from the previous section as:

```
  transform :: Stat → Stat
  desugar-inc :: Exp → Stat
```

The caller of a strategy with a type annotation needs to ensure that the term that it is applied to has the right type. The type checker checks that given the assumption on the input term, the strategy definition guarantees that the output term has the specified type, or that the strategy fails. The type system does not infer types for top-level declarations of strategies. If a type is omitted type dynamic is assumed, as we will discuss below.

Note that the addition of a top-level type can invalidate code that is acceptable when dynamically typed.

Type Checking Term Patterns. Given an expected input or output type, the type checker checks that a pattern is well-formed with respect to that type and the constructors in the signature. Figure 7 shows examples of errors caught in pattern matching and instantiation. These would not be errors if the type annotation was not present to restrict the strategies to a specific type. Thus, the errors in Figure 6 in the desugaring rule for `Inc` are all caught.

Inferring Types of Variables. While we opted to not infer types for top-level declarations, we do infer types for local variables. Variables in Stratego rules are declared implicitly by using them in a match position. To avoid clutter, we do not require explicit declaration of type annotations for local variables, but rather infer their type from context, where we take the types of the left- and right-hand sides of a rule as leading. For example, consider the `ssa` rules in Figure 8. The types of the local variables in the condition of the rule follow from the type of the strategy, and then from the types inferred in previous steps in the condition. Consider the errors that are found when the variables in the `Stat(_,_)` pattern match are swapped:

```
ssa : Min(e) → s2
  where <ssa> e ⇒ Stat(e', s1)
  ; !Var(<new>) ⇒ x
  ; !Stat(Seq(s1,                 // warn: not Stat
              Assign(x, Min(e'))),// warn: not Exp
          x) ⇒ s2                 // error: not Stat
```

Type Preserving Transformations. Generic term traversals, such as `bottomup`, apply an arbitrary transformation to the sub-terms of a term. As we illustrated in Figure 5, such generic traversals may construct (temporarily) non-well-formed terms. However, a particular class of traversals is more well-behaved and transforms each term to a term of the same type (or fails), possibly operating heterogeneously on multiple types. In other words, such strategies are *type preserving*. Following the work of Lämmel [14] we introduce the **TP** type to characterize type preserving strategies.

Type preserving transformations can be heterogeneous rewrite rules such as `desugar` in Figure 1, which operate on terms of different types, but ensure to always transform each term to a term of the same type[2]. For example, `desugar` transforms `Exp` terms to `Exp` terms, and `Stat` terms to `Stat` terms. Generic strategies can construct type preserving strategies from type preserving strategies. For example, if `s` is of type **TP**, then `try(s)` is also of type **TP**. Similarly, **all**`(s)` is **TP** if `s` is, and `s1; s2` is **TP** if `s1` and `s2` are. Given these ingredients and the annotation `bottomup(TP) :: TP`, we can conclude that the definition `bottomup(s) = all(bottomup(s)); s` is well-typed, and that `bottomup` is a type preserving strategy.

Given the type annotations in Figure 9, the rules in Figure 1 and the strategies in Figures 3 and 4 are completely statically typable, except for one detail, which we discuss next.

Type Match. The rule `min : e → Min(e)` takes any term and applies `Min` to it. That is, its applicability is not guarded by a pattern with a constructor. For example, in untyped Stratego we can write `<min>Skip() ⇒ Min(Skip())`, producing a non-well-formed term. When we give it the type annotation `min :: Exp → Exp` we express that *when applied to an* `Exp` *it will return an* `Exp`. The caller should guarantee to only apply

```
strategies
  is-stat :: Stat → Stat
  is-stat = ?Assign(_,_)        // ok
  is-stat = ?Var(_)             // error: not a Stat
  is-stat = ?Seq(Var(_),_)      // error: arg not Stat
  mk-stat :: () → Stat
  mk-stat = !Exp(Var("x"))      // ok
  mk-stat = !Var("x")           // error: not a Stat
  mk-stat = !Exp(Exp(Var("x")))// error: arg not Exp
```

Figure 7. Checking (nested) patterns

```
rules
  new :: () → string
  ssa :: Exp → Exp
  ssa : Var(x) → Stat(Skip(), Var(x))
  ssa : Min(e) → s2
    where <ssa> e ⇒ Stat(s1, e')
    ; !Var(<new>) ⇒ x
    ; !Stat(Seq(s1, Assign(x, Min(e'))), x) ⇒ s2
```

Figure 8. Inferring types of variables

```
rules
  transform :: Stat → Stat
  desugar :: TP
  try(TP) :: TP
  bottomup(TP) :: TP
  oncetd(TP) :: TP
  alltd(TP) :: TP
```

Figure 9. Type annotations

[2]The difference between **TP** and polymorphic strategy `a → a` is discussed at the end of Section 4.

it to `Exp` terms. In order to qualify for a type annotation `min :: TP`, stronger requirements apply. A `TP` strategy should be applicable to a term of any type and produce a well-formed term of the same type as output (or fail). Thus, a `TP` strategy should check its own input type requirements. Given that the new type checker relies on dynamic type checks for casts (see below), we also make this functionality available as a *type match*. For any declared sort *t*, the strategy `is(t)` *dynamically* checks that the subject term is of type *t* and *statically* guarantees that that is the case when it succeeds. Thus, we can define the `min` rule as `min : e@<is(Exp)> → Min(e)` and give it the `TP` annotation.

In Figure 3 we defined `is-simple-stat` to identify simple statements. Such strategies are often used to define fine grained recognizers of subsets of types [27]. Unfortunately, its use in the `desugar` rule in Figure 1 is not sufficient to convince the type checker that the rule is `TP`. Using the type match `is(Stat)` we can convey that it is:

```
desugar :
  s@<is(Stat); is-simple-stat> → Seq(s1, s2)
  where <oncetd-hd((Stat(s1, e) → e))> s ⇒ s2
```

With that edit, the rules in Figure 1 and the strategies in Figures 3 and 4 are completely statically typable against the type annotations in Figure 9.

Polymorphic Strategies. Our type checker supports parametric polymorphic types for rules and strategies. We use prenex polymorphism with lowercase names as type variables, which was already used informally (i.e. without type checker support) in signatures. In Figure 10 we define several polymorphic strategies on lists from the standard library (edited for space) and provide type annotations for them. Note that type match and imperative update make that we can not support parametricity [22, 31].

3.3 *Gradual* Types for Stratego

Not all Stratego programs can be statically type checked using the techniques discussed above. For example, the alternative transformation approach of Figure 5 constructs intermediate terms that are not well-formed with respect to the signature. Furthermore, there exists a significant amount of Stratego code without type annotations. Imposing the requirement that all strategies should be annotated, before the new type checker can be used would be prohibitive. This is one of the classical motivations for the introduction of gradual types [24].

We have extended the type system sketched above with dynamic types such that *any* existing Stratego program (that passes the static checks of the legacy compiler), will pass the type checker and will have the same runtime semantics. (Note that Stratego's syntax already enforces a distinction between strategies (functions) and terms.)

```
rules
  filter(a → b) :: List(a) → List(b)
  filter(s) : [] → []
  filter(s) : [x | xs] → <conc>(<opt(s)>x,
                                <filter(s)>xs)
  opt(a → b) :: a → List(b)
  opt(s) = ![<s>] <+ ![]

  conc :: List(a) * List(a) → List(a)
  conc : ([], xs) → xs
  conc : ([x | xs], ys) → [x | <conc>(xs, ys)]

  mapconc(a → List(b)) :: List(a) → List(b)
  mapconc(s) : [] → []
  mapconc(s) : [x|xs] → <conc>(<s>x, <mapconc(s)>xs)
```

Figure 10. Polymorphic strategies

Type Dynamic and Type Casts. We extend the set of types with the type dynamic `?`, which represents a dynamically checked type (not to be confused with the match operator). When a typed strategy is invoked on a dynamically typed term, the term needs to be checked in order to guarantee the input requirements. This check is done by a type cast, an assertion that verifies that the subject term has the expected type. Failing the assertion is a programming error and leads to an exception. The type checker (silently) inserts casts where needed, in the style of gradual typing [24]. When a top-level strategy does not have a type declaration, it is considered to have a dynamic type. The dynamic type can also be used explicitly in type annotations.

Gradually Typing Term Patterns. Term patterns in Stratego appear in either matching or building position. When we match against a dynamically typed term, we can not assume that it is a well-formed term. Therefore, we only check that the constructors that are used in the pattern are defined and have the right arity, compatible with the legacy Stratego compiler. When we build a term that is expected to be dynamically typed, we also check that the used constructors are defined and have the right arity. However, we can sometimes infer the type of a well-formed term in a build pattern (Section 4). This extra type information is propagated, and can prevent some unnecessary casts from being inserted.

Proxies. The type checker must also type check strategy arguments to other strategies. We do not support higher-order casts, i.e. type assertions on strategies, directly. Instead we do this dynamic type check lazily, at the time the strategy is called. We do so by creating a new closure for a strategy argument that includes a type cast. But if a strategy argument is passed through a couple of statically and dynamically typed strategies, this can lead to an accumulation of closures in closures, as noted by Herman et al. [9]. Figure 11 shows an example. When `<intToNat; if-even(!Z(), !S(Z()))> 10`

```
strategies
  if-odd(Nat → Nat, Nat → Nat) :: Nat → Nat
  if-odd(s1, s2): Z() → <s2>
  if-odd(s1, s2): S(t) → <if-even(s2, s1)> t

  if-even(s1, s2): Z() → <s1>
  if-even(s1, s2): S(t) →
    <if-odd(cast(Nat); s2; cast(Nat),
            cast(Nat); s1; cast(Nat))> t
  // proxy version:
  if-even(s1, s2): S(t) →
    <if-odd(proxy(|Nat,Nat|s2),
            proxy(|Nat,Nat|s1))> t
```

Figure 11. Explicit casts surrounding strategy arguments can accumulate into closures inside closures, a space and time leak we should avoid.

is executed, the first argument is wrapped in a type cast closure *five* times before it is executed. Therefore, the type checker inserts type proxies instead of normal casts. These are closures that contain a type assertion for the input and output of a strategy, and the strategy (closure) itself, as a special value that the runtime can inspect. When a new proxy is constructed around a proxy value, these are collapsed into one proxy value to avoid the accumulation of closures in closures:

```
proxy(|Nat,Nat|proxy(|Nat,Nat|s1))
--> proxy(|Nat;Nat,Nat;Nat|s1)
--> proxy(|Nat,Nat|s1)
```

Locally required type casts can be merged with the type casts of the proxy value. This makes dynamic type checks less lazy, as incompatible type assertions can be found while adding them to an existing proxy, similar to the work of Siek et al. [23].

Type Preserving and Dynamically Typed Traversals. While the `TP` type allows static typing of many transformations, the type restricts the legitimate use of useful standard library strategies. To prevent duplication of type preserving and dynamically typed versions of the same strategies, the type checker allows a dynamically typed fall back type annotation for a strategy. For example, the `bottomup` strategy can also be typed with the more permissive type annotation `bottomup(? → ?) :: ? → ?`. This strategy can be called on any term, well-formed or not, since the strategy argument is given no guarantees on what kinds of terms it is called upon. Perhaps a more interesting case is the `try` strategy, which still has some typing even when not type preserving: `try(a → b) :: a → ?`. A strategy wrapped in a try must still be called with the right input type for the strategy.

Type Unifying Strategies. Lämmel [14] coined the term *type unifying strategies* (and the special type `TU(b)`) for Stratego strategies that take any input and return a single, typed output. We use `?` for the input to allow any input and a type parameter for the output. Figure 12 shows an example of a type unifying generic traversal.

```
rules
  collect(? → b) :: ? → List(b)
  collect(s) = // all sub-terms for which s succeeds
    <conc>(<opt(s)>, <kids; mapconc(collect(s))>)
  kids :: ? → List(?) // list of children, using
  kids : _#(xs) → xs // generic term deconstruction
strategies
  vars :: ? → List(Var)  // all variables
  vars = collect(?Var(_)) // in a program
```

Figure 12. Type unifying strategies

Polymorphism Revisited. The prenex polymorphism in our type system has a limitation. The current runtime of Stratego does not support strategies with explicit type arguments. Therefore, type arguments must be completely abstract, and cannot be used in dynamic type assertions. Our type system gives a type error when a cast with a type variable must be inserted. This means that polymorphic strategies and rules are less gradual than the rest of gradually typed Stratego.

4 A Type System for Core Gradual Stratego

In this section, we present an algorithmic type system for Core Gradual Stratego, which formalizes the ideas of the previous section. Previous work on formalizations of Stratego were based on System S, the calculus of strategy combinators [14, 29]. We consider a larger core language, including signatures and definitions of rules and strategies, as these matter for the type system.

4.1 Core Gradual Stratego

Figure 13 defines the grammar of Core Gradual Stratego, which is Core Stratego extended with (dynamic) types, top-level type annotations, casts, and proxies. We use vector notation instead of a Kleene star for lists, to mirror the type rules. These are all zero-or-more, except for the second alternative of *ot*, which is one-or-more.

Signatures *o*, *ot* consist of constructor definitions with zero or more arguments, and injection definitions (an example is back in Figure 2). Types *t*, or sorts, include built-in types (string, int) and (parameterized) types. We extend types with the dynamic type `?`, and the ill-formed type. We also add strategy types *st* to describe strategy arguments.

Definitions *d* include strategy definitions and rule definitions. Including rewrite rule in the core is not necessary for dynamic expressivity; usually they are presented as sugar, by translation to a strategy sequence of a match, side-condition, and a build [29]. However, we give type annotated rules a more intuitive typing.

$$
\begin{array}{rcll}
sl & ::= & \textit{string literals} & \\
f, x & ::= & \textit{names} & \\
o & ::= & f : ot \mid : ot & \text{signatures} \\
ot & ::= & t \mid \overline{t} \rightarrow t & \text{sig types} \\
t & ::= & f(\overline{t}) \mid \texttt{string} \mid ? \mid \texttt{ill-formed} & \text{types} \\
st & ::= & (\overline{st})\ t \rightarrow t \mid ? & \text{strategy types} \\
d & ::= & f(\overline{f}) = s & \text{definitions} \\
 & \mid & f(\overline{f}) : e \rightarrow e \texttt{ where } s & \\
s & ::= & f(\overline{s}) \mid \texttt{fail} \mid \texttt{id} \mid ?e \mid !e & \text{strategies} \\
 & \mid & \{ x : s \} \mid s ; s \mid s < s + s & \\
 & \mid & \texttt{cast}(c) \mid \texttt{proxy}(\overline{sc}|c,c|s) & \\
e & ::= & x \mid \_ \mid sl \mid f(\overline{e}) \mid e :: t & \text{terms} \\
c & ::= & \texttt{id} \mid t & \text{coercions} \\
sc & ::= & \texttt{id} \mid st & \text{strategy coercions}
\end{array}
$$

Figure 13. The Stratego core grammar. Any Stratego program can be desugared to these core constructs.

Strategy expressions s consist of calls to strategies or rules, the explicit match failure strategy, the identity strategy, match, build, scoping a variable, sequences, guarded choice, and finally the additions, cast and proxy (note that the two vertical bars that are part of the syntax of proxy). Term expressions e consist of variables, wildcards, string literals, constructor applications, and additionally type ascription. Casts and proxies apply coercions, which can be a coercion c to a type or the identity coercion.

4.2 Algorithmic Type System

We present an algorithmic type system, written as a transformation on the program that inserts casts while type checking. The type system follows an abstract interpretation approach.

Meta-properties. We currently do not have a formal dynamic semantics for Core Gradual Stratego, but assuming a reasonable dynamic semantics as sketched in previous sections, we postulate that the type system presented here is sound for the typed subset of the core language, i.e. all programs accepted by the type system execute without type-errors. Furthermore, if a program after cast insertion executes without type-errors, we postulate that the same program without type annotations (but preserving type tests) will also execute and give the same result. In other words, type annotations do not affect the behaviour or results of type-correct programs. Finally, we believe that the presented transformation rules are deterministic if alternatives are tried in order, and that their execution terminates for all inputs.

Environments. The type system is meant to be modular, therefore we assume that environment Γ contains all available definitions and their types, so we can resolve calls. The environment also contains the types of defined constructors and the precomputed transitive relation of defined injections, i.e. every mapping from one type into another without constructor becomes a subtype fact, to be available for the [subinj] rule of Figure 14. Most of the information in the environment flows through the type system like a store. Local variables can be bound to different types at different points in the program, e.g. inside and after a guarded choice. The arity of dynamically typed strategy arguments is discovered during type checking.

Type coercion rules $\boxed{\Gamma \vdash t \rightsquigarrow t : c}$

$$
\frac{\Gamma \vdash t_1 <: t_2}{\Gamma \vdash t_1 \rightsquigarrow t_2 : \texttt{id}}\ [\mathsf{csub}]
\qquad
\frac{}{\Gamma \vdash ? \rightsquigarrow t : t}\ [\mathsf{ccheck}]
$$

Subtype rules $\boxed{\Gamma \vdash t <: t}$

$$
\frac{}{\Gamma \vdash t_1 <: ?}\ [\mathsf{dyntop}]
\qquad
\frac{}{\Gamma, t_1 <: t_2 \vdash t_1 <: t_2}\ [\mathsf{subinj}]
$$

$$
\frac{\Gamma \vdash \overline{t_1} <: \overline{t_2}}{\Gamma \vdash f(\overline{t_1}) <: f(\overline{t_2})}\ [\mathsf{subcovar}]
\qquad
\frac{}{\Gamma \vdash t <: t}\ [\mathsf{subrefl}]
$$

Figure 14. Algorithmic coercion and subtype rules. Alternative rules are tried in order.

Coercion, Subtyping and Bounds. Figure 14 defines the computation of coercions from one type to another. The [csub] rule defers to subtyping. The coercion from a subtype to its supertype is the identity coercion. The [ccheck] rule defines that it is also possible to go from the ? type to any specific type by coercing to that type.

The subtyping rules in Figure 14 define that ? is a supertype of any other type ([dyntop]). The [subinj] rule looks up an injection in the environment. The [subcovar] rule defines that parameterised sorts are co-variant in their parameters. Subtyping is reflexive ([subrefl]), but not transitive. Instead we took the transitive closure of the injections before the start of the program.

With this definition we can define the least-upper-bound and greatest-lower-bound on types. The representative type is used for any two types from an injection cycle. In all other cases it is the expected bound wherever injections form a lattice. Note that injections do not guarantee a lattice structure, so there may not be unique bounds. In those cases where we do not find a unique bound, we use the ? type.

Definitions. Figure 15 defines the typing judgements for rule and strategy definitions. Rules for definitions look up their type in the environment, and register their arguments and type variables. For strategy definitions [sdef], we then check the body, and insert a cast after the body with a coercion from the result type to the output type of the definition. We slightly abuse the syntax of vectors here for pairing the strategy arguments with their types ($\overline{f : st}$). These strategy names are paired with their arity and those together with

Definition type rules $\boxed{\Gamma \vdash d \Rightarrow d \dashv \Gamma}$

$$\dfrac{\begin{array}{c} len(\overline{f}) = j \qquad \Gamma_1 = \Gamma_1', \langle f, j\rangle : (\overline{st})\ t_1 \to t_2 \\ \Gamma_1, \overline{f} \mathrel{\overline{:}} \overline{st};\ t_1 \vdash s_1 \Rightarrow s_2 \dashv \Gamma_2; t_3 \qquad \Gamma_2 \vdash t_3 \rightsquigarrow t_2 : c \end{array}}{\Gamma_1 \vdash f(\overline{f}) = s_1 \Rightarrow f(\overline{f}) = s_2;\ \mathtt{cast}(c) \dashv \Gamma_2}\ [\mathsf{sdef}]$$

$$\dfrac{\begin{array}{c} len(\overline{f}) = j \qquad \Gamma_1 = \Gamma_1', \langle f, j\rangle : (\overline{st})\ t_1 \to t_2 \\ \Gamma_1, \overline{f} \mathrel{\overline{:}} \overline{st};\ t_1 \vdash_m e_1 \Rightarrow e_3; s_2 \dashv \Gamma_2; t_3 \qquad \Gamma_2; t_2 \vdash_b e_2 \Rightarrow e_4 \dashv \Gamma_3; t_2' \qquad \Gamma_3; t_3 \vdash s_1 \Rightarrow s_3 \dashv \Gamma_4; t_3' \end{array}}{\Gamma_1 \vdash f(\overline{f}) : e_1 \to e_2 \ \mathtt{where}\ s_1 \Rightarrow f(\overline{f}) : e_3 \to e_4 \ \mathtt{where}\ s_2;\ s_3 \dashv \Gamma_4}\ [\mathsf{rdef}]$$

Figure 15. Algorithmic typing rules for adding definitions to the environment. Alternative rules are tried in order.

Strategy typing $\boxed{\Gamma; t \vdash s \Rightarrow s \dashv \Gamma; t}$

$$\dfrac{\Gamma_1; t_1 \vdash_m e_1 \Rightarrow e_2; s \dashv \Gamma_2; t_2}{\Gamma_1; t_1 \vdash ?e_1 \Rightarrow ?x@e_2;\ s;\ !x \dashv \Gamma_2; t_2}\ [\mathsf{match}] \qquad \dfrac{\Gamma_1; t_1 \vdash_b e_1 \Rightarrow e_2 \dashv \Gamma_2; t_2}{\Gamma_1; t_1 \vdash !e_1 \Rightarrow !e_2 \dashv \Gamma_2; t_2}\ [\mathsf{build}]$$

$$\dfrac{\begin{array}{c} len(\overline{s_1}) = j \qquad \Gamma_1 = \Gamma_1', \langle f, j\rangle : (\overline{st})\ t_1 \to t_2 \\ \Gamma_1; \overline{st} \overrightarrow{\vdash_{sa}} \overline{s_1} \Rightarrow \overline{s_2} \dashv \Gamma_2 \qquad \Gamma_2 \vdash t_0 \rightsquigarrow t_1 : c \end{array}}{\Gamma_1; t_0 \vdash f(\overline{s_1}) \Rightarrow \mathtt{cast}(c); f(\overline{s_2}) \dashv \Gamma_2; t_2}\ [\mathsf{call1}] \qquad \dfrac{\begin{array}{c} len(\overline{s_1}) = j \qquad \Gamma_1 = \Gamma_2, \langle f, 0\rangle : ? \\ \Gamma_2, \langle f, j\rangle : (\overline{?})\ ? \to ?; \overline{?} \overrightarrow{\vdash_{sa}} \overline{s_1} \Rightarrow \overline{s_2} \dashv \Gamma_3 \end{array}}{\Gamma_1; t_1 \vdash f(\overline{s_1}) \Rightarrow f(\overline{s_2}) \dashv \Gamma_3; ?}\ [\mathsf{call2}]$$

Figure 16. Excerpt of the algorithmic typing rules for the core Stratego strategy expressions. Alternative rules are tried in order.

their type are put into the environment. Dynamically typed strategy arguments are put into the environment with arity 0, which is relevant later in Figure 18.

For rule definitions [rdef] we check their input and output term expressions against the type definition first, before we check the side-condition. This entails that we may check variables in the output term that are bound in the side-condition. The rules for terms in build position take this into account. Since rules and strategies can be overloaded in the number of strategy arguments, and these remain separate definitions at runtime, the types of these definitions are associated to a pair of name and number of arguments.

```
strategies // [sdef] example
  assert-Stat :: ? → Stat
  assert-Stat = id
// ⇒ assert-Stat = id; cast(Stat)
```

Strategies. We present an excerpt of the rules for strategy expressions in Figure 16. The typing judgement is $\Gamma; t \vdash s \Rightarrow s \dashv \Gamma; t$. These are the input context, type of the current term, and a strategy expression, which are transformed to a strategy expression with inserted casts, an output context, and the type of the current term after the strategy expression.

The [match] rule defers to the type rules for term expressions. Term expressions have a set of rules for match position and for build position. During a match of a term expression, there can be parts that need to be tested with a cast. Such casts are collected as a strategy expression, which is executed after the match. We use syntactic sugar to bind the current term to fresh variable x, and restore the current term value after the casts. The [build] rule defers to the build position term expression type rules, where casts can be inserted inline with some syntactic sugar.

For the strategy call we have two alternatives for typing. The [call1] and [call2] rules are attempted in order, the first to succeed is used. The [call1] rule looks up the strategy in the environment based on its name and arity. Then it type-checks the strategy arguments, and finally it computes the coercion required for the current type to match the input type of the found strategy. A cast is inserted before the call with that coercion. Of course an implementation of these rules may leave out the cast if the computed coercion is the identity coercion. The [call2] rule attempts to find the strategy under the arity 0 with the type **?** in the environment. This is how strategy arguments are registered in the environment, for an untyped strategy with strategy arguments. If found, the environment is updated to reflect the discovered arity of the strategy argument.

The rules for calls work on vectors of arguments, and therefore need a form of iteration over these vectors. We put an arrow over the turnstile of a rule to denote that we map that particular rule over the vector arguments, and thread the other parts (typically the environment). For example:

$$\begin{array}{l} \Gamma_1; ?, \overline{?} \overrightarrow{\vdash_{sa}} s_1, \overline{s_2} \Rightarrow s_3, \overline{s_4} \dashv \Gamma_3 \equiv \\ \Gamma_1; ? \vdash_{sa} s_1 \Rightarrow s_3 \dashv \Gamma_2 \wedge \Gamma_2; \overline{?} \overrightarrow{\vdash_{sa}} \overline{s_2} \Rightarrow \overline{s_4} \dashv \Gamma_3 \end{array}$$

Build term typing

$$\boxed{\Gamma; t \ \vdash_b e \Rightarrow e \dashv \Gamma; t}$$

$$\frac{\Gamma_1 \ \vdash t_1 <: \texttt{string}}{\Gamma_1; t_1 \ \vdash_b sl \Rightarrow sl \dashv \Gamma_1; t_1}\ \text{[bstring]} \qquad \frac{\Gamma_1 \ \vdash t_2 <: t_1 \qquad \Gamma_1; t_2 \ \vdash_b e_1 \Rightarrow e_2 \dashv \Gamma_2; t_2'}{\Gamma_1; t_1 \ \vdash_b e_1 :: t_2 \Rightarrow e_2 \dashv \Gamma_2; t_2}\ \text{[bascr]}$$

$$\frac{\Gamma_1 \ \vdash t_2 \rightsquigarrow t_1 \ : \ c \qquad t_3 = t_1 \sqcap t_2}{\Gamma_1, x : t_2; t_1 \ \vdash_b x \Rightarrow \texttt{<cast(}c\texttt{)>}\, x \dashv \Gamma_1, x : t_3; t_3}\ \text{[bvar1]} \qquad \frac{}{\Gamma_1; t_1 \ \vdash_b x \Rightarrow x \dashv \Gamma_1, x : t_1; t_1}\ \text{[bvar2]}$$

$$\frac{len(\overline{e_1}) = j \qquad \Gamma_1 = \Gamma_1', \langle\!\langle f, j \rangle\!\rangle : \overline{t_1} \rightarrow t_1 \qquad \Gamma_1; \overline{?} \ \overrightarrow{\vdash_b}\ \overline{e_1} \Rightarrow \overline{e_2} \dashv \Gamma_2; \overline{t_2} \qquad \Gamma_2 \ \overline{\vdash}\ \overline{t_2} <: \overline{t_1}}{\Gamma_1; ? \ \vdash_b f(\overline{e_1}) \Rightarrow f(\overline{e_2}) \dashv \Gamma_2; t_1}\ \text{[bconstr1a]}$$

$$\frac{len(\overline{e_1}) = j \qquad \Gamma_1 = \Gamma_1', \langle\!\langle f, j \rangle\!\rangle : \overline{t_1} \rightarrow t_1 \qquad \Gamma_1; \overline{?} \ \overrightarrow{\vdash_b}\ \overline{e_1} \Rightarrow \overline{e_2} \dashv \Gamma_2; \overline{t_2}}{\Gamma_1; ? \ \vdash_b f(\overline{e_1}) \Rightarrow f(\overline{e_2}) \dashv \Gamma_2; \texttt{ill-formed}}\ \text{[bconstr1b]}$$

$$\frac{len(\overline{e_1}) = j \qquad \Gamma_1 = \Gamma_1', \langle\!\langle f, j \rangle\!\rangle : \overline{t_1} \rightarrow t_1 \qquad \Gamma_1; \overline{?} \ \overrightarrow{\vdash_b}\ \overline{e_1} \Rightarrow \overline{e_2} \dashv \Gamma_2; \overline{t_2} \qquad \Gamma_2 \ \vdash t_1 <: t_0 \qquad \Gamma_2 \ \overline{\vdash}\ \overline{t_2} <: \overline{t_1}}{\Gamma_1; t_0 \ \vdash_b f(\overline{e_1}) \Rightarrow f(\overline{e_2}) \dashv \Gamma_2; t_1}\ \text{[bconstr2]}$$

Figure 17. Algorithmic typing rules for the core Stratego terms in build position. Alternative rules are tried in order.

Build Terms. Terms in build position are checked against the expected result type of the term (Figure 17). The resulting type is the type of the term that was actually built, while the environment can contain new bindings for local variables discovered in the term.

The rule for building string literals [bstring] defines that we can only have string literal where the expected type is a subtype of strings. The only subtypes of the built-in string type are aliases of the string type. The rule for type ascription [bascr] checks that the ascribed type is a subtype of the expected type, and propagates it to the subterm.

The rules for variables are again based on whether the variable was already bound to a type in the environment. If so, in [bvar1] we compute a coercion from the type of the variable to the expected type. We update the binding of the variable and the resulting type to the greatest-lower-bound of the two types. This can be done since the coercion to a subtype is only allowed if the supertype is ?. In that we have discovered that after this build the variable must be the more specific type or the cast failed that we insert. The inserted cast uses syntactic sugar from the full Stratego language to be able to apply a strategy during the build of a term and put the result of that application there. If the variable that is built is not in the environment, rule [bvar2] is in effect, which will record the discovered type information. This will typically occur when we type-check the output term of a rule before the side-condition of the rule binds the variable.

Constructors can be built in contexts where a dynamically or a statically typed term is expected. The [bconstr1a] and [bconstr1b] rules handle the first case. With a dynamically typed term expected we look up a constructor of the right arity[3], and in [bconstr1a] we handle the case where despite the dynamically typed context, the child terms build the correct types for the constructor. In that case we can return the static type of the constructor. When the child terms do not build the correct types for the constructor, [bconstr1b] returns the `ill-formed` type, which is only a subtype of ?. The [bconstr2] rule is the statically typed term construction, where we find a constructor of the right (sub)type for the expected output and require the child terms to have the correct child types. These constructor rules show another judgement with an adapted turnstile. In this case, with a bar over the turnstile, we have nothing to thread, but we lift the judgement point-wise over the vectors.

```
strategies // [rdef,bconstr2,bvar1] example
  exp-to-stat :: ? → Stat
  exp-to-Stat: maybe-exp → Exp(maybe-exp)
// ⇒ exp-to-Stat:
//      maybe-exp → Exp(<cast(Stat)> maybe-exp)
```

Match Terms. Terms in match position are checked against the type of the term they are matched against. The rules are mostly analogous to those of build terms, and therefore elided.

Strategy Arguments. Strategy arguments require special care (Figure 18). These arguments become closures, and if we add casts before we close over them, these casts will become invisible to the strategy that receives the arguments. A strategy argument that is passed around a few times can accumulate a number of casts, creating new closures for the cast and call sequence every time. This problem was identified

[3]We use slightly different ⟨⟨brackets⟩⟩ for this pair to visually distinguish it from the ⟨strategy⟩ pair.

Strategy argument typing $\boxed{\Gamma; st \ \vdash_{sa} s \Rightarrow s \dashv \Gamma}$

$$\frac{\Gamma_1 = \Gamma_1', \langle f, 0\rangle : ? \qquad \Gamma_1 \ \vdash ? \rightsquigarrow t_2 \ : \ c}{\Gamma_1; (\overline{st_1}) \ t_1 \longrightarrow t_2 \ \vdash_{sa} f() \Rightarrow \mathtt{proxy}(\overline{\mathtt{id}} | \mathtt{id}, c | f()) \dashv \Gamma_1} \text{[saref1]}$$

$$\frac{\begin{array}{c}\Gamma_1 = \Gamma_1', \langle f, 0\rangle : (\overline{st_2}) \ t_3 \longrightarrow t_4 \qquad \Gamma_1 \ \overline{\vdash} \ \overline{st_1} \rightsquigarrow \overline{st_2} \ : \ \overline{sc_1} \\ \Gamma_1 \ \vdash t_1 \rightsquigarrow t_3 \ : \ c_1 \qquad \Gamma_1 \ \vdash t_4 \rightsquigarrow t_2 \ : \ c_2\end{array}}{\Gamma_1; (\overline{st_1}) \ t_1 \longrightarrow t_2 \ \vdash_{sa} f() \Rightarrow \mathtt{proxy}(\overline{sc_1} | c_1, c_2 | f()) \dashv \Gamma_1} \text{[saref2]} \qquad \frac{\Gamma_1; t_1 \ \vdash s_1 \Rightarrow s_2 \dashv \Gamma_2; t_3 \qquad \Gamma_2 \ \vdash t_3 \rightsquigarrow t_2 \ : \ c}{\Gamma_1; () \ t_1 \longrightarrow t_2 \ \vdash_{sa} s_1 \Rightarrow \mathtt{proxy}(| \mathtt{id}, c | s_2) \dashv \Gamma_2} \text{[sa]}$$

Figure 18. Algorithmic typing rules for strategy arguments in a call. Alternative rules are tried in order.

and solved by Herman et al. [9], by constructing special values containing the casts and the passed function. The values are called proxies, and they can be introspected at runtime. When creating a proxy directly around another proxy, no new proxy needs to be allocated. Instead the coercions are merged into the existing proxy. This saves memory, and also means that some of the dynamic type checks happen at that time, instead of when the proxy is executed.

In this core language, strategies have two kinds of arguments, the default term argument, and strategy arguments. So proxies save the coercions for the extra arguments, a coercion for input and output, and the strategy argument itself. The [saref1] rule handles strategy arguments that are a references to dynamically typed strategies, [saref2] handles references to statically typed strategies (note the contravariance on the return type coercion), and [sa] handles other strategy expressions. Arbitrary strategy expressions need to become closures regardless of proxy objects. We can require the strategy expression to handle the exact type that it will receive as a strategy argument, but we still save the coercion on the return value in a proxy, so it is accessible at runtime.

Refer back to Figure 11 for an example of proxy insertion.

4.3 Polymorphism, Parametricity, and TP

We have purposefully not discussed support for polymorphism in the type system so far. The language features of this core subset of Stratego have enough moving parts already. In this subsection we list the modifications to the type system that handle polymorphism. The environment now additionally holds pairs of type variables α and their type. Top-level definitions now have type schemes so the type variables can be made fresh at lookup time. Definitions register strategy arguments as definitions *without* type schemes. The subtyping relation binds unbound type variables to the sub- or supertype they are compared to. Casts can only be inserted for types without type variables. No coercions can be used to a type that has an unbound type variable or a type variable from the type scheme of the definition. The latter restriction is due to type erasure in the runtime of Stratego.

Parametricity. Typical Stratego code can inspect any term with a match strategy or a rewrite rule. This means that, if allowed, Stratego code could inspect terms that are polymorphic in the strategy type. This would violate parametricity, the guarantee that terms of polymorphic type cannot be deconstructed, but it fits the style of Stratego programs rather well. Therefore we chose not to guarantee parametricity in the type system. Type variables from the type scheme of the definition receive special treatment, where they behave as **?** throughout the definition, since the definition can be instantiated with dynamic types for the type variables.

Type Preserving. The special type preserving type is an important piece of the puzzle to type generic traversals and reduce the number of casts inserted around generic traversals. **TP** may seem to be simply a $\rightarrow$ a, but there are three key differences. The first is that **TP** provides a limited form of higher-rank types. The second is that while parametrically polymorphic strategies may be called with terms that are dynamically typed and possibly ill-formed, a type preserving strategy must be called with a statically typed term. The third is that a type preserving strategy must either return the given term, call a type preserving strategy on it, or inspect the term with a pattern match or type test. In other words, we cannot call a typed strategy with the input term unless the type of the term has been inspected. We do not insert casts to assert its type as we do for polymorphic strategies, as a type preserving, heterogenous strategy should be useable in a generic traversal that tries to apply the strategy everywhere in a tree. When the strategy is not applicable, it should produce a match failure, not a cast error, otherwise the generic traversal mechanism does not work.

5 Evaluation

By design of the type system, the following requirements from Section 3.1 were met: existing programs are accepted without annotations, generic traversals are supported in both statically and dynamically typed code, we only needed to add minimal (top-level) type annotations which preserves Stratego's concise style, and the types can be modularly checked for integration with the incremental Stratego compiler.

In this section we evaluate the requirement that we can migrate code from untyped to typed. To do so, we have implemented a stand-alone prototype type checker for Stratego,

```
rules
  step(|state): IADD() →
    <pop(|2); push(|r); next> state
  where
    <top(|2)> state ⇒ [Int(v1), Int(v2)];
    <addS> (v2, v1) ⇒ r
```

Figure 19. An example of a step rule in the Jasmin interpreter.

```
top(|n) = ?JBCState(
  [JBCFrame(_, _, _, JBCStack(<take(|n)>), _)|_], _)
top = top(|1)
_______________Typed and corrected to:_______________
top(|int) :: State → List(Value)
top(|n) = ?JBCState(
  [JBCFrame(_, _, _, JBCStack(<take(|n)>), _)|_], _)
top :: State → Value
top = top(|1); \[head] → head\
```

Figure 20. A bug found by adding types and inspecting errors. The `top` strategy with term argument gives the first `n` values on the stack, whereas the top strategy without argument *should* give the top value (given how it was consistently used) but instead gave a singleton list with the top value.

and used it to type-annotate the pre-existing Stratego code of a Spoofax programming language project. This project is Jasmin, a Java Byte Code assembler language [18]. In particular, the project implements the JasminXT version of the language as defined by Reynaud [21]. The Jasmin language project is used in a compiler construction course, where students transform MiniJava to Jasmin, and then use the Jasmin definition to compile to bytecode files. As a result, the part of the codebase that does compilation is well-exercised. Another part of the codebase, a Jasmin interpreter written in Stratego, is not used in the course. The interpreter defines signatures for a state of the JVM and a step rewrite rule that rewrites the state based on a Jasmin instruction. A program is then a list of instructions, which can be executed over the empty state. Figure 19 shows a simple step rule that does integer addition by manipulation of the JVM operand stack and addition provided by Stratego.

The Jasmin codebase contains 3253 lines of manually written Stratego code divided over 36 files. We do not count blank lines and comments, nor generated constructor signatures from the grammar. There are 49 manual definitions of constructor signatures for intermediate representations, and 146 named strategies/rules (counted by unique name).

Changes and Bugs. During our evaluation we added 74 type signatures of standard library strategies that are used in the code, and 117 type signatures to the code of the project. In general, we find that the compilation of Jasmin (the well-exercised part) is free of type errors and is mostly written with a static typing discipline. Most type-annotated strategies (85 out of 117) can be given a static type without use of the dynamic type. Some of the manually defined constructors do not have correct types which was not checked before.

The interpreter clearly has not seen testing. We found numerous type-related bugs, such as inconsistent use of integers and strings containing integers, a pattern match one constructor too shallow for the case that was handled, and the use of the wrong strategy which gives back a list instead of an element from the list. Another example is given in Figure 20. Most strategies and rules were written with static type discipline in mind. This part did have some overloading (which could be split for static typing) as well as a messy combination of constructors from the Jasmin AST and a newly introduced intermediate representation without a combining supertype. The interpreter 'step' rewrite rule also takes explicitly ill-formed terms, variants of the defined terms where all strings with numbers are replaced by integers.

Interpretation and Conclusion. We conducted a small experiment that shows that using the gradual type system is useful to guide the transition to a system with a better type discipline. In a codebase which was written with some type discipline in mind, this was an exercise in understanding the existing code and adding type annotations. In some cases a small refactoring was easy enough to do and resulted in well-typed strategies. In the process we found that previously added type annotations helped us find the right annotations for the next strategy, and the type system helped us find some mistakes made when originally guessing the type annotation for a strategy. This reassured us that we were indeed adding a consistent set of type annotations to the code.

A noteworthy exception to the smooth experience was the interpreter 'step' rewrite rules. This part of the code was written on an intermediate representation that looked very close to the defined constructors, but with some amount of desugaring and preprocessing. This intermediate representation was close to the original representation, but with small changes consistently applied on a larger number of constructors. Writing the signatures for this would be tedious, and we did not do this as part of the case study. We are not yet sure if this situation is a good argument for our gradual types, a programming style we should discourage in future Stratego code, or if we should add language support for defining these types of intermediate representations more easily.

6 Related Work

Dynamically Typed Strategies. The ELAN language introduced rewrite systems with labeled rules that could be invoked from strategies [1]. Luttik and Visser [13] extended these strategies with modal operators (all, one) for generic traversal. Visser et al. [30] used this as the basis for the design of Stratego. Visser and Benaissa [29] define System S, a core language with operational semantics for rewriting, with matching and building as primitive operations. Visser [27] demonstrates the use of strategies for *strategic pattern matching* to dynamically check the (type) structure of terms, e.g. to recognize subsets of a type schema.

Stratego is applied in the Stratego/XT [2] transformation tool suite and the Spoofax language workbench [11] for the definition of program normalization [4], type checkers [8], program analyses [3], code generators [10], and more. Smits et al. [26] introduced an incremental compiler for Stratego.

Erdweg et al. [7] introduce typesmart constructors, smart constructors that dynamically check type-correctness of constructed terms. They integrate support for typesmart constructors in the Stratego runtime. Their approach is all-or-nothing, requiring all intermediate values to be well-typed, which precludes the dynamic typing scenarios of Section 2.

Statically Typed Strategies. Typing strategies was pursued by Lämmel and Visser [16] in the context of the Strafunski Haskell library for generic programming inspired by Stratego [17]. They identified the concepts of type preserving and type unifying strategies. Lämmel [14] formalizes these with the **TP** and **TU**(b) types in a type system for System S [29]. We have adopted the **TP** type, but have opted to model type unying strategies with type dynamic as $? \longrightarrow b$. SYB is a design pattern encoding the statically typed fragment of Stratego's strategies in Haskell [15], using type classes and higher-rank polymorphic functions to provide type transformations (type-preserving) and queries (type-unifying) on arbitrary data, made usable by deriving instances of the type classes automatically.

Gradual Types. The term gradual types was introduced by Siek and Taha [24] in a paper that adds optional type annotations to simply-typed lambda calculus, by an orthogonal extension with a *consistency* relation. This work kicked off an new line of research in adding gradual types to many different type systems and languages, entirely too many to enumerate here. As noted earlier in the paper, we took the work of Herman et al. [9] to heart and applied it in our type system. This work introduces proxies as special closures that can be introspected at runtime to add more coercions to the input and output. This solves a space leak from adding normal closures with casts around a function when it is passed back and forth between statically and dynamically typed higher-order functions.

Xie et al. [32] describe a gradual, higher-kinded polymorphic type system that guarantees parametericity, and manages to keep gradual types orthonogal from subtyping induced by polymorphic functions. This work gave us confidence that our type system with limited higher-kinded polymorphism without parametricity (**TP**) would be possible too. We took inspiration from their notation for algorithmic typing rules. We differ from the research into gradual types in functional type systems, which typically have a separate consistency relation for gradual types which is orthogonal to the rest of the type system. Our approach to subtyping and cast computation is very close to the pessimistic and optimistic subtyping of Muehlboeck and Tate [19].

Dynamic Rewrite Rules. In this paper we have not considered the extension of Stratego with *scoped dynamic rewrite rules* [3], which are used to define context-sensitive transformations such as type checking [8], function inlining [28], and transformation based on data-flow analysis [20]. Such rules are dynamic in the sense that new rule instances are added at run time. With respect to static typing, such dynamic rules can be type checked with the type system from this paper. When dynamic rules have type dynamic their outputs may be checked at run time. When dynamic rules have a static type annotation their definitions and applications are checked like regular rewrite rule definitions. So, while the behaviour of dynamic rules is dynamic their typing is static.

7 Conclusion

We have introduced the design of a gradual type system for Stratego with a series of idiomatic examples, and presented a formal definition of the type system for Core Gradual Stratego. This type system can statically type many stratego programs, including type preserving and type unifying generic traversals. The gradual types support partially dynamically typed Stratego programs and provide a migration path for existing dynamically typed Stratego code.

To evaluate the implementation of our type checker, we demonstrated this migration path on an existing Stratego codebase, and find it works rather well. What is left to future work is investigating whether we need more language support for easily defining the types of intermediate representations, and to experimentally evaluate the overhead of the casts that our type checker inserts. We also hope to track more properties statically, such as partiality of strategies and boundedness of variables, and use all this static information for optimisations in the back-end of the compiler.

A final word of warning for those who wish to implement a type checker for gradual types: The silent insertion of casts can easily hide bugs in your type system when you write small programs for exploratory testing. So test profusely, including the results after cast insertion.

Acknowledgments

We would like to thank Michael Greenberg and the anonymous reviewers for their valuable comments and suggestions.

This research was supported by a gift from the Oracle Corporation.

References

[1] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. 1998. An overview of ELAN.

Electronic Notes in Theoretical Computer Science 15 (1998), 55–70. http://www.elsevier.com/gej-ng/31/29/23/39/23/show/Products/notes/index.htt#022

[2] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. 2008. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming* 72, 1-2 (2008), 52–70. https://doi.org/10.1016/j.scico.2007.11.003

[3] Martin Bravenboer, Arthur van Dam, Karina Olmos, and Eelco Visser. 2006. Program Transformation with Scoped Dynamic Rewrite Rules. *Fundamenta Informaticae* 69, 1-2 (2006), 123–178. https://content.iospress.com/articles/fundamenta-informaticae/fi69-1-2-06

[4] Martin Bravenboer and Eelco Visser. 2004. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004*, John M. Vlissides and Douglas C. Schmidt (Eds.). ACM, Vancouver, BC, Canada, 365–383. https://doi.org/10.1145/1028976.1029007

[5] Luís Eduardo Amorim de Souza and Eelco Visser. 2020. Multi-Purpose Syntax Definition with SDF3. In *Software Engineering and Formal Methods - 18th International Conference, SEFM 2020 (Lecture Notes in Computer Science)*. Springer.

[6] Jasper Denkers, Louis van Gool, and Eelco Visser. 2018. Migrating custom DSL implementations to a language workbench (tool demo). In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05-06, 2018*, David Pearce 0005, Tanja Mayerhofer, and Friedrich Steimann (Eds.). ACM, 205–209. https://doi.org/10.1145/3276604.3276608

[7] Sebastian Erdweg, Vlad A. Vergu, Mira Mezini, and Eelco Visser. 2014. Modular specification and dynamic enforcement of syntactic language constraints when generating code. In *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*, Walter Binder, Erik Ernst, Achille Peternier, and Robert Hirschfeld (Eds.). ACM, 241–252. https://doi.org/10.1145/2577080.2577089

[8] Zef Hemel, Danny M. Groenewegen, Lennart C. L. Kats, and Eelco Visser. 2011. Static consistency checking of web applications with WebDSL. *Journal of Symbolic Computation* 46, 2 (2011), 150–182. https://doi.org/10.1016/j.jsc.2010.08.006

[9] David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient gradual typing. *Higher-Order and Symbolic Computation* 23, 2 (2010), 167–189. https://doi.org/10.1007/s10990-011-9066-z

[10] Lennart C. L. Kats, Martin Bravenboer, and Eelco Visser. 2008. Mixing source and bytecode: a case for compilation by normalization. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, Gail E. Harris (Ed.). ACM, 91–108. https://doi.org/10.1145/1449764.1449772

[11] Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, Reno/Tahoe, Nevada, 444–463. https://doi.org/10.1145/1869459.1869497

[12] Gabriël Konat, Michael J. Steindorfer, Sebastian Erdweg, and Eelco Visser. 2018. PIE: A Domain-Specific Language for Interactive Software Development Pipelines. *Programming Journal* 2, 3 (2018), 9. https://doi.org/10.22152/programming-journal

[13] Bas Luttik and Eelco Visser. 1997. Specification of Rewriting Strategies. In *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF 1997) (Electronic Workshops in Computing)*, M. P. A. Sellink (Ed.). Springer-Verlag, Berlin.

[14] Ralf Lämmel. 2003. Typed generic traversal with term rewriting strategies. *Journal of Logic and Algebraic Programming* 54, 1-2 (2003), 1–64. https://doi.org/10.1016/S1567-8326(02)00028-0

[15] Ralf Lämmel and Simon L. Peyton Jones. 2003. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of TLDI 03: 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, New Orleans, Louisiana, USA, January 18, 2003*, Zhong Shao and Peter Lee (Eds.). ACM, 26–37. https://doi.org/10.1145/604174.604179

[16] Ralf Lämmel and Joost Visser. 2002. Typed Combinators for Generic Traversal. In *Practical Aspects of Declarative Languages, 4th International Symposium, PADL 2002, Portland, OR, USA, January 19-20, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2257)*, Shriram Krishnamurthi and C. R. Ramakrishnan (Eds.). Springer, 137–154. http://link.springer.de/link/service/series/0558/bibs/2257/22570137.htm

[17] Ralf Lämmel and Joost Visser. 2003. A Strafunski Application Letter. In *Practical Aspects of Declarative Languages, 5th International Symposium, PADL 2003, New Orleans, LA, USA, January 13-14, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2562)*, Verónica Dahl and Philip Wadler (Eds.). Springer, 357–375. http://link.springer.de/link/service/series/0558/bibs/2562/25620357.htm

[18] Jon Meyer and Troy Downing. 1997. *Java Virtual Machine.* O Reilly.

[19] Fabian Muehlboeck and Ross Tate. 2017. Sound gradual typing is nominally alive and well. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017). https://doi.org/10.1145/3133880

[20] Karina Olmos and Eelco Visser. 2005. Composing Source-to-Source Data-Flow Transformations with Rewriting Strategies and Dependent Dynamic Rewrite Rules. In *Compiler Construction, 14th International Conference, CC 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3443)*, Rastislav Bodík (Ed.). Springer, 204–220. https://doi.org/10.1007/978-3-540-31985-6_14

[21] Daniel Reynaud. 2006. *JasminXT Syntax.* Available at http://jasmin.sourceforge.net/xt.html.

[22] John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*. 513–523.

[23] Jeremy G. Siek, Ronald Garcia, and Walid Taha. 2009. Exploring the Design Space of Higher-Order Casts. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5502)*, Giuseppe Castagna (Ed.). Springer, 17–31. https://doi.org/10.1007/978-3-642-00590-9_2

[24] Jeremy G. Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, Vol. 6. 81–92.

[25] Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4609)*, Erik Ernst (Ed.). Springer, 2–27. https://doi.org/10.1007/978-3-540-73589-2_2

[26] Jeff Smits, Gabriël D. P. Konat, and Eelco Visser. 2020. Constructing Hybrid Incremental Compilers for Cross-Module Extensibility with an Internal Build System. *Programming Journal* 4, 3 (2020), 16. https://doi.org/10.22152/programming-journal

[27] Eelco Visser. 1999. Strategic Pattern Matching. In *Rewriting Techniques and Applications, 10th International Conference, RTA-99, Trento, Italy, July 2-4, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1631)*, Paliath Narendran and Michaël Rusinowitch (Eds.). Springer, 30–44. https://doi.org/10.1007/3-540-48685-2_3

[28] Eelco Visser. 2001. Scoped Dynamic Rewrite Rules. *Electronic Notes in Theoretical Computer Science* 59, 4 (2001), 375–396. https://doi.org/10.1016/S1571-0661(04)00298-1

[29] Eelco Visser and Zine-El-Abidine Benaissa. 1998. A core language for rewriting. *Electronic Notes in Theoretical Computer Science* 15 (1998),

422–441. https://doi.org/10.1016/S1571-0661(05)80027-1

[30] Eelco Visser, Zine-El-Abidine Benaissa, and Andrew P. Tolmach. 1998. Building Program Optimizers with Rewriting Strategies. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, Matthias Felleisen, Paul Hudak, and Christian Queinnec (Eds.). ACM, Baltimore, Maryland, United States, 13–26. https://doi.org/10.1145/289423.289425

[31] Philip Wadler. 1989. Theorems for Free!. In *FPCA*. 347–359. https://doi.org/10.1145/99370.99404

[32] Ningning Xie, Xuan Bi, and Bruno C. D. S. Oliveira. 2018. Consistent Subtyping for All. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 3–30. https://doi.org/10.1007/978-3-319-89884-1_1